Propuesta de una metodología de programación de operaciones detallada para órdenes de trabajo en operaciones secuenciales a través de algoritmos heurísticos basados en lógica proposicional articulados sobre una serie de autómatas programables

Autor
MIGUEL ANGEL RINCON CHAPARRO

Director
JOSE JAIRO SORIANO MENDEZ
Msc. Ingeniería Industrial

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS
Maestría en Ingeniería Industrial
Énfasis en Automática e Informática Industrial
Bogotá, Colombia
marzo de 2017

CONTENIDO

RESUMEN 6
PALABRAS CLAVES 7
INTRODUCCION 8
1. PROBLEMA DE INVESTIGACION 11
1.1. Planteamiento del problema 11
1.2. Formulación del problema 12
1.3. Sistematización del problema 13
2. OBJETIVOS 14
2.1. Objetivo General 14
2.2. Objetivos Específicos 14
3. JUSTIFICACION 15
4. MARCO REFERENCIA 17
4.1. Marco teórico 17
4.2. Marco conceptual 19
4.2.1. Antecedentes 20
4.2.2. Algoritmos de programación 24
4.2.3. Control de entrada y salidas. 25
4.2.4. Cartas y gráficas de programación 25
5. HIPOTESIS 31
6. CONGRUENCIA METODOLOGICA 33
7. DESCRIPCION DEL PROBLEMA SECUENCIACION 39
7.1. PASO 1: Proceso de producción 42
7.2. PASO 2: Conjuntos 46
7.2.1. El lenguaje de las matemáticas: Axiomas y primeras definiciones 46
7.2.2. Complemento, unión e intersección 49
7.2.3. Producto cartesiano 53
7.3. PASO 3: Relaciones 55
7.3.1. Relaciones de equivalencia 58
7.3.2. Relaciones de equivalencia y particiones 62
7.3.3. Relaciones de orden 62
7.4. PASO 4: Funciones de recurrencia - Inducción matemática 66
7.4.1. Algoritmos recursivos 66

7.4.2. Funciones recursivas 67
7.4.2.1. Fundamentos de la teoría de funciones recursivas 68
7.4.2.2. Alcance de las funciones recursivas primitivas. 73
7.4.2.3. Funciones recursivas parciales. 78
8. DISEÑO E IMPLEMENTACION 82
8.1. Problema de secuenciación con Satisfacibilidad Booleana SAT 82
8.1.1. Programación Lógica Computacional Aplicada 97
8.2. PASO 5: Sistemas Lógicos Proposicionales: Algebra Booleana 103
8.2.1. El razonamiento lógico 103
8.2.2. Lógica computacional : Programación 104
8.2.3. Aplicación: formulas lógicas proposicionales para $3SAT - FNC$ 105
8.3. PASO 6: Sistema lógico combinacional 106
8.3.1. Procedimiento de análisis 106
8.3.2. Procedimiento de diseño : algoritmo combinacional 108
8.4. PASO 7: Sistema lógico secuencial 112
8.4.1. Análisis de la lógica secuencial con reloj 115
8.4.2. Reducción y análisis de estados 123
8.4.3. Procedimiento de diseño 127
9. DISENO E IMPLEMENTACION 143
9.1. PASO 8. Autómata finito 143
9.2. PASO 9. Lenguajes recursivos 144
9.3. PASO 10: Máquina de estados finitos: Máquina de Turing MT 146
10. REPRESENTACION GRAFICA 168
10.1. PASO 11. Secuenciación 168
10.1.1. Algoritmo 170
10.1.2. El poder de los lenguajes de programación 192
10.1.3. El lenguaje de la lógica proposicional 192
11. ANÁLISIS DE CASOS 202
11.1. CASO 1: Pasos del proceso de fabricación del neumático (llanta). 202
11.2. CASO 2: Pasos del proceso de fabricación de acero (Laminas) 217
12. CONCLUSIONES Y RECOMENDACIONES 220
12.1. Conclusiones 220
12.2. Recomendaciones 223
13. LIMITACIONES DE LA INVESTIGACION 226
14. REFERENCIAS BIBLIOGRAFICAS 229
15. ANEXOS 232

LISTA DE FIGURAS

FIGURA 1. FORMULACIÓN DE UN PROBLEMA NP-COMPLETO A 3SAT-FNC (AUTOR) 13
FIGURA 2. PLANTA DE PRODUCCIÓN – DISTRIBUCIÓN POR CENTROS DE TRABAJO 22
FIGURA 3. CARTA DE PROGRAMACIÓN – DIAGRAMA DE GANTT 26
FIGURA 4. CARTA DE PROGRAMACIÓN – CÓDIGO BINARIO 31
FIGURA 5. MAPA CONCEPTUAL DE LA METODOLOGÍA PLANTEADA PARA DESARROLLAR SECUENCIACIÓN DE TRABAJOS 37
FIGURA 6. PIRÁMIDE DE LA AUTOMATIZACIÓN INDUSTRIAL 38
FIGURA 7. REPRESENTACIÓN DEL CONJUNTO $A \subset E$ 50
FIGURA 8. REPRESENTACIÓN DEL CONJUNTO Ac 51
FIGURA 9. REPRESENTACIÓN GRÁFICA DE CONJUNTOS POR PRODUCTO pi ASOCIADOS A CENTRO DE TRABAJO cn 54
FIGURA 10. RELACIÓN DE PERTENENCIA $\{0, 1\}$ DE CENTRO DE TRABAJO cn A CADA PRODUCTO pi 54
FIGURA 11. CONJUNTOS CADA PRODUCTO pi ASOCIADOS A CENTROS DE TRABAJO cn 55
FIGURA 12. RELACIÓN DE PERTENENCIA Y EQUIVALENCIA $\{0, 1\}$ DE CENTRO DE TRABAJO cn A CADA PRODUCTO pi 57
FIGURA 13. UNA RELACIÓN REFLEXIVA Y UNA RELACIÓN IRREFLEXIVA. 58
FIGURA 14. JERARQUÍA PROVISIONAL DE LAS FUNCIONES COMPUTABLES 77
FIGURA 15. REVISIÓN DE LA JERARQUÍA PROVISIONAL MOSTRADA EN LA FIGURA 15 78
FIGURA 16. JERARQUÍA DE LAS CLASES DE FUNCIONES SEGÚN LA TEORÍA DE FUNCIONES RECURSIVAS 80
FIGURA 17. LÓGICA COMBINACIONAL APLICADA A RECURSOS POR CENTRO DE TRABAJO 107
FIGURA 18. DIAGRAMA DE BLOQUES DE UN SISTEMA COMBINACIONAL 108
FIGURA 18. (ÁRBOL SINTÁCTICO) SISTEMA LÓGICO COMBINACIONAL DEFINIDO EN CENTRO DE TRABAJO POR RECURSOS 110
FIGURA 19. (ÁRBOL SINTÁCTICO) SISTEMA LÓGICO COMBINACIONAL PARA MATERIALES mi POR PRODUCTOS pi 111
FIGURA 20. (ÁRBOL SINTÁCTICO) SISTEMA LÓGICO COMBINACIONAL PARA MÁQUINAS **ei** POR PRODUCTOS **pi** 111
FIGURA 21. (ÁRBOL SINTÁCTICO) SISTEMA LÓGICO COMBINACIONAL PARA MANO DE OBRA **li** POR PRODUCTOS **pi** 111
FIGURA 23. (ÁRBOL SINTÁCTICO) SISTEMA LÓGICO COMBINACIONAL | CENTRO DE TRABAJOS EN PLANTA DE PRODUCCIÓN **ERROR! BOOKMARK NOT DEFINED.**
FIGURA 24. DIAGRAMA DE BLOQUES DE UN SISTEMA SECUENCIAL 112
FIGURA 25. (ÁRBOL SINTÁCTICO) SISTEMA LÓGICO SECUENCIAL *BIESTABLE* CON CONECTORES $\wedge$ Y $\vee$ 114
FIGURA 33. SISTEMA LÓGICO SECUENCIAL CON PULSOS DE RELOJ 117
FIGURA 34. DIAGRAMA DE ESTADO PARA EL SISTEMA LÓGICO SECUENCIAL 119
FIGURA 35. IMPLEMENTACIÓN DE LAS FUNCIONES DE ENTRADA AL BIESTABLE $JA = BC'x + B'Cx'$ Y $KA = B + y$ 122
FIGURA 36. DIAGRAMA DE ESTADO 124
FIGURA 37. DIAGRAMA DE ESTADO 129
FIGURA 38. (ÁRBOL SINTÁCTICO) DIAGRAMA DE BLOQUES DEL SISTEMA LÓGICO COMBINACIONAL 131
FIGURA 39. MAPAS PARA EL SISTEMA COMBINACIONAL 132
FIGURA 40. (ÁRBOL SINTÁCTICO) DIAGRAMA LÓGICO DEL SISTEMA SECUENCIAL 133
FIGURA 41. (ÁRBOL SINTÁCTICO) DIAGRAMA LÓGICO PARA EL SISTEMA LÓGICO SECUENCIAL 135
FIGURA 42. DIAGRAMA DE ESTADO PARA EL SISTEMA LÓGICO COMBINACIONAL 136
FIGURA 43. DIAGRAMA DE ESTADO DE UN CONTADOR BINARIO DE 3-BIT 137
FIGURA 44. DIAGRAMA DE OPERACIONES – SALIDA DEL SISTEMA LÓGICO PROPOSICIONAL 168

LISTA DE TABLAS

TABLA 1. PLANEACIÓN Y CONTROL DE DATOS PARA EL CAP 9

TABLA 2. COMBINACIONES DE LOS VALORES $\mathbf{0}$ Y $\mathbf{1}$ **ERROR! BOOKMARK NOT DEFINED.**

TABLA 3. NEGACIÓN DE UNA PROPOSICIÓN $\boldsymbol{p}$ **ERROR! BOOKMARK NOT DEFINED.**

TABLA 4. CONJUNCIÓN DE DOS PROPOSICIONES $\boldsymbol{p}, \boldsymbol{q}$, DENOTADA $\boldsymbol{p} \wedge \boldsymbol{q}$ **ERROR! BOOKMARK NOT DEFINED.**

TABLA 5. DISYUNCIÓN DE DOS PROPOSICIONES $\boldsymbol{p}, \boldsymbol{q}$, DENOTADA $\boldsymbol{p} \vee \boldsymbol{q}$**ERROR! BOOKMARK NOT DEFINED.**

TABLA 6. LA IMPLICACIÓN DE DOS PROPOSICIONES $\boldsymbol{p}, \boldsymbol{q}$, DENOTADA $\boldsymbol{p} \rightarrow \boldsymbol{q}$ **ERROR! BOOKMARK NOT DEFINED.**

TABLA 7. LA DOBLE IMPLICACIÓN DE DOS PROPOSICIONES $\boldsymbol{p}, \boldsymbol{q}$, DENOTADA $\boldsymbol{p} \leftrightarrow \boldsymbol{q}$ **ERROR! BOOKMARK NOT DEFINED.**

TABLA 8. FORMULA LÓGICA PROPOSICIONAL FBF DE RECURSOS Y LA SALIDA DEL CENTRO DE TRABAJO $\boldsymbol{C11}$ 106

TABLA 9. FORMULA LÓGICA PROPOSICIONAL FBF DE PRODUCTOS $\boldsymbol{pi}$ POR CENTRO DE TRABAJO $\boldsymbol{Cij}$ 110

TABLA 10. BIESTABLE CON DESPEJE DIRECTO **ERROR! BOOKMARK NOT DEFINED.**

TABLA 11. TABLA DE ESTADOS PARA EL SISTEMA LÓGICO SECUENCIAL 117

TABLA 12. TABLAS DE EXCITACIÓN BIESTABLE **ERROR! BOOKMARK NOT DEFINED.**

TABLA 13. TABLA DE ESTADO 129

TABLA 14. TABLA DE EXCITACIÓN 130

TABLA 15. TABLA DE EXCITACIÓN PARA EL SISTEMA LÓGICO SECUENCIAL 134

TABLA 16. TABLA DE EXCITACIÓN PARA UN CONTADOR BINARIO DE 3 BITS 138

RESUMEN

La investigación se enfoca en las operaciones de programación en el corto plazo, debido a que se ha convertido en un aspecto crítico en el entorno actual por mantener los inventarios bajos y los tiempos de entrega cortos, y por lo tanto el obtener una secuencia de trabajos amerita que sea planteada una manera para poder visualizarla en el tiempo, fundamentado en que lo sistemas tradicionales de MRP y MRPII no hacen hincapié en la programación real de la capacidad finita.

Se parte de un razonamiento básico de los problemas de secuenciación de trabajos en programación de operaciones, en los diversos entornos de maquinas se dividen en varias clases: en *una sola máquina* se deben procesar todos los trabajos. Las maquinas pueden procesar a lo más de un trabajo a la vez. Una vez que un trabajo se ha procesado en la maquina, se termina. Muchos de ellos se resuelven con facilidad, pero no es probable que existan algoritmos eficientes para los modelos que incluyen tiempos de preparación o tardanza en diferentes métricas. Asimismo los métodos de búsqueda que se pueden modificar para resolver la mayor parte de los modelos de programación.

Después se incluyen los *modelos de programación paralela*, para máquinas múltiples que procesan por completo cualquier trabajo. Varias maquinas que puede realizar el mismo tipo de procesamiento se llaman maquinas paralelas. Un trabajo se puede procesar en cualquiera de las maquinas, y una vez procesado por cualquiera de ellas, queda terminado. A menos que se diga lo contrario, se supone que todas la maquinas paralelas son idénticas. El tiempo para procesar un trabajo en una de varias maquinas idéntica es independiente de que maquina lo haga. Para *máquinas idénticas*, los problemas de tiempo de flujo tienen una solución sencilla. Para otras medidas, las *listas heurísticas* son un buen enfoque. Se aprecia la cota del peor caso para el modelo del lapso de producción.

Para las *plantas de producción continua* que tienen múltiples operaciones para cada trabajo, las cuales se realizan en el mismo orden, solo se puede resolver con facilidad los problemas de dos máquinas y algunos de tres máquinas; otros modelos requieren soluciones heurísticas. Cada trabajo debe procesarse en cada máquina exactamente una vez. Más aun, todos los trabajos siguen la misma ruta; esto es, deben visitar las maquinas en el mismo orden. Sin pérdida de generalidad, se puede numerar las maquinas de manera que la 1 sea la primera, la 2 la segunda, y así sucesivamente. Un trabajo no puede comenzar su procesado en la segunda maquina hasta no terminar el de la primera. Las líneas de ensamble y las células son ejemplos típicos de producción continua. Se enuncian varios heurísticos, junto con las colas que se pueden usar en el algoritmo de ramificación y acotamiento. Los métodos de búsqueda también son útiles para encontrar buenos programas de *permutación* para producción continua.

Los talleres de *producción intermitente*, que son los modelos de programación más difíciles, es más general que el de producción continua; cada trabajo tiene una ruta única. Se analiza la heurística de despacho y varias reglas de prioridad. (Daniel Sipper, 1998)

Todos estos modelos en sus diferentes entornos de maquinas, hacen pensar que la secuenciación de trabajos es un problema de enumeración de carácter combinatorio y al ser representado en diversos escenarios de producción con centros de trabajos en su distribución; la elección de una secuencia entre todas las opciones se vuelve cada vez más compleja de obtener en la medida que aumente en número de trabajos y el numero de maquinas y se remite a trabajar con métodos de optimización y algoritmos heurísticos para hallar una posible solución.

Al plantear una propuesta metodología para determinar heurísticas basado en sistemas lógicos proposicionales en diferentes niveles; donde se tengan en cuenta el razonamiento lógico y las métricas de desempeño para poderlo evaluar como problema de secuenciación de trabajo en programación de operaciones, teniendo en cuenta que los problemas de secuenciación son considerados de gran complejidad algorítmica tipo NP-completo, pueda ser expresado como un problema de satisfactibilidad booleana SAT. Es decir un problema de optimización pueda ser expresado en un problema de decisión y poderlo tratar.

El desarrollo de la investigación de fundamenta en plantear una metodología para obtener expresiones o formulas atómicas (fbf formulas bien formadas), a través de conceptos de la matemática discreta y combinatoria (*conjunto, relaciones y funciones*) y principalmente del estudio de la lógica proposicional (denominada lógica de primer orden LPO), para efectuar operaciones con las variables lógicas, cumpliendo con las propiedades del algebra boolena.

Una vez obtenidas estas formulas, se representan los diferentes flujos de procesos alineado al flujo de información con razonamientos en los sistemas y subsistemas lógicos de índole combinacional y secuencial. Con estos elementos se modela el funcionamiento de una máquina de estado finitos universal de Turing, que es considerada un autómata más poderoso para modelar las funciones lógicas obtenidas. Se explicará su funcionamiento con respecto a la metodología planteada y los objetivos a conseguir.

Se cuenta son los elementos necesarios para la puesta en marcha de la máquina de estados finitos: la cinta (secuencia de registros binarios $\{0,1\}$ del sistema lógico), el cabezal (función de transición de estados en diagramas de decisión binario BDD) y el programa (algoritmos efectuados con conceptos del algebra booleana y funciones de conmutación: seleccionadores, codificadores, sumadores, comparadores de magnitud, etc.) con el cual se interpreta a través de una función de transición, a través del tiempo y como resultado obtener una salida de tipo binario, que represente la secuencia de operaciones a programar en una planta de producción asociado a centros de trabajo y relacionado con los recursos inherentes para la transformación de productos.

PALABRAS CLAVES

Secuenciación, Complejidad, Computabilidad, Satisfactibilidad - SAT, NP-completo, Lógica combinacional y secuencial, Indecidible, Intratable, máquina de estados finitos- FMS, Recursividad, Algoritmos computacionales

INTRODUCCION

La programación de las operaciones es una parte fundamental para asegurar un efectivo desempeño del sistema productivo de la organización. Tiene como función determinar que operaciones se van a realizar sobre los distintos pedidos, durante cada momento del horizonte de planificación, en cada centro de trabajo, de forma que, con la capacidad disponible en cada uno de ellos, se cumplan las fechas de entrega planificadas, empleando el menor volumen de recursos e inventarios posibles.

La programación de las operaciones está referida a la gestión de las actividades del sistema productivo de corto plazo entre un día y un mes en términos generales. El propósito es disponer de un mecanismo de *monitoreo* y *seguimiento* a las diversas actividades de los procesos productivos de una organización. La programación de operaciones requiere de la consideración de diversos aspectos, entre los cuales está el horizonte de tiempo, que contempla los plazos en los cuales se desarrollan las actividades productivas de la organización y que puede ser de corto plazo (por lo general inferior a un año), mediano (que comprende entre uno y tres años), y largo plazo (cuando se realiza a más de tres años). Por otra parte, la programación y control de las operaciones considera entre los factores al sistema de producción, el cual requiere comprender cómo funciona y los correspondientes efectos sobre la programación de la producción continua, la producción por lotes y la producción por proyectos.

Finalmente, la programación y control de la producción requiere de la utilización de las *herramientas e instrumentos* que faciliten su acción y garanticen el logro de los resultados esperados y posibiliten un adecuados sistema de monitoreo. Entre las herramientas más utilizadas en la programación y control de las operaciones se destacan: el *gráfico Gantt*, el método PERT y la ruta crítica.

La programación de las operaciones constituye *"un sistema que programa, rastrea, vigila y controla la producción en la planta".* Estos sistemas también proporcionan nexos, en tiempo real, con los sistemas MRP, la planeación de procesos y productos y los que llegan más allá de la fábrica, entre ellos la administración de la cadena de suministro, la ERP, las ventas y la administración de servicios." . (J.Chase Richard B, 2005.).

El control de actividades de producción (CAP) es el área donde se desarrollan la actividad de planeación de control de manufactura (MAP) y el control de piso (SFC), a través de técnicas administrativas de prioridad y capacidad usadas para programar y controlar las operaciones de producción. El control de la prioridad asegura que las actividades de producción siga un plan de prioridad (ejemplo, el plan de requerimientos de materiales) controlando las órdenes, los proveedores y la producción interna.

El sistema de *control de prioridad* está centrado en el *estado* y *secuencia* de los trabajos para los diferentes centros de trabajo enfocados en sistemas de producción *flow shop;* cuyas actividades incluyen liberación de órdenes, despacho (o programación del trabajo), y control del avance. Como las órdenes, las fechas y las cantidades cambian, el sistema de control de prioridad debe reflejar constantemente las prioridades válidas, ya sean mayores o menores que las planeadas previamente.

El sistema de *control de capacidad* está centrado en las cargas (en horas estándar) en las instalaciones productivas. Las actividades de control de capacidad incluyen control de tiempo y de entrega, balanceo de cargas de trabajo, y control de las entradas y salidas. Ayuda *monitoreando* los centros de trabajo para asegurarse de que están proporcionando la cantidad de mano de obra y tiempo de equipo que es necesario (y fue planeado) para realizar el trabajo programado. Controlando la carga en los centros de trabajo, los tiempos de entrega de producción pueden ser controlados y las capacidades pueden ser mejor utilizadas.

El CAP es aquel grupo de actividades directamente responsables de manejar la transformación de planes ordenados a una serie de salidas. Gobierna a corto plazo la planeación detallada de operaciones, la ejecución y las actividades de monitoreo necesarias para el control de flujo de una orden desde el momento en que la orden se libera por el sistema de planeación para su ejecución, hasta que la orden se completa cumpliendo las especificaciones. Asimismo es responsable de hacer la asignación detallada y final de trabajo, de la capacidad de máquinas, herramientas y materiales a las diferentes órdenes en competencia. Se obtienen los datos de las actividades que se llevan a cabo en la planta, las cuales involucran los progresos de varias órdenes y el estado de los recursos, lo que hace que la información esté disponible para el sistema de planeación. Finalmente el sistema es también responsable de asegurar que las ordenes liberadas a la planta por el sistema de planeación, se concluyan en un tiempo y a un costo efectivo.

Se considerara que CAP es un subsistema dentro del sistema de manufactura. Se complementa con otros sistemas como la planeación de requerimientos de materiales (MRP) y la planeación de requerimientos de capacidades (CRP). El sistema CAP debe proporcionar información segura sobre: el estado actual de los trabajos (ejemplo, qué órdenes están en proceso y dónde); trabajos pendientes; la adecuación de materiales y capacidad; utilización de equipo y mano de obra y progreso, eficiencia y los trabajos.

Tabla 1. Planeación y control de datos para el CAP

	Datos de prioridad	**Datos de Capacidad**
Información de planeación	Números de artículos y descripción. Tamaño de lote y tiempo de entrega de producción. Cantidad disponible, asignada en un proceso.	Número y capacidad de centros de trabajo (CT). Centro de trabajo alternativos Eficiencia y tiempos de espera
Información de control	Numero de órdenes internas. Prioridad y fecha de entrega. Cantidad ordenada y balance Cantidad terminada, desechada y gastada.	Numero de operaciones Tiempo de arranque y corrida Fecha por entregar y tiempo de entrega disponible.

El uso de un sistema CAP requiere *información realista, comprensible y oportuna*. La tabla 1 resume algunos de los requerimientos de datos para un sistema CAP. Además, son usados registros de ruteo para especificar que se está haciendo (centros de trabajo, y cuanto tiempo va a tardar hacerlo - tiempo estándar). Aunque la programación realmente comienza con la planeación agregada y la programación maestra de producción, las actividades de control de prioridad de programación de órdenes y asignación de trabajo que se realizarán, son también conocidas generalmente como *programación de operaciones*.

Las estrategias de programación difieren ampliamente entre las empresas y van desde programación muy detallada hasta ninguna programación. La programación detallada de trabajo para las máquinas en un tiempo muy a futuro es generalmente impráctica. Una programación acumulada de cargas de trabajo totales, es útil para la planeación a largo plazo y las necesidades aproximadas de capacidad. Algunos lineamientos generales para programación de trabajos y cargas en instalaciones: Proporcionar una programación realista; considerar tiempos adecuados para las operaciones; considerar tiempos adecuados antes, durante y después de las operaciones; no programar todos los trabajos internamente; no programar toda la capacidad disponible; cargar sólo centros de trabajos seleccionados; hacer cambios ordenadamente cuando sea necesario y responsabilizarse por la programación. El resultado de la programación deberá incluir la secuencia de los trabajos, el establecimiento de la prioridad de paso de los pedidos en los diferentes centros de trabajo para cumplir las fechas de entrega planificadas con la menos cantidad de inventarios y recursos.

Realizar todos los trabajos disponibles en el orden en que van llegando es una causa común del incremento de los tiempos de entrega de producción y del exceso de trabajo en proceso. Los buenos sistemas de programación realizarán el trabajo a una tasa razonable que eliminará la acumulación innecesaria y reducirá el tiempo de entrega a los clientes. En los sistemas de servicios, la programación y trabajos es más difícil debido a que la recepción son órdenes (clientes) y los tiempos de servicio generalmente son más variables que en los bienes producidos.

1. PROBLEMA DE INVESTIGACION

En el mundo actual, las técnicas para la resolución de problemas de secuenciación, se ha limitado en problemas estructurados, dejando a los no estructurados o complejos como una opción lejana sobre lo que se puede o no se puede calcular. Separar aquellos problemas que se pueden resolver de forma eficiente mediante computadora de aquellos problemas que, en principio, pueden resolverse, pero que en la práctica consumen tanto tiempo que las computadoras resultan inútiles, y se denominan “insolubles” o “NP-difíciles”.

1.1. Planteamiento del problema

El problema de la secuenciación en programación de operaciones, se puede describir desde la matemática discreta y combinatoria como un problema de enumeración, basada en los principios del conteo. Enumerar o contar, parece un proceso obvio y no se le presta atención; y se encuentran problemas que pueden ser muy sencillos de formular pero un tanto complicados de resolver debido a la cantidad de opciones posibles para tomar una decisión. Se debe advertir sobre la seriedad y dificultad del simple conteo, ya que en la realidad se buscan soluciones basadas en el propio escrutinio o de experiencias anteriores, independientemente de si concuerdan o no con las propuestas actuales planteadas. Por lo tanto se convierten en un reto para resolver, ya que actualmente suelen haber diversas formas de abordar un problema para obtener una respuesta que pueda ser lo más acertada posible. Al secuenciar, se contarán con disposiciones de trabajos según un orden o diseño específico, y estas disposiciones suelen denominarse permutaciones. Al no considerarse el orden, la secuenciación puede ser una combinación; y aun más cuando se repiten en el tiempo, puede ser planteada como una distribución. Partiendo de estos enunciados, la secuenciación de trabajos es un problema de enumeración (*conteo*), donde la cantidad de selecciones, a pesar de ser finita, generan un gran número de posibles formas de organizar los trabajos, y por ende este número implica operaciones de orden factorial $n!$. En la medida que se incrementa el número m de maquinas para transformar los n números de trabajos, aumenta la complejidad para determinar cuál puede ser la secuencia que cumpla con los requerimientos necesarios de producción según las medidas de desempeño definidas en el sistema. Los objetivos sustitutos comunes incluyen minimizar el tiempo de flujo total, la tardanza total, el tiempo máximo de terminación, la tardanza máxima o el número de trabajos retrasados. Todos (excepto C_{max}, L_{max}, T_{max}) son *simplemente sumas*, sobre todos los trabajos, de las cantidades respectivas para cada trabajo. Si los trabajos no tienen todos la misma importancia; se puede calcular una medida ponderada multiplicando la medida por el peso apropiado para el trabajo. Como la suma de los tiempos de procesamiento es constante, minimizar el lapso (C_{max}) es equivalente a minimizar el tiempo ocioso o maximizar la utilización de las maquinas. Si los costos de mantener inventario dominan, el tiempo de flujo ponderado es una medida equivalente. Minimizar el tiempo de flujo es lo mismo que minimizar los tiempos de terminación, de retraso y los tiempos de espera de los trabajos. El tiempo de espera de los trabajos es inventario en proceso, de manera que

minimizar el tiempo de flujo equivalente a minimizar el número de trabajos de proceso. El tiempo de flujo ponderado corresponde al valor del inventario en proceso. El número de trabajos retrasados, la tardanza máxima y la tardanza total son medidas de servicio al cliente. Se pueden encontrar más detalles sobre los costos y medidas sustitutas en (Rinnooy Kan, 1976).

La mayoría de los problemas de programación suponen certidumbre en los datos; los tiempos de preparación son independientes del orden de procesado; todos los trabajos están disponibles de inmediato ($r_i = 0$); no existe precedencia entre los trabajos, y una vez que comienza el procesamiento de un trabajo, no se puede interrumpir.

1.2. Formulación del problema

Al determinar la cantidad de posibles de selecciones, ahora hay que considerar el tiempo para poderlas evaluar antes de tomar una decisión, y se determina que no es viable su análisis, ya que para obtener un resultado habrá transcurrido demasiado tiempo (años, meses, días), al ser planteado como un problema estático, la solución por ende caduca y como resultado se da trámite a los trabajos, los materiales se transforman y se avanza el proceso de producción. Las variables consideradas sufren cambios través del tiempo, como se concibe un problema en entornos dinámicos. Por ejemplo, un programa a corto plazo se crea para un conjunto de trabajos, pero mientras que se están procesando, llegan más trabajos, y este enfoque se extiende a otras instancias que cambian de estado.

Las técnicas clásicas y convencionales de secuenciación se tratan de optimización y heurísticas para determinar una solución que se pueda ser lo más acertada posible, mediante métodos diversos. Los algoritmos heurísticos dan soluciones que se espera sean optimas o cercanas a la optima en cualquier caso no se garantiza el resultado. Para muchos de los modelos de programación, los únicos algoritmos exactos que se conocen están basados en la *enumeración,* como el de la ramificación y acotamiento o la programación dinámica. Ver figura 1. (K.Ida, 2009)

¿Cómo obtener un resultado computacionalmente confiable para que no sean prohibitivos a la hora de procesarlos por su naturaleza combinatoria al utilizar algoritmos heurísticos en problemas de secuenciación de trabajos para escenarios flow shop?

Los algoritmos heurísticos se juzgan según su calidad y eficiencia. La calidad es la diferencia entre una solución heurística y la óptima, mientras que la eficiencia se refiere al esfuerzo realizado para obtener la solución. Ambas se pueden expresar teórica o empíricamente. Los heurísticos garantizan la calidad y la eficacia (es decir, cotas del peor en su desempeño). Un algoritmo heurístico es una receta para obtener una solución de un modelo. Un *caso* es un conjunto de datos específicos para el modelo. Los *algoritmos exactos* proporcionan una solución óptima para todos los casos del problema. Los *algoritmos heurísticos* dan soluciones que se espera puedan ser optimas o cercanas a la optima en cualquier caso. La *cota del peor caso* sobre la eficacia determina el numero de cálculos que deben realizar el algoritmo para cualquier problema práctico de una tamaño especifico (de ahí el nombre de *"peor caso"*). Un buen algoritmo acota el número de cálculos por una función polinomial del tamaño del problema. Esto no se ha hecho para algunos algoritmos heurísticos que deben ser evaluados en forma empírica.

Figura 1. Formulación de un problema NP-completo a 3SAT-FNC (autor)

Funciones Computables

indecidibles

decidibles

NP- completo

0 n MT t_n

P

3SAT-FNC

tratables

intratables

t_0 t_1 t_2 t_3 . . . t_{m-1} t_m t_{m+1} . . . t_{n-3} t_{n-2} t_{n-1} t_n

Problema de optimización → Problema de decisión

1.3. Sistematización del problema

Es extremadamente improbable que incluso la mejora de carácter exponencial en la velocidad de cálculo que el computador ha experimentado tenga un impacto para procesar problemas de secuenciación de trabajos sean insolubles. En particular, la teoría de los problemas intratables nos permite deducir si podremos enfrentarnos a un problema y escribir un programa para resolverlo (porque no pertenece a la clase de problemas intratables) o si tenemos que hallar alguna forma de salvar dicho problema:

Sub-problema 1: *¿Se podrá hallar una aproximación, al emplear un método heurístico o algún otro método para limitar el tiempo que el programa invertirá en resolver el problema de sucuenciacion de trabajos qué lo pueda hacer una computadora?* (esta área de se conoce como “decidibilidad”, y los problemas que una computadora puede resolver se dice que son “decidibles”).

Sub-problema 2: *¿Qué puede hacer una computadora de manera eficiente?* (esta área se conoce como “intratabilidad”, y los problemas que una computadora puede resolver en un tiempo proporcional a alguna función que crezca lentamente con el tamaño de la entrada se dice que son “tratables”).

Sub-problema 3: *¿Si esta limitación de intratabilidad es debida al diseño del lenguaje, o si, por el contrario, es un reflejo de las limitaciones de los procesos con algorítmicos heurísticos en general?* es decir, nos preguntamos *¿si la imposibilidad de calcular una función por un programa en un determinado lenguaje significa que el lenguaje es incapaz de expresar un algoritmo heurístico, o más bien que no existe ningún algoritmo que pueda calcular esa función?*. La pregunta es con respecto al poder expresivo de los lenguajes de programación. Lo que nos concierne ahora seria la cuestión de que aspectos deben incluirse en un lenguaje de programación, para garantizar que una vez diseñado e implementado no descubramos que existen problemas computables cuyas soluciones no pueden especificarse con dicho lenguaje.

2. OBJETIVOS

2.1. Objetivo General

Proponer una metodología basada en razonamientos lógicos para determinar un algoritmo heurístico basado en formulas y/o expresiones booleanas que representen un problema de secuenciación de trabajos en programación de operaciones para sistemas de producción *flow shop*, para ser procesado en una máquina de estados finitos (FMS por sus siglas en ingles) y obtener un cronograma por centros de trabajo.

2.2. Objetivos Específicos

- Representar un escenario de producción, considerando el sistema de producción *flow shop* para ser aplicanda la metodología desarrollada propuesta y considerando los principios descritos en la literatura reciente para problemas de secuenciación.

- Definir en términos de razonamiento lógico, el funcionamiento de cada una de las máquinas de estados finitos requeridas dentro del sistema, y la forma en que trabaja embebidas en otra de mayor jerarquía (entendiéndose por jerarquía los niveles en que interactúan los centros de trabajos, líneas de producción, secciones, etc.).

- Definir los conceptos (conjuntos), relaciones y funciones necesarias para establecer las clausulas que definen una ecuación lógica para problemas SAT de satisfacibilidad boolena.

- Establecer escenarios que se retroalimenten en forma continua a través del tiempo, con la finalidad de estar acotando las alternativas avaluadas en forma simultánea y evitar procesar alternativas que matemáticamente puede llegar a plantear un conjunto solución pero en tiempos de procesamiento exponencial.

- Representar una planta de producción como un sistema lógico proposicional para que se obtenga una salida/ entradas binarias $\{0,1\}$ para cada uno de los sistemas y subsistemas lógicos como centros de trabajo y otros definidos, para ser procesado en máquina de estados finitos.

3. JUSTIFICACION

(Morton y Pentico ,1993) afirman: "programar es el proceso de organizar, elegir y dar tiempos al uso de recursos para llevar a cabo todas las actividades necesarias, para producir las salidas deseadas en los tiempos deseados, satisfaciendo a la vez un gran número de restricciones de tiempo y relaciones entre las actividades y los recursos". Los trabajos son las actividades y las máquinas son los recursos. Entonces un *programa* especifica en tiempo en el que comienza y termina un trabajo en cada máquina, al igual que cualquier recurso adicional que se necesite. Una *secuencia* es un orden simple de trabajos; 3-1-2 indica que el trabajo 3 se hace primero, el 1 es el segundo y el 2 es el último. Si cada trabajo comienza tan pronto como es posible y se procesa sin interrupción para un tiempo dado de procesamiento, la secuencia determina los tiempos de inicio y terminación y, por lo tanto, determina la programación.

Determinar la mejor secuencia parece sencillo; solo se aumentaría todas las secuencias y se elige la que optimiza alguna medida de desempeño. Para 32 trabajos, el número de secuencias posibles es $32! = 2.6 \text{ x } 10^{35}$ secuencias disponibles. Si se supone que una computadora puede examinar mil millones de secuencias por segundo, tomaría 8.4×10^{15} siglos enumerarlas todas. Una computadora que fuera un millón de veces más rápida todavía tardaría 8.4×10^{9} siglos para examinarla. Con solo 16 trabajos existen más de 20 billones de programas, que a una tasa de mil millones de secuencias por segundo, podrían enumerarse más o menos en 8 meses. Muy pocos problemas de programación son tan sencillos. Los recursos adicionales (mano de obra, materia prima, etc.) y las dependencias entre trabajos (como la preparación) complican aun más el problema. El cuadro presenta más detalles sobre la complejidad de los problemas de programación. Esta explosión combinatoria muestra porque es difícil resolver algunos problemas de programación y la razón por la que su estudio es interesante.

Complejidad Algorítmica. Este es un análisis intuitivo más que una descripción matemática precisa. Aunque las siguientes afirmaciones no son exactas, son bastantes buenas para las consideraciones practicas. Un tratamiento más preciso se puede encontrar en (Garey y Johnson, 1979).

Para dar una justificación teórica de la calidad de un algoritmo heurístico, deben probarse matemáticamente que genera una solución dentro de cierto porcentaje de optimalidad, sin importar el problema particular que está resolviendo. Igual que con la eficacia, si la calidad de un algoritmo heurístico no tiene justificación teórica, deben juzgarse de manera empírica. Las pruebas empíricas consisten en generar y resolver muchos problemas prácticos y analizar los resultados. Se pueden determinar el tiempo promedio para resolver el problema. Se puede encontrar la diferencia entre las soluciones heurística y óptima para problemas pequeños. Para casos en los que no se puede obtener la solución optima, la solución heurística se compara con una cota sobre la solución óptima. Si existe una diferencia pequeña, la calidad del heurístico parece buena. La causa de una diferencia grande puede ser una cota débil o un heurístico pobre. Si el algoritmo heurístico tiene un buen desempeño en los casos de prueba, se hace la

suposición de que se desempeña igual en otros casos. Esto no puede ser cierto para algún caso difiera de las pruebas. Al ponerlo en marcha debe compararse la solución heurística con la solución actual.

Todos los algoritmos heurísticos que se estudian han tenido un buen desempeño en las pruebas. Estas pruebas empíricas están publicadas y se dan las referencias en la literatura sobre la programación. Esto no es una garantía de soluciones buenas o ni siquiera que puedan usar. La implantación de un algoritmo heurístico debe ser siempre ir precedida de pruebas con ejemplos típicos de la aplicación. (I-Hsuan Huang, 2011)

Un algoritmo eficiente es aquel en que el esfuerzo dedicado a un problema esta acotado por un polinomio cuyo grado es el tamaño del problema, como el número de trabajos. Un ejemplo seria el algoritmo para el árbol de expansión mínima, que se puede resolver a lo más en n^2 iteraciones, donde n es el número de arcos. Se dice que el algoritmo es de orden n cuadrada [$O(n^2)$]. Si el esfuerzo exponencial, por ejemplo [$O(2^n)$], el algoritmo no es eficiente. Un ejemplo sería el algoritmo de ramificación para variable {1,0}, que puede requerir 2^n nodos que deben explorarse.

El conjunto NP es el conjunto de todos los problemas que se pueden resolver por *enumeración total*. El conjunto P es un subconjunto de NP que consiste en todos los problemas para los que se conocen algoritmos eficientes. De nuevo, el árbol de expansión mínima es un ejemplo de un problema que pertenece a P. El conjunto NP -duro también es un subconjunto de NP, pero estos problemas son los que se han probado que son más difíciles de NP. De hecho, si alguien encontrara un algoritmo eficiente para cualquier problema de NP -duro, ese algoritmo podría modificar para resolver todos los problemas de NP en tiempo polinominal lo que haría al inventor rico y famoso. Parece poco probable que tal algoritmo llegue a existir. El problema del viajero se conoce como P -duro. Existen problemas en NP que en la actualidad no se pueden clasificar como pertenecientes a P o P -duro; es decir, que su complejidad no se conoce, y se llaman problemas abiertos. Si un problema es NP-duro, a menos que todos los problemas en NP se puedan resolver polinomialmente, deberá recurrirse a algoritmos exponenciales (enumerativos) para obtener soluciones óptimas. Para ejemplos pequeños, o en algunos casos particulares, estos algoritmos enumerativos pueden ser aceptables. Conforme el problema crezca, la explosión combinatoria hará que sea posible resolver el problema en un tiempo razonable. Otro enfoque será usar *algoritmos heurísticos* basados en el razonamiento lógico proposicional, que por lo general son buenos, pero no necesariamente proporcionan soluciones óptimas. Es común que los algoritmos heurísticos sean algoritmos polinomiales adaptados para una estructura de problema específica.

4. MARCO REFERENCIA

4.1. Marco teórico

Siempre se ha hecho programación de la producción; sin duda, la construcción de las pirámides la requirió. (Gantt, 1911) fue tal vez, el primero el impulsar el enfoque cuantitativo para la programación. Como área de atención, la programación en realidad tuvo un inicio a mediados de los 50. (Johnson, 1954), (Smith, 1956), (Jackson, 1956) y (McNaughton, 1959) desarrollaron los modelos y las soluciones clásicas. Estas publicaciones dieron fundamento al trabajo que siguió. Los primeros resultados fueron consolidados por (Thompson, 1963). Se modelaron muchos problemas de programación complicados como problemas de programación estera mixta, pero su naturaleza combinatoria los hizo posibles de resolver en un tiempo razonable. Esto hizo que el esfuerzo se tornara a los algoritmos heurísticos. (Maxwell, 1967) escribieron el primer libro de texto sobre programación de la producción. Una generación de investigadores y profesionales aprendió programación en el riguroso pero comprensible libro de (Baker, 1974).Los años 70 produjeron más resultados teóricos. El trabajo pionero sobre análisis heurísticos para el peor caso (Graham, 1969) y *la complejidad de varios modelos de programación de la producción* (Karp, 1972) condujeron a resultados importantes. (Kan R. , 1976) popularizo estos enfoques.

La investigación actual sobre programación parece estar dedicada a resultados más prácticos. (Panwalkar, 1973) y (McKay, 1988) investigaron la programación de la producción en la industria e identificaron aspectos relevantes. La reorganización de los sistemas de manufactura, en particular el movimiento para reducir el inventario y la necesidad de reducir tiempos de entrega ha hecho que la programación sea mucho más importante. Los estimulantes, aunque controvertidos, resultados de los sistemas de tecnologías de producción optimizada OPT (por sus siglas en ingles), despertaron el interés de muchos profesionales e investigadores; esto llevo a fijar la atención en la programación de cuellos de botella. Las técnicas de solución generales, como la simulación de recocido y algoritmos genéticos han permitido que algunos modelos más complejos se puedan resolver. El trabajo actual sobre el análisis estocástico y programación con criterios múltiples sustenta una promesa para el futuro. Un buen punto de partida es el libro de (Pinedo, 1995) que combina resultados teóricos con el estudio de casos.

Las computadoras actuales más rápidas, al alcance de todos, y las interfaces graficas amigables han hecho que sea más sencillo implantar la programación de la producción. APICS: The performance Advantage, una revista orientada a los profesionales en la práctica, han cambiado su evaluación del software, de los paquetes de MRP II a los paquetes de programación de capacidad finita. Se han publicado varios artículos por ejemplo (Huettle, 1993) y (Gilman, 1994) sobre el creciente interés en la programación de la producción/capacidad. Conforme ocurren los avances metodológicos, se hace borrosa la distinción entre planeación y programación (Lasserre, 1992) y una compañía no pueda programar rápida y correctamente estará en una tremenda desventaja.

Se dispone de paquetes para programación de la producción en abundancia y debe tenerse cuidado al comprar un paquete, ya que las habilidades varían mucho. (Dewilde, 1994) dan algunas sugerencias para elegir un paquete de

programación de la producción. (Buyer's, 1995) contiene información sobre más de 70 paquetes. Se puede encontrar un estudio profundo de varios de ellos en (Pinedo, 1995). Las funciones básicas que puede realizar un paquete de programación de producción son: *exhibición del programa; evaluación del programa, ajustes al programa, generación de la programación y simulación de la programación.*

Los paquetes pueden tener una o más características. La exhibición del programa es una descripción de un programa dado. Esto se puede hacer mediante una simple lista de con tiempos de inicio y terminación de cada trabajo y la combinación de las máquinas. Con más frecuencia se presenta una grafica de Gantt. Las graficas de Gantt van desde barras de caracteres ASCII a graficas de alta resolución en colores. Es importante que el usuario entienda bien el programa y saber dónde viene el programa exhibido. El software menos costoso requiere que el usuario genere programa; los de mayor precio contienen generadores del programa.

Algunos paquetes hacen la evaluación del programa. Esto proporciona medidas de la bondad del programa, que pueden incluir el lapso de producción, el tiempo de flujo y la tardanza, los que el uso de recursos, personal y niveles de trabajo en proceso y alguna otra información relevante. Igual que con la exhibición, esta información se puede mostrar en graficas o en una lista. Algunos paquetes de software proporcionan al usuario la posibilidad de cambiar o ajustar los programas. Los más elaborados permiten tomar e insertar operaciones en la misma grafica de Gantt; estos requieren interfaces de usuario graficas complejas. Los paquetes menos costosos permiten hacer cambios en los tiempos de inicio y terminación de los trabajos. De la mano con la evaluación del programa, los ajustes permiten al usuario hacer un análisis sencillo de qué pasa si.

Para exhibir, evaluar o ajustar un programa, primero debe generarse. Esta es la parte crítica de la programación de la capacidad finita. Muchos paquetes dejan la generación del programa al usuario y nada más la exhiben o la evalúan. Cuando es difícil obtener un buen programa a mano, con frecuencia estos paquetes no son muy útiles. Los paquetes más avanzados generan los programas, pero muchos de ellos no son muy buenos. El paquete puede generar un solo programa, varios programas o el mejor de un subconjunto de programas. Las reglas de despacho simples, como primero en entrar, primero en servir o con TPC (tiempo de procesamiento más corto) pueden producir un programa. Al usar varias reglas de despacho se pueden generar varios programas; después el usuario elige uno o el paquete solo le muestra el mejor. Se pueden usar heurísticos más avanzados, incluso procedimientos de ramificación y acotamiento, para generar un programa. Como la programación es difícil, el enfoque debe hacerse a la medida el problema que se tiene; un enfoque complejo equivocado puede ser peor que las reglas de despacho sencillas. Si el sistema de producción es complicado (como una planta grande de producción intermitente), puede ser necesaria una simulación del programa. Al someter a prueba el programa con simulación se puede examinar muchos factores no considerados en las decisiones de programación. Este puede ser en extremo útil, en particular si el heurístico usa tiempos de entrega, que la simulación puede estimar mejor. Los paquetes que simulan la planta son costosos.

En la experiencia, muy pocas compañías obtienen buenos resultados con paquetes baratos o con software comercial. Debido a las variaciones en los entornos de programación, con frecuencia se requiere software hecho a

la medida. Esto casi siempre incluye la compra de un paquete y algunas consultas con quien lo desarrolla, lo cual puede resultar costoso. Existen paquetes educativos de programación. (Pentico, 1993) ofrecen software , Parsifal, que resuelve la mayor parte de los modelos de programación; puede usar varios heurísticos para modelos de una sola máquina, máquinas paralelas, producción continua y producción intermitente. Dispone de varios algoritmos generales, incluyendo simulación de recocido. Su desempeño es bastante bueno para muchos problemas pequeños, y es posible que pueda incorporarse a un sistema de programación QS: Quantitative Systems (Chang, 1995) es un software general que realiza programación de producción continua e intermitente. Se pueden usar muchos de los heurísticos para la producción continua y la mayoría de las reglas de despacho se encuentran disponibles.

4.2. Marco conceptual

El objetivo del sistema de *control piso* (SFC) es administrar el flujo de materiales para cumplir con los planes especificados de producción. Un sistema de control de piso efectivo hará que una compañía cumpla con las fechas de entrega y mantenga un buen nivel de servicio al cliente. La administración del sistema puede conducir a la reducción de los inventarios de trabajo en proceso y a la reducción de los lead time de producción.

En algunas compañías, se tienen otros objetivos relacionados con el uso eficiente de la capacidad, mano de obra, máquinas, tiempo y materiales. El conjunto de objetivos particulares de una compañía es crítico para ver en qué forma estará diseñado el sistema control de piso. Los criterios del sistema de control de piso, se enfocan en las órdenes de producción y la *capacidad de los centros de trabajo* a través de los cuales pasarán las órdenes de trabajo. La elección de los objetivos del sistema de control de piso, refleja la posición de la compañía frente a los competidores, clientes y proveedores. Asimismo, refleja el objetivo principal de la compañía y las *restricciones* bajo las cuales opera.

La carga para un trabajo en particular sobre un centro de trabajo, los elementos del *lead time* y los datos de entrada, vendrían a ser conceptos básicos para el control de piso. Una de las técnicas sobre el control de piso seria, las *reglas de prioridad secuencial* para los trabajos en un centro de trabajo. Las reglas de prioridad secuencial están relacionadas con los trabajos que se van a correr en un centro de trabajo. Es decir, el trabajo a correr enseguida se puede determinar solamente cuando el trabajo anterior se completa. La regla de prioridad secuencial, es justamente como su nombre lo indica, una regla para determinar que trabajo procesar enseguida.

Planeación y control de planta SFC. El control de planta incluye las siguientes actividades:

1. Asignar una prioridad a cada orden, es decir, alguna medida de la importancia relativa de cada orden. Esto ayuda a establecer la secuencia de producción de las órdenes en los centros de trabajo.
2. Emitir listas de despacho para cada centro de trabajo. Estas listas informan al supervisor de producción que ordenes deben producirse en ese centro de trabajo, sus prioridades y cuando deberá terminarse cada orden.

3. Mantener actualizado el inventario en proceso (WIP, por sus siglas en ingles). Esto incluye conocer la ubicación de cada una de las ordenes y el número de componentes en cada orden dentro del sistema; llevar control del movimiento de las ordenes entre centro de trabajo cuando se usan fichas de traslado o transferencia y saber la cantidad de componentes en buen estado que han sobrevivido en cada paso de la producción, el desperdicio, el trabajo que se requiere y la cantidad de unidades faltantes en cada una de las ordenes.
4. Proporcionar un control de entradas y salidas de todos los centros de trabajo. Esto significa desarrollar información sobre la forma en que fluyen los trabajo entre los centros de trabajo.
5. Medir la eficiencia, la utilización y la productividad de los trabajadores y las máquinas en cada centro de trabajo.

Los departamentos de planeación y control de la producción realizan estas actividades e informan de los resultados a los gerentes de operaciones, de manera que se puedan tomar acciones correctivas cuando las ordenes vayan a retrasarse o cuando en los centro de trabajo ocurran problemas de capacidad o de carga de trabajo. (Norman Gaither, 2000)

4.2.1. Antecedentes

Esta parte del trabajo se proporcionan antecedentes para los problemas, modelos y algoritmos de programación. Se analizaran el entorno de la programación y se definirá la terminología y notación. Se usa la terminología clásica para programar los trabajos en las maquinas, pero estos son términos genéricos y existen muchas aplicaciones fuera de la manufactura.

Trabajos. Los trabajos son actividades a realizar; serán las órdenes de los clientes. Se supone que cada trabajo tiene un tiempo de procesado conocido. A menos que se establezca de otra manera, una vez que se comienza a realizarse un trabajo, debe procesarse continuamente hasta terminarlo; es decir, no se permiten interrupciones. Puede tenerse una fecha de entrega en la que el trabajo sebe estar terminado. Un trabajo también puede tener una fecha de inicio (o fecha de distribución de la orden), antes de la cual el procesamiento no puede comenzar.
Un trabajo puede depender de otro. Un tipo de dependencia ocurre cuando un trabajo debe preceder a otro; por ejemplo, un orificio no se puede roscar ante de perforarse. Otro tipo de dependencia ocurre cuando el tiempo necesario para un trabajo depende de que el trabajo anterior sea procesado. Si el trabajo **1** necesita un conjunto de herramientas en una máquina y el trabajo 2 necesita otro, entonces después de procesar el trabajo **1** debe cambiarse el herramental antes de procesar el trabajo 2. Si el trabajo 3 necesita las mismas herramientas que el trabajo 1, entonces la secuencia 1-2-3 requiere más preparaciones que la secuencia 1-3-2. Esto se conoce como *tiempo de preparación dependiente de la secuencia.* Si los tiempos de preparación no dependen de la secuencia,

se puede incluir en el tiempo de procesado. Se supone que los trabajos son independientes a menos que se diga lo contrario. Se supone que además que, se debe procesarse un trabajo en más de una maquina, el trabajo solo puede procesarse en una máquina a la vez; es decir, no se puede hacer otro trabajo al mismo tiempo.

Maquinas. Las máquinas procesan trabajos. En la manufactura, una máquina puede ser una máquina de moldeo automático. En situaciones fuera de manufactura, una máquina puede representar un aparato de rayos X, un mesero en un restaurante, una computadora o una pista de aterrizaje.
Para el problemas de varias máquinas se deben procesarse en ella todos los trabajos que puede realizar el mismo tipo de procesamiento se llaman máquinas *paralelas***.** Las máquinas pueden procesar a lo más de un trabajo a la vez. Un trabajo se puede procesar en cualquiera de las maquinas, y una vez procesado por cualquiera de ellas, queda terminado. A menos que se diga lo contrario, se supone que todas la máquinas paralelas son idénticas. El tiempo para procesar un trabajo en una de varias máquinas idéntica es independiente de que máquina lo haga. Un caso de máquinas paralelas es un grupo de máquinas de moldeo por inyección, cada una de las cuales puede hacer varias partes de plástico diferentes.

Medición. El mejor programa implica una medida de desempeño. Maximizar la ganancia o minimizar el costo son medidas obvias. Desafortunadamente, es difícil estimar los parámetros financieros que relacionan un programa con costo o ganancia. Por otro lado, no se conocen algoritmos eficientes para optimizar la ganancia o el costo en modelos de programación de la producción.
Se usan objetivos sustitutos para aproximar algunos costos relevantes. Las medidas sustitutas, por lo general, son funciones de tiempos de terminación para un programa dado. La mayoría de los sustitutos son medidas normales.

Una **medida normal** es una función del tiempo de terminación, en la que el objetivo es minimizar la función y donde esa función solo se incrementa si al menos un tiempo de terminación en el programa se aumenta. Sean

n = Número de trabajos que serán procesados

m = Número de maquinas

P_{ik} = Tiempo de procesado del trabajo i en la máquina k (p_i si $m = 1$)

r_i = Tiempo de liberación de la orden (o fecha de distribución) del trabajo i

d_i = Fecha de entrega del trabajo i

w_i = Ponderacion (importancia o valor) del trabajo i respecto a los otros trabajos

Dado un programa especifico, se define para cada trabajo i

$L_i = C_i - d_i$, **retraso** del trabajo i ($L_i < 0$ denota anticipación)

$T_i = \max\{0, L_i\}$, **tardanza** del trabajo i

$E_i = \max\{0, -L_i\}$, **adelanto** del trabajo i

$\delta_i = 1$, si el trabajo i se atrasa (es decir , si $T_i > 0$)

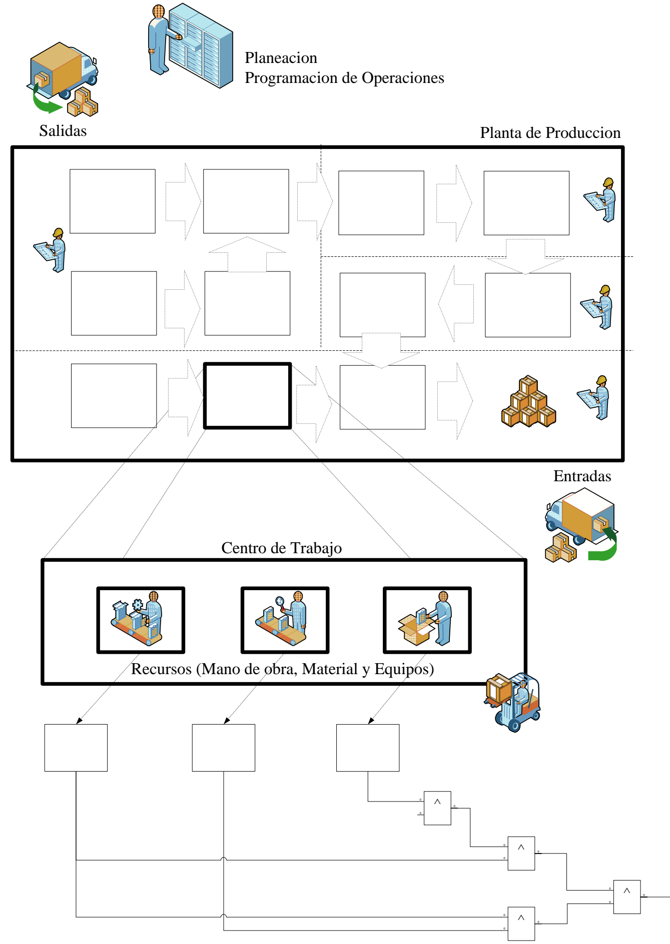
$\delta_i = 0$, si el trabajo i esta a tiempo o se adelanta (es decir, si $T_i = 0$)

$C_{max} = max_{i=1,n}\{C_i\}$, tiempo máximo de terminación de todos los trabajos o **lapso**

$L_{max} = max_{i=1,n}\{L_i\}$, **retraso máximo** de todos los trabajos

$T_{max} = max_{i=1,n}\{T_i\}$, **tardanza máxima** de todos los trabajos

Figura 2. Planta de producción – Distribución por centros de trabajo



Fuente : Autor

Los objetivos sustitutos comunes incluyen minimizar el tiempo de flujo total, la tardanza total, el tiempo máximo de terminación, la tardanza máxima o el número de trabajos retrasados. Todos (excepto C_{max} ,L_{max},T_{max}) son simplemente sumas, sobre todos los trabajos, de las cantidades respectivas para cada trabajo. Si los trabajos no tienen todos la misma importancia, se puede calcular una medida ponderada multiplicando la medida por el peso apropiado para el trabajo.

Como la suma de los tiempos de procesamiento es constante, minimizar el lapso (C_{max}) es equivalente a minimizar el tiempo ocioso o maximizar la utilización de las maquinas. Si los costos de mantener inventario dominan, el tiempo de flujo ponderado es una medida equivalente. Minimizar el tiempo de flujo es lo mismo que minimizar los tiempos de terminación, de retraso y los tiempos de espera de los trabajos. El tiempo de espera de los trabajos es inventario en proceso, de manera que minimizar el tiempo de flujo equivalente a minimizar el número de trabajos de proceso. EL tiempo de flujo ponderado corresponde al valor del inventario en proceso.

El número de trabajos retrasados, la tardanza máxima y la tardanza total son medidas de servicio al cliente. Se pueden encontrar más detalles sobre los costos y medidas sustitutas en Rinnooy Kan (1976).

La mayoría de los problemas de programación suponen certidumbre en los datos; los tiempos de preparación son independientes del orden de procesado; todos los trabajos están disponibles de inmediato ($r_i = 0$); no existe precedencia entre los trabajos, y una vez que comienza el procesamiento de un trabajo, no se puede interrumpir.

4.2.2. Algoritmos de programación

Un algoritmo es una receta para obtener una solución de un modelo. Un caso es un conjunto de datos específicos para el modelo. Los algoritmos exactos proporcionan una solución óptima para todos los casos del problema. Los algoritmos heurísticos dan soluciones que se espera sean optimas o cercanas a la optima en cualquier caso.
¿Por qué no siempre se usan los algoritmos exactos? Para muchos de los modelos de programación, los únicos algoritmos exactos que se conocen están basados en la enumeración, como el de la ramificación y acotamiento o la programación dinámica. En los casos prácticos, la *naturaleza combinatoria* del problema de programación los hace computacionalmente prohibitivos. Los algoritmos heurísticos se juzgan según su calidad y eficiencia. La calidad es la diferencia entre una solución heurística y la óptima, mientras que la eficiencia se refiere al esfuerzo realizado para obtener la solución. Ambas se pueden expresar teórica o empíricamente. Los heurísticos garantizan la calidad y la eficacia (es decir, cotas del peor en su desempeño). El trabajo actual sobre el análisis estocástico y programación con criterios múltiples sustenta una promesa para el futuro. Un buen punto de partida es el libro de (Pinedo, 1995) que combina resultados teóricos con el analisis de casos.

La cota del peor caso sobre la eficacia determina el numero de cálculos que deben realizar el algoritmo para cualquier problema práctico de una tamaño especifico (de ahí el nombre de “peor caso”). Un buen algoritmo acota el número de cálculos por una función polinomial del tamaño del problema. Esto no se ha hecho para algunos algoritmos heurísticos que deben ser evaluados en forma empírica. Para dar una justificación teórica de la calidad de un algoritmo heurístico, deben probarse matemáticamente que genera una solución dentro de cierto porcentaje de optimalidad, sin importar el problema particular que está resolviendo. Igual que con la eficacia, si la calidad de un algoritmo heurístico no tiene justificación teórica, deben juzgarse de manera empírica.

Las pruebas empíricas consisten en generar y resolver muchos problemas prácticos y analizar los resultados. Se pueden determinar el tiempo promedio para resolver el problema. Se puede encontrar la diferencia entre las soluciones heurística y óptima para los problemas pequeños. Para casos en los que no se puede obtener la solución optima, la solución heurística se compara con una cota sobre la solución óptima. Si existe una diferencia pequeña, la calidad del heurístico parece buena. La causa de una diferencia grande puede ser una cota débil o un heurístico pobre. Si el algoritmo heurístico tiene un buen desempeño en los casos de prueba, se hace la suposición de que se desempeña igual en otros casos. Esto no puede ser cierto para algún caso difiera de las pruebas. Al ponerlo en marcha debe compararse la solución heurística con la solución actual.

Todos los algoritmos heurísticos que se examinan han tenido un buen desempeño en las pruebas. Estas pruebas empíricas están publicadas y se dan las referencias en la literatura sobre la programación. Esto no es una garantía de soluciones buenas o ni siquiera que puedan usar. La implantación de un algoritmo heurístico debe ser siempre ir precedida de pruebas con ejemplos típicos de la aplicación.

4.2.3. Control de entrada y salidas.

El *control de las entradas y salidas* es una actividad clave que permitirá a los gerentes de operación identificar problemas como, capacidad insuficiente, capacidad en exceso y dificultades de producción entre grupos de estaciones de trabajo interconectados. Los gerentes de operaciones pueden determinar si la cantidad de trabajo que fluye hacia un centro de trabajo es la planeada y si la capacidad del centro de trabajo es la adecuada. Si esta fluyendo demasiado trabajo a un centro, en comparación con su capacidad, entonces lo que ocurrirá será un exceso en los inventarios en proceso (WIP) en los centros de trabajo que se preceden. Cuando los trabajos se acumulan, no solo el centro de trabajo se amontona y aglomera, sino que también los centros de trabajo que siguen a continuación pueden quedarse sin trabajo. Si, por otra parte, esta fluyendo poco trabajo hacia un centro de trabajo; en comparación con su capacidad, el centro de trabajo puede estar subutilizado y puede dar como resultado máquinas y trabajadores ociosos.

Una de las principales entradas para el sistema de control de piso, es el *recorrido y el lead time* para cada pieza. El recorrido especifica cada operación necesaria para hacer la parte y en qué centro de trabajo será llevada a cabo la operación. El lead time de producción está formado típicamente por 4 elementos:

1. *Tiempo de corrida:* Tiempo de corrida por operación, por pieza o por tamaño de lote.
2. *Tiempo de setup:* Tiempo para preparar el centro de trabajo inmediatamente del tamaño de lote.
3. *Tiempo para mover:* Tiempo empleado para mover un lote de un centro de trabajo a otro.
4. *Tiempo en la fila:* Tiempo que se espera para procesarse en un entro trabajo, este tiempo es el más crítico.

4.2.4. Cartas y gráficas de programación

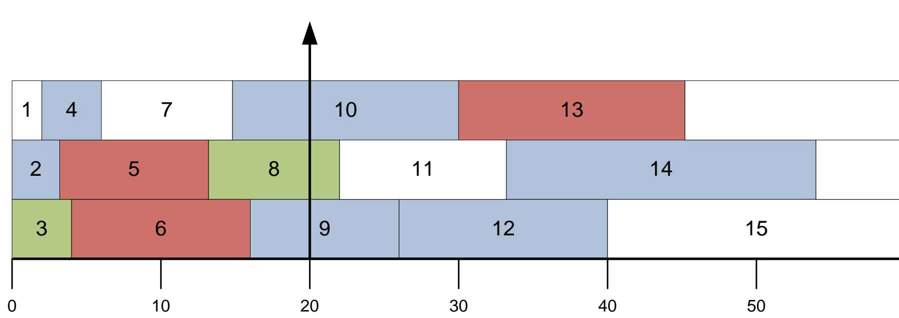
La coordinación de los programas de operaciones para los centros de trabajo ayuda en obtener un flujo ordenado de los trabajos entre estos mismos, para lo que son útiles los diagramas de Gantt.

Las gráficas de Gannt y las de barras muestran la *programación del trabajo y las cargas* en las instalaciones sobre un horizonte de tiempo y las cartas de programación (o avance) muestran las *actividades de trabajo secuenciales* necesarias para terminar un trabajo. Las cartas de carga muestran las horas de trabajo asignadas a un grupo de trabajadores o máquinas. Se pueden utilizar diagrama de Gantt para desplegar visualmente las cargas de trabajo de trabajo en cada centro de trabajo de un departamento. La figura 4 es un diagrama de Gantt utilizando para comparar el programa semanal de cinco centro de trabajo en un taller de modelado (planta de de producción de

productos experimentales). Los trabajos programados durante la semana se despliegan con nombres (A, B, C, etc. en colores, o números de código), y tiempos de inicio y de terminación representados por una barra sin relleno. Conforme avanza el trabajo en una tarea, una barra solida muestra como se está desempeñando el centro de trabajo en relación con el programa. Los periodos de revisión se muestran con una flecha vertical.

Los cambios en la maquinas, su mantenimiento y otras labores planeadas se indicaran mediante una s_i. Los espacios vacios indican tiempo ociosos planeado en el centro de trabajo; durante estos lapsos no se requieren operarios y, por lo tanto, estos pueden ser enviados a otros centros, o bien pueden programarse labores diferentes en estos periodos.

Figura 3. Carta de programación – Diagrama de Gantt



Fuente: Autor

Carga finita e infinita. Se emplean en ocasiones dos procedimientos para asignar trabajos a centro de trabajo; carga finita y carga infinita.

El procedimiento de carga infinita se utiliza cuando se asignan a centros de trabajo *sin tomar en consideración su capacidad*. Este procedimiento abandona la planeación de requerimientos de capacidad (CRP) y sus programas de carga. A menos que una empresa tenga capacidad excesiva de producción, en los centro de trabajo se presentaran *filas de espera inaceptables.*

El procedimiento de carga finita se utiliza cuando la capacidad de los centro de trabajo se asigna a una lista de labores. Utilizando un modelo de simulación por computadora o cualquier otro medio, y modificando los tiempos de inicio y terminación, la capacidad hora por hora de cada centro de trabajo se asigna a diversas tareas. El resultado final de este procedimiento es que en un centro de trabajo, durante cualquier hora, *no se programan más trabajos correspondientes a su capacidad*. Este procedimiento deberá estar integrado con el CRP.

Programación hacia adelante y hacia atrás. En la preparación de diagrama de Gantt, o de los programas de carga, hay dos formas para determinar la manera de asignar espacios de tiempo para trabajos dentro de los centro de trabajo: programación hacia adelante o programación hacia atrás.

En la programación hacia adelante, las tareas se asignan a los espacios de tiempo lo más temprano posible de los centros de trabajo. Consiste en programar todos los trabajos disponibles para que comiencen tan pronto como los requerimientos sean conocidos. Esta realización inmediata puede resultar en una terminación temprana del trabajo a costa de más trabajos en proceso y mayores costos de llevar más inventario del necesario. Este procedimiento supone que los clientes desean que sus trabajos se entreguen tan pronto como sea posible. Aunque es simple de utilizar, generalmente resultan inventarios en proceso excesivos, dado que las tareas tienden a esperar a que se les asignen a su siguiente centro de trabajo.

Para el proyecto de investigación a desarrollar se trabajará con el enfoque de la programación hacia adelante, evitando productos en proceso (WIP), mediante el control de entradas y salidas y los diagramas de Gantt y ofrecer a los gerentes de operaciones formas sistemáticas para ordenar el flujo de tareas en los centro de trabajos.

En la programación hacia atrás, el punto de inicio de la planeación es la fecha prometida de entrega para el cliente. Esta fecha se tomo como un hecho y se programa hacia atrás a través de los centro de trabajo utilizando los plazos de entrega para determinar cuando los trabajos deben pasar a través de cada una de las etapas de producción. Los trabajos se asignan a los espacios de tiempo de los centro de trabajo más tardíos posibles en que se pueda cumplir con la fecha de entrega prometida para eliminar tiempo de espera de MRP. Aunque este procedimiento requiere plazos de entrega precisos, tiene la tendencia a reducir los inventarios en proceso, porque los trabajos se terminan cuando se necesitan en el siguiente centro de trabajo de su plan de ruta. Los componentes son entregados " cuando se necesitan" más que " tan pronto como sea posible". Por esta razón, el procedimiento predominante utilizado por las empresas es la programación hacia atrás.

Las reglas de decisión de prioridad para “n” trabajos en un solo centro de trabajo son lineamientos simplificados (heurísticos) para determinar la secuencia en la cual están hechos los trabajos. Las reglas simplemente asignan trabajos con base en un sólo criterio, tal como primero que entra primero que sale (PEPS); fecha primera de Vencimiento (FPV); mínima holgura (o tiempo programado, menos tiempo de proceso) (MH); mayor tiempo de proceso (MATP), menor tiempo de Proceso (METP) y órdenes de clientes preferentes (OCP): la mayoría de las reglas son estáticas, pues no incorporan actualización.

Para cada una de estas reglas se deberán calcular ciertos parámetros que ayudarán a definir de acuerdo a las políticas de la empresa a seleccionar la regla de prioridad más ventajosa o equitativa entre nuestro cliente y nuestros intereses empresariales. (Norman Gaither, 2000)

Problemas de secuenciación de las órdenes. En los problemas de secuencia de las órdenes deseamos determinar la secuencia en la que produciremos un conjunto de órdenes que están a la espera en un centro de trabajo. Analizamos estos problemas al estudiar las diferentes reglas de secuencia, los criterios para la elaboración de

dichas *reglas de secuencia* y una *comparación de dichas reglas*, así como el control de los costos de cambios y la minimización de los costos de producción.

Reglas de secuenciación. Se pueden seguir *muchas* reglas de prioridad secuencial para establecer prioridades entre órdenes y trabajos que esperan en los centros de trabajo. Entre las más comunes están:

- *Primeras llegadas, primeros servicios* (FCFS, por sus siglas en ingles) el siguiente trabajo que se producirá, de entre los que están esperando, es el que llego primero.
- *Tiempo de procesamiento más breve* (SPT, por sus siglas en ingles) el siguiente trabajo que se producirá , de entre los que están esperando , es aquel cuyo tiempo de procesamiento es el más corto.
- *Fecha de entrega más cercana* (EDD por sus siglas en ingles) el siguiente trabajo que se producirá, de entre los que están esperando m es el que tiene la fecha de entrega más cercana (fecha prometida al cliente).
- *Aquel que tendrá menos holgura* (LS por sus siglas en ingles) el siguiente trabajo que se producirá, de entre los que están esperando, es el que tenga menor holgura (tiempo de fecha de entrega menos tiempo total de producción faltante).

  Holgura de la orden: Suma los tiempos de setup y de corrida para todas las operaciones que faltan, restando este del tiempo que falta (Desde ahora hasta la fecha de entrega de la parte), y el resto lo llama holgura. La regla es procesar el trabajo con la menor holgura.
  Holgura de la operación. Una variable de holgura de la orden es dividir la holgura entre el número de operaciones que faltan, procesando el trabajo con el valor más pequeño. El concepto de holgura por operación, es que sería más difícil completar los trabajos con más operaciones debido a que tienen que ser programadas a través de los centros de trabajo.
  La operación más corta. Esta regla ignora toda la información de fechas y de los trabajos que faltan. Dice simplemente; toma el siguiente trabajo, el cual será completado en el menor tiempo posible en el centro de trabajo. Esta regla maximiza el número de órdenes que pasaran por un centro de trabajo y minimiza el tiempo de espera en la fila.

- *Relación critica* (CR, por sus siglas en ingles) el siguiente trabajo que se producirá, de entre los trabajos que están esperando, es aquel que tenga la relación critica menor (tiempo de fecha de entrega dividido entre el tiempo total de producción faltante). Si la razón es 1.0 el trabajo será procesado a tiempo. Una razón menor que 1.0 implica un trabajo programado con retraso. Una razón es mayor que 1.0 indica un trabajo programado con anticipación.
- *Aquel que tenga el costo de cambio menor* (LCC, por sus siglas en ingles) en vista de que algunos trabajos siguen lógicamente a otros debido a la facilidad en los cambios; la secuencia de los trabajos en espera se determina al analizar el costo total de hacer todos los cambios de máquinas entre tareas.

También pudieran ser aplicables otras reglas, por ejemplo, el de clientes más valiosos, el trabajo más redituable y de línea de espera más corta en la siguiente operación. Estás reglas se mantienen a través del tiempo y no es conveniente ya que hay diversos sucesos que alteran el escenario de producción y por ende la regla dejara de estar vigente.

Criterios para la evaluación de las reglas de secuenciación. Al decidir cuál de las reglas de secuenciación se desempañará mejor para un conjunto de trabajo a la espera, comúnmente se utilizan varios criterios:

- Tiempo de flujo promedio: tiempo promedio que los trabajos se quedan en el taller.
- Cantidad promedio de trabajos en el sistema: cantidad promedio de trabajos en el taller.
- Retraso promedio del trabajo: tiempo promedio que la fecha de terminación del trabajo excede a su fecha de entrega prometida.
- Costo de cambios: Costo total de efectuar cambios en las máquinas para un conjunto de trabajos.

Una comparación de las reglas de secuenciación y el de los criterios de evaluación en un sistema de producción son para un solo centro de trabajo y no para diversos que cambian en forma continua a través del tiempo. Las reglas de secuencia del *tiempo más corto de procesamiento* y de la *relación crítica* con la política de primeras llegadas, son las que actualmente se utilizan en producción y definen en el control de piso (SFC).

Evaluación de las reglas de secuenciación. Por lo general, el departamento de programación analiza el desempeño de las diferentes reglas de secuenciación en conjunto de tareas o de trabajos representativos. Una vez que hayan seleccionado la regla que tiende a funcionar mejor en función de los criterios de mayor importancia, se incorpora como parte de su sistema de programación y de piso de taller, aunque se revise de vez en cuando, por lo tanto será inexacto ya que en la planta las variables cambian continuamente.

Control de los costos de cambio. Los costos de cambio son los costos de *cambiar un paso del proceso* de un sistema de producción de un trabajo a otro. Incluye los costos por elementos como el cambios de los ajustes en las máquinas, obtener las instrucción para el trabajo y el cambio en materiales y herramientas. Por lo general, *los trabajos deberán producirse en la secuencia que minimice el costo de estos cambios.* Por ejemplo, si dos trabajos utilizan prácticamente los mismos ajustes de máquina, las mismas herramientas y los mismos materiales, el cambiar del primer trabajo al segundo será muy rápido y poco costos. Una regla simple determina la secuencia de trabajo que mantendrá reducido el costo de los cambios entre un conjunto de trabajos en espera. El procedimiento selecciona el primero y segundo trabajos en la secuencia al averiguar cuál es el costo más bajo de cambio entre todos los cambios posibles. Del segundo trabajo en adelante, el siguiente trabajo siempre se determinara seleccionando el cambio de costo más bajo entre los restante. Esta regla pudiera no ser optima, pero en la práctica funciona bastante bien.

Otros *procedimientos matemáticamente más complejos* pueden lograr resultado óptimo. Se ha utilizado la programación lineal de enteros para minimizar los costos de cambio, dentro de un conjunto de restricción, que requieren que todos los trabajos se asignen en la secuencia una sola vez.

Secuencia de n trabajos a través de dos centros de trabajos. Problema clásico del tipo *flow shop.* $F2||C_{\max}$.Cuando las tareas deben secuenciarse a través de dos centros de trabajo, a menudo deseamos seleccionar una secuencia de trabajo que sea válida para ambos centros. Esta situación se puede analizar con efectividad utilizando la regla de Johnson. Los trabajos de los clientes deben pasar por maquinado (centro de trabajo 1) y acabado (centro de trabajo 2) en un misma secuencia de trabajos. La secuencia de trabajos que resulte tendrá para todos los trabajos el tiempo de producción total mínimo en estos dos centros.

Si existe empate para el tiempo más corto de procesamiento de diferentes centros de trabajo, no hay dificultad alguna en determinar la secuencia de trabajo. Sin embargo, si ocurre empate en el mismo centro de trabajo, será necesario evaluar dos secuencias de trabajo al comparar sus tiempos acumulados de producción. La secuencia de trabajo que tenga el menor tiempo acumulado ser la recomendada. Observe también que se puede utilizar la regla de Johnson con o sin el requisito de que los cambios de trabajo tienen que ocurrir simultáneamente en ambos centros de trabajo. (Pinedo, 1995)

Secuencia de n trabajos a través de m centros de trabajos. Problema clásico del tipo *flow shop.* $F_m||C_{\max}$. Las plantas de producción normalmente deben poner en secuencia muchas tareas a través de muchos centros de trabajo, *problema para el que no existen soluciones analíticas fáciles*. No obstante, los gerentes de operaciones y los programadores toman cotidianamente este tipo de *decisiones se secuenciamiento*. ¿Cómo se las ingenian para tomar estas decisiones complejas? Por lo común, una regla de secuencia como el tiempo más corto de procesamiento, la relación critica o la fecha más temprana de vencimiento se puede aplicar de manera uniforme. La *secuencia de trabajo* se modifica después, para aprovechar economías en los cambios. Si algunos trabajos están particularmente retrasados, las economías en los cambios deben sacrificarse para cumplir con los compromisos de fecha de entrega de los clientes. Dado que, de manera creciente, las decisiones de secuencia forman parte integral de los sistemas computarizados de programación, los procedimientos de secuencia deben quedar formalizados y programados en las computadoras. Este escenario es la base para desarrollar la metodología propuesta de secuenciación de trabajos. (Pinedo, 1995)

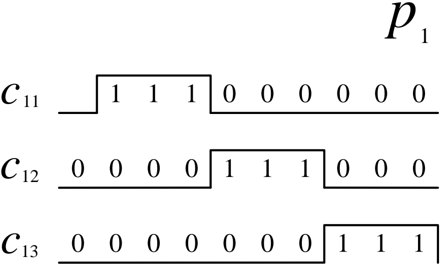
Aunque $F_2 \mid prmu \mid C_{\max}$ es fuertemente NP-duro es de interés estudiar casos especiales que tienen propiedades estructurales agradables. Una serie de casos especiales son importantes.

5. HIPOTESIS

"Al plantear una metodología para desarrollar un algoritmo heurístico, basado en fundamentos lógicos que exprese el problema de secuenciación de trabajos con otro tipo de expresiones algebraicas de tipo booleano, con el cual se pueda obtener una secuencia confiable al ser procesado como un problema de decisión y no de optimización".

El presente trabajo pretende determinar la secuencia de operaciones a través del uso de sistemas combinacionales y secuenciales basados en el álgebra de boole, con la finalidad de tomar decisiones de forma inmediata, de carácter afirmativo o negativo *"si o no"* para ser representados con valores **0** y **1** para poderlos relacionar con operaciones binarias suma (+) y producto (.) lógicos, que igualmente cumplan con los postulados conocidos por la matemática moderna y deducir funciones lógicas expresadas con ecuaciones boolenas, para efectuar operaciones para que se puedan deducir las salidas de *cronogramas*, que en forma analógica a los diagramas de Gantt , se representen con **1** y **0**, p.ej estando activo una máquina en un centro de trabajo expresado con el digito **1** y **0** en caso adverso.

Figura 4. Carta de programación – Código Binario



Fuente: Autor

Podemos disponer de la lógica proposicional, como un lenguaje en que se pueden describir de una forma alterna un problema real de toma de decisiones, de secuenciación, y en los que se puedan acotar las decisiones por resultados binarios; dentro de las posibles combinaciones de resultados se consideran las condiciones de indiferencia por lo regular ayudan a obtener una ecuación de transición mas simplificada, y por ende llegar a un resultado podría ser más eficiente, ya que tendrá menos operaciones lógicas que evaluar.

De esta manera se podrá llevar la representación de un flujo proceso físico alineado con la representación de flujo de información.

La ruta de fabricación en forma lógica, genera una relación de recurrencia para determinar el siguiente estado. Es una sucesión de estados. Por ejemplo **10110111**. Poder encontrar la fórmula para determinar la sucesión / secuencia.

Los supervisores y planeadores de producción se apoyan en diagrama de Gantt para detectar el *avance* de los centros de trabajo en comparación con su programa. Se desarrollará dentro de la metodología de investigación aplicada el planteamiento de una salida del sistema lógico proposicional para obtener este diagrama a través de funciones booleanas, para cada centro de trabajos c_i relacionados a cada producto p_i para luego ser combinados en uno solo o individual y efectuar el análisis de los trabajos en curso en forma continua a través del tiempo, y mitigar la generación de informes con información desactualizada como lo expresa la teoría de programación de operaciones.

Se desarrollará una metodología en la cual se puedan deducir un sistema lógico, que determine la disposición de cada uno de los centro de trabajo presentes en una planta de producción con operaciones discretas en el escenario *flow shop*.

La programación de operaciones a corto plazo deberá generar una reacción inmediata según los sucesos que ocurran directamente en la planta, para así determinar los posibles cambios y generar una secuencia de operaciones acorde a las necesidades a cubrir del plan maestro de producción.

El desarrollar una plataforma basada en sistemas lógicos combinacionales y secuenciales para toma de decisiones inmediatas se fundamenta en la ingeniería de producto PDM que hay que ejecutar cuando se está en firme una orden de trabajo. Expresar la ruta de fabricación y la lista de materiales como relaciones lógicas

Se está generando una alternativa de razonamiento matemático para apoyar las métricas de control de la ejecución de órdenes de trabajo en forma organizada y secuencial, cumpliendo en los requerimientos exigidos en el tiempo expresados en el plan de producción según la estrategia MTS.
MTO se vuelve más exigente ya que los requerimientos son expresados por el cliente sin considerar un tiempo para incorporaralo a la planeación de la producción. Esto amerita que la producción moderan sea cada vez más flexible y no pueda planear con el debido tiempo las diferentes ordenes de trabajo. Esta evolución en las tendencias de producción de hoy día, reflejan que la manufactura moderna debe estar alineada tanto en el flujo de procesamiento físico como con el flujo de información al mismo tiempo.

6. CONGRUENCIA METODOLOGICA

El primer paso para el perfeccionamiento del algoritmo de la elección de la jugada en el ajedrez es su comprensión. Es simultáneo, pensamos, ya que es útil recibir la imagen evidente del algoritmo con objeto del análisis consecuente.

La elección de la jugada es un proceso *intelectual complicado* y sin duda, *no conseguiremos representarlo* nunca sobre el papel y analizar este proceso por completo, *uno a uno*, con todos los detalles. Sin embargo podemos aplicar el método de la modelación, que en esencia consiste en lo siguiente:

- Se construye el modelo del proceso como el juego de las *correlaciones lógicas* y *matemáticas.*
- En el *modelo* se reflejan sólo los *atributos principales*, los más característicos y esenciales del fenómeno y sus enlaces, mientras que los factores secundarios y sus dependencias se desatienden. Se cumple el *análisis del modelo*, que tiene como objetivo el establecer, a cuales influencia y a que cambio es oportuno someter los atributos básicos, cuales significados dar para recibir los resultados necesarios.
- Las conclusiones recibidas en el modelo, son aplicadas al fenómeno real (*secuenciación*).

El método de la modelación sirve de medio eficaz del estudio de los procesos complicados desde hace mucho tiempo. A él, por ejemplo, se deben los éxitos de los matemáticos, pero es ahora aplicado en todas las regiones básicas de la actividad humana.

> *Se utiliza ampliamente el modelo del algoritmo de la elección de la jugada (secuencia de trabajo), desglosando las partes separadas de este algoritmo, no parando cada vez sobre la legitimidad de la modelación y no haciendo otras cláusulas especiales. El algoritmo de ajedrez de Alan Turing, conocido como Turochamp, fue diseñado a partir de la aplicación de ciertas reglas de oro con el objetivo de escoger la mejor movida posible, anticipando dos movimientos en la partida. Si tenemos en cuenta que los programas modernos de ajedrez pueden ir mucho más profundos que eso, y que Kasparov en una tarde normal anticipa diez jugadas, la descripción de primitivo parece adecuada para el algoritmo, pero no en un sentido que le quite valor, sino todo lo contrario. Es el primer programa de ajedrez jamás hecho. Si bien hay antecedentes de supuestas máquinas que jugaban al ajedrez (tal vez recuerden al falso "Turk"), el algoritmo de Turing es real, funciona, y a pesar de sus limitaciones, lo hace a la perfección.*

> *El juego queda apenas como una anécdota. Tampoco tiene sentido enfrentar a un algoritmo de 60 años con un ex campeón del mundo y esperar una sorpresa. Los dieciséis movimientos que llevaron a Kasparov a la victoria son irrelevantes en este caso. Lo que es realmente admirable aquí es que Turing hizo el primer algoritmo para un juego de ordenador incluso antes de que hubiese un ordenador capaz de ejecutarlo (trató de hacerlo en la Ferranti Mark 1, pero no fue posible). Turing se usó a sí mismo como ordenador para probar el algoritmo, recurriendo a papel y lápiz y tardando entre quince minutos y más media hora para procesar cada movimiento.*

El proceso de la partida de ajedrez comienza en la llamada posición inicial y se define reglamentariamente en la Posición de jaque mate, así mismo, los resultados de empates o tablas se verifican igualmente en determinadas posiciones. La posición constituye y contiene el problema al que debe enfrentarse el ajedrecista (*planificador de operaciones*). El dominio de todos los elementos técnicos del ajedrez está referido a ella, pero también lo están las *habilidades intelectuales* (*ser un estratega*) que debe poseer el atleta para actuar desempeñarse en las mismas.

Para definir un problema. Tendríamos que decir entonces que un problema es determinada situación en la cual existen *nexos, relaciones, cualidade*s de y entre los *objetos* que no son accesibles directa o indirectamente a la persona. Un problema es toda situación en la cual hay algo oculto para el sujeto que éste se esfuerza por hallar.

La posición de Ajedrez como problema. Una posición de Ajedrez se justifica así como problema, precisamente en los nexos, relaciones y cualidades de y entre los componentes de la misma: *el espacio*, *la fuerza* y *el tiempo*, lo que oculta la posición para los sujetos que intervienen en el proceso de la partida de ajedrez, también apoya el considerar esta como un problema.

Cómo se soluciona un problema?. El *pensamiento* se manifiesta por lo regular como un acto propositivo que se aplica predominantemente a la solución de problemas, puede ser regulado por el propio sujeto, distinguiéndose dos formas de regulación del pensamiento: tiene lugar a partir de la *contrastación* o *comparación* de las acciones realizadas y los productos obtenidos con *representaciones* (*modelos*) de lo que debe hacerse y lo que debe obtenerse en cada momento (*secuencia de operaciones a través del tiempo*).

Posiciones de Ajedrez como problemas. Los problemas que suponen las posiciones de ajedrez, se pueden caracterizar como tales precisamente por la indeterminación inicial de la acción concreta que hay que ejecutar en correspondencia con cierto criterio. Este criterio es la *decisión*. Para el proceso de la partida de ajedrez, la toma de decisiones habitualmente requiere la elección no univalente de cierto *conjunto de acciones* y la *transformación de la información*.

La decisión en Ajedrez. La decisión que se toma expresa la correspondencia de ésta con la finalidad de la acción que pretende alcanzar, el objetivo, por tanto la mejor decisión es aquella que debe ejecutar de la mejor manera el objetivo previamente seleccionado.

Concepto de estrategia general para el ajedrecista. En cualquier posición debo aumentar al máximo mis posibilidades o viceversa, disminuir al mínimo las posibilidades del adversario. Al operar el pensamiento en la solución de problemas se distinguen tres funciones: *función de análisis, función de ejecución* y *función de control* valorativo.

Proceso de solución: .....el proceso de solución de cualquier problema, al ser un acto esencial de pensamiento, puede considerarse, en una nueva dimensión, como *una sucesión - no lineal - de funciones* (*análisis, ejecución y control valorativo*), en la que cada una de esas funciones ocupa por cierto tiempo el lugar predominante, y luego cede el paso a otra función que *realiza los objetivos* (*u obtiene los productos*) programados o propuestos por el sujeto que resuelve el problema.

El análisis de un problema: ......el análisis del problema está dirigido a obtener la información útil para la solución, también es él quien permite destacar, filtrar o dejar a un lado aquella información contenida en el problema, o en la situación que en éste se plantea que no se revela - para el sujeto - como necesaria para el desarrollo de la solución.

Resultado más importante. El resultado más importante, al cual conduce la función de análisis, consiste en facilitar al sujeto el acceso gradual a la estructura del problema, es decir, aquel *conjunto de relaciones* y *nexos entre los elemento*s que constituyen la clave para la solución, en otros términos, la comprensión del problema puede denominarse aprehensión o determinación de la vía de solución.

Función ejecutiva: La función ejecutiva, al manifestarse como proceso de operación con símbolos, esquemas, fórmulas u otros elementos que realizan la transformación efectiva del problema, tiene que ver directamente, con las habilidades y hábitos para operar con dichos elementos; estos conocimientos y hábitos se asimilan de la experiencia individual (obtenida dentro de la enseñanza o fuera de ella).

Función de control valorativo. Mantiene la actividad de solución dentro del curso previsto (cuando se ha realizado de acuerdo con un plan preconcebido). Para lograr esto, el sujeto opera comparando los productos realmente obtenidos con los patrones o modelos de aquellos productos que normativamente, deberán ser obtenidos.

Un algoritmo en el Ajedrez. La caracterización de las posiciones de ajedrez como problemas y el estudio realizado acerca de cómo resolver estos en general, fundamentan la concepción de un algoritmo para ser aplicado por el ajedrecista (*planificador de operaciones a corto plazo- secuencias*) en la toma de decisiones dentro del proceso de la partida de ajedrez.

Este algoritmo refleja dos formas de regulación: Regulación a partir de patrones referidos a lo que debe hacerse u obtenerse en cada momento considerado para la toma de decisiones ante una posición de ajedrez. Regulación consciente por parte del competidor u atleta de sus propios procesos mentales aplicados a la toma de decisiones ante una posición de ajedrez.

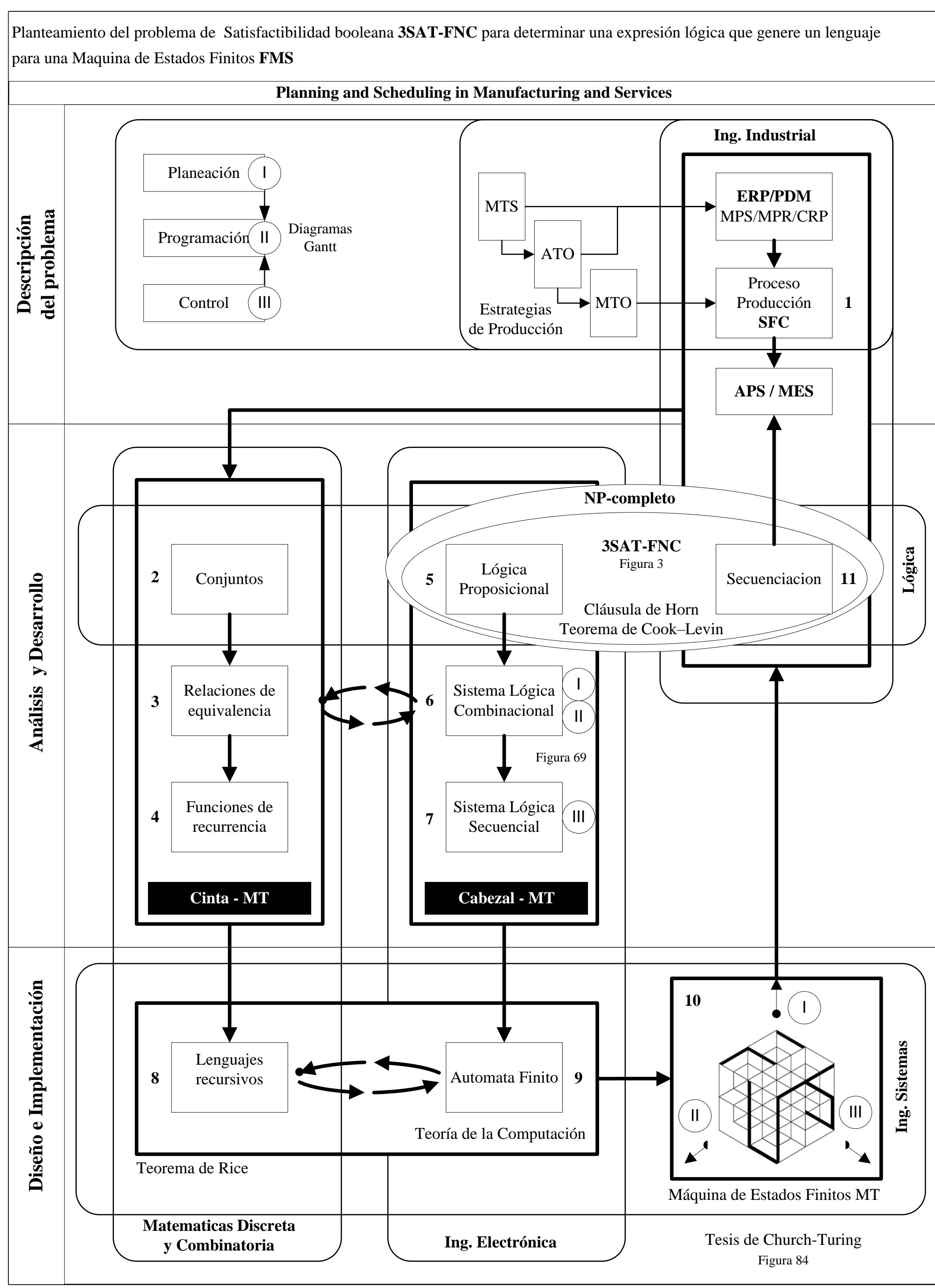
Las conclusiones recibidas sobre los modelos, se destinan a la aplicación directa práctica, ya que todos ellos en cierta medida se probaran. Los diagramas de bloques lógicos (*sistemas lógicos*) están numerados y están unidos por las líneas. Además de cada bloque del algoritmo es posible, dependiendo de la situación, moverse o de la izquierda a la derecha, o de arriba abajo.

Como vemos, la esencia del algoritmo presentado es escoger la jugada (*operaciones*) que detiene o que realiza una amenaza (esta puede ser la amenaza de tomar el peón, etc.). Probaremos formular las exigencias comunes a un algoritmo y al mismo tiempo examinaremos críticamente el diagrama de bloques.

En primer lugar, el algoritmo debe ser lógicamente no contradictorio. Cualquier situación debe acabarse por la elección de la jugada. Para empezar no se ha conseguido representar tal algoritmo – por ejemplo, sobre el esquema no es tomado en consideración que a veces es imposible parar la amenaza del adversario.

En la medida que se desarrolla el trabajo de investigación aplicada, se incorporan conceptos inherentes a la metodología descritos en la figura 7, para comprender los principios que fundamentan el razonamiento lógico para sustentar la alternativa de secuenciación de trabajos en producción.

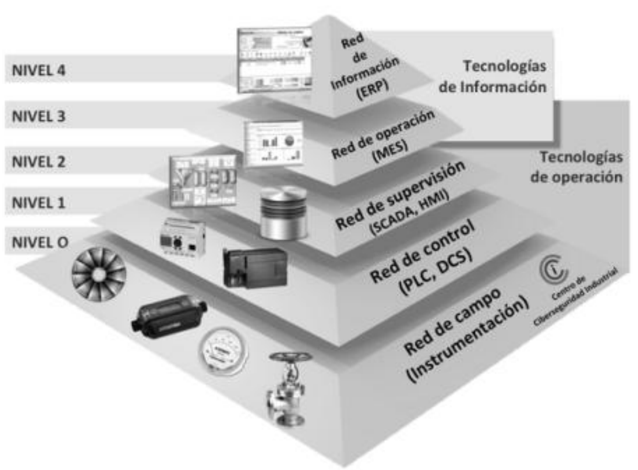
Figura 5. Mapa conceptual de la metodología planteada para desarrollar secuenciación de trabajos



Fuente: Autor

Al enmarcar la metodología propuesta, se dispone de sistemas de automatización que pueden ser divididos en distintos niveles conformando la pirámide CIM de automatización (*Computer Integrated Manufacturing* por sus siglas en ingles) descrito en la figura 6; donde se estaremos situados entre el nivel 3 red de operación (MES) donde se efectúa la planificación y el seguimiento del producto, gestión del stock y, a grandes rasgos, la ejecución de la producción. y el nivel 2 red de supervisión (SCADA), que corresponde a los sistemas de supervisión, control y adquisición de datos donde se enlazan, principalmente, celdas de producción (centro de trabajo) y computadoras con equipo de observación como puede serlo un sistema SCADA. Es por esto que en este nivel es importante contar con equipo que integre los diferentes protocolos de comunicación de los equipos en el nivel 1, es decir, los *lenguajes* que distintos PLCs hablan como PROFINET, ETHERNET IP, DEVICE NET, etc.

Figura 6. Pirámide de la automatización industrial



Asimismo, sobre estos niveles es donde las *tecnologías de información* y las *tecnologías de operación* se integran y en esta frontera se compartirá información de las variables de estado que provienen del sistema de control y visualización de procesos junto al sistema de gestión empresarial de operaciones MES/PS, donde se describen las características que contempla una aplicación para *Production Schedule* PS. Ver apartado recomendaciones.

7. DESCRIPCION DEL PROBLEMA SECUENCIACION

En la funcionalidad de *scheduler* del sistema operativo está orientada esta investigación; ya que este mismo es quien gobierna las decisiones a ser tomadas para el ordenador (en la planta de producción), y donde se encuentran los algoritmos que tramitan el procesamiento de los programas a ser ejecutados y por lo tanto la metodología a desarrollar se dirige la determinar el problema de decisión (en ciencias de la computación y matemáticas, en alemán : *entscheidungsproblem*) con lógica simbólica y determinar un algoritmo general que decidiera si una fórmula del lógica de primer orden es un teorema. Efectuando está analogía, dentro de la metodológica se representará el problema de secuenciación de operaciones de producción con los principios que rigen a un ordenador pero basado en la *programación declarativa* para algoritmos decidibles analizando los recursos que son necesarios para ser ejecutados físicamente (recursos de tiempo, numero de centros de trabajo, tipo de operaciones, materiales, etc.)…

El planificador (en inglés scheduler) es quien efectúa la programación de operaciones en la gestión planeación operativa de producción y desde el punto de vista de la teoría de computación, es una funcionalidad muy importante de los sistemas operativos *multitarea* y *multiproceso*, y es esencial en los sistemas operativos de *tiempo real*. Su función consiste en repartir el tiempo disponible de un microprocesador entre todos los procesos que están disponibles para su ejecución. Se denominará microprocesador a cada uno de los *centros de trabajo* presentes en un ambiente de producción.

En todo sistema operativo se gestionan los programas mediante el concepto de *proceso*. Se denomina este mismo, como los procesos de producción requeridos para cada uno de los productos a ser transformados. En un instante dado, en el ordenador (como la planta de producción) pueden existir diversos procesos listos para ser ejecutados. Sin embargo, solamente uno de ellos puede ser ejecutado (en el microprocesador). De ahí la necesidad de que una parte del sistema operativo gestione, de una manera equitativa, qué proceso debe ejecutarse en cada momento para hacer un uso eficiente del procesador. La gestión de procesos podría ser similar al trabajo en la administración de la producción industrial. Se puede tener una lista de trabajos a realizar y a estos fijarles prioridades alta, media, baja. Se debe comenzar haciendo los trabajos de prioridad alta primero y cuando se terminen seguir con las de prioridad media y después las de baja. Una vez realizada el trabajo se suprime. Esto puede traer un problema que los trabajos de baja prioridad pueden que nunca lleguen a ejecutarse y permanezcan en la lista para siempre. Para solucionar esto, se puede asignar alta prioridad a las tareas más antiguas.

Al suponer que un ordenador contiene un único microprocesador, que se representa como un centro de trabajo. Dicho microprocesador solamente puede ejecutar un programa en cada instante de tiempo. Además, cuando un programa está ejecutándose, nunca dejará de hacerlo por sí mismo. De manera que, en principio, cualquier

programa monopoliza el microprocesador impidiendo que otros programas se ejecuten. Es el caso de la programación de una sola máquina en la teoría de *scheduling* para programación de trabajos que deben ser procesados uno a la vez hasta terminar. (Negrón, 2009)

Por ello, la primera misión de un planificador es expulsar el programa en ejecución cuando decida que es pertinente. Esto se consigue de dos maneras, siempre con ayuda del propio hardware:

- Cuando expira un temporizador, que se activa a intervalos regulares de tiempo. En intervalos muy cortos de tiempo, generalmente cada 250 milisegundos. (para nuestro caso se representa como un sistema lógico secuencial síncrono en segundos para el recurso tiempo).
- Cuando el programa solicita una operación de entrada/salida. Dado que el programa no puede continuar hasta que termine dicha operación, es un buen momento para ejecutar otro programa (u orden de trabajo en firme correspondiente al MPS: *master production schedule* por sus siglas en ingles).

En ambos casos, el control del microprocesador pasa a manos del planificador gracias a que el hardware genera una interrupción. En este proceso de expulsión, se guarda el estado de ejecución del programa (programa y su estado se denomina contexto). Este hardware, se contextualiza como el equipo (máquina de trabajo) que se encuentra ubicado en cada centro de trabajo establecido.

A continuación, el planificador decide cuál será el siguiente proceso en ejecutarse. Naturalmente, solamente se escogen procesos que estén listos para hacerlo. Si un proceso sigue esperando por una operación de entrada/salida no será candidato a ejecutarse hasta que finalice tal operación.

La *selección* del proceso sigue alguna política de planificación preestablecida. Una vez seleccionado un proceso, se procede a ejecutarlo. Para ello, el planificador restaura su estado de ejecución (previamente salvado) y abandona el uso del microprocesador cediéndoselo a dicho proceso.

Se dice que el planificador procede a ejecutarlo a modo ilustrativo, entrando un poco más en detalle el planificador envía su lista de procesos al *'dispatcher'* (despachador) para que este adecue al proceso y su PCB para la inmediata ejecución Todo esto apenas dura unos pocos milisegundos, y se consideraría tomar *decisiones a corto plazo*.

Gracias a que el tiempo del microprocesador se reparte entre todos los procesos a intervalos muy cortos, el ordenador ofrece la sensación de que todos los procesos están ejecutándose a la vez (*ejecución concurrente*). Cuando un ordenador tiene varios microprocesadores, este esquema se repite para cada microprocesador.

En producción, varias máquinas que puede realizar el mismo tipo de procesamiento se llaman máquinas paralelas. Un trabajo se puede procesar en cualquiera de las máquinas, y una vez procesado por cualquiera de ellas, queda terminado. A menos que se diga lo contrario, se supone que todas la máquinas paralelas son idénticas. El tiempo para procesar un trabajo en una de varias máquinas idéntica es independiente de que máquina lo haga.

Una producción continua contiene máquinas diferentes. Cada trabajo (programa) debe procesarse en cada máquina exactamente una vez. Más aun, todos los trabajos siguen la misma ruta; esto es, deben visitar las máquinas en el mismo orden. Sin pérdida de generalidad, se puede numerar las máquinas de manera que la 1 sea la primera, la 2 la segunda, y así sucesivamente. Un trabajo no puede comenzar su procesado en la segunda máquina hasta no terminar el de la primera. Las líneas de ensamble y las células son ejemplos típicos de producción continua. Este caso resulta interesante, ya que un ordenador en la actualidad se le puede asignar a tareas específicas desde el sistema operativo a un procesador indicado pero no en secuencia.

Un sistema operativo en tiempo real se caracteriza por garantizar que todo programa se ejecutará en un límite máximo de tiempo. El planificador debe comportarse de manera que esto sea cierto para cualquier proceso.
En estos casos, la finalidad del planificador es balancear o equilibrar la carga del procesador, impidiendo que un proceso monopolice el procesador o que sea privado de los recursos de la máquina. En entornos de tiempo real, como los dispositivos para el control automático en la industria (por ejemplo, robots), el planificador también impide que los procesos se paren o interrumpan a otros que esperan que se realicen ciertas acciones. Su labor resulta imprescindible para mantener el sistema estable y funcionando.
El comportamiento del modelo será en tiempo real, y por lo tanto los sistemas lógicos interactuarán activamente con el entorno de producción junto con la dinámica conocida en relación con sus entradas, salidas y restricciones temporales, para darle un correcto funcionamiento de acuerdo con los conceptos de predictibilidad, estabilidad, controlabilidad y alcanzabilidad concebida en la programación de operaciones.
La principal característica que distingue el modelo STR en la secuenciación de trabajos es el tiempo de interacción. La palabra tiempo significa que el correcto funcionamiento del sistema depende no sólo del resultado lógico que devolverá la máquina de estados finitos deterministica, también depende del tiempo en que se produce ese resultado. La palabra real quiere decir que la reacción de un sistema a eventos externos debe ocurrir durante su evolución. El *determinismo* es una característica que se refiere al tiempo que tarda el modelo antes de responder a una interrupción. Este dato es importante saberlo porque casi todas las peticiones de interrupción se generan por eventos externos al sistema (por una petición desde algún centro de trabajo, un cambio en la demanda, etc.), así que es importante determinar el tiempo que tardara el modelo en aceptar esta petición de servicio. Como una consecuencia, el tiempo del sistema lógico (tiempo interno) debe ser medido usando la misma escala con que se mide el tiempo del ambiente controlado de programación (tiempo externo), cumpliendo con tres condiciones básicas: interactúa con el mundo real (proceso físico de transformación), emite respuestas correctas (variables de estado) y cumpliendo con las restricciones temporales (reglas).
La *responsividad* se enfoca en el tiempo que tarda una tarea en ejecutarse una vez que la interrupción ha sido atendida. Los aspectos a los que se enfoca en el modelo son: la cantidad de tiempo que se lleva el iniciar la ejecución de una interrupción (o un trabajo); la cantidad de tiempo que se necesita para realizar un trabajo que pidió la interrupción y los efectos de interrupciones anidadas que afectan la secuenciación de trabajos.

Una vez que el resultado del cálculo de determinismo y responsividad es obtenido, se convierte en una característica del sistema y un requerimiento para las aplicaciones que correrán en él, (si diseñamos una algoritmo eficiente en un sistema de producción en el cual el 95 % de los trabajos deben terminar en cierto período entonces es recomendable asegurarse que las trabajos ejecutadas en la aplicación no caigan en el 5 % de bajo desempeño).

La *confiabilidad* en un sistema de secuenciación de producción en tiempo real es otra característica clave. El sistema lógico no debe solamente estar libre de fallas pero más aún, la calidad de la información no debe degradarse más allá de un límite determinado. El sistema debe de seguir en funcionamiento a pesar de catástrofes, o fallas mecánicas. Usualmente una degradación en el servicio en un sistema de tiempo real lleva consecuencias catastróficas. Que el sistema sea estable, es decir, que si para el sistema es imposible cumplir con todas los trabajos sin exceder sus restricciones de tiempo, entonces la planeación de la producción cumplirá con las trabajos más críticos y de más alta prioridad. Cuando ocurra una falla, el sistema preserve la mayor parte de los datos y capacidades del sistema en la mayor medida posible.

7.1. PASO 1: Proceso de producción

El *flow shop* es un tipo de proceso de fabricación que se caracteriza básicamente en que sus tareas (series de trabajo) para llevarse a cabo necesariamente pasan a través de todos sus procesos (máquinas) en el mismo orden, es decir que sus productos tienen una relación de procesos y secuencias idénticas; las máquinas están dispuestas de manera que el flujo de todos los productos que se procesan en ellas es unidireccional. Existen m máquinas y puede haber que tengan menor operaciones que m. Este problema es complicado debido a de carácter combinatoria. Si los n trabajos se procesan en las m máquinas existen $(n!)^m$ alternativas para la programación. Si hay que planificar 5 trabajos en 3 máquinas se disponen de $1.728.000$ alternativas diferentes. Hasta el momento solo está resuelto de forma óptima el caso estático del *flow shop* de dos máquinas.

El *flow shop* conocido también en el ámbito de producción como enfoque estratégico repetitivo se aplica de manera conveniente en organizaciones que manejan una media flexibilidad de referencias y un nivel medio de volúmenes de fabricación, pero que a su vez se basa en el ensamble de módulos (elementos a ensamblar que son factor común en diversas referencias) los cuales fluyen en el sistema basados en un proceso continuo.
Si para sistemas *Job Shop*, en los que las secuencias de producción son particulares y no repetitivas los enfoques vanguardistas de distribución sugieren un enfoque de manufactura esbelta, para el *flow shop* es sin duda la configuración perfecta, esta consiste en que las máquinas e instalaciones deben disponerse en función al flujo de

producto, lo cual mejorará el tiempo, independiente de que exista más de una secuencia de producción, tal como observaremos a continuación.

En los sistemas de producción *flow shop* se observa un proceso simple, lineal, completamente visible, enfocado al flujo de producto y que además permite un mejor mantenimiento.

Enrutamiento (Orden de producción). Es común que se efectúe una producción orientada al pedido en la que si se llegaran a presentar vacíos de programación, estos se compensan con órdenes de procesos continuos orientados a los módulos con mayor rotación (para lo que se requiere un análisis previo), aumentando la utilización de la capacidad instalada (por ende disminuyendo los costos totales unitarios, aunque estos es relativo con base en la meta de la empresa) y alimentando un stock de subensambles o producto en proceso, que reducirá el ciclo logístico de las órdenes futuras.

Vale la pena considerar que en el caso en el que se presenten vacíos de programación, no necesariamente la mejor alternativa sea compensar los puestos de trabajo con órdenes de previsión; para ello es preciso tener un análisis detallado de la rotación de las unidades, y en caso de que dicho índice no sea representativo, no se aconseja utilizar a tope la capacidad instalada solo por el hecho de mejorar los rendimientos locales. Tenga en cuenta que un sistema con múltiples óptimos locales no necesariamente es un sistema óptimo en su conjunto; es decir, que en producción es vital equilibrar la demanda con el flujo, y no con la capacidad, ya que operando al máximo de la capacidad y sin órdenes de pedido (no de previsión) en pie, se incurre en sobreproducción con sus consecuentes efectos: excesos de inventario de producto en proceso, excesos de inventario de producto terminado, y costos de oportunidad o lucro cesante, es decir capital de trabajo, mientras no se está facturando a la misma tasa.

En los casos en los que no se trabajo sobre pedido, el proceso puede desarrollarse de manera continua (programación con poco estrés) para todas las operaciones relacionadas con la fabricación de módulos y estableciendo corridas lote por lote para los procesos de ensamble y personalización, las cuales deben ser lo más cercano al cliente posible, eso sí, esta es una alternativa que depende de la rotación de inventario.

La programación de un sistema *flow shop* es compleja, si bien es cierto que es mucho más simple que la programación de un sistema *job shop*, esto derivado de la alta cantidad de posibles alternativas de secuencia de entrada, tiempos de procesamiento, tamaño de lote, reutilización de estaciones de trabajo, entre otras. Para programar un sistema *flow shop* se recomienda utilizar las reglas de asignación, establecer y hacer seguimiento a los indicadores de producción más relevantes para la compañía, y evaluarlos con las alternativas de prioridad.

Qué empresas utilizan *flow shop*?

Algunos ejemplos de organizaciones con enfoques estratégicos repetitivos o *flow shop* son los lugares dedicados a la preparación de comidas rápidas, ensambladoras, siderúrgicas, neumáticos, pastas alimenticias u organizaciones que se enfocan en la producción de un solo producto.

Ventajas y Desventajas del *flow shop*

La principal ventaja del *flow shop* es que se basa en las ventajas de sus enfoques extremos, es decir en los beneficios respecto a la personalización de productos de un enfoque al proceso y en las ventajas económicas de la producción a media/alta escala que se perciben al llevar un proceso continuo de programación y fabricación de módulos cuya probabilidad de rotación hacia el ensamblado final es mayor que en un enfoque en el proceso.
Las ventajas más significativas respecto al *Job Shop* son sin duda alguna las ofrecidas en el proceso de programación, dado que existe la posibilidad de realizarse sobre los módulos que presentan una cantidad lógicamente inferior a la cantidad de referencias. Otra de las ventajas que presenta el *flow shop* es su alta capacidad de optimizarse con el balanceo de líneas.
Las desventajas son muy pocas, y consiste básicamente en que no todos los productos son susceptibles de una producción modular.

Niveles de planificación. Los niveles de planificación están basados en la frecuencia con la que se realiza cada uno. En analogía a los sistemas operativos de propósito general, existen tres tipos de planificadores que se asumen en la planificación de la producción. El planificador a corto plazo es el que se ha descrito aquí, siendo también el más importante. En inglés, se denomina *dispatcher* o *short term scheduler*, también existe un planificador a mediano plazo (en inglés, *mid term scheduler*) relacionado con aquellos procesos que no se encuentran en memoria principal (véase memoria virtual). Su misión es mover procesos entre memoria principal y disco (lo que se conoce como *swapping*) y por último existe el planificador a largo plazo (en inglés *long term scheduler*) es el encargado de ingresar nuevos procesos al sistema y de finalizarlos. Como analogía en la planeación de producción están presentes la planificación estratégica, táctica y operacional que se rigen en la misma escala de tiempo. En esta última se centra el problema de investigación para aumentar la confiabilidad en el programa.

Políticas de planificación. A continuación se enumeran diversas políticas de planificación. Lo habitual es utilizar políticas mixtas. Generalmente, el planificador a corto plazo utiliza *round-robin*, mientras que el planificador a largo plazo utiliza varias colas FIFO (*First In First Out*). Cada una de estas colas corresponde a una prioridad diferente.

- Planificación *Round-robin*
- *Round-robin* con pesos.
- Prioridades monótonas en frecuencia RMS *Rate-monotonic scheduling*
- Menor tiempo de respuesta primero EDF *Earliest deadline first scheduling*
- FIFO - También conocido como FCFS *First Come, First Served*.
- LIFO.
- SJF - *Shortest Job First*.
- CFS - *Completely Fair Scheduler* (ó Planificador Completamente Justo)

- SPT - *Shortest Process Time* (clásico en la programación de operaciones industriales).
- SRT - *Shortest Remaining Time*

Planificación mediante colas multinivel. Existen dos tipos de algoritmos de calendarización (políticas de planificación), *expropiativos* y *no expropiativos*. Los expropiativos asignan un tiempo de ejecución a cada proceso después del cual se calendariza otro proceso, hasta que cada proceso acabe su trabajo. También pueden expulsar a un proceso en ejecución si llega otro de mayor prioridad que necesita ejecutarse. Los no expropiativos permiten que se ejecute el proceso hasta que acabe su trabajo. Es decir, una vez les llega el turno de ejecutarse, no dejarán libre la **CPU** hasta que terminen o se bloqueen. La existencia de estos algoritmos serán codificados bajo *programacion lógica declarativa* para el problema de secuenciación (*scheduling)* y al preguntarse si lo sugerido con respecto a la teoría de la computación, el problema decisión planteado será una formula lógica es una *clausula de Horn*, con un conjunto de literales de longitud finita que tienen asociadas proposiciones resultantes también de longitud finita. El problema de decisión es un problema en donde las respuestas posibles son *sí* o *no*. Como un típico problema de decisión se pregunta: ¿hay inventario disponible?, ¿está disponible el centro de trabajo?, ¿el personal está asignado? Una instancia de este problema sería: ¿Es posible este proceso?

El problema de decisión también se podrá formalizar como el problema de decidir si cierto sistema lógico formal pertenece a un conjunto de elementos más simples que representan proposiciones, y cuyas constantes lógicas llamadas conectivas lógicas, representan operaciones sobre proposiciones, capaces de formar otras proposiciones de mayor complejidad donde se pueda analizar la inferencia lógica partir de estas mismas. Tanto la lista de materiales como la ruta de fabricación para cada unos de los artículos a transformar (tomados desde el PDM *product data management*: administración de datos de producto) serán codificados en el lenguaje lógico representándolos como sistemas lógicos tanto combinacionales y secuenciales, basados en el algebra booleana, que en informática y matemática son estructuras algebraicas que esquematizan operaciones lógicas $\wedge$, $\vee$, $\neg$ y $\rightarrow$ (*and, or, not,* y *if..then*) así como el conjunto de operaciones unión, intersección y complemento en el algebra de conjuntos.

De los sistemas lógicos formados, se genera el alfabeto de un sistema formal como el conjunto de símbolos que pertenecen al lenguaje del sistema (autómata finito programable *Máquina de Turing*). El conjunto contiene exactamente las frases para las cuales la respuesta a la pregunta es positiva. La pregunta anterior sobre los procesos se puede ver también como el lenguaje de todas las frases en el alfabeto $\{p_0, p_1, p_2, \ldots, p_n,\}$ tales que el proceso correspondiente es posible.

Si existe un algoritmo que pueda decidir para cada posible frase de entrada si esa frase pertenece al lenguaje, entonces se dice que el problema es decidible, de otra forma se dice que es un problema indecidible. Cuando

existe un algoritmo que puede responder positivamente cuando la frase está en el lenguaje, pero que corre indefinidamente cuando la frase no pertenece al lenguaje se dice que el problema es *parcialmente decidible*. En la teoría de la computabilidad, se estudia qué lenguajes son decidibles con diferentes tipos de máquinas. En la teoría de la complejidad computacional se estudia cuántos recursos necesita un algoritmo decidible para ejecutar (recursos de tiempo, espacio, número de procesadores, tipo de máquina, etc.).

La lógica proposicional y la computación. Debido a que los ordenadores trabajan con información binaria, la herramienta matemática adecuada para el análisis y diseño de su funcionamiento es el Álgebra de Boole que fue desarrollada inicialmente para el estudio de la lógica. Ha sido a partir de 1938, fecha en que Claude Shannon publicó un libro llamado *Análisis simbólico de circuitos con relés*, estableciendo los primeros conceptos de la actual teoría de la conmutación, cuando se ha producido un aumento considerable en el número de trabajos de aplicación del Álgebra de Boole a los computadores digitales. Hoy en día, está herramienta resulta fundamental para el desarrollo de los ordenadores ya que, con su ayuda, el *análisis* y *síntesis* de combinaciones complejas circuitos lógicos puede realizarse con rapidez.

El análisis dentro de la metodología comprende que la representación de una planta de producción sea descrita como un sistema lógico proposicional al cual sea análogo a los componentes de un ordenador y este gobernado con operaciones lógicas que determinen un razonamiento con el cual se tomen decisiones a corto plazo.

El hardware se representa la parte física del modelo, y está fundamentado en los sistemas lógicos tanto combinacionales como secuenciales. El conjunto de componentes está asociados a cada uno de los recursos tangibles como son los equipos, los materiales y la mano de obra asociados a cada uno de los centros de trabajo, que se refiere al procesamiento (procesador) de cada uno de los artículos que forman parte del producto final. El procesador (*centro de trabajo*) es el encargado de ejecutar los *programas* (de producción), desde el *scheduler* del *sistema operativo* (*clausulas de Horn* con formulas bien formadas); sólo ejecuta instrucciones programadas en un lenguaje de bajo nivel, realizando operaciones aritméticas y lógicas simples, tales como sumar, restar, multiplicar, dividir, las lógicas binarias y accesos a memoria.

El software principal (*scheduler*) o programa que hace parte del sistema operativo que gestiona los recursos de hardware y provee servicios a los programas de aplicación de software, ejecutándose en modo privilegiado respecto de los restantes.

7.2. PASO 2: Conjuntos

7.2.1. El lenguaje de las matemáticas: Axiomas y primeras definiciones

Los conjuntos se han convertido en los objetos matemáticos más fundamentales, sobre los que se construye el resto de las matemáticas. Y la teoría de conjuntos, a su vez, se edifica sólidamente sobre axiomas mediante las leyes de la lógica de las que se ha hecho un esbozo en el apartado precedente. Razón por la cual se parte de los conjuntos para efectuar la representación de una instalación de producción constituida por centros de trabajo, maquinas, equipo y materiales.

La colección de axiomas de la teoría de conjuntos es un tema complicado y nunca cerrado a la discusión. Actualmente se considera como esquema básico los nueve axiomas de Zermelo y Fraenkel. Sin embargo, para nuestro propósito, mucho más modesto, de introducir las operaciones entre conjuntos, basta con cinco axiomas y a ellos nos limitaremos.

Se introducen las operaciones del *complemento, unión e intersección*. Las definiciones son una traducción, paso por paso, de los conectores entre proposiciones lógicas: *negación, disyunción y conjunción* respectivamente. Por ello no debe extrañar que el algebra de conjuntos sea idéntica al algebra de proposiciones (es la estructura algebraica conocida como algebra de Boole) (Badesa C, 1998). Finalmente se define una operación más entre conjuntos, el producto cartesiano, que no tiene un análogo en el apartado anterior. Los conjuntos estarán constituidos por los recursos necesarios para expresar la estructura de productos para cada uno de los artículos a producir.

Cualquier intento de definir el concepto de conjunto está condenado a enumerar sinónimos como son colección, familia, agregado, agrupación, etc. Lo importante de la idea que asociamos al término conjunto es que contiene elementos y debemos poder expresar si un elemento pertenece o no a un conjunto: la pertenencia es el concepto sobre el que se construye la teoría de conjuntos, es el concepto primitivo (es decir, que no se define). El símbolo que indica que x pertenece al conjunto A es $x \in A$, y decimos que x es elemento de A, mientras que $x \notin A$ es su negación. Cada una de las máquinas se constituye como un elemento de conjunto equipos; cada operario será un elemento que pertenece al conjunto personal y cada artículo que sea parte de la materia prima, producto en proceso serán elementos del conjunto materiales.

La teoría de conjuntos tiene esencialmente dos actividades: *comparar conjuntos* y *construir nuevos conjuntos* a partir de unos dados (para, después, comparar los nuevos con los originales). En cualquiera de los dos casos, la teoría usa conjuntos ya existentes, no puede crear un conjunto de la nada. Por ello el primer axioma que enunciamos es el que dice que, al menos, existe un conjunto de modo que toda la teoría no se quede vacía.

Para cada artículo final habrá un conjunto constituido por los subconjuntos: personal, equipos y materiales propios de este mismo; el cual podremos comparar con otros conjuntos definidos para diferentes artículos y visualizar que recursos son comunes entre ellos.

Axioma (De existencia). Existe un conjunto. La primera tarea es, por tanto, comparar conjuntos. La comparación más básica es saber si dos conjuntos son iguales o no, que es lo que resuelve el segundo axioma.

Axioma (De igualdad). Dos conjuntos son iguales si, y solo si, contienen los mismos elementos. De manera simbólica lo escribimos $A = B \Leftrightarrow \forall x(x \in A \leftrightarrow x \in B)$

Lo que dice este axioma es que un conjunto queda completamente caracterizado por los elementos que contiene, y no importa si los elementos están ordenamos o están desordenados; solo importa cuáles son los elementos. También se llama axioma de extensión porque permite definir un conjunto describiendo todos y cada uno de sus elementos, es decir, describiendo su extensión. Para definir un conjunto por extensión se escriben sus elementos entre llaves, por ejemplo $A = \{1, 2, 3, 4, 5\}$.

En vista del axioma de igualdad, es claro que $\{1, 2, 3\} = \{2, 3, 1\}$ ya que los dos conjuntos tienen exactamente los mismos elementos. Más aun, también se cumple $\{a, a\} = \{a\}$ por la misma razón.

Ahora podemos definir otra forma de comparar conjuntos más poderosa que la mera igualdad: *la inclusión*, en la que definimos cuando un conjunto está contenido en otro y lo llamamos subconjunto.

Un conjunto B es subconjunto de otro conjunto A, y se denota $B \subset A$, si todo elemento de B es elemento de A. Es decir, $B \subset A \Leftrightarrow \forall x(x \in B \rightarrow x \in A)$

La inclusión es una comparación más poderosa que la igualdad, en el siguiente resultado se indica como verificar la igualdad de conjuntos usando la inclusión: *la igualdad es una doble inclusión*. Ver anexo Teorema 1.

Los siguientes tres axiomas son para construir nuevos conjuntos a partir de conjuntos ya conocidos. El primero de ellos, el axioma de especificación utiliza un enunciado abierto $p(x)$ y construye el conjunto formado por los elementos que hacen cierto el enunciado.

Axioma (De especificación). Dado un conjunto A y un enunciado abierto $p(x)$ existe el conjunto de los elementos de A que hacen cierto el enunciado. Es decir, existe el conjunto B que cumple $\forall x(x \in B \leftrightarrow x \in A \wedge p(x))$

El conjunto recién definido se representa mediante el símbolo $B = \{x \in A \mid p(x)\}$

Es claro que el nuevo conjunto B es subconjunto de A. En la investigación se explica la necesidad de exigir que los elementos del nuevo conjunto B se escojan únicamente entre los elementos de algún conjunto A ya conocido.

Al utilizar este axioma inmediatamente para definir un conjunto con nombre propio, el conjunto vacío, que es un conjunto sin elementos. Una definición por especificación es elegante y útil, más que una por extensión.
Sea A un conjunto cualquiera. Definimos el conjunto vacío como $\varnothing = \{x \in A | \ x \neq x\}$

Obsérvese que para definir el vacío así hace falta la existencia de, al menos, un conjunto. Pero el axioma de existencia asegura que si lo tenemos. Por su definición resulta inmediato que el vacío es subconjunto de cualquier otro. Ver anexo Teorema 2.
En el axioma de especificación se construye, a partir de uno dado, un conjunto más pequeño. En los siguientes axiomas la construcción es al revés: se construyen conjuntos más grandes. En ambos aparecen conjuntos cuyos elementos también son conjuntos.

Axioma (De la unión). Dada una familia de conjuntos F, existe un conjunto que contiene los elementos de los elementos de F. Al llamar E a dicho conjunto, entonces podemos definir el conjunto llamado unión de F utilizando el axioma de especificación como sigue.
Dada una familia de conjuntos F, la unión de la familia F es el conjunto formado exactamente por los elementos de los conjuntos que están en F:

$$\bigcup F = \{x \in E \mid \exists\, A \in F, x \in A\}$$

Sean los conjuntos $X = \{1, 2, 3\}$ e $Y = \{3, 4, 5\}$ y con ellos la familia $F = \{X, Y\}$. Entonces la unión de F es el conjunto

$$\bigcup F = \{1, 2, 3, 4, 5\}$$

Por último, si consideramos familias de conjuntos, hay una muy natural y útil: la familia formada por todos los subconjuntos de un conjunto. Pero de nuevo es necesario un axioma que asegure que tal cosa es un conjunto: este es el quinto y último axioma que se utiliza.

Axioma (Del conjunto potencia). Dado un conjunto A, existe el conjunto cuyos elementos son los subconjuntos de A, llamado conjunto potencia y denotado $P(A)$.
El conjunto potencia de $A = \{1, 2, 3\}$ es $P(A) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{3, 1\}, \{1, 2, 3\}\}$
Resumiendo en una lista los cinco axiomas enunciados que son los que usaremos en la teoría desarrollada en este trabajo, y serán la base para construir expresiones lógicas a partir de conjuntos físicos que constituyen el escenario de una planta de producción: a*xioma de existencia, axioma de igualdad, axioma de especificación, axioma de la unión y axioma del conjunto potencia*

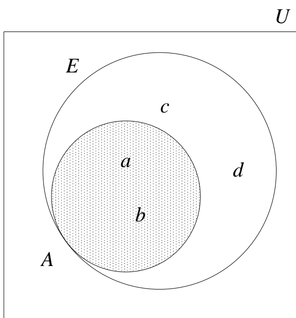
7.2.2. Complemento, unión e intersección

Se definen las operaciones de complemento, unión e intersección y sus principales propiedades, que constituyen el álgebra de conjuntos. También mencionamos las operaciones de diferencia y diferencia simétrica que enseguida escribimos en función de las otras.

La operación de unión de conjuntos no es más que el axioma de la unión ya enunciado y la definición que le sigue. Sin embargo lo volveremos a enunciar en el caso particular de dos conjuntos, que es la forma más habitual de manejarla. De hecho, el axioma de la unión es el que permite establecer los resultados algebraicos. Si tenemos dos conjuntos $A\ y\ B$, dicho axioma nos permite hablar de un conjunto E que contiene todos los elementos de A y todos los elementos de B. Utilizaremos el conjunto E para la explicación de las operaciones y deducir sus propiedades. Al definir las tres operaciones en el marco de un conjunto E ocurre que son una traducción directa de las operaciones entre *proposiciones lógicas:* el *complemento* corresponde a la *negación*, la *unión* a la *disyunción* y la *intersección* a la *conjunción.*

Hay una representación gráfica de los conjuntos que es particularmente apropiada para visualizar las operaciones entre conjuntos: los *diagramas de Venn.* Un diagrama de Venn representa al conjunto E por un rectángulo, y cualquier subconjunto del mismo por una curva cerrada dentro del rectángulo. Si es posible, los elementos del conjunto E se marcan como puntos dentro del rectángulo y la curva que representa a un subconjunto encierra sus elementos. Los conjuntos $E = \{a, b, c, d\}\ y\ A = \{a, b\} \subset E$ se representan en la figura 8.

Figura 7. Representación del conjunto $\boldsymbol{A \subset E}$

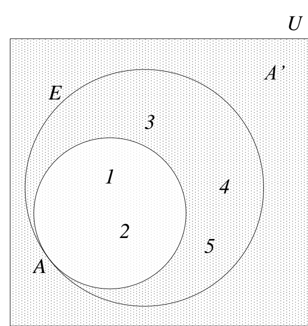


Fuente: Autor

Debe quedar claro que los diagramas de Venn no sirven como demostraciones de teoremas. Solo son ilustraciones de los mismos y con ellos podremos tener claras las relaciones entre conjuntos para cada uno de los recursos declarados y efectuar definiciones. Se representan con diagramas de Venn, la planta de producción y los subconjuntos formados por los centros de trabajo. Asimismo los subconjuntos específicos de los recursos: materiales, mano de obra y equipos por centro de trabajo. Estos conjuntos y subconjuntos definirán los niveles a los cuales se enmarcan los sistemas lógicos. Un sistema estará formado por subsistemas, y a su vez este mismo por otros subsistemas en un nivel inferior.

La primera operación que abordamos es el *complemento*. El complemento de un subconjunto A del conjunto E es el conjunto de todos los elementos de E que no están en A. Se denota A^c y se puede describir como $A^c = \{x \in E \mid x \notin A\}$. Gráficamente, lo vemos en la figura 9.

Figura 8. Representación del conjunto $\boldsymbol{Ac}$



Fuente: Autor

En el conjunto $E = \{1, 2, 3, 4, 5\}$, el complemento del conjunto $A = \{1, 2\}\ es\ Ac = \{3, 4, 5\}$

A continuación nos ocupamos de la *unión* y *la intersección*. En la unión de dos conjuntos se consideran los elementos que están en, al menos, uno de los dos conjuntos. En la intersección, sin embargo, se consideran los elementos que están en ambos conjuntos.
La unión de dos conjuntos A y B es el conjunto formado por los elementos que estan en A o están en B. Se denota $A \cup B = \{x \in E \mid (x \in A) \vee (x \in B)\}$

La intersección de dos conjuntos A y B es el conjunto formado por los elementos que están en A y están en B. Se denota $A \cap B = \{x \in E \mid (x \in A) \wedge (x \in B)\}$

Dados los conjuntos $A = \{1, 2\}\ y\ B = \{2, 3\}$ tenemos $A \cup B = \{1, 2, 3\}\ y\ A \cap B = \{2\}$.
Si dos conjuntos verifican $A \cap B = \varnothing$ se dice que son *disjuntos* porque no tienen elementos en común.

Estas operaciones básicas son importantes para representar la estructura de producto denominada la lista de materiales (o también llamadas BOM) y las instrucciones de ruta de fabricación que son parte esencial de la administración de datos de producto, que son parte de los datos maestros de la planificación de producción.

Las operaciones de diferencia y diferencia simétrica consisten, como indica el nombre, en quitar elementos a un conjunto. La diferencia del conjunto A menos el conjunto B es el conjunto formado por los elementos que están en A pero no en B. Se denota $A \setminus B = \{x \in E \mid (x \in A) \wedge (x \notin B)\}$

Las diferencias de los conjuntos $A = \{1,2\}\ y\ B = \{2,3\}$ son $A \setminus B = \{1\},\ B \setminus A = \{3\}$.
La diferencia de conjuntos, como en los números, no es conmutativa; en general $A \setminus B \neq B \setminus A$. Sin embargo, la diferencia simétrica se construye de forma que si lo es.
La diferencia simétrica de dos conjuntos $A\ y\ B$ es el conjunto formado por los elementos que están en A o están en B excepto los comunes ambos. Se denota $A \Delta B$ y se puede escribir como $A \Delta B = (A \setminus B) \cup (B \setminus A)$
La diferencia simétrica de los conjuntos $A = \{1,2\}\ y\ B = \{2,3\}\ es\ A \Delta B = B \Delta A = \{1,3\}$.

Después de ver en forma resumida las descripciones de las operaciones y su aplicación a la representación física del modelo de producción, se observan las propiedades que se deben satisfacer: las llamadas *leyes del algebra de conjuntos*.

La diferencia y la diferencia simétrica se pueden escribir en función de *unión, intersección* y *complemento*. Como en el caso de *proposiciones lógicas*, también la unión se puede expresar en términos del complemento y la intersección, o la intersección en función del complemento y la unión pero es habitual considerar estas tres por el *paralelismo con las proposiciones*. Ver anexo Teorema 3.
Las *leyes del algebra de conjuntos* son las *leyes del algebra de las operaciones lógicas* de complemento, unión e intersección. A continuación se enumeran algunas de tales leyes. No son todas, pues se pueden deducir otras nuevas a partir de estas. Tampoco son independientes entre ellas, pues algunas de la lista se pueden deducir de otras. Es una elección arbitraria de las más útiles y habituales. Se deduce que la representación gráfica es posible para los conjuntos enunciados en producción y su aplicabilidad se fundamenta en probar que las leyes del algebra se cumplen satisfactoriamente entre los conjuntos y elementos planteados en un escenario de producción junto con las operaciones lógicas para expresar un problema SAT. Ver anexo Teorema 4.

Analizando en detalle una de las propiedades que hacen parte de la representación lógica a plantear, se muestra la primera ley de De Morgan. Escribimos el conjunto de la izquierda según su definición.

$$(A \cup B)^c = \{x \in E \mid x \notin (A \cup B)\}$$

La proposición $x \notin (A \cup B)$ significa $\neg((x \in A) \vee (x \in B))$. Ahora aplicamos la ley de De Morgan de *proposiciones lógicas* y llegamos a que es equivalente a $(x \notin A) \wedge (x \notin B)$. Pero esta proposición define, precisamente, el conjunto $A^c \cap B^c$ y la propiedad queda demostrada.

La existencia de la propiedad asociativa permite definir el símbolo $A \cup B \cup C$ como cualquiera de $A \cup (B \cup C)$ o bien $(A \cup B) \cup C$, pues son iguales. El conjunto $A \cup B \cup C$ esta formado por los elementos que pertenecen, al menos, a uno de los tres conjuntos y, por tanto, coincide con la unión de la familia $\{A, B, C\}$ tal y como se enuncian en el axioma de la unión. Una forma habitual de escribir la unión de una familia grande de conjuntos es mediante

índices: $\{A\}_{\alpha \in I}$ es una familia formada por los conjuntos A_α , donde el subíndice toma diferentes valores (para cada valor de A_α es un conjunto). Los valores que puede tomar α forman el conjunto de índices, que hemos llamado I. Con esta notación la unión de esta familia se escribe $\bigcup_{\alpha \in I} A_\alpha$. Esta notación se aplica a cada uno de los elementos que pertenecen a un conjunto definido.

Sea $I = \{1, 2, 3, 4, 5\}$ un conjunto de índices, y sea $\{A_k\}_{k \in I}$ una familia de intervalos de la recta real dada por $A_k = [k, 3k]$. Entonces $\bigcup_{k \in I} A_k = [1, 15]$

Del mismo modo podemos pensar en la intersección de tres conjuntos, pues también hay asociatividad. El conjunto $A \cap B \cap C$ está formado por los elementos que pertenecen a todos y cada uno de los tres conjuntos. Análogamente, si $\{A_\alpha\}_{\alpha \in I}$ es una familia de conjuntos, definimos la intersección de la familia, escrita $\bigcap_{\alpha \in I} A_\alpha$, como el conjunto de los elementos que pertenecen a todos y cada uno de los A_α. Con los mismos datos anteriores, $\bigcap_{k \in I} A_k = \varnothing$.

En las propiedades enunciadas se puede observar el llamado *principio de dualidad*. Este asegura que dado un teorema de la teoría de conjuntos con los símbolos $\cup, \cap, E \; o \; \varnothing$, su expresión dual (la que se obtiene al cambiar $\cap$ por $\cup$ y cambiar E por $\varnothing$) también es un teorema de la teoría. La base de este teorema es expresar el problema SAT en su FNC *forma normal conjuntiva* o su dual FND *forma normal disyuntiva.*

7.2.3. Producto cartesiano

El producto cartesiano de dos conjuntos es el conjunto formado por parejas ordenadas, con un elemento de cada conjunto. Pero no hemos definido que es una pareja ordenada. Obsérvese que el símbolo $\{a, b\}$ denota el conjunto cuyos elementos son $a \; y \; b$; es una pareja. Pero no es ordenada ya que, según el axioma, $\{a, b\} = \{b, a\}$ pues tienen los mismos elementos. Necesitamos definir el símbolo (a, b) en el que, en general, $(a, b) \neq (b, a)$. ¿Cómo hacerlo? Al usar el símbolo $\{a, b\}$ y añadir la información de cuál de los dos elementos es el primero. Considerar esta aplicación para generar pares ordenados entre productos y centros de trabajos.

La pareja ordenada (a, b) es el conjunto $\{\{a\}, \{a, b\}\}$. Es correcto llamar conjunto a (a, b) pues obsérvese que si $a \in A \; y \; b \in B$, entonces $\{a, b\}$ es un subconjunto de $A \cup B$, es decir, un elemento de $P(A \cup B)$. Entonces (a, b) es subconjunto de $P(A \cup B)$ y, por tanto elemento de $P(P(A \cup B))$, todo ello apoyado en la existencia del conjunto potencia que asegura el axioma del mismo nombre.
Con el concepto de pareja ordenada, que es diferente del símbolo $\{a, b\}$ podemos definir el producto cartesiano como un subconjunto de $P(P(A \cup B))$. El producto cartesiano es fundamental, debido a que en cada nivel se determinan vínculos de elementos entre conjuntos centros de trabajo y subconjuntos recursos asociados al mismo.

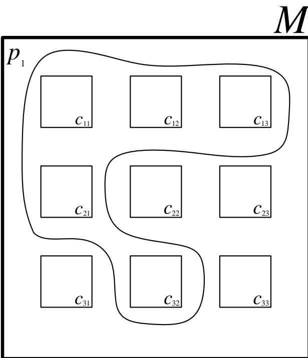
El producto cartesiano de dos conjuntos $A\, y\, B,$ denotado $A\times B,$ es el conjunto formado por todas las parejas ordenadas cuyo primer elemento es del conjunto A y cuyo segundo elemento es del conjunto B.

$$A\times B=\{(a,b)\mid a\in A\wedge b\in B\}$$

Sean los conjuntos $A=\{1,2\}\, y\, B=\{a,b\}$. Entonces, su producto cartesiano es el conjunto $A\times B=\{(1,a),(1,b),(2,a),(2,b)\}$. Puesto que las parejas son ordenadas, $A\times B$ no es lo mismo que $B\times A$.

La representación de la planta de producción a través del diagrama de Venn proporcionara una vista pictórica de los conjuntos a ser nombrados dentro de este universo nombrado como M, donde los elementos designados para cada conjunto serán los centros de trabajo $c_{11},c_{12},\dots,c_{ij}$ que estarán asociados para cada uno de los productos $p_1,p_2,\dots,p_{10}$. Figura 10.

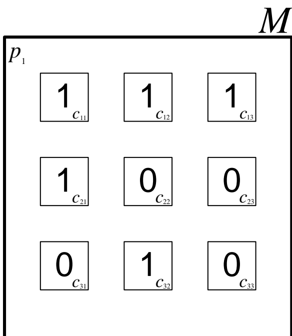
Figura 9. Representación gráfica de conjuntos por producto $\boldsymbol{p_i}$ asociados a centro de trabajo $\boldsymbol{c_n}$



Fuente: Autor

Se enmarcan de una forma no homogénea el conjunto, donde se encierra los respectivos centros de trabajo que intervienen en el proceso de producción para definir los elementos del conjunto.
De esta forma se obtienen los diferentes conjuntos en forma individual por producto, sobre el universo planteado. Figura 11. Dentro de los conjuntos formados por producto, la identificación de pertenencia de los centros de trabajo relacionados en cada conjunto se indica con el valor **1** uno, o de lo contrario se identificara con el valor **0** cero, y será parte del complemento del mismo $\neg p_n$.

Figura 10. Relación de pertenencia $\{\mathbf{0},\mathbf{1}\}$ de centro de trabajo $\boldsymbol{c_n}$ a cada producto $\boldsymbol{p_i}$



Fuente: Autor

Al sobreponer cada uno de los escenarios se obtienen todas las posibles intersecciones, donde se determinan que centros de trabajo tienen mayor participación dentro del proceso de fabricación para cada uno de los productos e igualmente la cantidad de procesos por producto definidos en su arquitectura, y se describe los diferentes procesos que están asociados para obtener el producto final. Ver anexo de la representación obtenida para cada uno de los productos planteados. Figura 12.

Figura 11. Conjuntos cada producto $\boldsymbol{p_i}$ asociados a centros de trabajo $\boldsymbol{c_n}$

Fuente: Autor

7.3. PASO 3: Relaciones

En la metodologia desarrollada hasta este punto la única referencia que se ha hecho a los elementos de un conjunto es la pertenencia a dicho conjunto. No hay ninguna conexión entre los elementos de un conjunto (aparte de la de pertenecer al mismo) y, mucho menos, entre elementos de diferentes conjuntos. El papel de las relaciones y las funciones es, precisamente, establecer dichas conexiones.

Las relaciones son la forma más básica (y por ello de más alcance) de imponer una estructura en un conjunto. La explicación general de relación y enseguida nos concentramos en los dos tipos más importantes: las *relaciones de equivalencia* y las *relaciones de orden.*

Las equivalencias son las que permiten clasificar los elementos de un conjunto. El objetivo de las equivalencias es ver el resultado de que toda equivalencia da lugar a una clasificación de los elementos del conjunto y viceversa, toda clasificación (o partición) de un conjunto procede de una relación de equivalencia. Se reflejara en las relaciones de una orden de trabajo (*WO wokr order* por sus siglas en inglés) con las especificaciones de ingeniería de producto dado (*PDM product data management* por sus siglas en inglés).
Las órdenes son los que ordenan los elementos de un conjunto. El objetivo es conocer diferentes tipos de órdenes que existen y, en particular, entender la estructura de orden de los naturales, de los enteros, de los racionales y de los reales. Se enuncian las propiedades que distinguen cada uno de estos órdenes de todos los demás.

Partamos del conjunto de productos en una planta de producción. En el podemos relacionar entre sí a los productos que se transforman en máquinas de la planta, con lo cual cada producto estará conectado con algunos otras máquinas –en centros de trabajo adyacentes– y no lo estará con algunos más. Se podra pensar en otro ejemplo en la misma planta de producción si relacionamos a cada producto con subproductos, si los tiene y se producen. Definimos relación como un objeto matemático para describir conexiones entre los elementos de un conjunto. En particular se examinan dos tipos de especial importancia*:* las *relaciones de equivalencia y las relaciones de orden.* Estas son, respectivamente, la forma matemática de establecer clasificaciones, como ocurre en el caso de los centros de trabajo por secciones en una planta de producción, y de ordenar artículos en la medida que se transforman, que es lo que ocurre entre subensambles y productos finales.

Se trata, por tanto, de decir que elementos están relacionados con cuales, y una forma es escribiendo parejas ordenadas (a, b) que signifiquen que el elemento a esta relacionado con el b. Vemos que el *orden* es importante (no es igual que a sea padre de b o que b lo sea de a).

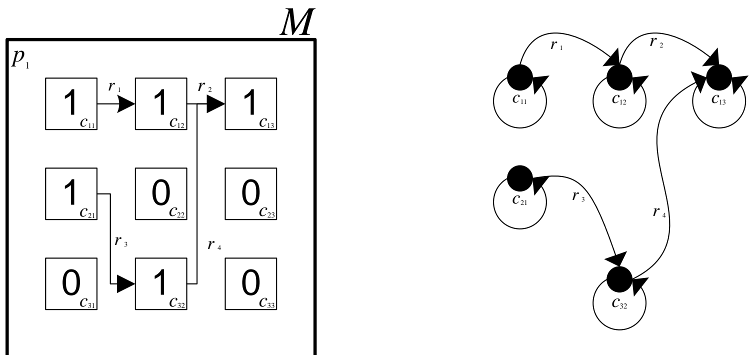
Una relación R en un conjunto A es un subconjunto no vacio del producto cartesiano $A \times A$. Para denotar que un elemento a esta relacionado con otro b por la relación R escribimos $(a, b) \in R$ o también aRb (y su negación aRb).
En el conjunto $A = \{1, 2, 3\}, R_1 = \{(1, 2), (1, 3), (2, 3)\}$ es una relación que podríamos llamar la relación del orden habitual, ya que indica que el primer elemento del par precede al segundo según el orden habitual de los números. Otra relación es $R_2 = \{(1, 1), (1, 3), (2, 2), (3, 1), (3, 3)\}$, que podríamos llamar la relación de paridad, pues los elementos de cada pareja son ambos pares o ambos impares.

En la explicación no se exige que todos los elementos del conjunto estén relacionados con algún otro, ni que todos reciban la relación de alguno. Por ello describimos el *dominio* y el *contradominio* a continuación. El dominio de una relación R en el conjunto A es el subconjunto de A de elementos que están relacionados con algún otro. Lo denotamos $D(R)$ y lo podemos expresar como $D(R) = \{x \in A| \exists y, xRy\}$.

El contradominio de una relación R en el conjunto A es el subconjunto de A de elementos con los que alguno está relacionado. Lo denotamos $D0(R)$ y lo escribimos como $D'(R) = \{x \in A| \exists y, yRx\}$
La relación inversa de una relación R es la relación formada por las parejas de R invirtiendo el orden de los elementos en cada pareja. Se denota por R^{-1}. Es decir, $R^{-1} = \{(a,b) \in A \times A| bRa\}$
De la aclaración se desprende inmediatamente que $D(R^{-1}) = D'(R)$ y $D'(R^{-1}) = D(R)$

Hay una forma de representación gráfica para relaciones que es muy intuitiva e ilustrativa (cuando se puede poner en práctica). Se representa el conjunto y sus elementos como en un diagrama de Venn, y cada pareja (a,b) de la relación se representa por una flecha que nace en el punto a y muere en el punto b. Una vez definidos los elementos del conjunto por centros de trabajo/producto, se efectúan las relaciones entre los mismos, para determinar el escenario de cómo se establece el orden de las operaciones para obtener el producto final. Figura 13.

Figura 12. Relación de pertenencia y equivalencia $\{\mathbf{0},\mathbf{1}\}$ de centro de trabajo $\boldsymbol{c_n}$ a cada producto $\boldsymbol{p_i}$



Fuente: Autor

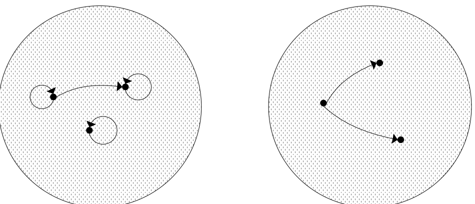
La relación $R = \{(a,a),(a,b),(a,c)\}$ definida en el conjunto $A = \{a,b,c\}$. El análisis matemático de las relaciones se concentra en la estructura que impone la relación en el conjunto, independientemente de cómo se haya definido. Para ello examina propiedades de las relaciones que se definen independientemente del significado de la relación. A continuación se describe formalmente tales propiedades.

Una relación definida en un conjunto se llama reflexiva si cada elemento del conjunto está relacionado consigo mismo. Es irreflexiva si ningún elemento se relaciona consigo mismo.

$$R\ reflexiva\ , \Leftrightarrow \forall x\ \in\ A, xRx$$
$$R\ irreflexiva, \Leftrightarrow \forall x\ \in\ A, xRx$$

La relación “ser empleado de” no es ni una ni otra, pues algunos empresarios son empleados de sí mismos, mientras que muchos trabajadores son empleados de otra persona y no de ellos mismos. La figura 14 representa una relación reflexiva y una irreflexiva.

Figura 13. Una relación reflexiva y una relación irreflexiva.



Fuente: Autor

Una relación es simétrica si para cada pareja de la relación, la pareja en orden inverso también forma parte de la relación. Es asimétrica si para toda pareja en la relación se cumple que su inversa no está en la relación. Por último, una relación anti simétrica es aquella en que las únicas parejas cuyas inversas también son parte de la relación son las parejas de elementos iguales.

$$R\ simetrica\ , \Leftrightarrow \forall x, y(xRy\ \rightarrow\ yRx)$$
$$R\ asimetrica\ , \Leftrightarrow \forall x, y(xRy\ \rightarrow\ y6Rx)$$
$$R\ antisimetrica\ , \Leftrightarrow \forall x, y((xRy \wedge yRx)\ \rightarrow\ x\ =\ y)$$

Una relación es transitiva si siempre que un elemento se relaciona con un segundo elemento, y este con un tercero, entonces el primero está relacionado con el tercero.

$$R\ transitiva\ , \Leftrightarrow \forall x, y, z((xRy \wedge yRz) \rightarrow\ xRz)$$

Se puede representar una relación transitiva y otra que no lo es. Estas propiedades no son todas independientes. Para empezar, las propiedades reflexiva e irreflexiva son incompatibles, al igual que las propiedades de simetría y antisimetrıa. Se describen los dos tipos más importantes de relaciones.

7.3.1. Relaciones de equivalencia

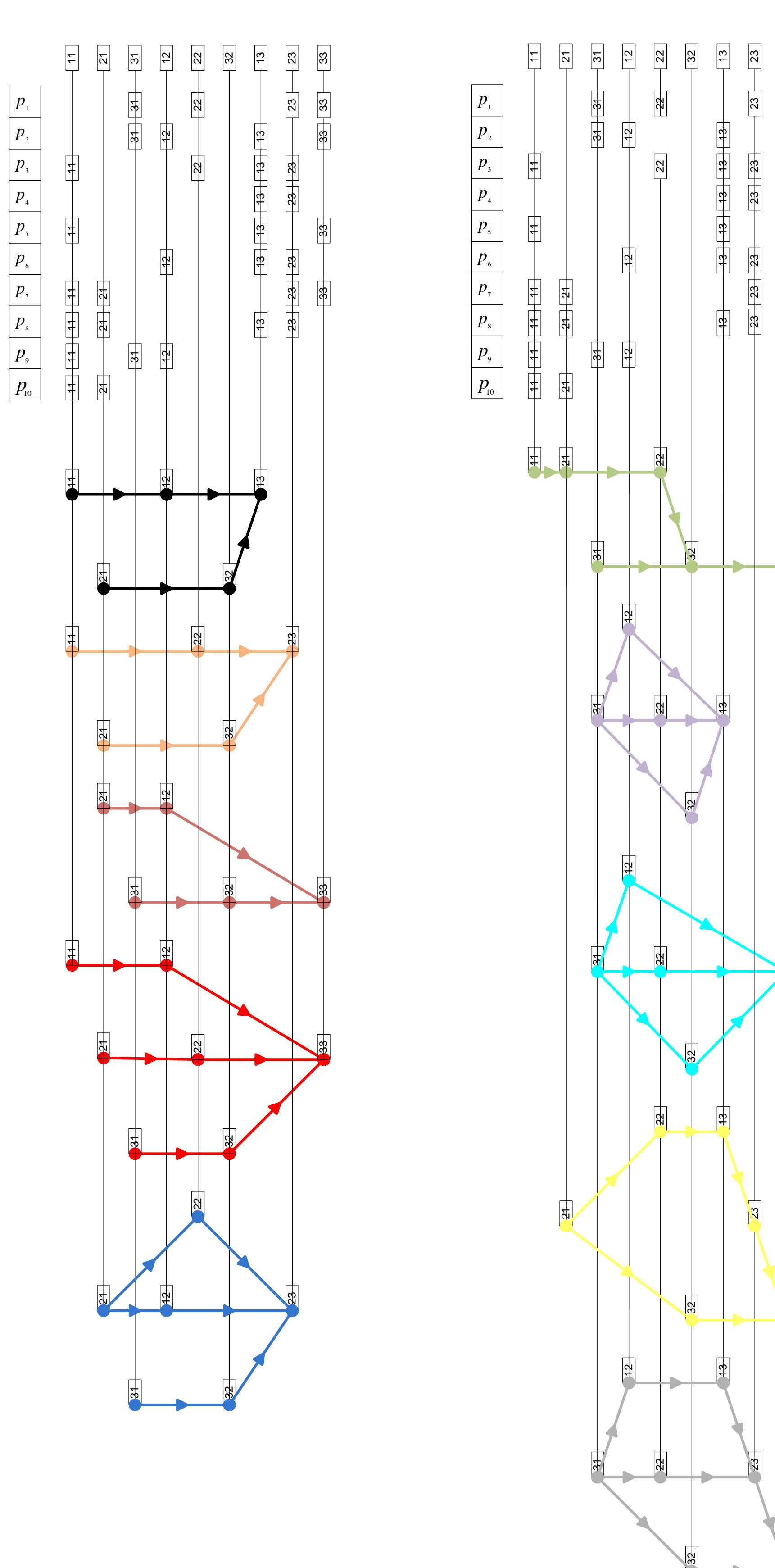
La *relación de equivalencia*, o simplemente equivalencia, es la herramienta matemática para hacer clasificaciones. Primero se define equivalencia; luego clasificación o, como se le suele llamar, partición; finalmente el teorema fundamental, que afirma que ambas son la misma cosa. (Raposo, 2010)

Una relación es una equivalencia si es reflexiva, simétrica y transitiva. Si el elemento a esta relacionado con b, lo denotamos $a \sim b$. Ver anexo Teorema 5.

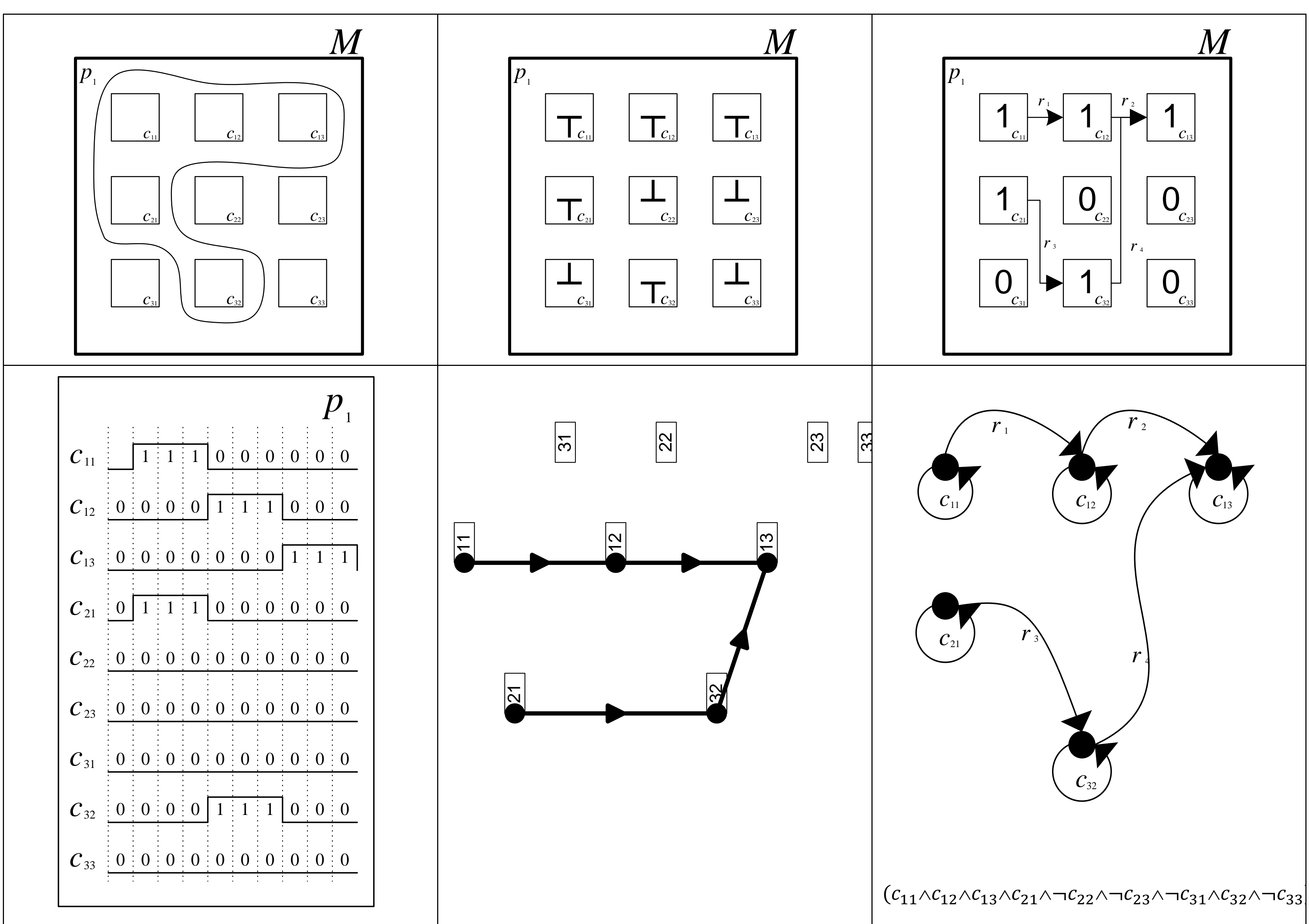
Representando una equivalencia se observa enseguida que los elementos se agrupan en subconjuntos disjuntos entre sí. Este es el resultado central de la *teoría de equivalencias*. Una equivalencia produce una clasificación de los elementos del conjunto. Cada clase contiene elementos que están relacionados todos entre si y no están relacionados con ningún otro fuera de su clase. Estas clases se denominan *clases de equivalencia*.

Una equivalencia en el conjunto $A = \{a, b, c, d\}$. Aparecen tres clases de equivalencia: $\{a, b\}, \{c\}, \{d\}$.
Dada una equivalencia R definida en A y un elemento a del conjunto A, la clase de equivalencia de a es el subconjunto de los elementos relacionados con el. La denotamos por $[a]$, y decimos que a es un representante de dicha clase. Se puede escribir como $[a] = \{b \in A | a \sim b\}$
El representante de una clase es parte de la clase (como era de esperar), ya que la relación es reflexiva. Pero, por otro lado, cualquier elemento de la clase puede ser un representante. Finalmente, elementos no relacionados entre sí están en clases diferentes. Estas ideas en un teorema. Ver anexo Teorema 6.

Por último se tiene que $[a] = [b] \Rightarrow a \sim b$. Como $[a]\ y\ [b]$ son una misma clase, entonces $a\ y\ b$ estan en dicha clase, por el primer punto de este teorema.
Por definición de los símbolos $[a]\ y\ [b]$, se cumple tanto $a \sim b$ como $b \sim a$.



p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
p_9
p_{10}
11
21
31
12
22
32
13
23
33



M
p_1
c_{11} c_{12} c_{13}
c_{21} c_{22} c_{23}
c_{31} c_{32} c_{33}
M
p_1
⊤ c_{11} ⊤ c_{12} ⊤ c_{13}
⊤ c_{21} ⊥ c_{22} ⊥ c_{23}
⊥ c_{31} ⊤ c_{32} ⊥ c_{33}
M
p_1
1 c_{11} r_1 1 c_{12} r_2 1 c_{13}
1 c_{21} 0 c_{22} 0 c_{23}
r_3 r_4
0 c_{31} 1 c_{32} 0 c_{33}
p_1
c_{11} 1 1 1 0 0 0 0 0 0
c_{12} 0 0 0 0 1 1 1 0 0 0
c_{13} 0 0 0 0 0 0 0 1 1 1
c_{21} 0 1 1 1 0 0 0 0 0 0
c_{22} 0 0 0 0 0 0 0 0 0 0
c_{23} 0 0 0 0 0 0 0 0 0 0
c_{31} 0 0 0 0 0 0 0 0 0 0
c_{32} 0 0 0 0 1 1 1 0 0 0
c_{33} 0 0 0 0 0 0 0 0 0 0
31 22 23 33
11 12 13
21 32
r_1 r_2
c_{11} c_{12} c_{13}
c_{21} r_3 r_4
c_{32}
$(c_{11} \wedge c_{12} \wedge c_{13} \wedge c_{21} \wedge \neg c_{22} \wedge \neg c_{23} \wedge \neg c_{31} \wedge c_{32} \wedge \neg c_{33}$

7.3.2. Relaciones de equivalencia y particiones

El principal resultado de la teoría de equivalencias, como se ha dicho, es que las clases de equivalencia constituyen una clasificación de los elementos del conjunto. Es decir, todo elemento está en una clase de equivalencia y solo en una.
Se enuncia primero una aclaración precisa del concepto de *clasificación* o *partición* para, después, enunciar y demostrar el teorema. Nótese que la definición de partición no habla de elementos, sino de subconjuntos, pero es equivalente a decir que cada elemento esta en uno, y solo uno, de tales subconjuntos.

Una partición de un conjunto A es una familia de subconjuntos de $A, \{A_\alpha\}_{\alpha \in I}$, don$de I$ es un conjunto de índices, tal que la unión de todos los subconjuntos es el conjunto A y la intersección de dos diferentes cualesquiera es vacía.
Es decir, $\{A_\alpha\}_{\alpha \in I}$ es una partición de A si $\forall \alpha \in I, A_\alpha \subset A$, $\cup_{\alpha \in I} A_\alpha = A$ y $\forall \alpha, \beta \in I (\alpha \neq \beta \rightarrow A_\alpha \cap A_{\boldsymbol{\beta}} = \emptyset)$
Dado el conjunto $A = \{1, 2, 3, 4, 5, 6\}$, las familias $P_1 = \{\{1, 2\}, \{3, 4, 5\}, \{6\}\} \, y \, P_2 = \{\emptyset, A\}$ son particiones de A, mientras que $P_3 = \{\{1, 2, 3, 4\}, \{4, 5, 6\}\} \, y \, P_4 = \{\{1, 2\}, \{6\}\}$ no lo son. Pero, precisamente esta implicación también se ha probado en el teorema anterior. Ver anexo Teorema 7.
La conexión entre *equivalencias* y *particiones* que establece el teorema anterior queda totalmente complementada por el siguiente teorema, que es su reciproco. En definitiva, toda equivalencia produce una partición y toda partición procede de una equivalencia. Ver anexo Teorema 8.

Ahora veamos que las clases de equivalencia de esta relación son exactamente los subconjuntos de la partición. Por la explicación de la relación, la clase $[a]$ está formada por los elementos que se hallan en el mismo subconjunto que a. Es, pues, dicho subconjunto.
Al finalizar el análisis de las equivalencias introduciendo un concepto muy sencillo pero de gran alcance en matemáticas: el *conjunto cociente*. Este concepto atrapa la esencia de la estructura adquirida por un conjunto cuando se define en él una equivalencia: *su partición en clases*. El conjunto cociente es, precisamente, el conjunto de las clases de equivalencia en que queda dividido el conjunto original (por ello el termino cociente).

Dada una equivalencia R definida en un conjunto A, el conjunto cociente, denotado A/R (o también $A/\sim$) es el conjunto de las clases de equivalencia de la relación, esto es $A/R = \{[a] \in P(A) \mid a \in A\}$.

7.3.3. Relaciones de orden

Como su nombre indica, la relación de orden, o simplemente orden, es la herramienta matemática diseñada para ordenar los elementos de un conjunto.

En la teoría básica de conjuntos los elementos de un conjunto no están ordenados, y $\{1,2,3\}$ es el mismo conjunto que el $\{3,2,1\}$ debido al axioma. El *orden* supone una estructura añadida al conjunto, y se adquiere mediante la descripción de una relación apropiada.
Para establecer un orden hemos de señalar que elementos preceden a cuales, lo cual se indica mediante la relación: si a precede a b, entonces $(a,b) \in R$. (Barnes, 1978)

Claramente la pareja inversa no puede ser parte de la relación, por lo cual pediremos que esta sea asimétrica. Además, si un elemento precede a otro y este a un tercero, entonces el primero debe preceder al tercero para evitar relaciones cíclicas por lo cual exigiremos transitividad.

Una relación R definida en un conjunto A es un orden si todo elemento de A esta en el dominio o en el contradominio y es asimétrica y transitiva. El símbolo $a < b$ se usa para denotar $(a,b) \in R$. El par (A,R) se llama *conjunto ordenado*. El símbolo $a < b$ se lee "a precede a b", "b sucede a a", "a es menor que b" o "b es mayor *que* a".

Se puede sustituir en las dos aclaraciones precedentes la condición de asimetría por la de antisimetrıa. Se tiene entonces *un orden no estricto*, y el símbolo utilizado es ≤ . Este símbolo también se puede definir como $a \leq b \Leftrightarrow a < b \vee a = b$. Un resultado muy sencillo y esperable es el siguiente referido a la relación inversa de un orden. Ver anexo Teorema 9.

Un elemento de A es maximal si no precede a ningún elemento, es decir a *maximal*, $\Leftrightarrow \forall x, a\ 6 < x$. Análogamente se define elemento *minimal* como aquel que no sucede a ningún elemento.
Un elemento de A es máximo si sucede, o es igual, a cada elemento del conjunto. Se denota $max\ A$.

$$a = max\ A \Leftrightarrow \forall x, x \leq a$$

Análogamente se define el elemento mínimo como aquel que precede, o es igual, a todo elemento, denotado $min\ A$.
Conviene hacer notar que no siempre existen *máximos* y *mínimos* en un conjunto ordenado, ni tampoco *maximales* o *minimales*. Ver anexo Teorema 10. Otro concepto estrechamente relacionado con el de *máximo* es el de *supremo* (y el *ínfimo* con el *mínimo*). Para definir supremo e ínfimo hay que definir primero *cota superior* y *cota inferior*.

Sea B un subconjunto de A. Decimos que B esta acotado superiormente si existe un elemento de A que es mayor o igual que cualquier elemento de B, lo cual podemos escribir como B acotado superiormente, $\Leftrightarrow \exists a, \forall b, (b \in B \rightarrow b \leq a)$.

El elemento a se llama una cota superior de B. Análogamente se define conjunto acotado inferiormente y cota inferior. Un subconjunto acotado puede tener muchas cotas superiores. En ocasiones existe una que se distingue de las demás, la menor de todas ellas, llamada *supremo*.
Sea B un subconjunto de A, acotado superiormente. Si el conjunto de cotas superiores de B tiene mínimo, este elemento se llama supremo de B, denotado $sup\ B$.

Sea B un subconjunto de A, acotado inferiormente. Si el conjunto de cotas inferiores de B tiene máximo, este elemento se llama *ínfimo* de B, denotado $\inf B$.
Puesto que no todo conjunto tiene máximo o mínimo, no todo conjunto tiene supremo o ínfimo. Pero lo que sí es cierto es que, si el supremo o el ínfimo existen, entonces son únicos. Vemos un conjunto que tiene supremo e ínfimo, aunque no máximo ni mínimo, un conjunto que tiene tanto supremo como máximo e ínfimo y mínimo y un conjunto sin supremo ni ínfimo.

Al considerar el conjunto de los números racionales (pero no como subconjunto de los reales, sino los racionales solamente). El intervalo cerrado $[\mathbf{0}, \mathbf{1}]$, es decir, los racionales entre **0** y **1**, estos incluidos, es un subconjunto acotado. Tiene máximo, el **1**, que es también el supremo, y tiene mínimo, el **0**, que es ínfimo.
Sin embargo el intervalo abierto $]\mathbf{0}, \mathbf{1}[$ de los racionales, es decir, el anterior pero quitando el **0** y el **1**, es de nuevo un conjunto acotado. Tiene supremo, el **1**, pero no tiene máximo porque **1** ya no pertenece al subconjunto. Igualmente tiene ínfimo, el **0**, que no es mínimo por la misma razón.
Algo más complicado es describir un subconjunto acotado pero que no tenga supremo o ínfimo. La relación precisa entre supremo y máximo de un subconjunto (así como ínfimo y mínimo) se da en el siguiente resultado. Ver anexo Teorema 11.

Orden total. Visto conjuntos ordenados donde existen elementos que no son comparables. Esta situación no se da en el ejemplo más común de conjunto ordenado en matemáticas: *los conjuntos numéricos.* En ellos todos los elementos son comparables. Esta propiedad le da características especiales al orden y por ello recibe un nombre. (Ershov, 1990)

Un orden R en el conjunto A es total si dados dos elementos cualesquiera, o bien son iguales, o bien uno precede al otro.

$$\forall x, y \in A, (x = y) \vee (x < y) \vee (y < x)$$

Esta propiedad también recibe el nombre de tricotomía, por las tres opciones que permite a cada pareja de elementos x, y. Cuando un orden no es total, para recalcar la ausencia de esta propiedad a veces se denomina orden parcial.

En un conjunto E de al menos dos elementos, el orden definido por la inclusión de conjuntos es un orden parcial. El orden habitual de los números naturales, N de los enteros, Z, de los racionales, Q, y de los reales, R, son todos órdenes totales. Una diferencia notable entre los órdenes parciales y totales es la relación de elementos máximos con maximales (y mínimos con minimales). En el teorema se estableció que todo máximo es maximal; en un orden total tenemos también el recíproco: todo maximal es máximo. Por tanto, ambos son la misma cosa. Ver anexo Teorema 12.

Un conjunto está bien ordenado si todo subconjunto tiene mínimo. En un conjunto totalmente ordenado A el elemento y es inmediato sucesor del elemento x si y sucede a x y cualquier otro elemento que sucede a x también sucede a y, es decir, si $(x < y) \wedge \forall z, (x < z \rightarrow y \leq z)$. En este caso, decimos que x es inmediato antecesor de y. Es fácil demostrar que, en caso de existir, el inmediato sucesor de un elemento es único. El comportamiento de los conjuntos numéricos respecto a la existencia de inmediatos sucesores se recoge en las siguientes dos propiedades: las propiedades del *inmediato sucesor* y del *inmediato antecesor* y el *orden lineal*.

Un orden total tiene la propiedad de inmediato sucesor si todo elemento que no sea máximo tiene un inmediato sucesor. Análogamente, un orden total tiene la propiedad de inmediato antecesor si todo elemento que no sea mínimo tiene un inmediato antecesor.

Un orden total R en A es lineal si para cualquier pareja de elementos diferentes existe un elemento situado entre los dos. Es decir, R es lineal si $\forall x, y \in A, (x < y \rightarrow \exists z, x < z < y)$.

Esta propiedad es incompatible con las anteriores en un conjunto (aunque pueden coexistir si cada una se verifica en un subconjunto diferente). Ver anexo Teorema 13.

Falta definir una propiedad que permita diferenciar el orden en los racionales y el orden en los reales: *la propiedad del supremo*. Un orden total verifica la propiedad del supremo si todo subconjunto acotado superiormente tiene supremo. Un orden total verifica la propiedad del ínfimo si todo subconjunto acotado inferiormente tiene ínfimo. La propiedad del supremo y la del ínfimo son equivalentes. Ver anexo Teorema 14.

Las relaciones y las funciones establecen una estructura para desarrollar las operaciones lógicas. Es decir, se evalúan alternativas basadas en estas relaciones para evaluar una serie de alternativas en forma recurrente, dicho

de un proceso que se repite, que vuelve a ocurrir o aparecer después de un intervalo de tiempo. De cierta forma los productos definidos en una planta de producción se vuelven a transformar en algún momento una vez se cumpla el requerimiento de una orden de trabajo y se repite en diferente o igual cantidad que las anteriores solicitudes.

7.4. PASO 4: Funciones de recurrencia - Inducción matemática

La *recursión* y la *inducción* están íntimamente relacionadas. Una demostración por inducción puede considerarse como una demostración recursiva. La demostración de las propiedades de los procedimientos recursivos se simplifica mucho por su similitud estructural. Incluiremos "función" en el significado general de "procedimiento"; la terminología de Java es "método").
Mostrar los árboles de recursión para contar con un marco general en el cual analizar las necesidades de tiempo de los procedimientos recursivos, e introducir estructuras de datos recursivas junto con procedimientos y funciones recursivos.

El entender con claridad cómo funciona realmente la recursión en una máquina de computación lógica que ayuda mucho a pensar recursivamente, a ejecutar código recursivo, y permite analizar el tiempo de ejecución de los procedimientos recursivos. Cómo se implementan las invocaciones de procedimientos con *marcos de activación*, y el apoyo que brindan las invocaciones a la recursión. Sin embargo, para la mayor parte de las actividades relacionadas con el diseño y análisis de procedimientos recursivos queremos pensar en un nivel más alto que el de los marcos de activación, que en realidad es un truco mental que permite diseñar soluciones recursivas.

7.4.1. Algoritmos recursivos

Un algoritmo es un método paso a paso para resolver algunos problemas, y este enfoque no es una novedad. En el contexto actual, "algoritmo" lo referiré a una solución ejecutable en una máquina de computo abstracta, y la preocupación principal será que se pueda ejecutar en una máquina de Turing planteada, que ejecute instrucciones paso a paso. (Turing, 1936) Al introducir los algoritmos y proporcionar varias pruebas, se dedicara tiempo al análisis de los algoritmos, es decir, al tiempo y espacio requeridos para ejecutarlos y concluiremos con un análisis de los *algoritmos recursivos*.

Una *función recursiva* (en seudocódigo binario) es una función que se invoca a sí misma. Un algoritmo recursivo es un algoritmo que contiene una función recursiva. La recursión es una forma poderosa, elegante y natural de resolver una clase amplia de problemas. (Cormen T. H., 2010) Un problema de esta clase se resuelve mediante una técnica de *divide y vencerás* en la que el problema se descompone en problemas del mismo tipo que el

problema original. Cada subproblema, a su vez, se descompone aún más hasta que el proceso produce subproblemas que se pueden resolver de manera directa. Por último, las soluciones de los subproblemas se combinan para obtener una solución del problema original. Cada subproblema está representado en una MT individual por unidades físicas de operación en planta que se relacionan de forma adyacente entre ellas, según lo tratado en la relaciones de equivalencia.

Los argumentos anteriores se pueden generalizar usando *inducción matemática* para probar que el algoritmo regresa un valor lo más acertado posible. Si se ejecuta en una computadora, en general el algoritmo no será tan eficiente como la versión no recursiva a causa de todas las llamadas recurrentes generales. Debe haber algunas situaciones en las que una función recursiva no se invoque a sí misma; de otra manera, se invocaría a sí misma sin detenerse. Estos valores para los cuales una función recursiva no se invoca a sí misma se llaman *casos base*. Para resumir, toda función recursiva debe tener casos base.

Demostrar que al utilizar la *inducción matemática* para probar que un algoritmo recursivo calcula el valor que asegura calcular. La relación entre la inducción matemática y los algoritmos recursivos es profunda. Con frecuencia, una prueba por inducción matemática se considera como un algoritmo para calcular un valor binario $\{\mathbf{1},\mathbf{0}\}$ y llevar a cabo una construcción en particular. El paso base de una prueba por inducción matemática corresponde a los casos base de una función recursiva, y el paso inductivo de una prueba por inducción matemática corresponde al paso de una función recursiva donde la función se invoca a sí misma. *La clave para escribir una función recursiva es encontrar una instancia más pequeña del problema dentro del problema más grande.*

7.4.2. Funciones recursivas

Actualmente, los programadores están familizarizados con las funciones recursivas. Sin embargo, estas funciones recursivas parecen no tener nada que ver con las máquinas de Turing que siempre se paran. Lo contrario (no recursivo o indecidible) hace referencia a los lenguajes que no pueden ser reconocidos por ningún algoritmo, aunque estamos acostumbrados a pensar en que "no recursivo" se refiere a cálculos que son tan simples que para resolverlos no se necesitan llamadas a *funciones recursivas.*
El término "*recursivo*" como sinónimo de "*decidible*" nos lleva al campo de las matemáticas, ya que se conocen antes que las computadoras. Entonces, los formalismos para el cálculo basados en la recursión (pero no en la iteración o en los bucles) se empleaban habitualmente como un concepto de computación. Estas notaciones, que no vamos a abordar aquí, guardan cierto parecido con los lenguajes de programación funcionales como $C\#$ o ML. En este sentido, decir que un problema era "recursivo" tenía un matiz positivo, ya que significaba que "era lo

suficientemente sencillo como para poder escribir una función recursiva para resolverlo y que además siempre terminara". Éste es exactamente el significado que tiene hoy día el término, y que lo conecta con las máquinas de Turing. (John, 2008).

En el contexto de las máquinas de Turing, el cual era un argumento plausible para la existencia de problemas que podrían ser resueltos por computadora. El problema es que estábamos forzados a ignorar las limitaciones reales que toda implementación en lenguaje C (o en cualquier lenguaje de programación) tiene en cualquier computadora real. Dichas limitaciones, tales como el tamaño del espacio de direccionamiento, no son limitaciones esenciales. Más bien, puede pensarse que, a medida que pase el tiempo, las prestaciones de las computadoras seguirán creciendo indefinidamente en aspectos como el tamaño del espacio de direccionamiento, el tamaño de la memoria principal, etc.

El poder de las máquinas de Turing como aceptadores de lenguajes corresponde a los poderes generativos de las gramáticas. En este tema investigaremos otros casos en los cuales el poder computacional de las máquinas de Turing corresponde a las capacidades de otros sistemas computacionales.

Al seguir la investigación se considera el poder expresivo de un determinado lenguaje de programación. Recordar que existen funciones que no pueden calcularse con ningún programa escrito en ese lenguaje. La pregunta que se plantea ahora es si esta limitación es debida al diseño del lenguaje, o si, por el contrario, es un reflejo de las limitaciones de los procesos algorítmicos en general. Es decir, ¿si la imposibilidad de calcular una función por un programa en un determinado lenguaje significa que el lenguaje es incapaz de expresar un algoritmo, o más bien que no existe ningún algoritmo que pueda calcular esa función?.

Para responder esta pregunta, primero se identifica una clase de funciones que contiene la totalidad de las funciones computables, es decir, todas aquellas funciones que pueden calcularse por medios algorítmicos, sin importar como pueda expresarse o implementarse ese algoritmo.

Después que las funciones de esta clase pueden calcularse por medio de las máquinas de Turing, y también mediante algoritmos expresados en un subconjunto muy sencillo de la mayoría de los lenguajes de programación. A partir de esto llegaremos a la conclusión de que los limites detectados en las máquinas de Turing y en la mayor parte de los lenguajes de programación son el reflejo de las limitaciones de los procesos computacionales, no del diseño de la máquina o del lenguaje que se emplee.

7.4.2.1. Fundamentos de la teoría de funciones recursivas

El análisis de los procesos computacionales y sus capacidades se ha basado en un enfoque operativo, no funcional. Se ha centrado el proceso en cómo se lleva a cabo un cálculo, no en lo que este cálculo logra. Sin

embargo, nuestro objetivo es identificar las funciones que pueden calcularse con al menos un sistema computacional, así que debemos alejarnos de cualquier método especifico para expresar o ejecutar los cálculos. En otras palabras, queremos utilizar el término computable para decir que una *función* puede calcularse mediante algún algoritmo, más que por un algoritmo que pueda implementarse en algún sistema especifico.

Para obtener esta generalidad, adoptaremos un enfoque funcional que nos permita examinar la computabilidad, centrándonos en determinar que funciones pueden ser calculadas, y no en la forma de calcularlas. Más concretamente, se utiliza el método que han seguido los matemáticos para el análisis de las *funciones recursivas*. Por tanto, al comenzar con un conjunto de funciones, llamadas *funciones iniciales*, cuya sencillez es tal que no hay dudas sobre su computabilidad, para luego mostrar que estas funciones se pueden combinar formando otras cuya computabilidad se deduce de las *funciones originales*. De esta manera, se obtiene una colección de funciones que contiene todas las *funciones computables*. Así pues, si un sistema computacional, como un lenguaje de programación o un ordenador, abarca todas estas funciones, concluiremos que el sistema tiene todo el poder posible. De lo contrario, dicho sistema estará necesariamente restringido. (Brookshear, 1993)

Funciones parciales. Las funciones pueden abarcar una diversidad muy amplia de dominios y codominios. Para enfrentarnos a esta diversidad, es necesario observar que cualquier dato se puede codificar mediante una cadena de *ceros* y *unos* $\{\mathbf{0,1}\}$ y, en el contexto de un sistema de codificación adecuado, es posible considerar entonces cualquier función computable como una función cuyas entradas sean tuplas de enteros no negativos. No obstante, esto no quiere decir que toda función computable tenga la forma $f : \mathbb{N}^m \rightarrow \mathbb{N}^n$
donde m y n son enteros de $\mathbb{N}$. De hecho la función división definida por

$$\text{división}(x;\, y) = \text{la parte entera de } x/y, \text{ donde } x;\, y \in \mathbb{N} \text{ y } y \neq 0$$

cuyo dominio contiene pares de enteros, deberá estar en nuestra colección de funciones computables. Sin embargo, división no es una función de la forma $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ ya que no está definida para los pares donde la segunda componente es cero.

Para tener en cuenta las funciones cuyos dominios no incluyen todo $\mathbb{N}^m$ para un m dado, presentare el concepto de función parcial. Una función parcial de un conjunto X es aquella cuyo dominio constituye un subconjunto de X. Para indicar que el subconjunto es propio, es decir, que una función parcial de X efectivamente se encuentra indefinida por lo menos para un elemento de X, se refiere a ella como estrictamente parcial. Por otra parte, se llamara función total de X a una función parcial de X cuyo dominio es todo el conjunto X. De esta forma, tanto la función división definida anteriormente, como la función suma definida por $suma(x;\, y) = x + y$ son funciones

parciales de $\mathbb{N}^2$ en $\mathbb{N}$. Para mayor precision, suma es una función total en $\mathbb{N}^2$, mientras que división es una función parcial en $\mathbb{N}^2$.

Por tanto, aplicando los sistemas de codificación adecuados, es posible identificar cualquier función computable como una función parcial de la forma $f : \mathbb{N}^m \to \mathbb{N}^n$, para m y n en $\mathbb{N}$. Así pues, nuestra búsqueda de todas las funciones computables se puede restringir a las funciones parciales de $\mathbb{N}^m$ en $\mathbb{N}^n$, donde m y n estan en $\mathbb{N}$.

Por último, dado que con frecuencia haremos referencia a n-tuplas arbitrarias de enteros, es conveniente adoptar la sencilla notación $\bar{x}$ para representar una tupla de la forma $(x_1, x_2,, \dots, x_n,)$ en aquellos casos donde no se requieren los detalles de la forma expandida de dicha tupla.

Funciones iniciales. La base de la jerarquía de las funciones computables está constituida por el siguiente conjunto de funciones, que llamaremos funciones iniciales: Una de estas funciones es la función cero, representada por ζ. Esta función establece una correspondencia entre la cero-tupla o tupla vacía y el entero cero, y se escribe $\zeta\,() = 0$.

Así pues, ζ corresponde al proceso de escribir un cero en un trozo de papel en blanco o en una celda de memoria de un ordenador. Ambas tareas están de acuerdo con nuestro concepto intuitivo de proceso computacional, por lo que resulta sencillo aceptar que ζ es una función que debe clasificarse como computable.

Otra de las funciones iniciales es la función sucesor, representada por σ. Esta función establece una correspondencia entre tuplas de una sola componente, de tal manera que $\sigma(x) = x + 1$, para cada entero no negativo x. Dado que existe un proceso computacional para la suma de enteros, también debemos considerar a como una función computable.

La clase de funciones iniciales se completa con una colección de funciones conocidas como funciones de proyección. Cada una de estas funciones extrae como salida una componente especifica de su tupla de entrada. Para representar una función de proyección, utilizamos el símbolo π junto con un superíndice que indica el tamaño de la tupla de entrada y un subíndice que indica la componente extraída. Por ejemplo, $\pi_2^3(7;\ 6;\ 4) = 6$. Como caso especial, consideraremos π_0^n , que establece una correspondencia entre tuplas de tamaño n y la tupla vacía, por lo que, $\pi_0^2$0(6; 5) = ( ).

Cualquier función π_m^n podría calcularse aplicando el procedimiento de recorrer la n-tupla de entrada hasta localizar el elemento m-esimo y después extraer el valor entero allá encontrado. Así pues, al igual que con el resto de las funciones iniciales, es sencillo asumir que las proyecciones deben estar también en la clase de las funciones computables.

Las funciones iniciales forman la base de la jerarquía que existe en la teoría de funciones recursivas. Por supuesto, se trata de funciones que por solas no pueden lograr mucho. Por lo tanto, nuestro siguiente objetivo será examinar cómo pueden emplearse estas funciones para construir otras más complejas.

Funciones recursivas primitivas. Existen varias formas de construir funciones más complejas a partir de las iniciales, de las cuales consideraremos las siguientes:

La primera técnica es la denominada combinación. La combinación de dos funciones $f : \mathbb{N}^k \to \mathbb{N}^m y\, g : \mathbb{N}^k \to \mathbb{N}^n$ es la función $f \times g : \mathbb{N}^k \to \mathbb{N}^{m+n}$, la cual viene definida por $f \times g\,(\bar{x}) = (f(\bar{x});\, g(\bar{x}))$, donde $\bar{x}$ es una k-tupla. Es decir, la función $f \times g$ toma como entrada tuplas de tamaño k y produce como salida tuplas de tamaño $m + n$, donde las primeras m componentes se obtienen a partir de la salida de f y las n restantes a partir de la salida de g. Dado, $\pi_1^3 \times \pi_3^3$ 3 $(4;\ 6;\ 8) = (4;\ 8)$.

Suponiendo que existe una forma de calcular las funciones f y g, podemos calcular $f \times g$ calculando primero por separado f y g, y después combinando sus salidas. Llegamos entonces a la conclusión de que la combinación de funciones computables también es computable.

La composición representa otro método para formar funciones más complejas. La composición de dos funciones $f : \mathbb{N}^k \to \mathbb{N}^m\, y\, g : \mathbb{N}^m \to \mathbb{N}^n$ es la función $f : \mathbb{N}^k \to \mathbb{N}^n$ definida por $g \circ f\,(\bar{x}) = g(f(\bar{x}))$, donde x es una k-tupla. Por ejemplo, $\sigma \circ \zeta(\,) = \sigma(\zeta(\,)) = \sigma(0) = \mathbf{1}$.

Calculando primero f y utilizando su salida como entrada para g, si f y g son computables, se deduce que la composición de ambas es también computable.

La última técnica de construcción en esta etapa se denomina recursividad primitiva. Supongamos que al definir una función $f : \mathbb{N}^2 \to \mathbb{N}$ tal que $f(x, y)$ sea el numero de nodos en un árbol completo y equilibrado en el cual cada nodo que no es una hoja tiene exactamente x hijos, y cada camino desde la raíz a un nodo hoja contiene y arcos (se dice entonces que este *árbol tiene profundidad* y). El primer paso es observar que cada nivel del árbol tiene exactamente x^n nodos, donde n es el número de nivel.

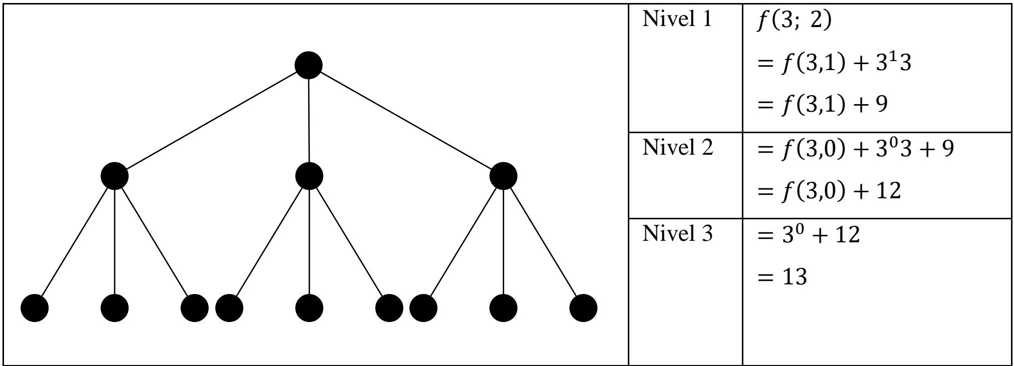
De esta forma, con un valor fijo de x, para determinar el número de nodos de un árbol con profundidad $y + 1$, basta con sumar $xy + 1 = xyx$ al número de nodos del árbol de profundidad y. Esto, unido al hecho de que un árbol con un solo nodo raíz contiene x^0 nodos, permite definir f recursivamente con el siguiente par de formulas:

$$f(x, 0) = x^0$$

$$f(x, y + 1) = f(x, y) + x^y x$$

Esta técnica es la aplicada inicialmente para identificar los niveles en que se ejecutara el algoritmo y de igual forma se identifican el proceso de recursividad entre MT embebidas .Ver figura 69

Al calcular $f(3;\ 2)$ tendremos:



Nivel 1	$f(3;\ 2)$ $= f(3,1) + 3^1 3$ $= f(3,1) + 9$
Nivel 2	$= f(3,0) + 3^0 3 + 9$ $= f(3,0) + 12$
Nivel 3	$= 3^0 + 12$ $= 13$

En un contexto más general, lo que hemos hecho es definir f en términos de otras dos funciones. Una de ellas es $g : \mathbb{N} \rightarrow \mathbb{N}$ tal que $g(x) = 1;\ \forall x \in \mathbb{N}$. Y la otra es $h : \mathbb{N}^3 \rightarrow \mathbb{N}$ tal que $h(x;\ y;\ z) = z + x^y x$. Al emplear estas dos funciones, f esta definida recursivamente por las formulas:

$$f(x,0) = g(x)$$
$$f(x,y + 1) = h(x,y,f(x,y))$$

En este caso diremos entonces que f esta construida a partir de g y de h por medio de recursividad primitiva. De manera aun más general, la recursividad primitiva es una técnica que permite construir una función f que establece la correspondencia entre $\mathbb{N}^{k+1}\ y\ \mathbb{N}^m$ a partir de otras dos funciones que relacionan $\mathbb{N}^k$ con $\mathbb{N}^m$ y $\mathbb{N}^{k+0+1}$ con $\mathbb{N}^m$, respectivamente, tal y como se muestra en las siguientes ecuaciones:

$$f(\bar{x},0) = g(\bar{x})$$
$$f(\bar{x},y + 1) = h(\bar{x};\ y;\ f(\bar{x};\ y))$$

donde $\bar{x}$ representa una k-tupla.

Empleando la recursividad primitiva junto con las funciones y operaciones que hemos presentado anteriormente, tenemos que la función suma se puede definir como:

$$suma(x,0) = x$$
$$suma(x,y + 1) = 1 + suma(x,y)$$

es decir:

$$suma(x, 0) = \pi_1^1(x)$$
$$suma(x, y + 1) = \sigma[\pi_3^3(x, y, suma(x; y))]$$

donde $g = \pi_1^1$ y $h = \sigma \circ \pi_3^3$

Una función construida mediante recursividad primitiva a partir de funciones computables es necesariamente computable. Más concretamente, si f está definida a partir de g y h, el papel de estas dos últimas funciones es el de permitir que $f(\bar{x}, y)$ se calcule en función de $f(\bar{x}, 0), f(\bar{x}, 1), f(\bar{x}, 2),$ hasta llegar a $f(\bar{x}, y)$. Por tanto, si g y h son computables, f también lo será.

En este momento estamos en condiciones de definir el siguiente escalón de la clasificación, el de la clase de las funciones recursivas primitivas. Esta clase está formada por las funciones que pueden construirse a partir de las funciones iniciales aplicando *un número finito de combinaciones*, *composiciones* y *recursividades primitivas*. Esta clase es muy extensa e incluye, si no todas, la mayoría de las funciones totales que se requieren en las aplicaciones de ordenador tradicionales.

Por último, debemos señalar que si f es una función recursiva primitiva, entonces f debe ser efectivamente total. De hecho, las funciones iniciales son totales, y las técnicas de construcción (combinación, composición y recursividad primitiva) producen funciones totales cuando se aplican a funciones totales. Por tanto, la clase de las funciones recursivas primitivas no completa nuestra jerarquía de funciones computables. La división es computable, pero no es recursiva primitiva ya que no es total. Además, veremos también que existen funciones totales computables que no son recursivas primitivas. Por consiguiente, se ampliara el repertorio de funciones recursivas primitivas, y después continuaremos la búsqueda de funciones computables más allá de esta clase.

7.4.2.2. Alcance de las funciones recursivas primitivas.

El primer objetivo es ampliar el repertorio de las funciones que sabemos que son recursivas primitivas, proporcionando una serie de casos útiles que suelen aparecer en las aplicaciones de ordenador tradicionales. Posteriormente, el segundo objetivo sería el de esclarecer la diferencia entre la clase de las funciones recursivas primitivas y la clase de las funciones totales computables.

Continuando con la investigación de las funciones recursivas primitivas considerando la colección de las funciones constantes, las cuales producen una salida fija predeterminada, sin importar cuál sea la entrada. Las funciones constantes cuyas salidas son tuplas de un solo elemento están representadas por K_m^n , donde n indica la dimensión del dominio y m es el valor de salida de la función. Así, K_3^2 establece la correspondencia entre

cualquier tupla de dos elementos y el entero 3. Cualquier función constante de la forma K_m^n es recursiva primitiva, ya que puede calcularse mediante la composición de las siguientes funciones iniciales: la proyección π_0^n, para la obtención de la tupla vacía a partir de cualquier n-tupla; la función ζ para la obtención de 0; y m aplicaciones de la función ζ para la obtención del valor m. Por ejemplo: $K_3^2\ (x_1;\ x_2) = \sigma \circ \sigma \circ \sigma \circ \zeta \circ \pi_0^2\ (x_1;\ x_2) = 3$

También son recursivas primitivas las funciones constantes cuyas salidas tienen más de una componente, ya que no son más que combinaciones de funciones de la forma K_m^n. La función que establece la correspondencia entre cualquier tupla de tamaño 3 y el par $(2;\ 5)$ es $K_3^2 \times\ K_5^3$.

Para evitar notaciones complicadas, en aquellos casos en los que la dimensión del dominio sea evidente o no tenga importancia para el análisis, representaremos las funciones constantes directamente con su valor de salida, es decir, 3 en lugar de K_3^2 o $(2;\ 5)$ en lugar de $K_3^2 \times\ K_5^3$.

Utilizando las funciones constantes y la función suma definida en la sección anterior, podemos mostrar de la manera siguiente que la función producto es también recursiva primitiva:

$$\text{producto } (x, 0)\ =\ 0$$

$$\text{producto } (x, y\ +\ 1)\ =\ x\ + \text{producto } (x, y)$$

es decir:

$$\text{producto}(x, 0)\ =\ K_0^1\ (x)$$

$$\text{producto}(x, y\ +\ 1)\ = \text{suma}[\pi_0^2 \times \pi_0^2(x, y \text{ producto}(x, y))]$$

donde $g\ =\ K_0^1 y\ h\ = \text{suma} \circ (\pi_0^2 \times \pi_0^2)$

De forma similar, podemos mostrar que la función potencia es también recursiva primitiva:

$$\text{potencia}(x, 0)\ =\ 1$$

$$\text{potencia}(x, y\ +\ 1)\ =\ x * \text{potencia}(x;\ y)$$

es decir:

$$\text{potencia}(x, 0)\ =\ K_1^1\ (x)$$

$$\text{potencia}(x, y\ +\ 1)\ = \text{producto}(x; \text{potencia}(x, y))$$

Obsérvese que $g\ =\ K_0^1$ y que realmente $h\ = \text{producto} \circ (\pi_1^3 \times \pi_3^3)$. Sin embargo, a veces usare notaciones menos formales, como producto $(x,$ potencia$(x, y))$ en este caso, con el fin de que determinadas expresiones sean más legibles.

La función predecesor establece la correspondencia entre tuplas de un solo elemento de tal forma que 0 corresponde a 0, y los números mayores que 0 se hacen corresponder con su predecesor en el orden natural de

N. Es decir, predecesor(1) $= 0$, predecesor(2) $= 1$, predecesor(3) $= 2$, etc. Para ver que predecesor es recursiva primitiva, observamos que puede definirse como:

predecesor(0) $= \zeta\,()$

predecesor($y + 1$) $= \pi_1^2(y$, predecesor $(y))$

En cierta forma, predecesor es la inversa de σ de hecho, puede utilizarse para definir la función de substracción propia, que llamaremos monus, de la misma manera que usamos para desarrollar la suma:

monus$(x, 0) = x$

monus$(x, y + 1)$ = predecesor$(monus(x;\ y))$

Así pues, $monus(x;\ y)$ es $x - y$ si $x \geq y$, y 0 en caso contrario.

Con frecuencia $monus(x, y)$ se representa como $x - y$. Otra tarea computacional muy común es determinar si dos valores son iguales. Este proceso esta modelado por la función igual, definida como:

$$\text{igual}(x;\ y) = \begin{cases} 1\ si\ x = y \\ 0\ si\ x \neq y \end{cases}$$

Se observa que la función igual es recursiva primitiva, ya que: $\text{igual}(x;\ y) = 1 - ((y - x) + (x - y))$

Dado el caso,

$$\begin{aligned} \text{igual}(5;\ 3) &= 1 - \big((3 - 5) + (5 - 3)\big) \\ &= 1 - (0 + 2) \\ &= 1 - 2 \\ &= \mathbf{0} \\ \text{igual}(5;\ 5) &= 1 - \big((5 - 5) + (5 - 5)\big) \\ &= 1 - (0 + 0) \\ &= 1 - 0 \\ &= \mathbf{1} \end{aligned}$$

también podemos “negar" cualquier tipo de función $f : \mathbb{N}^n \to \mathbb{N}$ para producir otra función : $\neg f : \mathbb{N}^n \to \mathbb{N}$ que sea 1 cuando f es 0 y 0 cuando f es 1, definiendo $\neg f$ como $monus \circ (K_0^1 \times f)$ es decir, $\neg f(x) = 1 - f(x)$. Por ejemplo:

$$\neg\ \text{igual}(x;\ y) = \begin{cases} \mathbf{1}\ si\ x \neq y \\ \mathbf{0}\ si\ x = y \end{cases}$$

Otra colección de funciones recursivas primitivas es aquella que comprende las que pueden definirse en una tabla en la cual se presenta explícitamente un número finito de entradas posibles junto con sus valores de salida correspondientes, y todas las demás entradas se asocian a un único valor común. A estas funciones se denominan *funciones tabulares*. Un caso sería la función f definida por:

$$f(x) = \{ \begin{array}{c} 3\ cuando\ x = 0 \\ 5\ cuando\ x = 4 \\ 2\ en\ los\ demas\ casos \end{array}$$

Cualquier función tabular es recursiva primitiva ya que puede expresarse como una suma finita de: una serie de productos de funciones constantes y funciones de igualdad, correspondientes a los casos de los valores de entrada citados explícitamente; y un último producto de una función constante y de una serie de funciones de igualdad negadas, correspondiente al caso de todas las demás entradas. Una descripción equivalente de la función f anterior es la siguiente:

$$f(x) = producto\big(3, igual(x,0)\big) + producto\big(5, igual(x,4)\big) + producto\big(2, producto(\neg igual(x,0), \neg igual(x,4)\big)$$

Como último caso, mostramos que la función cociente : $\mathbb{N}^2 \rightarrow \mathbb{N}$ definida por:

$$cociente(x,y) = \{ \begin{array}{c} la\ parte\ entera\ de\ x/y\ si\ y \neq 0 \\ 0\ si\ y = 0 \end{array}$$

es recursiva primitiva. Efectivamente, se trata de una versión total de la función parcial división.

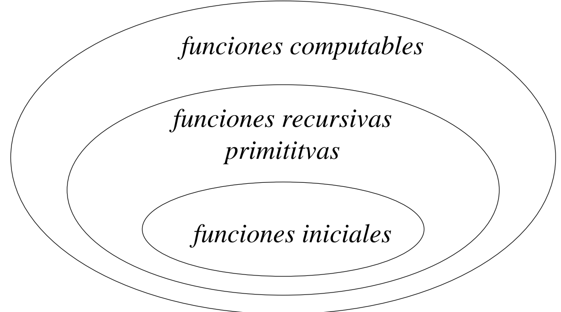
Aunque la recursividad primitiva se define formalmente empleando como índice la ultima componente de la tupla de entrada, el uso de proyecciones y combinaciones nos permite aplicar la recursividad en base a otros índices y permanecer en la clase de las funciones recursivas primitivas. Así pues, cociente es recursiva primitiva ya que puede definirse como:

$$cociente(0,y) = 0$$
$$cociente(x+1,y) = cociente(x,y) + igual(x+1, producto(cociente(x,y),y) + y)$$

Más allá de las funciones recursivas primitivas. Regresando ahora al objetivo principal de este tema, la jerarquía de funciones computables que hemos presentado hasta ahora, la cual se muestra en la figura 15. Se ha visto las funciones iniciales y como se pueden emplear estas funciones elementales para construir las funciones recursivas primitivas, todas las cuales son funciones totales computables. Sin embargo, tal y como se muestra en dicha figura 15, hemos dicho también que la clase de las funciones recursivas primitivas no abarca toda la colección de funciones computables, ya que:

a. por una parte, existen funciones computables que no son totales, como la función división,
b. y, por otra parte, existen funciones computables totales que no son recursivas primitivas, como la función de Ackermann.

Figura 14. Jerarquía provisional de las funciones computables



Fuente: Autor

Efectivamente, Ackermann presento en 1928 un ejemplo de función $A : \mathbb{N}^2 \rightarrow \mathbb{N}$ definida por las ecuaciones:

$$A(0, y) = y + 1$$
$$A(x + 1, 0) = A(x, 1)$$
$$A(x + 1, y + 1) = A(x, A(x + 1, y))$$

y demostró que dicha función es computable y total, pero no recursiva primitiva. No incluiremos esta demostración aquí, ya que lo importante es que en el análisis no se necesita una función computable que no sea recursiva primitiva. Lo único que necesita saber es que existe este tipo de funciones, tal y como enuncia el siguiente teorema: *El conjunto de funciones recursivas primitivas es un subconjunto propio del conjunto de funciones totales computables.*

Para demostrar este teorema, es suficiente con demostrar el siguiente enunciado equivalente: *Existe una función total* computable de $\mathbb{N}$ a $\mathbb{N}$ que no es recursiva primitiva.

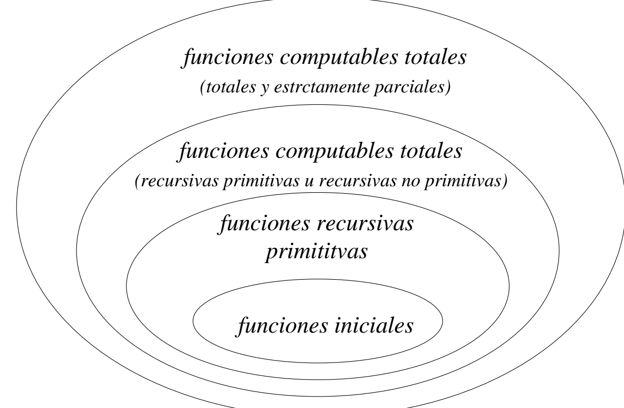
La demostración parte de la idea de que, con una codificación adecuada, es posible representar mediante una cadena finita de símbolos a toda función primitiva recursiva de $\mathbb{N}$ a $\mathbb{N}$ que se construya a partir de las funciones iniciales por medio de un número finito de combinaciones, composiciones y recursividades primitivas. Por tanto, es posible también asignar un orden a las funciones recursivas primitivas, colocando las codificaciones más cortas precediendo a las más largas y, dentro de las de igual longitud, respetando el orden alfabético de dichas codificaciones. (Brookshear, 1993)

En términos de este ordenamiento, podemos entonces hablar de la primera función recursiva primitiva (representada por f_0), de la segunda función (representada por f_1), de la tercera (representada por f_2), etc. Ahora definimos la $g : \mathbb{N} \rightarrow \mathbb{N}$ tal que $g(n) = f_n(n) + 1$, para todo $n \in \mathbb{N}$. Entonces g es total y computable, ya que puede calcularse como sigue: primero localizamos la función recursiva primitiva f_n, Después la aplicamos sobre n y finalmente sumamos 1.

Sin embargo, aplicando la *técnica de diagonalizacion*, deducimos que g no puede ser recursiva primitiva. Si lo fuera, g tendrá que ser igual a f_m para algún $m \in \mathbb{N}$, y entonces $g(m)$ sera $f_m(m)$, y no $f_m(m) + 1$ como indica la propia descripción de g. En consecuencia, g es total y computable, pero no recursiva primitiva.

Este teorema indica que la figura 15 se puede refinar para obtener la figura 16. En la siguiente sección examinaremos la parte de la figura 16 que cae fuera de las funciones computables totales.

Figura 15. Revisión de la jerarquía provisional mostrada en la figura 15



Fuente: Autor

7.4.2.3. Funciones recursivas parciales.

El análisis nos lleva hasta ahora a las clases de las funciones iniciales, las funciones recursivas primitivas y las funciones totales computables, todas las cuales son funciones totales. Ahora ampliaremos la explicación para incluir las funciones computables estrictamente parciales.

Para ampliar el análisis de las funciones computables mas allá de las funciones totales computables, aplicamos la técnica de construcción conocida como minimalización. Esta técnica nos permite construir una función $A : \mathbb{N}^2 \rightarrow \mathbb{N}$ a partir de otra función $g : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ declarando a $f(\bar{x})$ como la menor y en $\mathbb{N}$ tal que $g(\bar{x}; y) = 0$ y $g(\bar{x}; z)$ está definida para todos los enteros no negativos z menores que y. Esta construcción se representa con la notación:

$$f(\bar{x}) = \mu y\, [g(\bar{x};\, y) = 0]$$

Supongamos que $g(x, y)$ se define de acuerdo con los valores que aparecen en la tabla 1, y que $f(x)$ se define como $\mu y\,[g(\bar{x};\ y) = 0]$. Entonces, $f(0) = 4, f(1) = 2$, y $f(2)$ no está definido ya que, aunque 4 es el menor valor de y para el cual $g(2;\ y) = 0$, $g(2;\ z)$ no está definido para todos y cada uno de los valores de z menores que 4. En lo que se refiere al valor de $f(3)$, la tabla no nos proporciona suficiente información para determinar si se encuentra definido o no.

g(0; 0) = 2	g(1; 0) = 3	g(2; 0) = 8	g(3; 0) = 2	...
g(0; 1) = 3	g(1; 1) = 4	g(2; 1) = 3	g(3; 1) = 6	...
g(0; 2) = 1	g(1; 2) = 0	g(2; 2) = no definido	g(3; 2) = 7	...
g(0; 3) = 5	g(1; 3) = 2	g(2; 3) = 6	g(3; 3) = 2	...
g(0; 4) = 0	g(1; 4) = 0	g(2; 4) = 0	g(3; 4) = 8	...
g(0; 5) = 1	g(1; 5) = 0	g(2; 5) = 1	g(3; 5) = 4	...

Tabla 1. Algunos valores de una cierta función $g(x;\ y)$

Es importante recalcar que la minimalizacion puede producir funciones que no están definidas para ciertas entradas. Otro caso es la función $f : \mathbb{N} \rightarrow \mathbb{N}$ definida por $f(x) = \mu y[suma(x;\ y) = 0]$.

En este caso, $f(0) = 0$, pero no está definida para las demás entradas ya que, para $x > 0$, no existe ningún y en $\mathbb{N}$ tal que $x + y = 0$.

La función división $f : \mathbb{N} \rightarrow \mathbb{N}$ definida por:

$$\text{división}(x;\ y) = \{ \begin{matrix} la\ parte\ entera\ de\ x = y\ si\ y\ 6 = 0 \\ no\ definida\ si\ y = 0 \end{matrix}$$

que con la minimalizacion puede construirse como: división$(x;\ y) = \mu t[(x + 1) -$ (producto$(t, y) + y) = 0]$

Por otra parte, en algunos casos la minimalizacion produce funciones totales, como $f(x) = \mu y[monus(x;\ y) = 0]$, que no es otra cosa que la función identidad: el menor y tal que $x - y = 0$ es el propio x.

Hay que considerar la cuestión de si una función definida mediante la técnica de minimalización es o no computable. Si la función parcial g es computable, entonces se puede calcular $f(x) = y[g(\bar{x}, y) = 0]$ calculando los valores de $g(\bar{x}, 0)$, $g(\bar{x}, 1)$, $g(\bar{x}, 2)$, etc., hasta obtener el valor 0 o hasta llegar a un valor z para el cual $g(\bar{x};\ z)$ no esté definido. En el primer caso, el valor de $f(\bar{x})$ es el valor de y para el cual se encontró que $g(\bar{x}, y)$ era 0. En el segundo caso, $f(\bar{x})$ no está definido. Por tanto, el proceso de minimalización aplicado a una función computable produce otra función computable. Y además, aplicado a una función computable estrictamente parcial produce otra función computable también estrictamente parcial.

La jerarquía de las clases de funciones. La anterior observación nos permite entonces rebautizar la clase de las funciones computables mostrada en la figura 16. Esta clase seria denominada la clase de las funciones -recursivas o funciones recursivas parciales, tal y como vemos en la figura 17, la cual muestra ya la jerarquía completa de las clases de funciones según la teoría de funciones recursivas.

Figura 16. Jerarquía de las clases de funciones según la teoría de funciones recursivas

todas las funciones
(computables y no computables)
funciones μ-recursivas
o recursivas parciales
(totales o estrictamente parciales)
funciones computables totales
(recursivas primitivas y recursivas no primitivas)
funciones recursivas
primititvas
funciones iniciales

Fuente: Autor

Para ser más precisos, diremos entonces que la clase de funciones recursivas parciales es la clase de las funciones que pueden construirse a partir de las funciones iniciales aplicando un número finito de combinaciones, composiciones, recursividades primitivas y minimalizaciones.

Se subraya que el termino parcial no quiere decir que todas las funciones de la clase de las funciones recursivas parciales sean estrictamente parciales: se ha visto que el propio proceso de minimalizacion puede a veces producir funciones totales y, de hecho, la clase de las funciones recursivas parciales incluye a la clase de todas las funciones computables totales. Eso es, en este contexto, funciones tales como la función división, por ser estrictamente parciales, están dentro de la clase de las funciones recursivas parciales, pero fuera de clase la de las funciones computables totales.

Equivalencia de las tesis de Turing y de Church. Con la presentación de las funciones μ-recursivas o recursivas parciales llegamos al final de nuestra jerarquía de *funciones computables*. En este punto determino el uso de las maquinas de Turing con funciones recursivas primitivas iniciales para luego construir una más complejas.
Al igual que la tesis de Turing propone que la clase de las máquinas de Turing posee el poder computacional de cualquier sistema de computación, se plantea también la hipótesis de que la clase de las funciones recursivas

parciales contiene todas las funciones computables. Este último argumento se conoce como la tesis de Church, aunque Church la propuso originalmente en un contexto algo distinto. Un hecho que apoya la tesis de Church es que nadie ha podido demostrar que sea falsa, es decir, nadie ha encontrado una función parcial que sea computable pero no recursiva parcial. No obstante, un razonamiento aun más convincente es que la tesis de Church y la tesis de Turing representan lo mismo, tal y como enuncian los dos siguientes teoremas:

a. Toda función recursiva parcial es computable por una máquina de Turing.
b. Todo proceso computacional realizado por una máquina de Turing es en realidad el cálculo de una función recursiva parcial.

En resumen, las funciones μ -recursivas o recursivas parciales son computables por máquinas de Turing y que cualquier máquina de Turing no hace más que calcular una función recursiva parcial. Esto significa que las tesis de Turing y de Church son equivalentes aunque se muestren en contextos distintos. Con frecuencia, el concepto común que subyace en ambas se conoce con el nombre de tesis de Church-Turing en lugar de tesis de Church o tesis de Turing. Sin embargo, la equivalencia de estas hipótesis es más importante que la mezcla de la terminología, pues implica que hemos llegado al mismo limite aparente para el poder de los procesos computacionales siguiendo dos enfoques distintos, uno *operacional* y otro *funcional*, lo cual refuerza nuestra confianza en estas conjeturas. Se desarrolla siguiendo el flujo de proceso físico de un(os) productos en la planta de producción bajo tecnologías operacionales para enlazarla con tecnologías de información. (ver figura 6.) y por eso el objetivo de representar la misma planta de producción como un sistema lógico proposicional, para que pueda ser computable las operaciones.

8. DISEÑO E IMPLEMENTACION

8.1. Problema de secuenciación con Satisfacibilidad Booleana SAT

Una vez, catalogado el problema de secuenciación (desde la matemática discreta y combinatoria como un problema de enumeración), sea planteado como un problema de decisión, basado en lógica proposicional de primer orden, y declarando que se efectué con las actuales practicas modernas de la programación declarativa (*con programación lógica*), sea expuesto desde la teoría de la complejidad computacional como un problema de la *satisfacibilidad boolenana* que fue el primer problema identificado como perteneciente a la clase de complejidad NP-completo

Al presentar ahora el primer problema NP-completo, se demostró que este problema (si una expresión booleana puede satisfacerse) es NP-completo reduciendo explícitamente el lenguaje de cualquier MT no determinista que opera en tiempo polinómico al problema de la satisfacibilidad. En la teoría de la *complejidad computacional*, la clase de complejidad NP-completo es el subconjunto de los *problemas de decisión* en NP tal que todo problema en NP se puede *reducir* en cada uno de los problemas de NP-completo. De este modo; se concibe dentro de la metodología los pasos para deducir el planteamiento para obtener los literales que expresan una el problema de secuenciación y satisfacer la expresión lógica.

El término reducción se utilizado en el sentido transformar las instancias de un problema en instancias de otro (*reducciones many-one).* Otro tipo de reducción consiste en la reducción en *tiempo polinómico de Turing*. El problema secuenciación sea reducible en tiempo polinómico de Turing si dada una función que resuelve en tiempo polinómico, podría escribirse un programa que llamando a la subrutina anterior lo resuelva en tiempo polinómico. Esto contrasta con el uso del término reducción del que hablábamos al principio ya que este tiene la restricción de que el programa solamente puede llamar una vez al subalgoritmo y el valor retornado por este debe ser el valor de retorno del programa.

Si se definen el análogo a NP-completo con reducciones de Turing en lugar de reducciones many-one, el conjunto de problemas resultante no sería menor de NP-completo, de hecho se cuestiona si serían más grandes. Si los dos conceptos fuesen lo mismo, se seguiría que NO = Co-NP. Esto se mantiene porque por definición las clases de los problemas NP-completos y co-NP-completos bajo las *reducciones de Turing* son las mismas gracias a que las clases definidas con reducciones many-one son subclases de estas mismas. Por lo tanto si ambas definiciones de la NP-completitud son iguales hay un problema co-NP-completo (bajo ambas definiciones) como por ejemplo el complementario del problema de la satisfacibilidad booleana que es también NP-completo (bajo ambas definiciones).

Basado en estas afirmaciones, la metodología se encamina inicialmente a llevar el problema de programación de operaciones como un problema de satisfacibilidad boolenana, a través de expresiones lógicas para ser un problema de decisión, que pueda ser reducido y tratable; para que sea pueda ser decidible, y obtener una posible secuencia que represente una posible solución que sea confiable en un corto plazo de tiempo.

Es un problema NP-completo demostrando que el lenguaje de todas las MTN que funcionan en tiempo polinómico poseen una reducción en tiempo polinómico al problema SAT. Sin embargo, una vez que dispongamos de algunos problemas NP-completos, podremos demostrar que un nuevo problema es NP-completo reduciéndolo a alguno de los problemas NP-completos conocidos, empleando para ello una reducción en tiempo polinómico. El siguiente teorema demuestra por qué una reducción prueba que dicho problema es NP-completo.

S. Cook, en su artículo original sobre el tema, definía un problema P como NP-completo si, dado un oráculo para el problema P, es decir, un mecanismo que en una unidad de tiempo respondiera a cualquier pregunta sobre la pertenencia de una cadena dada a P, sería posible reconocer cualquier lenguaje de NP en tiempo polinómico. Este tipo de problemas NP-completos se conoce como completitud de Cook. En cierto sentido, la completitud de Karp es un caso especial en el que sólo se le plantea una pregunta al oráculo. Sin embargo, la completitud de Cook también permite la completamentación de la respuesta; por ejemplo, se puede plantear al oráculo una pregunta y luego responder lo contrario de lo que el oráculo dice. Una consecuencia de la definición de Cook es que los complementarios de los problemas NP-completos también son problemas NP-completos. Utilizando el concepto más restringido de completitud de Karp, podemos establecer una importante distinción entre los problemas NP-completos (en el sentido de Karp) y sus complementarios.

El problema de la satisfacibilidad. Las expresiones booleanas se construyen a partir de:

1. Variables cuyos valores son booleanos; es decir, toman el valor 1 (*verdadero*) o el valor 0 (*fals*o).
2. Los operadores binarios $\wedge$ y $\vee$, que representan las operaciones lógicas y y o de dos expresiones.
3. El operador unario $\neg$ que presenta la negación lógica.
4. Paréntesis para agrupar los operadores y los operandos, si fuera necesario para modificar la precedencia predeterminada de los operadores: $\neg$ es el operador de mayor precedencia, le sigue $\wedge$ y finalmente$\vee$.

Una asignación de verdad para una expresión booleana dada E asigna el valor verdadero o falso a cada de las variables que aparecen en E. El valor de la expresión E para una asignación de verdad T se designa como $E(T)$, y es el resultado de evaluar E reemplazando cada variable x por el valor $T(x)$ (verdadero o falso) que T asigna a x. Una asignación de verdad T satisface la expresión booleana E si $E(T) = 1$; es decir, la asignación de verdad T hace que la expresión E sea verdadera. Se dice que una expresión booleana E es satisfacible si existe al menos una asignación de verdad T que satisface E.

El problema de la satisfacibilidad es: Dada una expresión booleana, ¿es satisfacible?
Generalmente, se hará referencia al problema de la satisfisfacibilidad con la abreviatura SAT. Definido como un lenguaje, el problema SAT es el conjunto de expresiones booleanas (codificadas) que son satisfacibles. Las cadenas que no son ni códigos válidos para una expresión booleana ni son códigos de una expresión booleana insatisfacible no son problemas SAT.

Representación de problemas SAT. Los símbolos que se emplean en una expresión booleana son ∧, ∨, ¬, los paréntesis de apertura y cierre, y los símbolos que representan variables. La satisfacibilidad de una expresión no depende de los nombres de las variables, sólo de si dos apariciones de las variables corresponden a la misma variable o a variables diferentes.
Por tanto, podemos suponer que las variables son $x_1, x_2 \dots$, aunque en los ejemplos continuaremos empleando nombres de variable como y o z, así como x. También supondremos que las variables se renumeran con el fin de utilizar los subíndices más bajos posibles. Por ejemplo, no utilizaremos x_5 a menos que también empleemos x_1 hasta x_4 en la misma expresión.
Dado que existe un número infinito de símbolos que en principio pueden aparecer en una expresión booleana, nos encontramos con el familiar problema de disponer de un código con un alfabeto finito y fijo para representar expresiones con una cantidad arbitraria de variables. Sólo entonces podremos hablar del SAT como de un *problema*, es decir, como un lenguaje con un alfabeto fijo que consta de los códigos de aquellas expresiones booleanas que son satisfacibles. El código que utilizaremos es el siguiente:

1. Los símbolos ∧, ∨, ¬ , y ( ) se representan tal como están.
2. La variable x_i se representa mediante el símbolo i seguido de ceros y unos que representan i en binario.

Por tanto, el alfabeto del problema/lenguaje SAT está compuesto sólo por ocho símbolos. Todos los casos de SAT son cadenas de este alfabeto finito y fijo.

La forma de obtener la expresión de las clausulas de Horn para el problema de secuenciación, es identificando que recursos están relacionados a cada uno de los centros de trabajo, asociándolos en bloques (la división entre ellos) para identificar de forma individual que recursos son los asignados a cada producto en la medida que son requeridos para su transformación durante el proceso donde sea necesario el paso por un centro de trabajo C_{ij} determinado.
En cierta forma las clausulas están definidas por los tres recursos nombrados (*mano de obra, material* y *equipo*), dando lugar a que el problema de satisfacibilidad 3SAT, sea definido por cada centro de trabajo. Se resalta, que los literales que forman las clausulas, están identificados los recursos de forma individual a través de variables

lógicas definidas de la siguiente manera $I_1, I_2, I_3, I_4, I_5, I_6$ ...; M_1, M_2, M_3, M_4 ... y L_1, L_2, L_3 como se describe en la tabla 8.

El problema SAT es NP-Completo. Ahora se relaciona la demostración del *Teorema de Cook*, el hecho de que el problema SAT es NP-completo. Al demostrar que un problema es NP-completo, se necesita demostrar en primer lugar que está en NP. A continuación hay que demostrar que todo lenguaje de NP se reduce al problema en cuestión. En general, demostramos la segunda parte ofreciendo una reducción en tiempo polinómico de algún otro problema NP-completo, e invocando después el teorema. Pero ahora no conocemos otros problemas NP-completos a los que poder reducir el problema SAT. Por tanto, la única estrategia de la que disponemos es la de reducir todo problema de NP a SAT.

(Teorema de Cook) SAT es NP-completo. La primera parte de la demostración consiste en demostrar que SAT pertenece a NP. Esta parte es fácil:

1. Se utiliza la capacidad no determinista de una MTN para conjeturar una asignación de verdad T para la expresión dada E. Si la E codificada tiene longitud n, entonces un tiempo $O(n)$ basta en una MTN de varias cintas. Observe que esta MTN tiene muchas opciones de movimiento y puede tener tantas configuraciones diferentes como 2^n al final del proceso, donde cada camino representa la conjetura correspondiente a cada distinta asignación de verdad.
2. Al evaluar E para la asignación de verdad T. Si $E(T) = 1$, entonces acepta. Observe que esta parte es determinista. El hecho de que otros caminos de la MTN no lleven a la aceptación no tiene consecuencias, ya que incluso aunque sólo se satisfaga una asignación de verdad, la MTN acepta.

La evaluación puede llevarse a cabo fácilmente en un tiempo $O(n^2)$ sobre una MTN de varias cintas. Por tanto, el reconocimiento del problema SAT por parte de una MTN de varias cintas tarda un tiempo $O(n^2)$. La conversión a una MTN de una sola cinta puede elevar al cuadrado el tiempo, por lo que un tiempo $O(n^4)$es suficiente en una MTN de una sola cinta.

Ahora tenemos que demostrar la parte difícil: si L es cualquier lenguaje perteneciente a NP, entonces existe una reducción en tiempo polinómico de L a SAT. Podemos suponer que existe alguna MTN de una sola cinta M y un polinomio $p(n)$ tal que M no emplea más de $p(n)$ pasos para una entrada de longitud n, a lo largo de cualquier camino. Además, las restricciones del Teorema, que hemos demostrado para las MTD, pueden demostrarse de la misma manera para las MTN. Por tanto, podemos suponer que M nunca escribe un espacio en blanco y que nunca mueve su cabeza a la izquierda de la posición inicial de la misma.

Por tanto, si M acepta una entrada w y $|w| = n$, entonces existe una secuencia de movimiento de M tal que:

1. α_0 es la configuración inicial de M para la entrada w.
2. $\alpha_0 \ \alpha_0 \ \cdots \alpha_k$, donde $k \leq p(n)$.

3. α_k es una configuración con un estado de aceptación.
4. Cada α_i está formada sólo por símbolos distintos del espacio en blanco (excepto si α_i termina en un estado y en un espacio en blanco) y se extiende desde la posición inicial de la cabeza (el símbolo de entrada más a la izquierda) hacia la derecha.

Esta estrategia puede resumirse como sigue.

a) Cada α_i puede escribirse como una secuencia de símbolos $X_i0X_i1 \cdots X_i, p(n)$. Uno de estos símbolos es un estado, y los restantes son símbolos de cinta. Como siempre, suponemos que los estados y los símbolos de cinta son disjuntos, por lo que podemos decir que X_{ij} es el estado y, por tanto, dónde se encuentra la cabeza de la cinta. Observe que no existe ninguna razón para representar los símbolos a la derecha de los $p(n)$ primeros símbolos de la cinta [que junto con el estado definen una configuración de longitud $p(n) + 1$], porque no pueden influir en un movimiento de M si está garantizado que M se para después de $p(n)$ movimientos o menos.

b) Para describir la secuencia de configuraciones en función de variables booleanas, creamos la variable $y_{ij}A$ para representar la proposición de que $X_{ij} = A$. Aquí, i y j son cada uno de los enteros pertenecientes al intervalo de 0 a $p(n)$, y A es cualquier símbolo de cinta o un estado.

c) Expresamos la condición de que la secuencia de las configuraciones representa la aceptación de una entrada w escribiendo una expresión booleana que es satisfacible si y sólo si M acepta w mediante una secuencia de, como máximo, $p(n)$ movimientos. La asignación que satisface será aquella que *diga la verdad* sobre las configuraciones; es decir, $y_{ij}A$ será verdadera si y sólo si $X_{ij} = A$.

Para garantizar que la reducción en tiempo polinómico de $L(M)$ a SAT es correcta, escribimos esta expresión de modo que representa el cálculo:

i. Inicio correcto. Es decir, la configuración inicial es q0w seguida por espacios en blanco.
ii. El siguiente movimiento es correcto (es decir, el movimiento sigue correctamente las reglas de la MT). Entonces cada configuración subsiguiente procede de la anterior gracias a uno de los posibles movimientos válidos de M.
iii. Terminación correcta. Es decir, existe alguna configuración que es un estado de aceptación.

Antes de poder construir una expresión booleana precisa debemos comentar algunos detalles importantes.

En primer lugar, hemos especificado que las configuraciones terminan cuando comienza la cola infinita de espacios en blanco. Sin embargo, es más conveniente cuando se simula un cálculo en tiempo polinómico pensar que todas las configuraciones tienen la misma configuración $p(n) + 1$. Por tanto, podemos tener una cola de espacios en blanco en una configuración.

En segundo lugar, es mejor suponer que todos los cálculos duran exactamente $p(n)$ movimientos [y por tanto tienen $p(n) + 1$ configuraciones], incluso aunque la aceptación tenga lugar antes. Así, conseguimos que cada configuración con un estado de aceptación sea su propio sucesor. Es decir, si α tiene un estado de aceptación, permitimos un *movimiento* α α. Por tanto, podemos suponer que si existe un cálculo de aceptación, entonces $\alpha p(n)$ tendrá una configuración de aceptación y es todo lo que tendremos que comprobar para asegurar la condición *terminación correcta*.

Ahora proporcionamos un algoritmo para construir a partir de M y w una expresión booleana EM_w. La forma general de EM_w es $U \wedge S \wedge N \wedge F$, donde S, N y F son expresiones que establecen que M se inicia, mueve y termina correctamente, y U indica que sólo existe un símbolo en cada celda.

Conclusión de la demostración del Teorema de Cook. Aunque hemos descrito la construcción de la expresión: $EM_w = U \wedge S \wedge N \wedge F$ como una función tanto de M como de w, el hecho es que sólo el *inicio correcto* de la parte S depende de w de una manera muy sencilla (w está en la cinta de la configuración inicial). Las otras partes, N y F, dependen sólo de M y n, la longitud de w.

Por tanto, para cualquier MTN M que opere en un tiempo polinómico $p(n)$, podemos diseñar un algoritmo que tome una entrada w de longitud n, y generar EM_w. El tiempo de ejecución de este algoritmo en una MT determinista de varias cintas es $Op2(n)$, y dicha MT de varias cintas puede convertirse en una MT de una sola cinta que opere en un tiempo O.

$p4(n)$. La salida de este algoritmo es una expresión booleana EM_w que es satisfacible si y sólo si M acepta w en como máximo $p(n)$, movimientos.

Para resaltar la importancia del Teorema de Cook, veamos cómo se le aplica el Teorema. Suponga que disponemos de una MT determinista que reconoce en tiempo polinómico, por ejemplo en un tiempo $q(n)$, casos del problema SAT. Entonces, todo lenguaje aceptado por una MTN M que acepta en un tiempo polinómico $p(n)$ sería aceptado en un tiempo polinómico determinista por la MTD cuyo modo de operación se muestra en la Figura. La entrada w a M se convierte en una expresión booleana EM_w. Esta expresión se aplica al comprobador de SAT y lo que éste responda acerca de EM_w, nuestro algoritmo lo responde sobre w.

Problema de la satisfacibilidad restringido. Ahora se demuestra que en una amplia variedad de problemas, como el problema del viajante de comercio, son NP-completos. En principio, se hará buscando reducciones en tiempo polinómico del problema SAT al problemas de secuenciación que es el interés de la investigación. Sin embargo, existe un importante problema intermedio, conocido como 3SAT, que es mucho más fácil de reducir a problemas típicos que SAT.

3SAT también es un problema sobre la satisfacibilidad de expresiones booleanas, pero estas expresiones tienen una forma regular: están compuestas por operaciones lógicas Y de *cláusulas*, siendo cada una de ellas la operación lógica o de exactamente tres variables negadas o no.

En esta parte se presenta una terminología importante sobre las expresiones booleanas. Se reduce entonces la satisfacibilidad de cualquier expresión a la satisfacibilidad de expresiones en forma normal correspondientes al problema 3SAT. Es interesante observar que, aunque toda expresión booleana E tiene una expresión equivalente F en la forma normal de 3SAT, el tamaño de F puede ser exponencial repecto al tamaño de E. Por tanto, la reducción en tiempo polinómico de SAT a 3SAT tiene que ser más sutil que una simple manipulación con álgebra booleana. Necesitamos convertir cada expresión E de SAT en otra expresión F en la forma normal correspondiente a 3SAT. Aún así no necesariamente F es equivalente a E. Sólo podemos garantizar que F es satisfacible si y sólo si E lo es.

FN Formas normales de las expresiones booleanas optenidas. Las siguientes definiciones son fundamentales: Un literal es cualquier variable, o cualquier variable negada. Ejemplos de literales son x y $\neg y$. Para ahorrar espacio, a menudo utilizaremos la notación y' en lugar de $\neg y$. Una cláusula es la operación lógica O aplicada a uno o más literales. Algunos ejemplos son: $x, x \lor y\, y\, x \lor y \lor z$.

Una expresión booleana se dice que está en forma normal conjuntiva 3 o FNC si es un $\land$ lógico de cláusulas.

Para reducir aún más las expresiones, adoptaremos la notación alternativa en la que $\lor$ se trata como una suma, utilizando el operador $+$ y $\land$ se trata como un producto. Para los productos, normalmente utilizamos la yuxtaposición, es decir, ningún operador, al igual que en el caso de la concatenación en las expresiones regulares.

También es habitual hacer referencia a una cláusula como una *suma de literales* y a una expresión FNC como un *producto de clásulas.*

Conversión de expresiones a la FNC. Se dice que dos expresiones booleanas son equivalentes si proporcionan el mismo resultado para cualquier asignación de verdad de sus variables. Si dos expresiones son equivalentes, entonces o bien ambas son satisfacibles o no lo son. Por tanto, convertir expresiones arbitrarias en expresiones en FNC equivalentes es un método prometedor para desarrollar una reducción en tiempo polinómico del problema SAT al CSAT. Dicha reducción demostraría que CSAT es NP-completo.

Sin embargo, las cosas no son tan simples. Aunque podamos convertir cualquier expresión a la forma normal conjuntiva, la conversión puede tardar un tiempo mayor que el tiempo polinómico. En particular, puede producir una expresión de longitud exponencial, por lo que requerirá un tiempo exponencial para generar la salida.

Afortunadamente, la conversión de una expresión booleana arbitraria en una expresión en la forma FNC es sólo un método para reducir el problema SAT a CSAT, y demostrar por tanto que CSAT es NP-completo. Todo lo que tenemos que hacer es tomar un caso del problema SAT E y convertirlo en un caso de CSAT F, tal que F sea

satisfacible si y sólo si E lo es. No es necesario que E y F sean equivalentes. Ni siquiera es necesario que E y F tengan el mismo conjunto de variables y, de hecho, generalmente F tendrá un superconjunto de las variables de E.

La reducción de SAT a CSAT constará de dos partes. Primero empujaremos todos los ¬ hacia abajo del árbol de la expresión de modo que sólo queden negaciones de variables; es decir, la expresión booleana se transforma en una serie de literales relacionados mediante los operadores lógicos ∧ y ∨ Esta transformación produce una expresión equivalente y tarda un tiempo que es como máximo un tiempo cuadrático respecto al tamaño de la expresión. Una computadora convencional, con una estructura de datos cuidadosamente diseñada, sólo tarda un tiempo lineal.

El segundo paso consiste en escribir la expresión que hemos obtenido como un producto de cláusulas; es decir, en forma FNC. Introduciendo nuevas variables, podemos realizar esta transformación en un tiempo que es polinómico respecto al tamaño de la expresión dada. En general, la nueva expresión F no será equivalente a la antigua expresión E. Sin embargo, F será satisfacible si y sólo si E lo es. Más específicamente, si T es una asignación de verdad que hace que E sea verdadera, entonces existe una extensión de T, por ejemplo S, que hace que F sea verdadera; decimos que S es una extensión de T si S asigna el mismo valor que T a cada variable a la que asigna T, pero S también puede asignar un valor a las variables que no están en T.

El primer paso consiste en aplicar los operadores ¬ a los operadores ∧ y ∨. Las reglas que necesitamos son:

1. $\neg(E \wedge F) \Rightarrow \neg(E) \vee \neg(F)$. Esta regla, que es una de las leyes de DeMorgan, nos permite aplicar ¬ a cada operando del operador ∧. Observe que como efecto colateral, ∧ se convierte en ∨.
2. $\neg(E \vee F) \Rightarrow \neg(E) \wedge \neg(F)$. La otra ley de DeMorgan aplica el operador ¬ a los operandos de ∨. El efecto colateral en este caso es que ∨ se convierte en ∧.
3. $\neg\neg(E) \Rightarrow E$. Esta ley de doble negación cancela un par de operadores ¬ que se aplican a la misma expresión.

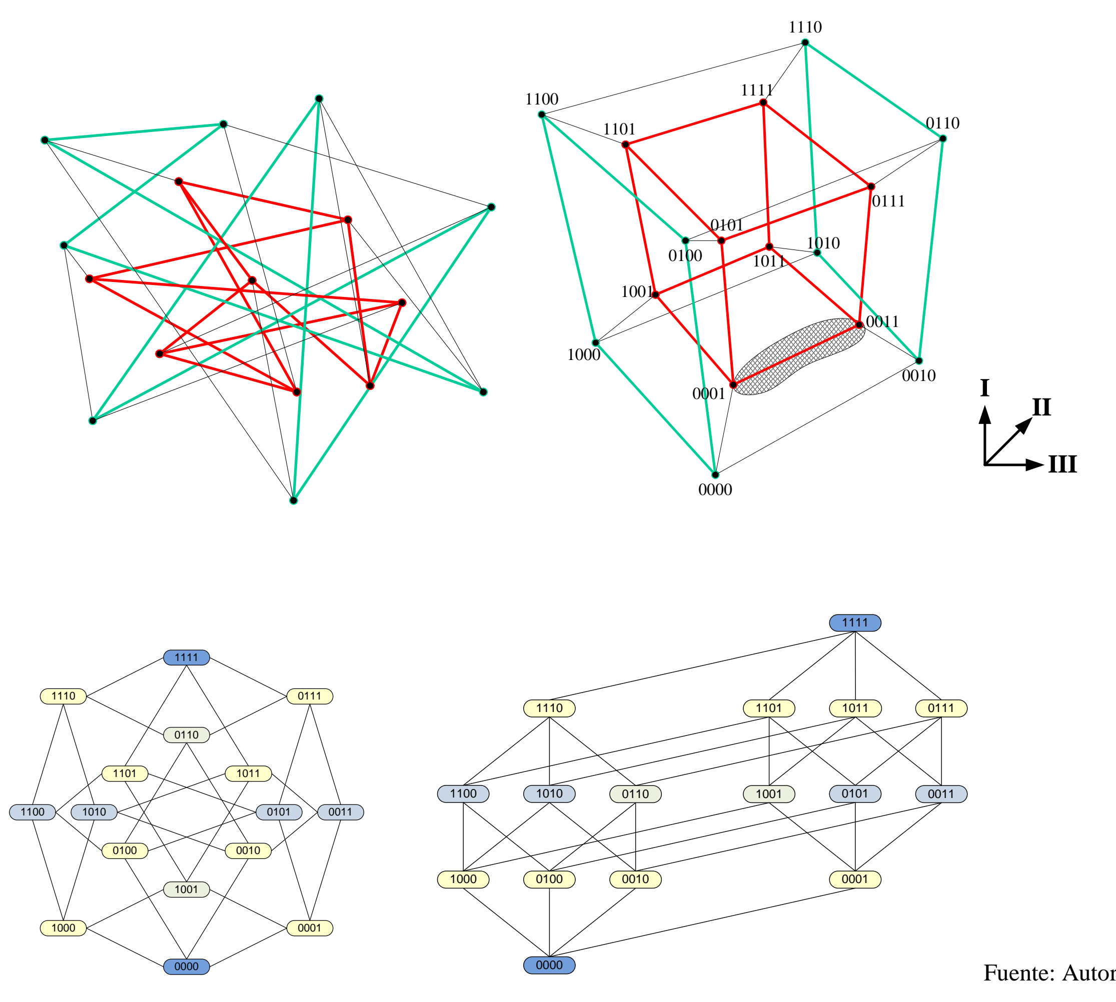
Descripciones de algoritmos. Aunque formalmente el tiempo de ejecución de una reducción es el tiempo que tarda en ejecutarse en una máquina de Turing de una sola cinta, estos algoritmos son innecesariamente complejos. Sabemos que los conjuntos de problemas que se pueden resolver en computadoras convencionales, en las MT de varias cintas y en las MT de una sola cinta en un tiempo polinómico son los mismos, aunque los grados de los polinomios pueden ser diferentes. Por tanto, cuando describamos algunos algoritmos bastante sofisticados para reducir un problema NP-completo a otro, mediremos los tiempos mediante implementaciones eficientes en una computadora convencional. Esto nos permitirá evitar los detalles que dependen de la manipulación de las cintas y nos permitirá centrarnos en las ideas importantes del algoritmo.

Los algoritmos asociados se conocen como *algoritmos paralelos*. Muchos problemas se resuelven con mayor rapidez usando computadoras paralelas en lugar de seriales. Se analizará un modelo de computación paralela de

máquinas de estados finitos conocido como el cubo-n o *hipercubo*. El cubo- n tiene 2^n procesadores, $n \geq 1$, que se representan por vértices (figura) etiquetados $0, 1, \ldots, 2^n - 1$. El cubo- n puede representar las combinaciones de la tabla de verdad para la cantidad de variables a considerar. Cada máquina de Turing lo efectúa en la unidad de control en cada uno de los niveles I, II y III. Ver figura).

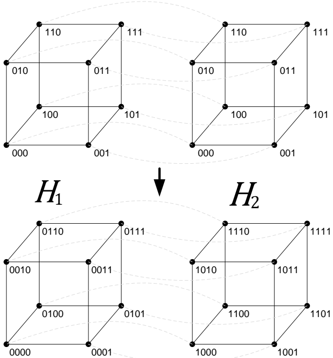
Una arista conecta a dos vértices si la representación binaria de sus etiquetas difiere exactamente en un bit. Durante una unidad de tiempo, todos los procesadores en el cubo-n pueden ejecutar una instrucción simultáneamente y después comunicarse con el procesador adyacente. Si un procesador necesita comunicarse con uno no adyacente, el primer procesador envía un mensaje que incluye la ruta al receptor, y el destino final del mismo. Puede tomar varias unidades de tiempo que un procesador se comunique con otro no adyacente.

Combinación de dos **2**-cubos 3-SAT para obtener la tabla de verdad

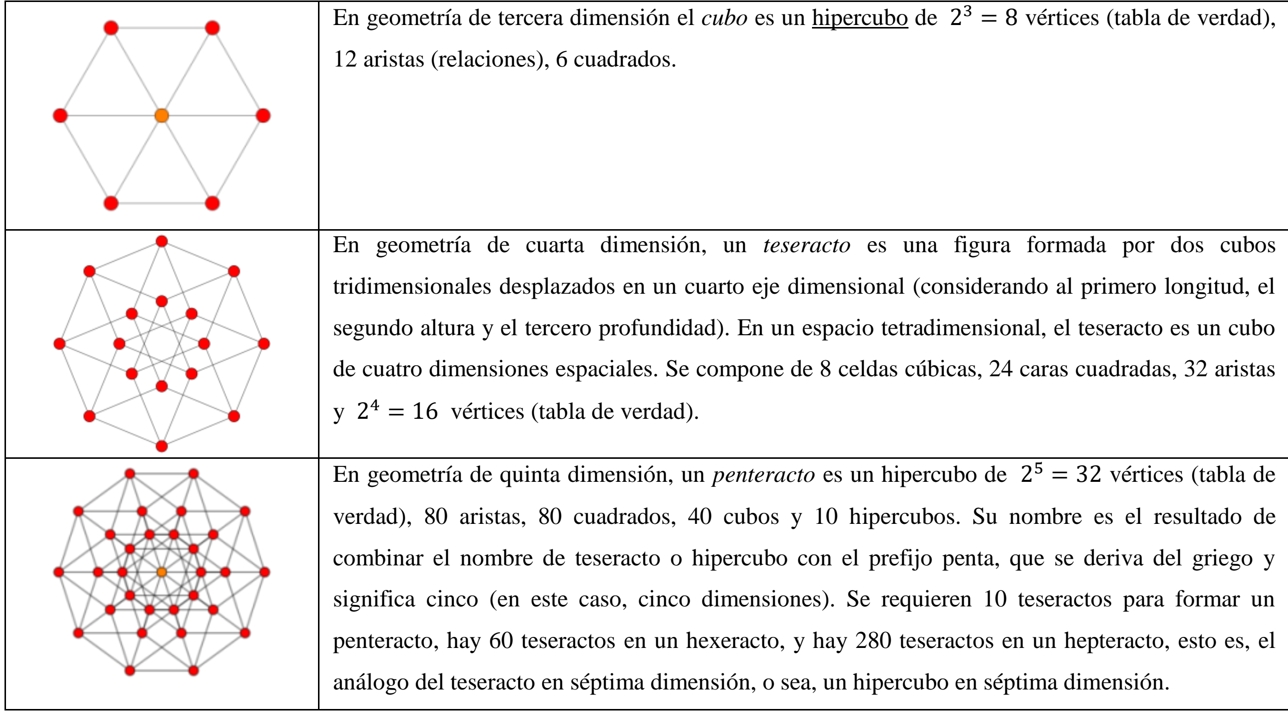


Fuente: Autor

También es posible describir el cubo- n de manera recursiva. El cubo-1 tiene dos procesadores, etiquetados $\mathbf{0}$ y 1, y una arista. Sean H_1 y H_2 dos cubos-($n-1$) cuyos vértices se etiquetan en binarios $0, 1, \ldots, 2^{n-1} - 1$. Se coloca una arista entre cada par de vértices, una desde H_1 y otra desde H_2, siempre que los vértices tengan etiquetas idénticas. Se cambia la etiqueta L en cada vértice de H_1 a $0L$ y la etiqueta L en cada vértice de H_2 se cambia a $1L$. Se obtiene un cubo- n.



El cubo-n es un modelo importante de computación porque se han construido varias máquinas de este tipo y están trabajando. Todavía más, algunos otros modelos de computación paralela se pueden simular con el *hipercubo*.



	En geometría de tercera dimensión el *cubo* es un hipercubo de $2^3 = 8$ vértices (tabla de verdad), 12 aristas (relaciones), 6 cuadrados.
	En geometría de cuarta dimensión, un *teseracto* es una figura formada por dos cubos tridimensionales desplazados en un cuarto eje dimensional (considerando al primero longitud, el segundo altura y el tercero profundidad). En un espacio tetradimensional, el teseracto es un cubo de cuatro dimensiones espaciales. Se compone de 8 celdas cúbicas, 24 caras cuadradas, 32 aristas y $2^4 = 16$ vértices (tabla de verdad).
	En geometría de quinta dimensión, un *penteracto* es un hipercubo de $2^5 = 32$ vértices (tabla de verdad), 80 aristas, 80 cuadrados, 40 cubos y 10 hipercubos. Su nombre es el resultado de combinar el nombre de teseracto o hipercubo con el prefijo penta, que se deriva del griego y significa cinco (en este caso, cinco dimensiones). Se requieren 10 teseractos para formar un penteracto, hay 60 teseractos en un hexeracto, y hay 280 teseractos en un hepteracto, esto es, el análogo del teseracto en séptima dimensión, o sea, un hipercubo en séptima dimensión.

	En geometría de sexta dimensión, un *hexeracto* es el nombre de un miembro de la familia de los hipercubos, con $2^6 = 64$ vértices (tabla de verdad), 192 aristas, 240 cuadrados, 160 cubos, 60 teseractos, así como 12 penteractos. Su nombre es el resultado de combinar el nombre de teseracto o hipercubo con el prefijo hexe-, que se deriva del griego y significa seis (en este caso seis dimensiones).
	En geometría de séptima dimensión, un *hepteract* es el nombre de un miembro de la familia de los hipercubos, con $2^7 = 128$ vértices (tabla de verdad), 448 aristas, 672 cuadrados, 560 cubos, 260 hipercubos, así como 84 penteract, y 14 hexeract. Su nombre es el resultado de combinar el nombre de teseract o hipercubo con el prefijo hepte que se deriva del griego y significa siete (en este caso siete dimensiones).
	En geometría de octava dimensión, un Octoract es el nombre de un miembro de la familia de los hipercubos, con $2^8 = 256$ vértices (tabla de verdad), 1024 aristas, 1792 cuadrados, 1792 cubos, 1120 hipercubos, así como 448 penteract, 112 hexeract, y 16 hepteract. Su nombre es el resultado de combinar el nombre de tesseract o hipercubo con el prefijo octo que se deriva del griego y significa ocho (en este caso ocho dimensiones).
	En geometría de novena dimensión, un *Eneract* es el nombre de un miembro de la familia de los hipercubos, con $2^9 = 512$ vértices (tabla de verdad), 2304 aristas, 4608 cuadrados, 5376 cubos, 4032 hipercubos, así como 2016 penteract, 672 hexeract, 144 hepteract, y 18 octoract. Su nombre es el resultado de combinar el nombre de tesseract o hipercubo con el prefijo ene que se deriva del griego y significa nueve (en este caso nueve dimensiones).
	En geometría de décima dimensión, un *Decaract* es el nombre de un miembro de la familia de los hipercubos, con $2^{10} = 1024$ vértices (tabla de verdad), 5120 aristas, 11520 cuadrados, 15360 cubos, 13440 hipercubos, así como 8064 penteract, 3360 hexeract, 960 hepteract, 180 octoract, y 20 eneract. Su nombre es el resultado de combinar (Acrónimo) el nombre de teseract o hipercubo con el prefijo deca-1 que se deriva del griego δέκα diez y significa diez. (en este caso diez dimensiones). También se le puede llamar icosaxennon o icosa-10-topo. Es parte de una familia infinita de *politopos* n dimensionales conocida como hipercubos. El politopo dual de un decaract puede ser llamado un 10-ortoplex o decacruce, y es una parte de la familia infinita de los politopos de cruce. El término fue creado por Alicia Boole Stott, hija del matemático y filósofo irlandés George Boole.

Los autómatas finitos se pueden representar mediante grafos particulares también llamados diagramas de estados, y base del planteamiento basado en la metodología es efectuarlos con *hipercubos* o cubo-n, de la siguiente manera: (a) Los estados Q se representan como los vértices, etiquetados con su nombre en el interior, y (b) Una transición δ desde un estado a otro, dependiente de un símbolo del alfabeto, se representa mediante una *arista* dirigida que une a estos vértices, y que está etiquetada con dicho símbolo. Hay que tener en cuenta la adyacencia solo entre vértices en el mismo nivel que son la base en las reducciones en los mapas de Karnaugh utilizados en el algebra de Booleana para efectuar reducciones. Partiendo de un cubo, que se compone de 6 caras cuadradas (*hipercaras*), 12 aristas y $2^3 = 8$ vértices (tabla de verdad); donde $n = 3$ representa la cantidad de estados en la unidad de control para la FMS). Sobre este mismo construir otros hasta obtener relaciones complejas entre

vértices de índole $2^{10} = 1024$. Hay que observar que existe una relación entre la cantidad de dimensiones representa una nueva proposición (variable lógica) que igualmente es n en la potencia de 2^n que forman la tabla de verdad en el modelo propuesto.

3SAT es NP-completo. Ahora vamos a demostrar que incluso una clase más pequeña de expresiones booleanas da lugar a un problema de satisfacibilidad NP-completo. Recuerde que el problema 3SAT consiste en: Dada una expresión booleana E que es el producto de cláusulas, siendo cada una de ellas la suma de tres literales distintos, ¿es E satisfacible?

Aunque las expresiones FNC-3 son una fracción de las expresiones en la FNC, son bastante complejas como para que la prueba de su satisfacibilidad sea NP-completa, como demuestra el siguiente teorema.

3SAT es NP-completo. Evidentemente, 3SAT está en NP, ya que SAT está en NP. Para demostrar que es NP-completo,

Reduciremos CSAT a 3SAT. La reducción se hace como sigue. Dada una expresión en $FNCE = e_1 \wedge e_2 \wedge \cdot \cdot \cdot \wedge e_k$, reemplazamos cada cláusula e_i de la forma siguiente, con el fin de crear una nueva expresión F. El tiempo que se tarda en construir F es lineal respecto de la longitud de E, y veremos que una asignación de verdad satisface E si y sólo si puede extenderse a una asignación de verdad que satisfaga F.

1. Si e_i es un único literal, por ejemplo (x).5 Introducimos dos nuevas variables u y v. Reemplazamos (x) por las cuatro cláusulas $(x + u + v)(x + u + v)(x + u + v)(x + u + v)$. Dado que u y v. aparecen en todas las combinaciones, la única forma de satisfacer las cuatro cláusulas es asignando el valor verdadero a x. Por tanto, todas y sólo las asignaciones que satisfacen E pueden extenderse a una asignación que satisfaga F.

2. Supongamos que e_i es la suma de dos literales, $(x + y)$. Introducimosuna nueva variable z, y reemplazamos e_i por el producto de dos cláusulas $(x + y + z)(x + y + z)$. Como en el caso 1, la única forma de satisfacer ambas cláusulas es satisfacer $(x + y)$.

3. Si ei es la suma de tres literales, ya está en la forma requerida para la FNC-3, por lo que dejamos e_i en la expresión F que estamos construyendo.

4. Supongamos que $e_i = (x_1 + x_2 + \cdot \cdot \cdot + x_m)$ para algúnm≥4. Introducimos nuevas variables $y_1, y_2, \ldots, y_m - 3$ y reemplazamos e_i por el producto de las cláusulas: $(x_1 + x_2 + y_1)(x_3 + y_1 + y_2)(x4 + y_2 + y_3) \cdot \cdot \cdot (x_m - 2 + y_m - 4 + y_m - 3)(x_m - 1 + x_m + y_m - 3)$

Una asignación T que satisface E tiene que asignar al menos a un literal de e_i el valor verdadero, por ejemplo, sea x_j verdadero (recuerde que x_j puede ser una variable sin negar o una variable negada).

Entonces, si hacemos que $y_1, y_2, \ldots, y_{j-1}$ sea verdadero y que $y_j, y_{j+1}, \ldots, y_{m-3}$ sea falso, satisfacemos todas las cláusulas Por tanto, T puede extenderse para satisfacer estas cláusulas. Inversamente, si T hace que todas las x sean falsas, no es posible extender T para que sea verdadera. La razón de ello es que existen $m - 2$ cláusulas y

cada una de las $m-3$ y sólo pueden asignar el valor verdadero a una cláusula, independientemente de si es verdadera o falsa.

Por tanto, hemos demostrado cómo reducir cada caso de E de CSAT a un caso F de 3SAT, tal que F es satisfacible si y sólo si E también lo es. Evidentemente, la construcción requiere un tiempo que es lineal respecto a la longitud de E, ya que ninguno de los cuatro casos anteriores expande una cláusula en más de un factor de 32/3 (es decir, la relación de símbolos en el caso 1), y es fácil calcular los símbolos necesarios de F en un tiempo proporcional al número de dichos símbolos. Dado que CSAT es NP-completo, se deduce que 3-SAT también lo es.

Otros problemas NP-completos. Ahora vamos a proporcionar una pequeña muestra de los procesos en que un problema NP-completo lleva a demostraciones de que otros problemas también son NP-completos. Este proceso de descubrimiento de nuevos problemas NP-completos tiene dos efectos importantes:

Cuando descubrimos que un problema es NP-completo, sabemos que existen pocas posibilidades de que pueda desarrollarse un algoritmo eficiente para resolverlo. Es aconsejable y animamos a buscar soluciones heurísticas, soluciones parciales, aproximaciones u otras formas con el fin de evitar afrontar el problema directamente. Además, podemos hacerlo con la confianza de que no se nos está escapando la *solución adecuada*.

Cada vez que añadimos un nuevo problema NP-completo P a la lista, estamos reforzando la idea de que todos los problemas NP-completos requieren un tiempo exponencial. El esfuerzo que sin duda se ha dedicado a encontrar algoritmos en tiempo polinómico para un problema P ha sido, inconscientemente, un esfuerzo dedicado a demostrar que P = NP. Es el peso acumulado de los intentos sin éxito hechos por muchos matemáticos y científicos expertos para demostrar algo que es equivalente a P = NP lo que fundamentalmente nos lleva al convencimiento de que es muy improbable que P = NP, el lugar de ello lo más probable es que todos los problemas NP-completos requieran un tiempo exponencial.

En esta sección veremos varios problemas NP-completos que implican el uso de grafos. Estos problemas se encuentran entre los problemas de grafos más comunmente utilizados en la solución de cuestiones de importancia práctica. Hablaremos del problema del viajante de comercio (PVC), que vimos en la Sección 10.1.4.

Demostraremos que una versión más simple y también más importante, conocida como problema del circuito hamiltoniano (PCH), es un problema NP-completo, demostrando a continuación que el problema PVC más general también es NP-completo. En varios problemas de *recubrimiento* de grafos, tales como el problema del *recubrimiento de nodos*, que exige determinar el conjunto de nodos mínimo que cubre todos los arcos, en el sentido de que al menos un extremo de cada uno de los nodos se encuentra dentro del conjunto seleccionado.

Descripción de problemas NP-completos. Para introducir nuevos problemas NP-completos, utilizaremos una forma resumida de definición, que es la siguiente:

1. El nombre del problema, y usualmente una abreviatura, como 3SAT.
2. La entrada al problema: lo que se representa, y cómo.
3. La salida deseada: ¿bajo qué circunstancias será la salida *sí*?
4. El problema del que se hace una reducción para demostrar que el problema es NP-completo.

¿Son los problemas *Sí-No* más fáciles? Puede preocuparnos que la versión *sí/no* de un problema sea más fácil que la versión de optimización. Por ejemplo, puede ser complicado determinar el conjunto independiente más grande, pero dado un límite k pequeño, puede ser fácil verificar que existe un conjunto independiente de tamaño k. Aunque esto es cierto, también puede darse el caso de que nos proporcionen una constante k que tenga exactamente el tamaño más grande para el que existe un conjunto independiente. Si es así, entonces resolver la versión $sí/no$ requiere determinar un conjunto independiente maximal.
De hecho, para todos los problemas comunes que son NP-completos, sus versiones $sí/no$ y de optimización son equivalentes en complejidad, difiriendo al menos en un polinomio. Típicamente, como en el caso del problema PCI, si tuviéramos un algoritmo en tiempo polinómico para determinar conjuntos independientes maximales, entonces podríamos resolver el problema de $sí/no$ determinando un conjunto independiente maximal, y comprobando si era al menos tan grande como el límite k. Dado que vamos a demostrar que la versión $sí/no$ es NP-completa, la versión de optimización tiene que ser también intratable.
La comparación puede hacerse también de otra forma. Supongamos que tenemos un algoritmo en tiempo polinómico para la versión $sí/no$ del PCI. Si el grafo tiene n nodos, el tamaño del conjunto independiente maximal se encuentra entre 1 y n. Ejecutando el PCI con todos los límites entre 1 y n, podemos determinar el tamaño de un conjunto independiente maximal (aunque no necesariamente el propio conjunto) en un tiempo que es n veces el tiempo que tarda en resolver el PIC una vez. De hecho, utilizando la búsqueda binaria, sólo necesitamos un factor de $log2\ n$ en el tiempo de ejecución.

¿Para qué son buenos los conjuntos independientes? El objetivo de la investigación no es cubrir las aplicaciones de los problemas NP-completos. Sin embargo, la selección de problemas se ha tomado de un artículo importantísimo acerca de los problemas NP-completos de R. Karp, en el que examinaba los problemas más importantes del campo de la *investigación operativa* y demostraba que muchos de ellos son NP-completos. Por tanto, existe una amplia evidencia de *problemas reales* que se resuelven utilizando estos problemas *abstractos*.

El problema de los conjuntos independientes. Sea G un grafo no dirigido. Decimos que un subconjunto I de los nodos de G es un conjunto independiente si no hay dos nodos de I conectados mediante un arco de G. Un conjunto independiente es maximal si es tan grande (tiene tantos nodos) como cualquier conjunto independiente del mismo grafo.

PROBLEMA. Conjunto independiente (PCI) (*Independent Set*, IS).
ENTRADA. Un grafo G y un límite inferior k, que tiene que estar comprendido entre 1 y el número de nodos de G.
SALIDA. Sí *si* y *sólo si* G tiene un conjunto independiente de k nodos.
REDUCCIÓN DESDE. 3SAT.
Tenemos que demostrar que PCI es NP-completo mediante una reducción en tiempo polinómico de 3SAT. Esta reducción se hace en el siguiente teorema.

El problema del conjunto independiente es NP-completo. No es difícil ver cómo pueden construirse el grafo G y el límite k a partir de la expresión E en un tiempo que es proporcional al cuadrado de la longitud de E, por lo que la conversión de E en G es una reducción en tiempo polinómico. Tenemos que demostrar que 3SAT se reduce correctamente al PCI. Es decir, E es satisfacible si y sólo si G tiene un conjunto independiente de tamaño m.
Parte Si. En primer lugar, observe que un conjunto independiente puede no incluir dos nodos de la misma cláusula, $[i, j_1]$ e$[i, j_2]$ para algún $j_1 = j_2$. La razón es que existen arcos entre cada par de tales nodos, y por lo tanto, si existe un conjunto independiente de tamaño m, este conjunto tiene que incluir exactamente un nodo de cada cláusula.
Además, el conjunto independiente puede no incluir nodos que correspondan tanto a la variable x como a su negación x. La razón es que todos los pares de tales nodos también tienen un arco entre ellos. Por tanto, el conjunto independiente I de tamaño m proporciona una asignación de verdad T que satisface E como sigue: si un nodo que corresponde a una variable x está en I, entonces $T(x) = 1$; si un nodo que corresponde a una variable negada x está en I, entonces seleccionamos $T(x) = 0$. Si no existe ningún nodo en I que se corresponda con x o x, entonces elegimos $T(x)$ arbitrariamente. Observe que el punto (2) anterior explica por qué no podemos llegar a una contradicción si en I hubiese a la vez nodos que corresponden tanto a x como a x.
Afirmamos que T satisface E. La razón es que cada cláusula de E tiene el nodo correspondiente a uno de los literales de I, y T se elige de modo que dicho literal tome el valor verdadero. Entonces, cuando existe un conjunto independiente de tamaño m, E es satisfacible.
Parte *sólo-si.* Ahora supongamos que E se satisface mediante alguna asignación de verdad, por ejemplo T. Dado que T hace que cada cláusula de E sea verdadera, podemos identificar un literal de cada cláusula al que T hace verdadero. Para algunas cláusulas, podemos elegir entre dos o tres de los literales, y si es así, elegiremos uno de ellos de forma arbitraria. Construimos un conjunto I de m nodos seleccionando el nodo correspondiente al literal seleccionado de cada cláusula.
Afirmamos que I es un conjunto independiente. Los arcos entre nodos que proceden de la misma cláusula no pueden tener ambos extremos en I, porque sólo seleccionamos un nodo de cada cláusula. Un arco que conecta una

variable y su negación no puede tener ambos extremos en I, ya que sólo elegimos para I nodos que correspondan a literales que la asignación de verdad T haga que tomen el valor verdadero. Por supuesto T hará que x o x sea verdadero, pero nunca ambas. Concluimos que si E es satisfacible, entonces G tiene un conjunto independiente de tamaño m.
Por tanto, existe una reducción en tiempo polinómico de 3SAT a PIC. Dado que sabemos que 3SAT es NP-completo, de acuerdo con el teorema, el problema PIC también lo es.

8.1.1. Programación Lógica Computacional Aplicada

Como se escribirá un programa que tienen por entrada otros programas y esta tiene que ser una cadena binaria, se tendrá que codificar los programas mediante sucesión de palabras sobre $\{0,1\}$; es de notar que los programas de los ordenadores reales ya están codificados en binario usando el código ASCII. Se usara aquí el mismo principio de codificación y <P> denota la palabra binaria que representa el programa P. La codificación de expresiones por números fue un concepto muy nuevo; hace setenta años y tienen el origen en la codificación de Godel de formulas aritméticas por medio de números. Hoy en día, no solo es que los programas son códigos, sino que estamos acostumbrados a que toda la información sobre personas, aviones, relaciones, imágenes, sonido, etc. este codificada por números, porque de otra manera no se podría introducir en el ordenador. Bajo este planteamiento, todo modelo matemático generado para que se pueda ejecutar en un ordenador, finalmente estará codificado en lenguaje de máquina (*código binario*) para que el mismo lo pueda procesar. De cierta forma se puede pensar que toda expresión matemática tendrá su representación binaria que junto a conectivas lógicas permitirán relacionar proposiciones simples para convertirlas en proposiciones compuestas se puedan efectuar operaciones aritméticas, que emulen el razonamiento de un experto (director de planta que lleva un sinnúmero de años efectuado la programación de operaciones, y sabrá por inducción la conclusión general a partir de observaciones de programas ya efectuados o casos particulares).

Se incorpora en la metodología el paradigma de la *programación lógica*, teniendo como base los procedimientos de deducción de fórmulas válidas en sistemas de axiomas. Al interpretar cada uno de los conjuntos formados por cláusulas de Horn como un programa lógico diseñado en forma sencilla para cada uno de los recursos; se utilizan adecuadamente sus predicados más característicos basados en sistemas lógicos para cada centro de trabajo aplicados en el campo industrial dentro la *planeación operativa* como un modelo para representar la secuenciación de operaciones a corto plazo, con el fin de obtener un procedimiento coherente y completo para dictaminar la satisfacibilidad de las formulas (formula lógica fbf). La ejecución del programa consistirá en la prueba de satisfacibilidad de la sentencia aplicando método eficiente de deducción: la resolución SLD (que significa *Selección Lineal de programas Definidos*)

Declaración del programa versus algoritmo de solución. El algoritmo planteado en la metodología es un problema particular de secuenciación que consta de dos componentes: *la lógica o declarativa*, y *el control*. El componente lógico especifica el conocimiento útil para la resolución del problema, y el componente de control determina la forma en que ese conocimiento puede utilizarse para resolverlo.

En la programación algorítmica, el énfasis se pone en diseñar algoritmos que resuelvan problemas, de modo que al ejecutarlos se calcule una posible solución. Con la programación lógica desarrollada, por el contrario, el énfasis se sitúa en el problema de secuenciación que se pretende presentar una posible solución. En el programa se especifica el componente lógico de los algoritmos (que *insinúa* un procedimiento de cálculo), es decir, el conjunto de condiciones que las soluciones satisfacen. Los mecanismos genéricos de inferencia incorporados en los intérpretes del lenguaje lógico en C, permiten deducir las soluciones a partir de las condiciones que satisfacen.

Algunos lenguajes de programación lógica permiten especificar ambas componentes específicas de un programa lógico: la pura lógica o declaración de axiomas que caracteriza las soluciones, y el control o procedimiento de deducción de conclusiones a partir de los axiomas. En este caso la eficiencia de los programas puede incrementarse cambiando el componente de control sin modificar el componente lógico. Sin embargo el ideal dentro de la programación lógica es despreocuparse del procedimiento de solución, que estaría predeterminado por el entorno de programación en que el programa se ejecuta. El programa consiste exclusivamente, en este caso ideal, en su parte *lógica o declarativa*, y no existe ningún modo en que se pueda optimizar la búsqueda de la solución para la máquina de estados finitos. A continuación se verá cómo las técnicas de resolución pueden verse como un tal procedimiento genérico de solución.

La resolución como algoritmo para la solución de problema planteado. Las técnicas de resolución se aplican a sistemas de axiomas expresados en forma clausulada y cualquier fórmula lógica deducida de un sistema lógico presentado en la metodología, podría expresarse de esta forma. Los axiomas se declaran a modo de reglas:

Si *materiales (premisa1)* y *mano de obra* (*premisa2*) y *equipos* (*premisa3*) y... entonces *disponibles* (*conclusiones*)

Formalmente, se escribe: $A = \forall x_1 \ldots \forall x_k \, (B_1 \wedge \ldots \wedge B_m \rightarrow B)$ donde B y B_i son los átomos que representan, respectivamente, la conclusión (o consecuente) y las correspondientes premisas (o antecedentes). En el caso degenerado en que no hay antecedentes, la fórmula se escribe: $A = \forall x_1 \ldots \forall x_k \, B_1$.
La traducción a forma normal conjuntiva de una regla, prescindiendo de los cuantificadores y eliminando el condicional, es inmediata: $A = \neg(B_1 \wedge \ldots \wedge B_m) \vee B = \neg B_1 \vee \ldots . \vee \neg B_m \vee B$ con a lo sumo un literal positivo.

El literal positivo A, se denomina cabeza y los literales negativos B_i, se denominan cuerpo. Su semántica informal es: para cada asignación de cada variable, si B_1, ..., B_m son fórmulas ciertas, A es una fórmula cierta.
Podemos ahora aplicar el método de resolución para dictaminar si una fórmula escrita en forma normal conjuntiva $G = G_1 \wedge \ldots \wedge G_l$ es una consecuencia lógica del conjunto de axiomas: basta con añadir $\neg G$ al conjunto de axiomas, expresados en forma clausulada, y refutarla por resolución. $\neg G$ se denomina habitualmente la *cláusula objetivo*.
Un conjunto de axiomas es presumiblemente satisfacible, de modo que en la refutación no tiene sentido buscar resolventes entre los axiomas; en lugar de ello se busca resolver la cláusula objetivo con un axioma. Como $\neg G$ se expresa como una disyunción de literales negativos, $\neg G = \neg(G_1 \wedge \ldots \wedge G_m) = \neg G_1 _ \ldots _ \neg G_l$, sólo existe la posibilidad de que uno de tales literales negativos, $\neg G_i$, se resuelva con el único literal positivo del axioma, B. El resolvente será por tanto una cláusula que constará sólo de literales negativos: $\neg B_1 \vee \ldots \vee \neg B_m \vee \neg G_1 _ \ldots \vee \neg G_{i-1} \vee \neg G_{i+1} \vee \ldots \vee \neg G_l$.
Por tanto, todos los resolventes que aparezcan a lo largo de la refutación contendrán sólo literales negativos. Eventualmente, la resolución tendrá lugar con axiomas que no tengan antecedentes, es decir, axiomas cuyas formas clausuladas consistan en un único literal positivo. Progresivamente se producirán resolventes más cortos y, finalmente, la cláusula vacía. De este modo quedará refutada la fórmula $\neg G$ y se habrá probado la satisfacibilidad de la fórmula G.

Programación lógica versus programación algorítmica. La lógica se convierte en una poderosa herramienta para interrogar el mundo en que vivimos y es la idea acerca de cómo la lógica de primer orden podría ser usada como un lenguaje de programación dentro del modelo planteado. El paradigma de la programación lógica se encuadra dentro del paradigma más general de la programación declarativa, cuyo enfoque es esencialmente distinto del de la convencional programación algorítmica. Frente a los lenguajes algorítmicos (tales como Modula-2, C/C++, Java, etc.), también llamados procedimentales o imperativos, los lenguajes declarativos no sirven para especificar *cómo resolver un problema*, sino *qué problema se desea resolver* (la secuenciación de operaciones).

En la *programación algorítmica* un programa consiste en una secuencia de instrucciones que una computadora ha de ejecutar para resolver un problema. El diseño del control de ejecución de tales instrucciones forma también parte del programa. Por el contrario, idealmente en la *programación declarativa* la labor efectuada no consiste en codificar instrucciones explícitas, sino en especificar conocimiento y suposiciones acerca del problema.
Los intérpretes de los lenguajes declarativos tienen incorporado un motor de inferencia genérico que resuelve los problemas a partir de su especificación. Entre los lenguajes declarativos se distinguen los lenguajes funcionales y los lenguajes lógicos.

Mientras que en la *programación funcional* el mecanismo de inferencia genérico se basa en la reducción de una expresión funcional a otra equivalente simplificada, en el paradigma de la computación lógica, el problema de secuenciación se formaliza como una serie de *sentencias lógicas* que ha de probarse bajo los supuestos de un sistema axiomático, que constituye el programa. La ejecución del programa consiste en la prueba de *satisfacibilidad de la sentencia*.

Existe, sin embargo otra diferencia clave entre ambos paradigmas de programación: los lenguajes algorítmicos sólo tienen significado con referencia al procedimiento que realizan en una máquina Von Neumann. La mayoría de los ordenadores modernos se basan en este modelo de máquina, caracterizado por un conjunto homogéneo de celdas de memoria y una unidad de procesamiento con algunas celdas locales denominadas registros. La unidad de procesamiento puede cargar datos desde la memoria a los registros, realiza operaciones lógicas y aritméticas en éstos y almacena valores en la memoria. Los lenguajes de programación se han diseñado para *comunicar* al ordenador los procedimientos de resolución de los problemas desde la perspectiva de la ingeniería del ordenador. De este modo, un programa diseñado para una máquina Von Neumann consiste en una secuencia de instrucciones encaminadas a realizar las operaciones anteriores y un conjunto adicional de instrucciones de control. Para alguien no familiarizado con las restricciones de la ingeniería que conducen al diseño Von Neuman, pensar en términos de un conjunto restringido de operaciones no resulta nada fácil. De ahí que surja la división de tareas (o trabajos) entre el diseño de métodos de solución y la codificación o traducción de las instrucciones de los diseñadores a las instrucciones que puede *entender* el ordenador.

Partiendo del denominado *lenguaje máquina*, directamente comprendido por el ordenador, se han ido desarrollando formalismos y notaciones más convenientes para la expresión humana, aunque en un principio también en correspondencia bastante directa con el lenguaje subyacente de la máquina, al que los programas deben finalmente traducirse para ser ejecutados. El programa consiste en una cadena de estas instrucciones más un conjunto de datos sobre el cual se trabaja. Estas instrucciones son normalmente ejecutadas en secuencia, con eventuales cambios de flujo causados por el propio programa o eventos externos. El *lenguaje de máquina* es específico de la arquitectura de la máquina (disposición de centros de trabajo en la planta de producción), aunque el conjunto de instrucciones disponibles pueda ser similar entre arquitecturas distintas.

Los lenguajes de programación lógica, por el contrario, se derivan del lenguaje abstracto de la lógica matemática, y no guardan dependencia ni relación directa con ningún modelo de máquina. *La lógica proporciona un lenguaje preciso para expresar explícitamente los objetivos, conocimiento y presunciones característicos de un problema, en términos más cercanos a los que un ser humano utiliza en sus razonamientos*. De este modo, no obliga a pensar en términos de las operaciones de un ordenador. Este lenguaje está compuesto por un conjunto de

instrucciones que determinan acciones a ser tomadas por la máquina de estados finitos para cada uno de los niveles.

Formalismo lógico para la representación del problema de secuenciación. La utilidad de las cláusulas de Horn radica en la existencia de un eficiente método de resolución que constituye un método coherente y completo de deducción cuando se aplica a cláusulas de este tipo. Este método de resolución, denominado SLD, haciendo así referencia a la esencia del método, que radica en la selección sucesiva de cláusulas y literales para resolver de acuerdo a criterios especificados por *reglas de computación* y *búsqueda*.

Derivación por resolución SLD: Sea P un conjunto de cláusulas de programa, R una regla de computación y G una cláusula objetivo. Una derivación por resolución SLD de $P \cup \{G\}$ se define como una secuencia de etapas de resolución entre cláusulas objetivo y cláusulas de programa. La primera cláusula objetivo es $G_0 = G$. Asumiendo que se ha derivado G_i, G_{i+1}se define seleccionando un literal $A_i \in G_i$ de acuerdo a la regla de computación R, escogiendo una cláusula $C_i \in P$ tal que la cabeza de C_i pueda unificarse con A_i mediante una sustitución MGU (Unificador de Máxima generalidad) θ_i y resolviendo:

$$G_i = \leftarrow A_i, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n$$

$$C_i = A \leftarrow B_1, \ldots, B_k$$

$$A_i\theta_i = A\theta_i$$

$$G_{i+1} = \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n)\theta_i$$

Una refutación SLD de $P \cup \{G\}$ es una derivación SLD finita de $P \cup \{G\}$ que tiene la cláusula vacía, □, como último objetivo en la derivación. Si G_n es la cláusula vacía, decimos que la longitud de refutación es n.

Respuesta computada: Sea P un conjunto de cláusulas de programa, R una regla de computación y G una cláusula objetivo. Una respuesta computada θ para $P \cup \{G\}$ es la sustitución obtenida por la composición $\theta_1, \ldots, \theta_n$ de todas las variables de G, donde $\theta_1, \ldots, \theta_n$ es la secuencia de MGUs usados en una refutación SLD de $P \cup \{G\}$. Se consideran incluidas sólo las θ que sean sustituciones de las variables del programa. El conjunto de respuestas computadas de un programa define su semántica operacional $S_O(P)$.

Consistencia de la resolución SLD. Sea P un conjunto de cláusulas de programa, R una regla de computación y G una cláusula objetivo. Se supone la existencia de una refutación SLD de G. Sea $\theta = \theta_1, \ldots, \theta_n$ la secuencia de unificadores usados en la refutación y sea σ la restricción de θ a las variables de G, es decir, la respuesta computada en la refutación. Entonces, σ es una respuesta correcta para G. Como las respuestas computadas definen la semántica operacional del programa $S_O(P)$, y las respuestas correctas la semántica declarativa $S_D(P)$, se concluye que $S_O(P) \subseteq S_D(P)$

Completitud de la resolución SLD aplicada a cláusulas de Horn. Sea P un conjunto de cláusulas de programa, R una regla de computación y G una cláusula objetivo. Sea s una respuesta correcta para G. Entonces existe una refutación SLD de G a partir de P tal que s es la restricción de la secuencia de unificadores $\theta = \theta_1, \ldots, \theta_n$ a las variables de G, es decir, la respuesta computada en la refutación. Del mismo modo, ya que para cada respuesta correcta existe una respuesta computada, podemos afirmar que $S_D(P) \subseteq S_O(P)$.
De los dos teoremas anteriores se deduce que $S_D(P) = S_O(P)$., es decir, la semántica declarativa de un programa lógico descrito en términos de cláusulas de Horn coincide con su semántica operacional cuando el control de ejecución consiste en la derivación SLD.

Árbol SLD. Sea P un conjunto de cláusulas de programa, R una regla de computación y G una cláusula objetivo. Todas las posibles derivaciones SLD pueden representarse en una estructura de tipo árbol denominada árbol SLD. La raíz se etiqueta con la cláusula objetivo G. Dado un nodo n etiquetado por la cláusula objetivo G_n, se crea un nodo ni para cada nueva cláusula objetivo G_{n_i}que puede obtenerse resolviendo el literal escogido por R con la cabeza de una cláusula en P.

Rama de éxito de un árbol SLD: Rama que conduce a una refutación
Rama de fallo de un árbol SLD: Rama que conduce a una cláusula objetivo cuyo literal seleccionado no pude unificarse con ninguna cláusula del programa.
Rama infinita de un árbol SLD: Rama correspondiente a una derivación no terminante.
Regla de búsqueda en un árbol SLD: Procedimiento de búsqueda de una refutación en un árbol SLD.
Procedimiento de refutación SLD: Algoritmo de refutación SLD junto con la especificación de una regla de computación y una regla de búsqueda.

Del teorema de completitud se deduce que la resolución SLD es un procedimiento completo independientemente de la elección de la regla de computación, pero sólo indica que existe una refutación. La elección de la regla de búsqueda determinará el que la refutación se encuentre o no. Surge de este modo un compromiso entre *completitud* y *eficiencia*. Una búsqueda *en anchura* en un árbol SLD, donde se comprueban todos los nodos de cada nivel antes de adentrarse más en el árbol, garantiza que se encontrará la rama de éxito siempre que exista. Por el contrario, si se aplica una estrategia de búsqueda en profundidad, siempre existe la posibilidad de explorar una rama no terminante. La regla de búsqueda en anchura es completa en el sentido de que siempre encuentra la respuesta correcta cuando ésta existe. Sin embargo, el tamaño del árbol que requiere almacenar en una estructura de datos crece exponencialmente con la profundidad de la búsqueda, de modo que la aplicación pura de esta estrategia no resulta una regla práctica.

8.2. PASO 5: Sistemas Lógicos Proposicionales: Algebra Booleana

La lógica en su formalización simbólica ha demostrado una íntima relación con las matemáticas, y dio lugar a la lógica matemática. En el siglo XX la lógica ha pasado a ser principalmente la lógica simbólica, un cálculo definido por símbolos y reglas de inferencia, lo que ha permitido su aplicación a la informática. Desde este punto de vista, cualquier operación que se pueda realizar en la computadora se expresa en términos lógicos, siendo esta una máquina electrónica que recibe y procesa datos para convertirlos en información conveniente y útil. Un ordenador está formado, físicamente, por numerosos circuitos integrados (*sistemas lógicos*) y otros muchos componentes de apoyo, extensión y accesorios, que en conjunto pueden ejecutar diversos algoritmos con suma rapidez y bajo el control de un programa. Se extractan las operaciones básicas que realizan los sistemas lógicos y su funcionalidad parte de realizar operaciones con variables lógicas, y por tal motivo nos referimos al algebra boolena para dominar estas mismas.

8.2.1. El razonamiento lógico

Finalmente el objetivo de la lógica: obtener proposiciones verdaderas a partir de otras proposiciones verdaderas ya conocidas. Esta deducción se efectúa mediante lo que se llama un razonamiento lógico. Por ello, se define que es un razonamiento lógico y cómo construirlo. La finalidad no es desarrollar la capacidad de crear nuevos razonamiento lógicos, sino poder analizar razonamientos ya hechos y determinar si son correctos.
Lo que se pretende con un razonamiento lógico es deducir una proposición verdadera nueva a partir de otras proposiciones verdaderas ya conocidas. Al analizar esta idea, empezando por nombrar sus elementos. Las proposiciones que son conocidas se llaman *hipótesis* o *premisas*. La proposición que se deduce es la tesis, resultado o consecuencia. El hecho de que las premisas nos lleven a deducir la consecuencia se puede expresar por medio de una implicación: si las premisas son ciertas, entonces la consecuencia debe ser cierta.
Si $p_1, p_2, \cdots, p_n$ son las premisas y q es la consecuencia, quiero expresar la idea de que si todas las premisas son ciertas, entonces la consecuencia es cierta. Es decir $p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow q$.

Nos interesa el razonamiento lógico independientemente del contenido de las proposiciones. Si es cierto que de las premisas $p_1, p_2, \cdots, p_n$ se sigue necesariamente la consecuencia q, entonces la expresion $p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow Q$ debe ser siempre verdadera.
Una proposición de la forma $p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow Q$ es un razonamiento lógico si es una tautología. Entonces el razonamiento se denota $(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \rightarrow$ q, las proposiciones $p_1, p_2, \cdots, p_n$ se llaman premisas y la proposición q consecuencia.

Al considerar las proposiciones: “si un numero entero no es cero, entonces su valor absoluto es positivo” y “-4 no es cero”. Puedo deducir que el valor absoluto de −4 es un número positivo. Pero la deducción no depende de que estar hablando acerca del valor absoluto de números enteros, sino de su estructura lógica. Las premisas son $p \rightarrow q$ (“si un numero entero no es cero, entonces su valor absoluto es positivo”) y p (“un numero entero, -4, no es cero”), es decir, $(p \rightarrow q) \wedge p$, y la consecuencia es q (“su valor absoluto, $|-4|$, es positivo”). El razonamiento ha sido $(p \rightarrow q) \wedge p \Rightarrow q$ y es valido sean quienes sean p y q.

Por ello tiene nombre propio: modus ponendo ponens (del latın, que se puede traducir como el razonamiento que afirmando p (ponendo) afirma q (ponens)).

Con la misma implicación de antes “si un numero entero no es cero, entonces su valor absoluto es positivo” y además con la proposición “el valor absoluto del numero a no es positivo”, puedo deducir que “el numero a es cero”. El razonamiento en este caso ha sido $(p \rightarrow q) \wedge \neg q \Rightarrow \neg p$, y se llama modus tollendo tollens (el razonamiento que negando q (tollendo) niega p (tollens).
Este razonamiento, cuya estructura es $(p \rightarrow q) \wedge (q \rightarrow r)) \Rightarrow (p \rightarrow r)$, se llama silogismo.

La pregunta ahora es, dado una expresión de la forma $p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow q$, ¿como saber si es un razonamiento lógico o no? Las dos formas de probar que es un razonamiento lógico son: Primera, demostrar directamente que dicha proposición es una *tautología,* mediante una tabla, parte de la investigación es expresar que el problemas $3SAT - FNC$ es una tautologia. Es útil para razonamientos sencillos, cuyas tablas no sean grandes. Segunda, descomponer el razonamiento en razonamientos más simples ya conocidos. Es decir, partiendo de las premisas y aplicando razonamientos simples conocidos ir deduciendo nuevas proposiciones ciertas hasta llegar a la que se enunciaba como consecuencia. Este segundo es el método habitual de probar la validez de razonamientos lógicos complejos. (Van Dalen, 1983)

Para poder utilizar cadenas de razonamientos simples en la prueba de un razonamiento complicado se necesita tener algunos razonamientos ya demostrados de forma directa. Se exponen algunos de estos razonamientos que son muy habituales y, prácticamente, se podría decir que responden en gran medida al sentido común: se llaman *reglas de inferencia* y tienen nombres propios. Ver anexo Teorema 28. (Enderton, 1972)
Aunque tal vez haya un camino más corto para determinar los valores de verdad de una proposición P formada al combinar las proposiciones $p_1, \ldots, p_n$ usando operadores como $\neg$ y $\vee$, la tabla de verdad siempre proporcionará todos los valores de verdad posibles de P para diferentes valores de las proposiciones que la constituyen $p_1, \ldots, p_n$.

8.2.2. Lógica computacional : Programación

El uso del lenguaje de especificación formal como C, es beneficioso en todos los desarrollos, ya que promueve la definición de modelos estructurados, concisos y precisos en diferentes niveles de abstracción, y facilita el razonamiento sobre ellos incluso a un nivel informal. Cuando a las notaciones formales se les asigna una semántica operacional, es posible diseñar herramientas automáticas que detecten ambigüedades en los requisitos iniciales, verifiquen y validen modelos a lo largo del ciclo de desarrollo, ayuden en la evolución y el mantenimiento de los productos y generen automática o semi-automáticamente prototipos o incluso partes del código final del algoritmo de secuenciación. (Torres, 2010)

La lógica ha llegado a ser una herramienta cada vez más utilizada en aplicaciones industriales en la última década y por lo tanto la orientación en esta investigación es a la programación de operaciones detallada para efectuar la planificación a corto plazo en cada uno de los trabajos de una planta de producción.

Una vez definidos los sistemas lógicos, se cubren en tres aspectos formales: (a) primero la interpretación del lenguaje formal, (b) después se define con precisión cómo evaluar el valor de verdad de una expresión del lenguaje y (c) y por último se facilitan sistemas deductivos para decidir, mediante cálculos, sobre propiedades o relaciones entre unas expresiones y otras (más precisamente, entre sus valores de verdad). Adicionalmente un marco formal sencillo donde plantearse propiedades e interrelaciones sobre los valores de las expresiones (*satisfacibilidad, validez, consecuencia, equivalencia*), sobre sus métodos de decisión y sobre los sistemas deductivos creados para abordar estas cuestiones. Y lo que es más importante: tales conceptos son todos exportables a otros sistemas lógicos, con las restricciones propias de cada sistema.

8.2.3. Aplicación: formulas lógicas proposicionales para $3SAT - FNC$

Las formulas lógicas proposicionales fbf para los demás centro de trabajo $\boldsymbol{C_i}$ y recursos definidos: Materiales I_i, Equipos M_i, y Mano de Obra L_i (se relacionan en el anexo A). La salida de la MT se expresa con S_i para el problema $3SAT - FNC$. La generación de la tabla se genera con un algoritmo realizado en Lenguaje C. Ver tabla 8.

Tabla 2. Formula lógica proposicional fbf de Recursos y la salida del Centro de Trabajo $\boldsymbol{C_{11}}$

<table>
<tr><td colspan="3">$\boldsymbol{C_{11}}$</td></tr>
<tr><td>$I_1, I_2, I_3, I_4, I_5, I_6$</td><td>$M_1, M_2, M_3, M_4$</td><td>$L_1, L_2, L_3$</td></tr>
<tr><td colspan="3">Materiales

$S_0 = (I_1 \wedge I_2)$
$S_1 = (I_3 \wedge I_4)$
$S_2 = (I_5 \wedge I_6)$
$S_3 = (S_0 \wedge S_1) = (I_1 \wedge I_2) \wedge (I_3 \wedge I_4)$
$S_4 = (S_3 \wedge S_2) = [(I_1 \wedge I_2) \wedge (I_3 \wedge I_4)] \wedge (I_5 \wedge I_6)$</td></tr>
<tr><td colspan="3">Equipos

$S_5 = (M_1 \wedge I_5 \wedge I_6 \wedge L_1)$
$S_6 = (M_2 \vee M_3)$
$S_7 = (S_0 \wedge S_1 \wedge S_5 \wedge S_6 \wedge S_9) = (I_1 \wedge I_2) \wedge (I_3 \wedge I_4) \wedge (M_1 \wedge I_5 \wedge I_6 \wedge L_1) \wedge (M_2 \vee M_3) \wedge (L_1 \vee L_2)$
$S_8 = (M_4 \wedge S_7 \wedge S_2 \wedge S_3 \wedge S_{10})$
$= M_4 \wedge [(I_1 \wedge I_2) \wedge (I_3 \wedge I_4) \wedge (M_1 \wedge I_5 \wedge I_6 \wedge L_1) \wedge (M_2 \vee M_3) \wedge (L_1 \vee L_2)] \wedge (I_5 \wedge I_6)$
$\wedge [(I_1 \wedge I_2) \wedge (I_3 \wedge I_4)] \wedge [(L_1 \vee L_2) \vee L_3]$</td></tr>
<tr><td colspan="3">Mano de obra

$S_9 = (L_1 \vee L_2)$
$S_{10} = (S_9 \vee L_3) = (L_1 \vee L_2) \vee L_3$</td></tr>
<tr><td colspan="3">$\boldsymbol{S_{11} = (S_4 \wedge S_8 \wedge S_{10}) = \{[(I_1 \wedge I_2) \wedge (I_3 \wedge I_4)] \wedge (I_5 \wedge I_6)\} \wedge \{M_4 \wedge [(I_1 \wedge I_2) \wedge (I_3 \wedge I_4) \wedge (M_1 \wedge I_5 \wedge I_6 \wedge L_1) \wedge}$
M2∨M3 ∧ L1∨L2∧ I5∧ I6∧ I1∧ I2∧ I3∧ I4∧[L1∨L2 ∨ L3]}∧{ [L1∨L2 ∨ L3]}</td></tr>
</table>

8.3. PASO 6: Sistema lógico combinacional

8.3.1. Procedimiento de análisis

La finalidad es proponer el diseño de una planta de producción como un sistema lógico proposicional y demostrar que es posible y funcional. Con la aplicación del álgebra booleana se podría construir cualquier relación lógico-numérica y su uso para efectuar operaciones con variables lógicas y algoritmos de secuenciación.
Shannon[1] demostró con su tesis, que las aplicaciones electrónicas de álgebra booleana podrían sentar las bases de la teoría del diseño del ordenador digital y el circuito digital.

[1] Claude Elwood Shannon (30 de abril de 1916 – 24 de febrero de 2001) fue un matemático, ingeniero eléctrico y criptógrafo estadounidense recordado como «el padre de la teoría de la información.

Figura 17. Lógica combinacional aplicada a recursos por centro de trabajo

Fuente: Autor

Los sistemas lógicos pueden ser combinacionales o secuenciales. Un sistema combinacional consta de compuertas lógicas cuyas salidas en cualquier momento están determinadas en forma directa por la combinación presente de las entradas sin tomar en cuenta las entradas previas. Un sistemas combinacional realiza una operación especifica de procesamiento de información, especificada por completo en forma lógica por un conjunto de funciones booleanas. Los sistemas lógicos secuenciales emplean elementos de memoria (celdas binarias) además de las compuertas lógicas. Sus salidas son una función de las entradas y el estado de los elementos de memoria. Es estado de los elementos de memoria, a su vez, es una función de las entradas previas. Como consecuencia, las salidas de un sistema secuencial dependen no solo de las entradas presentes, sino también de las entras del pasado y, el comportamiento del sistema debe especificarse en una secuencia de tiempo de entradas y de estados internos.

Al reconocer los números binarios y los códigos binarios que representan cantidades discretas de información. Estas variables binarias se representan por voltajes eléctricos o alguna otra señal. Las señales pueden manipularse en las compuertas lógicas digitales para realizar las funciones requeridas. En el apartado anterior se introdujo el algebra booleana como una forma para expresar de manera algebraica las funciones lógicas. Se simplificaran las funciones booleanas para lograr la implementación económica de compuertas. El objetivo es formular varios procedimientos sistemáticos de diseño y análisis de los sistemas lógicos combinacionales. La solución de algunos casos típicos proporcionara un catalogo útil de funciones elementales importante para el entendimiento de las computadoras y sistemas digitales.

Un sistema combinaciónal consta de variables de entras compuertas lógicas y variables de salida. Las compuertas lógicas aceptan las señales de las entradas y generan señales a las salidas. Este proceso transforma la información

binaria de los datos de entrada en los datos requeridos de salida. En forma obvia, tanto los datos de entrada y salida se representan por señales binarias, esto es, existen en dos valores posibles, uno representa la lógica **1** y el otro la lógica **0**. En la figura 21 se muestra un diagrama de bloques de un sistema lógico. Las n variables binarias de entrada provienen de un fuente externa; las m variables de salida van a un destina externo. En muchas aplicaciones, la fuente y/o destino son registros de almacenamiento localizados ya sea en la proximidad del sistema combinacional o en un dispositivo externo remoto. Un registro externo no influencia el comportamiento del sistema combinacional ya que si lo hace el sistema total se vuelve un sistema secuencial.

Para las n varaibles de entrada, hay 2^n combinaciones posibles de los valores binarios de entrada. Para cada combinación posible de entrada, hay una y solo una combinación posible de salida. Un sistema combinacional puede describirse por n funciones booleanas m para cada variable de salida. Cada función de salida se expresa en términos de las n variables de entrada.

Cada variable de entrada a un sistema combinacional puede tener uno o dos alambres. Cuando está disponible solo un alambre, puede representar la variable, ya sea en la forma normal (sin prima) o en la forma complementario (con prima). Ya que una variables es una expresión booleana puede aparecer con prima y/o sin prima, es necesario proporcionar un inversor para cada literal que no está disponible en el alambre de entrada. Por otra parte, una variable de entrada puede aparecer en dos alambres, suministrando las formas tanto normal como complementaria a la entrada del sistema lógico. En este caso, no es necesario incluir inversores para las entradas. El tipo de celdas binarias utilizadas en la mayoría de los sistemas digitales son sistemas lógicos biestables, que tienen salida para los valores tanto normal como complementario de la variable binaria almacenada. Se supondrá que cada variable de entrada aparece en dos alambres, suministrando en forma simultánea valor normal al igual que complementario. (Morris, 2003)

8.3.2. Procedimiento de diseño : algoritmo combinacional

El diseño de los sistemas lógicos combinacionales surge del planteamiento verbal del problema y termina en un diagrama de sistema lógico, o un conjunto de funciones booleanas del cual puede obtenerse con facilidad el diagrama lógico. El procedimiento sigue estos pasos:

Figura 18. Diagrama de bloques de un sistema combinacional

Fuente: Autor

1. Se enuncia el problema
2. Se determina el número de las variables de entrada disponibles y de las variables de salida requeridas.
3. Se asignan símbolos de letra a las variables de entrada y salida
4. Se deriva la tabla de verdad que define las relaciones requerida entre las entradas y las salidas.
5. Se obtiene la formula booleana simplificada para cada salida
6. Se dibuja el diagrama lógico

La tabla de verdad para un sistema combinacional consta de columnas de entrada y columnas de salida. Los **1** y **0** en las columnas de entrada se obtienen de las 2^n combinaciones binarias disponibles para las n variables de entrada. Los valores binarios para las salidas se determinan del examen del problema enunciado. Una salida puede ser igual ya sea a **0** y **1** para cada combinación valida de entrada. Sin embargo, las especificaciones pueden indicar que algunas combinaciones de entrada no ocurrirán. Estas combinación se vuelven condiciones no importa.

Las funciones booleanas de salida de la tabla de verdad se simplifica por cualquier método disponible, como manipulación algebraica, el método de mapa, o el procedimiento de tabulación. Por lo común, habrá una variedad de expresiones simplificadas a elegir. No obstante, en cualquier aplicación particular ciertas restricciones. Limitaciones y criterios servirán como guía en el proceso de escoger una expresión algebraica particular. Un método practico de díselo seria tener que considerar tales restricciones como (1) número mínimo de compuertas, (2) número mínimo de entradas a una compuerta (3) tiempo mínimo de propagación de la señal a través del sistema lógico, (4) número mínimo de interconexciones y (%) limitaciones de las capacidades de impulsión de cada compuerta. Ya que todos estos criterios no pueden satisfacerse en forma simultánea, y ya sea que la importancia de cada restricción se dicta por la aplicación particular, es difícil hacer un enunciado general de los que constituye una simplificación aceptable. En la mayoría de los casos, la simplificación principio por satisfacer un objetivo elemental, como producir una función booleana simplificada en una forma estándar y proceder de ese punto a cumplir cualquiera otros criterios de comportamiento. (Morris, 2003)

En la práctica, los diseñadores tienen a ir de la función booleana a una lista de alambrado que muestra las interconexiones entre varias compuertas lógicas estándar. En este caso, el diseño no va más allá de la función booleana simplificada de salida requerida. Sin embargo un diagrama lógico es de ayuda para visualizar la implementación de compuertas de las expresiones.

Tabla 3. Formula lógica proposicional fbf de Productos $\boldsymbol{p_i}$ por Centro de Trabajo $\boldsymbol{C_{ij}}$

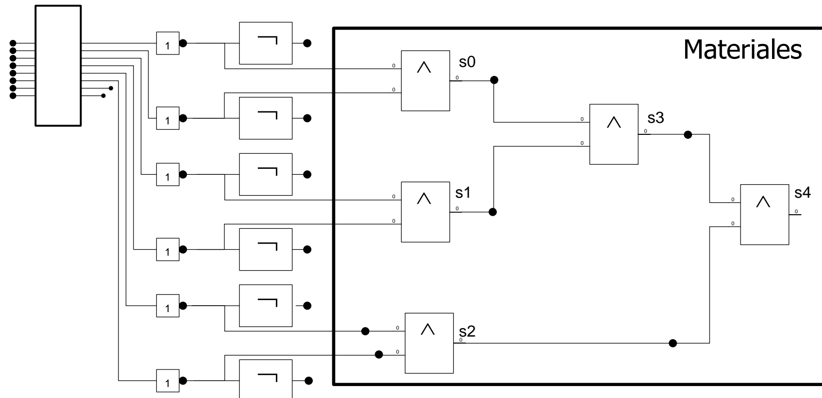
Producto	**Formula lógica combinacional fbf.** Centro de Trabajo	**Nivel FMS MT**
p_1	$(c_{11} \wedge c_{12} \wedge c_{13} \wedge c_{21} \wedge \neg c_{22} \wedge \neg c_{23} \wedge \neg c_{31} \wedge c_{32} \wedge \neg c_{33})$	I
p_2	$(c_{11} \wedge \neg c_{12} \wedge \neg c_{13} \wedge c_{21} \wedge c_{22} \wedge c_{23} \wedge \neg c_{31} \wedge c_{32} \wedge \neg c_{33})$	I
p_3	$(\neg c_{11} \wedge c_{12} \wedge \neg c_{13} \wedge c_{21} \wedge \neg c_{22} \wedge \neg c_{23} \wedge c_{31} \wedge c_{32} \wedge c_{33})$	I
p_4	$(c_{11} \wedge c_{12} \wedge \neg c_{13} \wedge c_{21} \wedge c_{22} \wedge \neg c_{23} \wedge \neg c_{31} \wedge c_{32} \wedge c_{33})$	I
p_5	$(\neg c_{11} \wedge c_{12} \wedge \neg c_{13} \wedge c_{21} \wedge c_{22} \wedge c_{23} \wedge c_{31} \wedge c_{32} \wedge \neg c_{33})$	I
p_6	$(c_{11} \wedge \neg c_{12} \wedge \neg c_{13} \wedge c_{21} \wedge c_{22} \wedge \neg c_{23} \wedge c_{31} \wedge c_{32} \wedge c_{33})$	I
p_7	$(\neg c_{11} \wedge c_{12} \wedge c_{13} \wedge \neg c_{21} \wedge c_{22} \wedge \neg c_{23} \wedge c_{31} \wedge c_{32} \wedge \neg c_{33})$	I
p_8	$(\neg c_{11} \wedge c_{12} \wedge \neg c_{13} \wedge \neg c_{21} \wedge c_{22} \wedge \neg c_{23} \wedge c_{31} \wedge c_{32} \wedge c_{33})$	I
p_9	$(\neg c_{11} \wedge \neg c_{12} \wedge c_{13} \wedge c_{21} \wedge c_{22} \wedge c_{23} \wedge \neg c_{31} \wedge c_{32} \wedge c_{33})$	I
p_{10}	$(\neg c_{11} \wedge c_{12} \wedge c_{13} \wedge \neg c_{21} \wedge c_{22} \wedge c_{23} \wedge c_{31} \wedge c_{32} \wedge c_{33})$	I

Figura 19. (Árbol sintáctico) Sistema lógico combinacional definido en centro de trabajo por recursos

CT – CENTRO DE TRABAJO (C_{11})

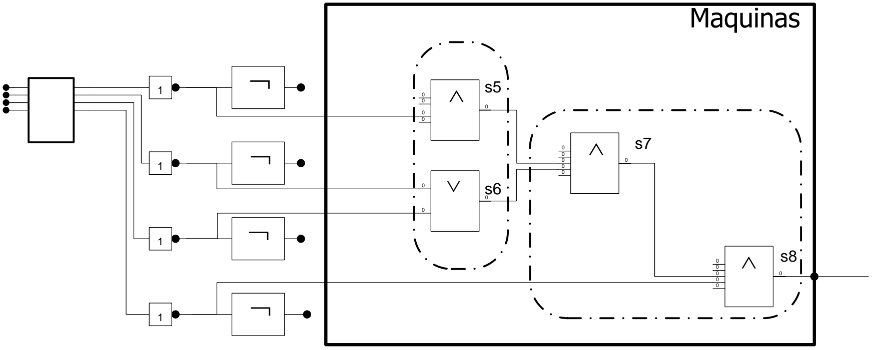
Fuente: Autor

Figura 20. (Árbol sintáctico) Sistema lógico combinacional para Materiales $\boldsymbol{m_i}$ por productos $\boldsymbol{p_i}$



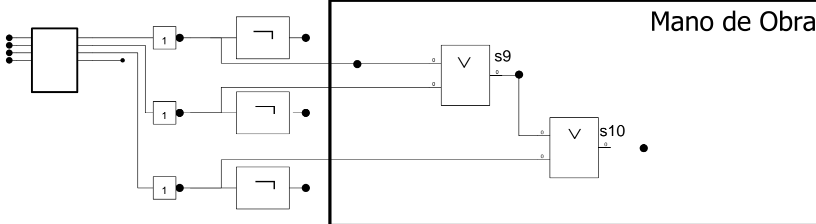
Fuente: Autor

Figura 21. (Árbol sintáctico) Sistema lógico combinacional para máquinas $\mathbf{e_i}$ por productos $\mathbf{p_i}$



Fuente: Autor

Figura 22. (Árbol sintáctico) Sistema lógico combinacional para Mano de Obra $\mathbf{l_i}$ por productos $\mathbf{p_i}$



Fuente: Autor

Las funciones de salida que se especifican en la tabla de verdad dan la descripción exacta del sistema combinacional. Es importante que las especificaciones verbales se interpreten correctamente en una tabla de verdad. Algunas veces el diseñador debe usar su intuición y experiencia para llegar a la interpretación correcta. Las especificaciones verbales rara vez son muy completas y exactas. Cualquier interpretación equivocada que resulte en una tabla de verdad incorrecta producirá un sistema combinacional que no cubriría los requisitos enunciados.

8.4. PASO 7: Sistema lógico secuencial

Los sistemas lógicos digitales que hasta ahora se han considerado han sido combinacionales, esto es, las salidas en cualquier momento dependen por completo de las entradas presente es ese tiempo. Aunque cualquier sistema digital es susceptible de tener sistemas lógicos combinacionales, la mayoría de los sistemas que se encuentra en la práctica también incluyen elementos de memoria, los cuales requieren que el sistema se describa en términos de lógica secuencial.

Figura 23. Diagrama de bloques de un sistema secuencial

Fuente: Autor

Consta de un sistema combinacional al que se conectan elementos de memoria para formar una trayectoria de retroalimentación. Los elementos de memoria son dispositivos capaces de almacenar dentro de ellos información binaria. La información binaria almacenada en los elementos de memoria en cualquier momento dado se define el estado del sistema secuencial. El sistema secuencial recibe información binaria de entradas externas. Estas entradas, junto con el estado presente de los elementos de memoria, determinan el valor binario en las terminales de salida. También determinan las condiciones para cambiar el estado en los elementos de memoria. El diagrama de bloque demuestra que las salidas externas en un sistema secuencial son funciones no solo de las entradas externas sino también del estado presente de los elementos de memoria. El siguiente estado de los elementos de memoria también es una función de las entradas externas y del estado presente. Por tanto, un sistema secuencial esta especificado por una secuencia de tiempo de entradas, salidas y estados internos. (Morris, 2003)
Hay dos tipos principales de sistemas lógicos secuenciales. Su clasificación depende del temporizado de sus señales. Un sistema secuencial síncrono es un sistema cuyo comportamiento puede definirse por el conocimiento de sus señales en instantes discretos de tiempo. El comportamiento de un sistema secuencial asíncrono depende del orden en el cual cambian sus señales de entrada y pueden afectarse en cualquier instante de tiempo. Los elementos de memoria que por lo común se utilizan en los sistemas lógicos secuenciales asíncronos son dispositivos de retardo de tiempo. La capacidad de memoria de un dispositivo de retardo de tiempo se debe al hecho de que toma un tiempo finito para que la señal se propague a través del dispositivo. En la práctica, el retardo de propagación interno en las compuertas lógicas es de suficiente duración para producir el retardo necesario, de modo que pueden ser innecesarias unidades físicas de retardo de tiempo. En los sistemas asíncronos

de tipo de compuerta, los elementos de memoria en la figura 49 constan de compuertas lógicas cuyos retardos de propagación constituyen la memoria requerida. Por consiguiente, un sistema secuencial asíncrono puede considerarse como un sistema combinacional con retroalimentación. Debido a la retroalimentación entre compuertas lógicas, un sistema secuencial asíncrono a veces puede llegar a ser inestable. El problema de la inestabilidad se impone muchas dificultades al diseñador. Los sistemas lógicos secuenciales asíncrono se presentan.

Un sistema lógico secuencial asíncrono, debe emplear señales que afecten los elementos de memoria solo en instantes discretos de tiempo. Una forma de lograr este objetivo es usar pulsos de duración limitada a través del sistema, de modo que una amplitud de pulso represente la lógica **1** y otra amplitud del pulso (o la ausencia de un pulso) represente la lógica 0. La dificultad con un sistema de pulsos es que cualesquiera dos pulsos que lleguen de fuentes independientes separadas a las entradas de la misma compuerta exhibirán retardos impredecibles, que separaran los pulsos ligeramente y resultaran en operación poco confiable.

Los sistemas lógicos secuenciales síncronos usan amplitudes fijas, como niveles de voltaje para las señales binarias. La sincronización se logra por un dispositivo temporizador llamado reloj maestro generador, el cual genera un tren periódico de pulsos de reloj. Los pulsos de reloj se distribuyen a través del sistema en tal forma que los elementos de memoria están afectados solo por la llegada del pulso de sincronización. En la práctica los pulsos de reloj se aplican a compuertas $\wedge$ junto con las señales que especifican el cambio requerido en los elementos de memoria. Las salidas de la compuerta $\wedge$ pueden transmitir señales solo a los instantes que coinciden con la llegada de los pulsos de reloj. Los sistemas lógicos secuenciales síncronos que usan pulsos de reloj en las entradas de los elementos de memoria se denominan sistemas lógicos secuenciales de reloj. Los sistemas lógicos secuenciales de reloj son de tipo que se encuentra con más frecuencia. No manifiestan problemas de inestabilidad y su temporizado se desglosa fácilmente en pasos discretos independientes, cada uno de los cuales se considera por separado. Los sistemas lógicos secuenciales que se exponen en este apartado son exclusivamente del tipo de reloj.

Los elementos de memoria que se usan en los sistemas lógicos secuenciales de reloj se llaman biestables. Estos sistemas lógicos son celdas binarias capaces de almacenar un bit de información. Un sistema biestable tiene dos salidas, una para el valor normal y otra para el valor complementario del bit almacenado en el. La información binaria puede entrar a un biestable en una gran variedad de formas, hecho que da lugar a diferentes tipos de biestables. Se examinaran los diversos tipos de biestables y se definirán sus propiedades lógicas.

Biestables. Un sistema biestable puede mantener un estado binario en forma indefinida (en tanto se suministre potencia al sistema lógico) hasta que recibe la dirección de una señal de entrada para cambiar estado. La

diferencia principal entre los diversos tipos de biestables está en enumero de entradas que poseen y en la manera en la cual las entradas afectan al estado binario. Los tipos comunes de biestable se exponen a continuación.

Sistema lógico básico biestable. Se menciono que un sistema biestable puede construirse mediante dos compuertas ⊼ o dos compuertas ⊽. Estas construcciones se muestran en los diagramas lógicos de las figuras 50. Cada sistema forma un biestable básico bajo el cual pueden construirse otros tipos más complicados. La conexione y acoplamiento cruzado mediante la salida de una compuerta a la entrada de otra constituye una trayectoria de retroalimentación. Por esta razón, los sistemas lógicos se clasifican como sistemas lógicos secuenciales asíncronos. Cada biestable tiene dos salidas Q y $Q',$ dos entradas, ajustar (set) y restaurar (reset). Este tipo de biestable algunas veces se denomina biestable RS directamente acoplado o seguro (latch) SR. La R y S son las iniciales de los dos nombres de la entrada (set y reset en ingles).

Para analizar la operación del sistema en la figura, debe recordarse que la salida de una compuerta ⊽ es **0** si cualquier entrada es **1**, y que la salida es **1** solo cuando todas las entradas son 0. Como punto de inicio, se supone que la entrada ajuste (set) es **1**, y la entrada restaurar (reset) es 0. Ya que la compuerta 2 tiene una entrada de 1, su salida Q' debe ser 0, la cual pone ambas entradas de la compuerta **1** en **0**, de modo que la salida Q es **1**. Cuando la entrada ajuste se regresa a **0**, la salida permanece igual, debido a que la salida Q permanece en **1**, dejando una entrada de la compuerta 2 en 1.

Figura 24. (Árbol sintáctico) Sistema lógico secuencial *biestable* con conectores ⊼ y ⊽

Fuente: Autor

Esto causa que la salida Q' permanezca en 0, lo cual deja ambas entradas de compuerta **1** en **0**, de modo que la salida Q esta en **1**. En la misma forma es posible mostrar que un **1** en la entrada de restaurar cambia la salida Q a **0** y Q'a **1**. Cuando la entrada de restaurar vuelve a 0, las salidas no cambian.

Cuando se aplica un **1** a ambas entradas de ajuste (set) y restaurar (reset) tanto la salida Q como la Q' van a 0. Esta condición viola el hecho de que las salidas Q y Q' son complementos una de otra. En la operación normal esta condición debe evitarse al tener la seguridad de que los **1** no son aplicables en forma simultánea a ambas entradas.

Un biestable tiene dos estados útiles, cuando $Q = \mathbf{1}$ y $Q' = \mathbf{0}$, está en estado ajuste (o estado **1**). Cuando $Q = \mathbf{0}$ y $Q' = \mathbf{1}$ está en el estado despejado (o estado **0**). Las salidas complementaria, respectivamente. El estado binario del biestable se toma para que sea el valor de la salida normal.

Bajo la operación normal, ambas entradas permanecen en **0** a menos que tenga que cambiarse es estado del biestable. La aplicación de un **1** momentáneo a la entrada de ajuste provoca que el biestable pase al estado ajuste. La entrada ajuste debe volver a **0** antes de que un **1** se aplique a la entrada de restaurar. Un **1** momentáneo aplicado a la entrada de restaurar causa que el biestable vaya al estado despejado. Cuando ambas entradas son inicialmente 0, un **1** aplicado a la entrada de puesto mientras el biestable está en el estado ajuste o un **1** aplicado a la entrada de restaurar mientras el biestable este en el estado despejado dejas las salidas son cambio. Cuando se aplica un **1** a ambas entras de ajuste y restaurar, ambas salidas pasan a 0. Este estado es indefinido y por lo común se evita. Si ambas entradas ahora van a 0, el estado del biestable es indeterminado y depende de cual entrada permanezca en **1** más tiempo antes de la transición a 0.

El sistema biestable $\barwedge$ básico en la figura 50 opera con ambas entradas normalmente en **1**, a menos que el estado del biestable tenga que cambiarse. La aplicación de un **0** momentáneo a la entrada de ajuste causa que la salida Q vaya a **1** y Q' a **0**, [poniendo por tanto el biestable en el estado de ajuste. Después de que la entrada de ajuste regresa a 1, un **0** momentáneo en la entrada de restaurar provoca una transición al estado despejado. Cuando ambas entradas van a **0**, ambas salidas irán a **1**, una condición que se evita en la operación normal del biestable. (Ver Anexo 11. Sistemas Secuenciales Biestables).

8.4.1. Análisis de la lógica secuencial con reloj

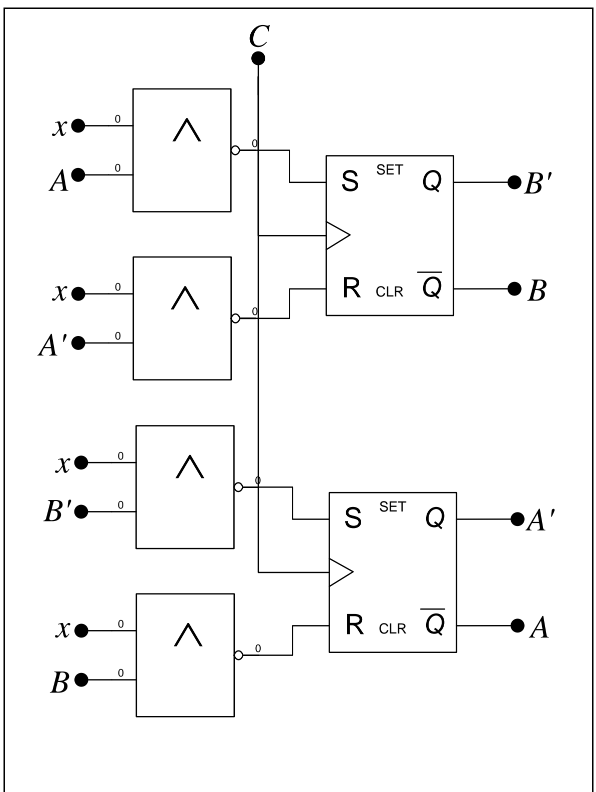
El comportamiento de un sistema secuencial se determina mediante las entradas, las salidas y los estados de sus biestables. Tanto las salidas como el estado siguiente son función de las entradas y del estado presente. El análisis de los sistemas lógicos secuenciales consiste en obtener una tabla o un diagrama de las secuencias de tiempo de las entradas, salidas y los estados internos. También es posible escribir expresiones booleanas que describen el comportamiento de los sistemas lógicos secuenciales. Sin embargo, esas expresiones deben incluir la secuencia de tiempo necesaria ya se en forma directa o indirecta. (Morris, 2003)
Un diagrama lógico se reconoce como el sistema de un sistema secuencia e incluye los biestables puede ser de cualquier tipo y el diagrama lógico puede o no incluir la compuerta combinacionales. Se introduce primero un sistema secuencial temporizado y entonces se presentan diversos métodos para describir el comportamiento de los sistemas lógicos secuenciales. El modelo específico se usara en la exposición para ilustrar diversos métodos.

Modelo de un sistema secuencial. En la figura 58 se muestra un sistema secuencial temporizado. Tiene una variable de entrada x, una variable de salida y dos biestables RS temporizados etiquetados A y B. Las conexiones cruzadas de las salidas de los biestables a las entradas de las compuertas no se muestran por líneas de dibujo, de modo que se facilite el trazado izquierdo del sistema lógico. En lugar de esto, se reconocen las conexiones por el símbolo de letra que se marca en cada entrada. La entrada marcada x' en la compuerta **1** indica una entrada del complemento de x. La segunda entrada marcada A indica una conexión a la salida normal de biestable A.

Se supone un disparo de borde negativo tanto en los biestables como en la fuente que producir a entrada externa x. En este caso, las señales para un estado presente dado están disponibles durante el tiempo desde la terminación de un pulso de reloj hasta la terminación del siguiente pulso de reloj, a cuyo tiempo el sistema pasa al estado siguiente,

Tabla de estado. La secuencia en tiempo de las entradas, salidas y estados de biestable puede enumerarse en una tabla de estado. La tabla de estado para el sistema en la tabla 23 se muestra en la tabla. Consta de tres secciones etiquetadas estado presente, estado siguiente y salida. El estado presente indica los estados de los biestables antes de la ocurrencia del pulso de reloj. El estado siguiente muestra los estados de los biestables después de la aplicación de un pulso de reloj, y la sección de salida lista los valores de las variables de la aplicación de un pulso de reloj. El estado siguiente muestra los estados de los biestables después de la aplicación de un pulso de reloj, y la sección de salida lista los valores de las variables de salida durante el estado presente. Las secciones de estado siguiente al igual que la salida tienen dos columnas, una para $x = 0$ y las otras para $x = 1$.

Figura 25. Sistema lógico secuencial con pulsos de reloj



Fuente: Autor

La derivación de la tabla de estado principia desde un estado inicial supuesto. El estado inicial de la mayoría de los sistemas lógicos secuenciales prácticos se define como el estado con números **0** en todos los biestables. Algunos sistemas lógicos secuenciales tienen un estado inicial diferente y otros no tienen ninguno en absoluto. En cualquier caso, los análisis siempre pueden principiar desde cualquier estado arbitrario. Se principia derivando la tabla de estado desde el estado inicial 00.

Tabla 4. Tabla de estados para el sistema lógico secuencial

Estado presente	**Estado siguiente**		**Salida**	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
AB	AB	AB	y	y
00	00	01	0	0
01	11	01	0	0
10	10	00	0	1
11	10	11	0	0

Cuando el estado presente es 00, $A = 0$ y $B = 0$. Mediante el diagrama lógico, se ve que ambos biestables están despejados $y x = 0$. Ninguna de las compuertas AND produce una señal lógica **1**. Por lo tanto, el estado siguiente permanece sin cambio.

Con $AB = 0$, y $x = 1$ la compuerta 2 produce una señal lógica **1** y la entrada S del biestable B y la compuerta 3 produce una señal lógica **1** a la entrada del biestable A.
Cuando un pulso de reloj dispara los biestable, A se despeja y B esta ajustado, haciendo que el estado siguiente sea 01. Esta información se lista en el primer renglón de la tabla de estado.

En forma similar, puede derivarse el siguiente estado principiando desde los otros tres posibles estados presentes. En general, el estado siguiente es una función de las entradas, del estado presente y del tipo de biestable que se utilice. Con biestables RS, debe recordarse que un **1** en la entrada S establece el biestable y un **1** en la entrada R despeja el biestable, sin importar su estado previo. Un **0** en las entradas S y R deja el biestable sin cambio, mientras que un trato en la entrada S como en la R evidencia un mal diseño y una tabla de estado indeterminada.

Las entradas para la sección de salida son fáciles de derivar. La salida y es igual a **1** solo cuando $X = \mathbf{1}, A = \mathbf{1}\ y\ B = \mathbf{0}$. Por tanto, las columnas de salida se marcan con 0, excepto cuando el estado presente es **10** y la entrada $x = \mathbf{1}$, por lo cual y se marca con un **1**.

La tabla de estado de cualquier sistema secuencial se obtiene por el mismo procedimiento que se usa en el modelo. En general, un sistema secuencial con m biestables y n variables de entrada tendrá 2^m renglones, uno para cada estado. Cada una de las secciones de estado siguiente y salida tendrán 2^n columnas, una para cada combinación de entrada.

Las salidas externas de un sistema secuencial pueden tener procedencia de compuertas lógicas o de elementos de memoria. La sección de salida en la tabla de estado es necesaria solo si hay salidas de compuertas lógicas. Cualquier salida externa que se toma en forma directa de un biestable ya esta listada en la columna de estado presente en la tabla de estado. Por lo tanto, la sección de salida de la tabla de estado puede excluirse si no hay salidas externas de compuertas lógicas.

Diagrama de estado. La información disponible en una tabla de estado puede representarse en forma gráfica en un diagrama de estado. En este diagrama se representa con un circulo y la transición entre estado se indica con líneas dirigidas que conectan los círculos. El diagrama de estado de un círculo secuencial se muestra en la figura 59. El número binario dentro de cada círculo identifica el estado que representa el circulo. Las líneas dirigidas

están etiquetadas con dos números binarios separados por un a /. El valor de entrada que provoca la transición de estado se etiqueta primero; el numero después del símbolo / da un valor de la salida durante el estado presente. La línea dirigida desde el estado 00 al 01 se etiqueta **1/0**, lo cual significa que el sistema secuencial esa en un estado presente 00 mientras $x = \mathbf{1}$ y $y = \mathbf{0}$, y que a la terminación del siguiente pulso de reloj, el sistema pasa al siguiente estado 01. Una línea dirigida que conecta un circulo con si misma indica que ocurre cambio de estado.

Figura 26. Diagrama de estado para el sistema lógico secuencial

Fuente: Autor

El diagrama de estado proporciona la misma información que la tabla de estado y se obtiene en forma directa de la tabla.

No hay diferencia entre una tabla de estado y un diagrama de estado excepto en la forma de representación. La tabla de estado es más fácil de derivar mediante un diagrama lógico dado y en el diagrama de estado continúa en forma directa de una tabla de estado. El diagrama de estado da una imagen de las transacciones de estado y se encuentra en una forma adecuada para la interpretación humana de la operación del sistema lógico. El diagrama de estado se utiliza con frecuencia como la especificación inicial de diseño de un sistema secuencial.

Ecuaciones de estado. Una ecuación de estado (también conocida como una ecuación de aplicación) es una expresión algebraica que especifica las condiciones para una transición de estado de biestable. El primer miembro de la ecuación denota el estado siguiente de un biestable en el segundo miembro, una función booleana que especifica las condiciones de estado siguientes que hacen que el siguiente estado sea igual a Una ecuación de estado es similar en forma a una ecuación característica de biestable, excepto que especifica las condiciones de estado siguiente en términos de las variables externas de entrada y otros valores del biestable. La ecuación de estado se deriva en forma directa mediante una tabla de estado. La ecuación de estado para el biestable A se deriva

mediante la inspección de la tabla. En las columna del estado siguiente, se observa que el biestable A pasa al estado cuatro veces: cuando $x = 0$ y $AB = \mathbf{01}$ o $\mathbf{10}$, o $\mathbf{11}$ o cuando $x = \mathbf{1}$ y $AB = \mathbf{11}$. Esto se puede expresarse en forma algebraica en una ecuación de estado como sigue: $A(t+1) = (A'B + AB' + AB)x' + ABx$

El segundo miembro de la ecuación de estado es una función booleana para un estado presente. Cuando esta función es igual a **1**, la ocurrencia de un pulso de reloj provoca que el biestable tenga un estado siguiente de **1**. Cuando la función es igual a 0, el pulso de reloj causa que A tenga un estado siguiente de 0. El primer miembro de la ecuación identifica al biestable por su símbolo de letra, seguido por la función de tiempo designada $(t+1)$ para hacer más énfasis en que el biestable alcanzara este valor una secuencia de pulsos después.

La ecuación de estado es una función booleana con tiempo incluido. Es aplicable solo en sistemas lógicos secuenciales temporizados, ya que $A(t+1)$ se define para cambiar valor con la ocurrencia de un pulso de reloj en instantes discretos de tiempo.
La ecuación de estado para el biestable A se simplifica mediante un mapa como se muestra. Con alguna manipulación algebraica, la función puede expresarse en la siguiente forma: $A(t+1) = Bx' + (B'x)'A$

Si se hace $Bx' = S$ y $Bx' = R$, se obtiene la relación: $A(t+1) = S + R'A$

Que es la ecuación característica de un biestable RS. Esta relación entre la ecuación de estado y la ecuación característica del biestable puede simplificarse por la inspección del diagrama lógico. De esta manera puede verse que la entrada S del biestable es igual a la función booleana Bx' y que la entrada R es igual a Bx'. La sustitución de esas funciones en la ecuación característica del biestable produce la ecuación de estado para este sistema secuencial.

La ecuación de estado para un biestable en un sistema secuencial puede derivarse de una tabla de estado o de un diagrama lógico. La derivación mediante la tabla de estado consiste en obtener la función del biestable sea un **1**. La derivación mediante un diagrama lógico consiste en obtener las funciones de las entradas del biestable y sustituirlas en la ecuación característica de biestable.

La derivación de la ecuación de estado el biestable B mediante la tabla de estado. Los **1** marcados en el mapa son el estado presente y las combinaciones de entrada que provocan que el biestable pase a un estado siguiente de **1**. Estas condiciones se obtienen de manera directa mediante la tabla. La forma simplificada que se obtiene en el mapa se manipula algebraicamente, y la ecuación de estado obtenida es:

$$A(t+1) = A'\,x + (A'x)'\,B$$

La ecuación de estado puede derivarse en forma directa mediante el diagrama lógico. A partir de la figura, puede verse que la señal para la entrada S del biestable B se genera por la función $A'\,x$ y la señal para la entrada R por la función $A'\,x$. La sustitución de $S = A'\,x$ y $R = A'\,x$ en la ecuación característica de un biestable RS dada por:

$$A(t+1) = S + R'\,B$$

Se obtienen la ecuación de estado derivada antes.

Las ecuaciones de estado de todos los biestable junto con las funciones de salida, especifican en forma completa un sistema secuencial. Representan, algebraicamente, la misma información que una tabla de estado presenta la forma tabular y un diagrama de estado representa en forma grafica.

Funciones de entrada de un biestable. El diagrama lógico de un sistema secuencial consta de elementos de memoria y compuertas. EL tipo de biestables y sus tablas características especifican las propiedades lógicas de los elementos de memoria. Las interconexiones entre las compuertas forman un sistema combinacional que puede especificarse en forma algebraica con funciones booleanas. Por consiguiente, el conocimiento del tipo biestables y una lista de funciones booleana del sistema combinacional proporcionan toda la información necesaria para dibujar el diagrama lógico de un sistema secuencial. La parte del sistema combinacional que genera las salidas externas se describe en forma algebraica por las funciones de salida del sistema lógico. La parte del sistema que genera las entradas a biestables se describe de manera algebraica por un conjunto de funciones booleanas llamadas funciones de entradas de biestables o algunas veces, ecuaciones de entrada.

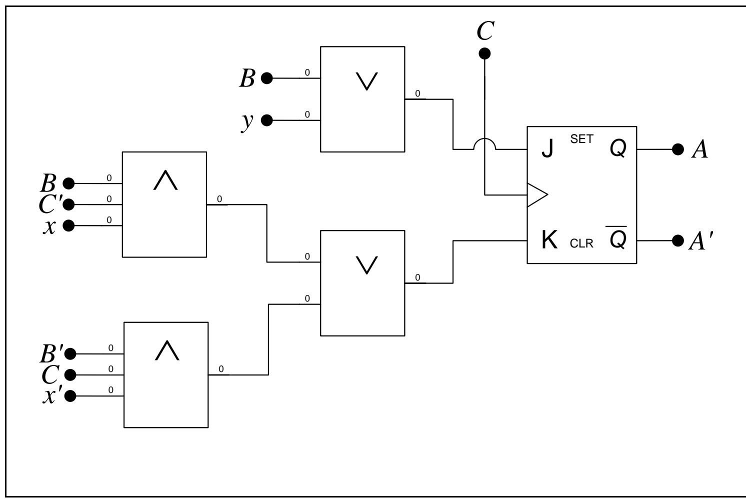
Se adoptara la convención de usar dos letras para denotar una variable de entrada biestable: la primera para designar el nombre de la entrada y la segunda el nombre del biestable. Considérense las siguientes funciones de entrada biestable:

$$JA = BC'x + B'Cx'$$

$$KA = B + y$$

JA y KA denotas dos variables booleanas. La primera letra en cada una denota la entrada J y K respectivamente, de un biestable JK. La segunda letra A es el símbolo del nombre del biestable. El segundo miembro de cada ecuación es una función booleana para la variable correspondiente de entrada al biestable. La implicación de las dos funciones de entrada se muestra en el diagrama lógico en la figura 60. El biestable JK tiene un símbolo de salida A y las dos entradas etiquetadas J y K. El sistema combinacional dibujado en el diagrama es la implantación de la expresión algebraica dada por las funciones de entrada. Las salida del sistema combinacional se denotan por JA y KA en las funciones de entrada si van a las entradas J y K respectivamente , del biestable A.

Figura 27. Implementación de las funciones de entrada al biestable $JA = BC'x + B'Cx'$ y $KA = B + y$



Fuente: Autor

Mediante este sistema logico, puede verse que una función de entrada biestable es una expresión algebraica para un sistema combinacional. La designación de dos letras es un nombre de una variable para una salida del sistema combinacional. Esta salida siempre está conectada a la entrada (denotada por las primeras letras) de un biestable (designado por la segunda letras).

El sistema secuencial en la figura 60 tiene una entrada $x,$ una salida y y dos biestable RS denotados por A y B. El diagrama lógico puede expresarse en forma algebraica con cuatro funciones de entrada biestable y una función de salida del sistema como sigue:

$$SA = Bx' \qquad SB = A'x$$

$$RA = B'x \qquad RB = Ax'$$

$$y = AB'x$$

Este conjunto de funciones booleanas especifica por completo el diagrama lógico. Las variables SA y RA especifica un biestable RS etiquetado A; las variables SB y RB especifican un segundo biestable RS etiquetado B. La variable y denota la salida. Las expresiones booleanas para las variables especifican el sistema combinacional que es parte del sistema secuencial.

Las funciones de entrada biestable constituyen una forma algebraica conveniente para especificar un diagrama lógico de un sistema secuencial. Implican el tipo de biestable mediante la primera letra de las variables de entrada y especifican por completo el sistema combinacional que impulsa al biestable. El tiempo no se incluye en forma explícita en estas ecuaciones, pero se implica mediante la operación del pulso de reloj. Algunas veces es conveniente especificar en forma algebraica un sistema secuencial con las funciones de salida del sistema y las funciones de entrada del biestable, en lugar de dibujar el diagrama lógico.

8.4.2. Reducción y análisis de estados

El análisis de los sistemas lógicos secuenciales principia mediante un diagrama de sistema y culmina en una tabla de estado o diagrama. El diseño de un sistema secuencial se inicia mediante un conjunto de especificación y termina en un diagrama lógico. Los procedimiento de diseño se presentan comenzando desde la sección. En esta sección se exponen ciertas propiedades de los sistemas lógicos secuenciales que pueden usarse para reducir el número de compuertas y de biestable durante el diseño.

Reducción de estado. En cualquier proceso de diseño debe considerarse el problema de minimizar el costo de sistema final. Las dos reducciones de costos más obvios son la reducción en el número de biestable y el número de compuertas. Debido a que estos dos detalles parecen los más evidentes, se han analizado e investigado extensamente. De hecho una gran parte del tema de la teoría de conmutación se dedica a la búsqueda de algoritmos para minimizar el número de biestables y compuertas en los sistemas lógicos secuenciales.

La reducción del número del biestables en un sistema combinacional secuencial se conoce como el problema de reducción de estado. Los algoritmos de reducción de estado tratan con procedimientos para reducir el número de estados en una tabla de estados mientras se mantienen sin cambio los requisitos de entrada-salida externa. Ya que m biestables producen 2^m estados, una reducción ene l numero de estados puede (o no puede) dar por resultado una reducción en el numero de biestables. Un efecto no predecible al reducir el número de biestables es que algunas veces el sistema equivalente (con menos biestables) puede requerir mas compuertas combinacionales.

Se ilustrara la necesidad de la reducción de estado con un caso. Se principia con un sistema secuencial cuya especificación está dada en el diagrama de estado en la figura. En este caso, solo son importantes las secuencias de entrada-salida; los estados internos solo se usan para proporcionar las secuencias requeridas. Por esta razón, los estados que se marcan dentro de los círculos se denotan por símbolos alfabéticos en lugar de sus valores binarios. Esto es en contraste a un contador binario, donde la secuencia de calores binarios de los estados por si mismos se toma como las salidas.

Hay número infinito de secuencias de entras que pueden aplicarse al sistema lógico, cada una conduce a una secuencia única de salida. Considérese la secuencia de entrada 01010110100 principiando desde el estado inicial A. Cada entrada de **0** o **1** produce una salida de **0** o **1** y causa que el sistema pase al estado siguiente. Mediante el diagrama de estado, se obtiene la secuencia de salida y estado para la secuencia dada de entrada como sigue: en el sistema en el estado inicial A, una entrada de **0** produce una salida de **0** y el sistema permanece en el estado a. Con el estado presente b y entrada de **1**, la salida es **0** y el estado siguiente es b. Con el estado presente b y la entrada

de 0, la salida es **0** y el estado siguiente es b. Continuando este proceso se encuentra l secuencia completa como sigue:

estado	$\boldsymbol{a}$	$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{c}$	$\boldsymbol{d}$	$\boldsymbol{e}$	$\boldsymbol{f}$	$\boldsymbol{f}$	$\boldsymbol{g}$	$\boldsymbol{f}$	$\boldsymbol{g}$
entrada	0	1	0	1	0	1	1	0	1	0	0
salida	0	0	0	0	0	1	1	0	1	0	0

En cada columna, se tiene el estado presente, el valor de entrada y el valor de salida. El estado siguiente se escribe en la parte superior de la siguiente columna. Es importante tomar en cuenta que es este sistema los estados por si mismos son de importancia secundaria, ya que se tiene interés solo en las secuencia de salida que provocan las secuencias de entrada.

Ahora se suponen que se ha encontrado un sistema secuencial cuyo diagrama de estado tiene menos de siete estados y se desea compararlos con el sistema cuyo diagrama de estado esta dado en la figura 61. Si se aplican secuencias de entrada idénticas a los dos sistemas lógicos y ocurren salidas idénticas para todas las secuencias de entrada, entonces se dice que los dos sistemas lógicos son equivalentes (en lo que respecta a la entrada – salida) y uno puede reemplazarse por el otro. El problema de reducción de estado es encontrar formas de reducir el número de estados en un sistema secuencial y alterar las relaciones de entrada-salida.

Figura 28. Diagrama de estado

Fuente: Autor

Ahora se procede a reducir el número de estado. Primero, se necesita la tabla de estado; es más conveniente aplicar los procedimiento para reducción de estados aquí que en los diagrama de estado. La tabla de estado del sistema se lista en la tabla y se obtienen en forma directa mediante el diagrama de estado en la figura 61.

Aquí se presenta, sin prueba, un algoritmo para la reducción de estado de una tabla de estado por completo especificada: "Se dice que dos estados son equivalentes si, para cada miembro del conjunto de entradas, dan exactamente la misma salida y envían al sistema ya sea al mismo estado o a su estado equivalente. Cuando dos estados son equivalentes, uno de ellos puede eliminarse sin alterar las relaciones de entrada-salida.

Se aplica este algoritmo a la tabla. Al pasar a través de la tabla de estado, se busca dos estados presentes que vayan al mismo estado siguiente que tengan la misma salida para ambas combinaciones de entrada. Los estados g y e son dos de dichos estados; ambos van a los estados a y f y tiene salidas de $\mathbf{0}$ y $\mathbf{1}$ para $x = \mathbf{0}$ y $x = \mathbf{1}$, respectivamente. Por eso, los estados g y e son equivalentes; pueden eliminarse uno. El procedimiento de eliminar un estado y reemplazarlo por su equivalente de demuestra en la tabla. El renglón con el estado presente g se cruza y el estado g se reemplaza por el estado e cada vez que ocurre en las columnas de estados siguientes.

El estado presente f ahora tiene estado siguientes e y f y salidas $\mathbf{0}$ y $\mathbf{1}$ para $x = \mathbf{0}$ y $x = \mathbf{1}$, respectivamente. Los mismos estados siguientes y salidas aparecen en el renglón con el estado presente d. Por lo tanto, los estados f y g son equivalentes. El estado f puede eliminarse y reemplazarse por d. La tabla repetida final se muestra en la tabla. El diagrama de estado para la tabla reducida consta solo de cinco estados y se muestra en la figura. Este diagrama de estado satisface las especificaciones originales de entrada-salida y producirá la secuencia referida de salida para cualquier secuencia dada de entrada. La siguiente lista que se deriva mediante el diagrama de estado en la figura es para la secuencia de entrada que se utilizo con anterioridad. Se observa que resulta la misma secuencia de salida aunque la secuencia de estado sea diferente:

estado	$\boldsymbol{a}$	$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{c}$	$\boldsymbol{d}$	$\boldsymbol{e}$	$\boldsymbol{d}$	$\boldsymbol{d}$	$\boldsymbol{e}$	$\boldsymbol{d}$	$\boldsymbol{e}$
entrada	0	1	0	1	0	1	1	0	1	0	0
salida	0	0	0	0	0	1	1	0	1	0	0

De hecho, esta secuencia es exactamente la misma que se obtuvo de la tabla y se reemplaza e por g y d por f

Vale la pena observar que la reducción en el número de estados de un sistema secuencial es posible si se tienen interés en las relaciones externas de salida-entrada. Cuando se toman en forma directa salidas externas de los biestables, las salidas deben ser independientes del número de estados antes de que se apliquen algoritmos de reducción de estado.

El sistema secuencial se redujo de siete a cinco estados. En cualquier caso, la representación de los estados con componente físicos requiere que se usen tres biestables, ya que m biestables pueden representar hasta 2^m estados distintos. Con tres biestables, pueden formularse hasta ocho estados binarios denotados por número binarios **000** hasta **111**, con cada bit designando el estado de un biestable. Si se utiliza la tabla de estado en la tabla, se deben asignarse valores binarios a siete estados; el estado restante no se usa. Si se utiliza la tabla de estado, solo cinco estados necesitan asignación binaria, y quedan tres estados sin uso. Los estados sin uso se tratan como condiciones no importa durante el diseño del sistema lógico. Ya que las condiciones no importa por lo común ayudar a obtener funciones booleanas más simples, es más probable que el sistema con cinco estado requiera menos compuertas combinacionales que el sistema con siete estados. En cualquier caso, la reducción de siete a cinco estado no reduce el número de biestables. En general, la reducción del número de compuertas en una taba de estado es probable que resulte en un sistema con menor equipo. Sin embargo, el hecho de que una tabla de estado se ha reducido a menos estados no garantiza un ahorro en el número de biestables o de compuertas.

Asignación de estado. El costo de un sistema combinacional parte de un sistema secuencial puede reducirse por el uso de métodos conocidos de simplificación para los sistemas lógicos combinacionales.

Sin embargo, hay otro factor, conocido como el problema de asignación de estado, que entra en el juego al minimizar las compuertas combinacionales. Los procedimientos de asignación de estado se ocupan con métodos para asignar valores binarios a los estados, en tal forma que reducen el costo de un sistema combinacional que impulsa a los biestables. Esto es de particular ayuda cuando se considera secuencial desde sus terminales externas de entrada-salida. Tal sistema puede seguir una secuencia de estados internos, pero los valores binarios de los estados individuales pueden no ser de consecuencia mientras el sistema produzca la secuencia referida de salida para una secuencia dada de entradas. Esto no se aplica a sistemas lógicos cuyas salidas externas se aplican de manera directa mediante biestables son secuencia binarias especificadas por completo.

Las alternativas disponibles de asignación al estado binario pueden demostrarse junto con sistema secuencial que se especifica en la tabla. Recuérdese que, los valores binarios de los estados son inmateriales mientras que su secuencia mantenga las relaciones apropiadas de entrada-salida. Por esta razón, cualquier asignación de número binario es satisfactoria en tanto que cada estado este asignado a un número único. En la tabla se muestran tres casos de asignaciones binarias posibles para los cinco estados de la tabla reducida. La asignación **1** es una asignación binaria directa para la secuencia de estados desde a hasta e. Las otras dos asignaciones se eligen en forma arbitraria. De hecho, hay 140 diferentes asignaciones distintas para este sistema lógico.

La tabla es la tabla de estado reducida con asignación binaria **1** sustituida por los símbolos de letra de los cinco estados. Es obvio que una asignación binaria diferente causara una tabla de estado con distintos valores binarios para los estados, en tanto las relaciones de entrada-salida permanezcan iguales. La forma binaria de la tabla de estado se utiliza para derivar el sistema combinacional parte del sistema secuencial. La complejidad del sistema combinacional depende de la asignación binaria de estado que se escoja, El diseño del sistema secuencial que se presenta se completa en el siguiente modelo.

Se han sugerido diversos procedimientos que conducen a una asignación binaria particular de las muchas disponibles. El criterio más común es que la asignación que se escoja debe producir un sistema combinacional simple para las entras biestable. Sin embargo, a la fecha, no hay procedimientos de asignación de estado que garanticen un sistema combinacionales de mínimo costo. La asignación de estado es uno de los problemas de reto de la teoría de conmutación. El lector interesado encontrara una rica y cada vez más abundante literatura sobre el tema. Las técnicas para tratar los problemas de asignación de estado rebasan el alcance de este libro.

8.4.3. Procedimiento de diseño

El diseño de un sistema secuencial temporizado principia mediante un conjunto de especificaciones y culmina en un diagrama lógico o en una lista de funciones booleanas, mediante las cuales puede obtenerse el diagrama lógico. En contraste con un sistema combinacional, que esta especificado por completo por una tabla de verdad, un sistema secuencial requiere una tabla de estado para su especificación. El primer paso en el diseño de sistemas lógicos secuenciales es obtener una tabla de estado o una representación equivalente, como un diagrama de estado o ecuaciones de estado.

Un sistema secuencial síncrono esta hecho de biestables y compuertas combinaciones. El diseño del sistema consiste en escoger los biestables y entonces encontrar una estructura de compuertas combinacionales que, junto con los biestables, producen un sistema que cumple con las especificaciones establecidas. El número de biestable se determina mediante el número de estados necesarios en el sistema lógico. El sistema combinacional se deriva mediante la tabla de estado por métodos que se presentan en este capítulo. De hecho, una vez que se determina el tipo y numero de biestables, el proceso de diseño implica una transformación del problema del sistema secuencial en un problema de sistema combinacional. En esta forma, puede aplicarse las técnicas del sistema combinacional.

Se presenta un procedimiento para el diseño de sistemas lógicos secuenciales. Aunque se intenta que sirva como una guía para el principiante, este procedimiento puede reducirse con la experiencia. El procedimiento primero se resume en una lista de pasos consecutivos que se recomiendan como sigue:

1. La descripción verbal del comportamiento del sistema se establece. Esta descripción puede ir acompañada con un diagrama de estado, un diagrama de temporizado o bien otra información pertinente.
2. A partir de la información dada sobre el sistema lógico, se obtienen la tabla de estado.
3. El número de estados puede reducirse por los métodos de reducción de estado y el sistema secuencial puede caracterizarse por las relaciones entrada-salida independientemente del número de estados.
4. Se asignan valores binarios a cada estado si la tabla de estado obtenida en los pasos 2 o 3 contiene símbolos alfabéticos
5. Se determina el número de los biestables necesarios y se asigna un símbolo alfabético a cada uno.
6. Se escoge el tipo del biestable que va a usarse.
7. Mediante la tabla de estado se derivan las tablas de excitación y salida del sistema lógico.
8. Por el uso del método de mapa o cualquier otro método de simplificación, se derivan las funciones de salida y las de entrada del biestable.
9. Se dibuja el diagrama lógico.

La especificación verbal del comportamiento del sistema por lo común supone que el lector está familiarizado con la terminología de la lógica digital. Es necesario que el diseñador utilice su intuición con la terminología de la lógica digital. Es necesario que el diseñador utilice su intuición y experiencia para llegar a la interpretación correcta de la especificación del sistema lógico, debido a que las descripciones verbales pueden ser incompletas e inexactas. Sin embargo, una vez que se ha establecido dicha especificación y se ha obtenido la tabla de estado, es posible hacer uso del procedimiento formal para diseñar el sistema lógico.

La reducción del número de estados y la asignación de valores binarios a los estados se expuso anteriormente. Se supone que se conocen el número de estado y la asignación binaria para los estados. Como consecuencia, los pasos 3 y 4 del diseño no se consideraran en las exposiciones subsecuentes.

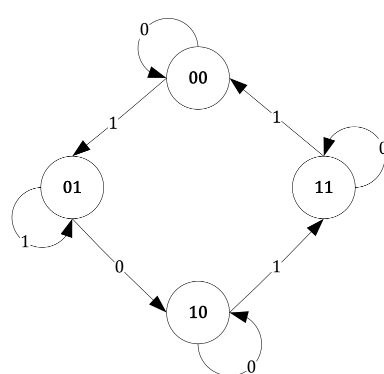
Ya se ha mencionado que el número de biestables está determinado por el número de estados. Un sistema puede tener estados binarios sin usar, si el número total de estados es menor que 2^m. Los estados que no se usan se toman como condiciones no importa durante el diseño del sistema combinacional parte del sistema lógico.

El tipo de biestable que va a usarse puede incluirse en las especificaciones de diseño o puede depender de los que esté disponible para el diseñador. Muchos sistemas digitales se construyen por completo con biestable JK porque son el tipo más versátil disponible. Cuando están disponibles muchos tipos de biestables, es aconsejable usar el biestable RS o D para aplicaciones que requieren transferencia de datos (como registradores con corrimiento), el

tipo T para aplicaciones que implican complementación (como contadores binarios) y el tipo JK para aplicaciones generales.

La información de la salida externa se especifica en la sección de salida en la tabla de estado. Mediante ella pueden derivarse las funciones de salida del sistema lógico.
La tabla de excitación para el sistema es similar a la de los biestables individuales, excepto que las condiciones de entrada están listadas por la información disponible en las columnas de estado presente y estado siguiente de la tabla de estado. El método para obtener la tabla la tabla de excitación y las funciones simplificadas de entrada biestables se ilustran mejor con un caso.
Se desea diseñar el sistema secuencial temporizado cuyo diagrama de estado esta dado en la figura 62. El tipo de biestable que va a usarse es JK.

Figura 29. Diagrama de estado



Fuente: Autor

El diagrama de estado consta de cuatro estados con valores binarios ya asignados. Ya que las líneas dirigidas están marcadas con un solo digito binario sin una /, se concluye que hay una variable de entrada y no hay variables de salida. (El estado de los biestables puede considerarse como las salidas del sistema lógico). Los dos biestables necesarios para representar los cuatro estados se denotan A y B. La variable de estado se denota x.
La tabla de estado para este sistema lógico, derivada mediante el diagrama de estado, se muestra en la tabla. Obsérvese que no hay sección de salida para este sistema lógico. Ahora se mostrara el procedimiento para obtener la tabla de excitación y la estructura de compuertas combinacionales.

Tabla 5. Tabla de estado

estado presente		estado siguiente			
		$x = 0$		$x = 1$	
A	B	A	B	A	B

0	0	0	0	0	1
0	1	1	0	0	1
1	0	1	0	1	1
1	1	1	1	0	0

La derivación de la tabla de excitación se facilita si se ordena la tabla de estado en una forma diferente. Esta forma se muestra en la tabla, donde el estado presente y las variables de entrada están ordenadas en la forma de una tabla de verdad.

El valor del estado siguiente para cada estado presente y las condiciones de entrada se copian de la tabla. La tabla de excitación de un sistema es una lista de las condiciones de entrada biestable que causaran las transiciones requeridas de estado y es una función del tipo biestable que se utilice. Este caso especifica biestables JK, se necesitan columna para las entradas J y K de los biestable A (denotadas por JA y KA) y B (denotadas por JB y KB).

La tabla de excitación para el biestable JK de derivo en la tabla. Esta tabla se usa ahora para derivar la tabla de excitación del sistema lógico. En el primer renglón de la tabla se tiene una transición para el biestable A desde **0** en el estado presente hasta **0** en el estado siguiente. En la tabla se encuentra que una transición de estados desde **0** hasta **0** requiere que la entrada $J = 0$ y la entrada $K = X$ de modo que **0** y X se copian del primer renglón bajo JA y KA, respectivamente. Ya que el primer renglón también muestra una transición para el biestable B desde **0** en el estado presente hasta **0** en el estado siguiente, **0** y X copian en el primer renglón bajo JB y KB.

Tabla 6. Tabla de excitación

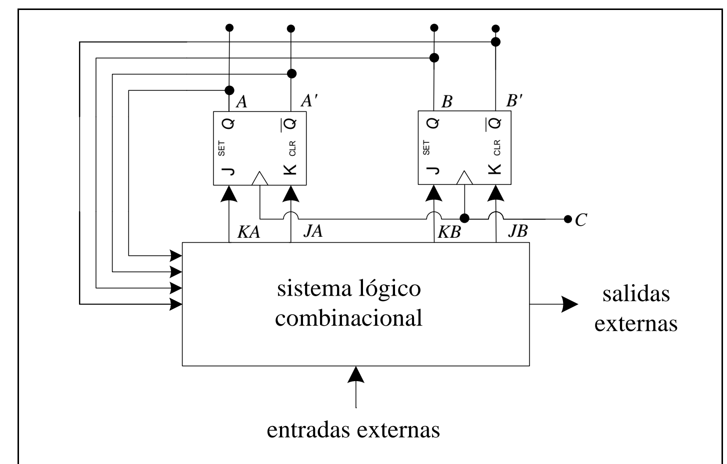
Entradas sistema combinacional					Salidas sistema combinacional			
estado presente		entrada	estado siguiente		entradas del biestable			
A	***B***	***x***	***A***	***B***	***JA***	***KA***	***JB***	***KB***
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

El segundo renglón de la tabla muestra una transición para el biestable B desde **0** en el estado presente hasta **1** en el estado siguiente. Mediante la tabla se encuentra que una transición desde **0** hasta **1** requiere que la entrada $\mathrm{J} = \mathbf{1}$

y la entrada $K = X$. De modo que **1** y X se copian en el segundo renglón bajo JB y KB, respectivamente. Este proceso se continúa para cada renglón de la tabla y para cada biestable, con las condiciones de entrada como se especifican en la tabla que se copian en el renglón apropiado del biestable particular que se esté considerando.

Se hace ahora una pausa y se considera la información disponible de una tabla de excitación como la tabla. Se sabe que un sistema secuencial consta de un número de biestables y un sistema combinacional.

Figura 30. (Árbol sintáctico) Diagrama de bloques del sistema lógico combinacional



Fuente: Autor

En la figura 63 se muestran los dos biestables JK necesarios para el sistema y una caja que representa el sistema combinacional. Mediante el diagrama de bloques, es claro que las salidas del sistema combinacional van a las entradas biestable y las salidas externas (si se especifica). Las entradas al sistema combinacional son las entradas externas y los valores de estado presentes de los biestables. Además, las funciones booleanas que especifican un sistema combinacional se derivan mediante una tabla de verdad que muestra las relaciones de entrada-salida del sistema lógico.

La tabla de verdad que describe al sistema combinacional está disponible en la tabla de excitación. Las entradas al sistema combinación se especifican bajo la columna de estado presente y entrada, y las salidas del sistema combinacional se especifican bajo las columnas de entrada biestable. Por lo tanto, una tabla de excitación transforma un diagrama de estado en la tabla de verdad necesaria para el diseño del sistema combinacional parte del sistema secuencial.

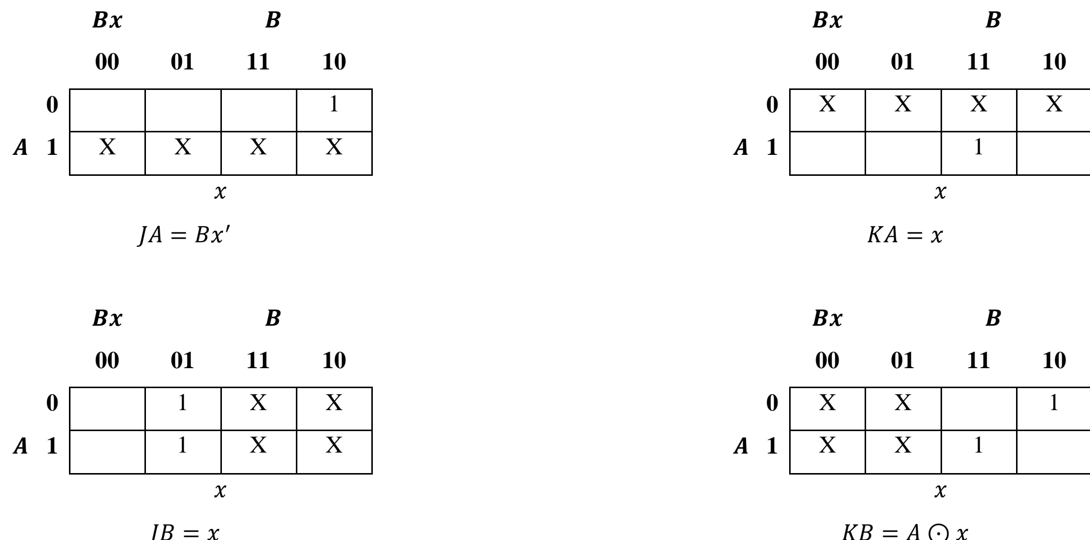
Las funciones booleana simplificadas para el sistema combinacional pueden derivarse ahora. Las entradas son las variables A, B y x ; las salidas son las variables $\mathrm{JA}, \mathrm{KA}, \mathrm{JB}$ y KB. La información de la tabla de verdad se transfiere a los mapas en la figura, donde se derivan las cuatro funciones simplificadas de entrada biestable:

$$JA = Bx' \qquad KA = Bx$$

$$JB = x \qquad KB = A \odot x$$

El diagrama lógico se dibuja en la figura 64 y consta de dos biestables, dos compuertas $\wedge$, una compuerta de equivalencia y un inversor.

Figura 31. Mapas para el sistema combinacional



Fuente: Autor

Con alguna experiencia, es posible reducir la cantidad de trabajo implicado en el diseño del sistema combinacional. Es posible obtener la información de los mapas en la figura 64 en forma directa en la tabla, sin tener que derivar la tabla. Esto se hace por el paso sistemático a través de cada estado presente y combinación de entrada en la tabla y comparando con los valores binarios del estado siguientes correspondiente. Entonces, se determinan las condiciones requeridas de entrada como las especifica la excitación biestable en la tabla. En lugar de insertar los **0, 1** o X obtenidos dentro de la tabla de excitación, pueden escribirse en forma directa en el cuadro adecuado del mapa apropiado.

La tabla de excitación de un sistema secuencial con m biestables, k entradas por biestable y n impulsos externos esta formada por $m + n$ columnas para el estado presente y variables de entrada y hasta 2_{m+n} renglones listados en alguna cuenta binaria conveniente. La sección de estado siguiente tienen xxxx columnas una para cada biestable. Los valores de entrada biestable se listan en mk columnas, una para cada entrada de cada biestable. Si el sistema contiene j salidas, la tabla debe incluir j columnas. La tabla de verdad del sistema combinacional se toma de la tabla de excitación

considerando los $m + n$ estados presentes y columnas de entrada como entradas y los $mk + j$ valores de entrada biestable y las salidas externas como salidas.

Figura 32. (Árbol sintáctico) Diagrama lógico del sistema secuencial

Fuente: Autor

Diseño con estados sin uso. Un sistema con m biestables puede tener 2^m estados. Hay ocasiones en la que un sistema secuencial puede utilizar menos de este número máximo de estado. Los estados que no se usan en la especificación del sistema secuencial no se listan en la tabla de estado. Cuando se simplifican las funciones de entrada a los biestables, los estados sin uso pueden tratarse como condiciones no importa.

Se completa el diseño del sistema secuencial que se presenta. Use la tabla de estado reducida con asignación **1** como se da en la tabla. El sistema empleara biestables $RS.$
La tabla de estados de la tabla vuelve a dibujarse en la tabla en la forma conveniente para obtener la tabla de excitación. Las condiciones de entrada biestable se derivan mediante las columnas de estado presente y estado siguiente de la tabla de estado. Ya que se usan biestables RS, se necesitan consultar la tabla para las condiciones de excitación de este tipo de biestables. Los tres biestables reciben nombres de variables $A, B \; y \; C$ La variable de entrada es x y la variable de salida es y. La tabla de excitación del sistema proporciona toda la información necesaria para el diseño.

Hay tres estados sin usar de este sistema lógico: estado binarios **000, 110** y **111**. Cuando se incluye una entradas de **0** o **1** con estos estados no usados, se obtienen seis minterminos no importa 0, 1, 12, 13, 14 y 15.
Estas seis combinaciones binarias no se listan en la tabla bajo estado presente y entrada y se tratan como términos no importa.

El sistema combinacional parte del sistema secuencial se simplifica en los mapas de la figura 65. Hay siete mapas en el diagrama: seis mapas son para simplificar las funciones de entrada para los tres biestables $RS.$ El séptimo mapa es para

simplificar la salida y. Cada mapa tiene seis X en los cuadros de los minterminos no importa 0, 1 y 2, 13, 14 y 15. Los otros términos no importa en el mapa provienen de las X columnas de entrada biestable de la tabla. Las funciones simplificadas se listan bajo cada mapa. El diagrama lógico obtenido mediante esas funciones booleanas se dibuja en la figura 66.

Un factor que hasta este punto no se ha considerado en el diseño es el estado inicial de un sistema secuencial. Cuando se conecta primero la potencia de un sistema digital, no se conoce en qué estado se asentaran los biestables. Es costumbre proporcionar una entrada maestra de restaurar cuyo propósito es inicializar los estados de todos los biestables en forma sincrónica antes de iniciar las operaciones temporizadas.
En la mayoría de los casos los biestables se despejan en **0** por la señal maestra de restaurar paro algunos pueden ajustarse en **1**. El sistema en la tabla 27 puede restaurarse en forma inicial en un estado $ABC = 001$, ya que el estado 000 no es un estado valido para este sistema lógico.

¿Pero qué pasa si un sistema no se restaura a un estado inicial valido? O peor aún, ¿Qué sucede si debido a una señal de ruido o cualquier otra razón no prevista el sistema mismo se encuentra en uno de sus estados inválidos? En este caso es necesario asegurar que el sistema en forma eventual pasara a uno de los estados validos de modo que pueda reasumir la operación normal. En otra forma, si el sistema secuencial circula entre estado inválidos, no habrá forma de devolverlo a su secuencia intentada de transiciones de estado. Aun cuando puede suponerse que esta condición indeseable no se supone que ocurra, un diseñador cuidadoso debe asegurarse de que esta situación no ocurra jamás.

Se estableció previamente que los estados sin usar en un sistema secuencial pueden tratarse como condiciones no importa. Una vez que se diseña el sistema lógico, los m biestables en el sistema pueden estar en cualquiera de los 2^m estados posibles. Si alguno de los estados se toma como condiciones no importa, debe investigarse el sistema para determinar el efecto de estos estados sin usar. El siguiente estado de los estados inválidos puede determinarse mediante el análisis del sistema lógico. En cualquier caso, siempre es prudente analizar un sistema obtenido mediante un diseño para asegurar que no se incurrió en errores durante el proceso de diseño.

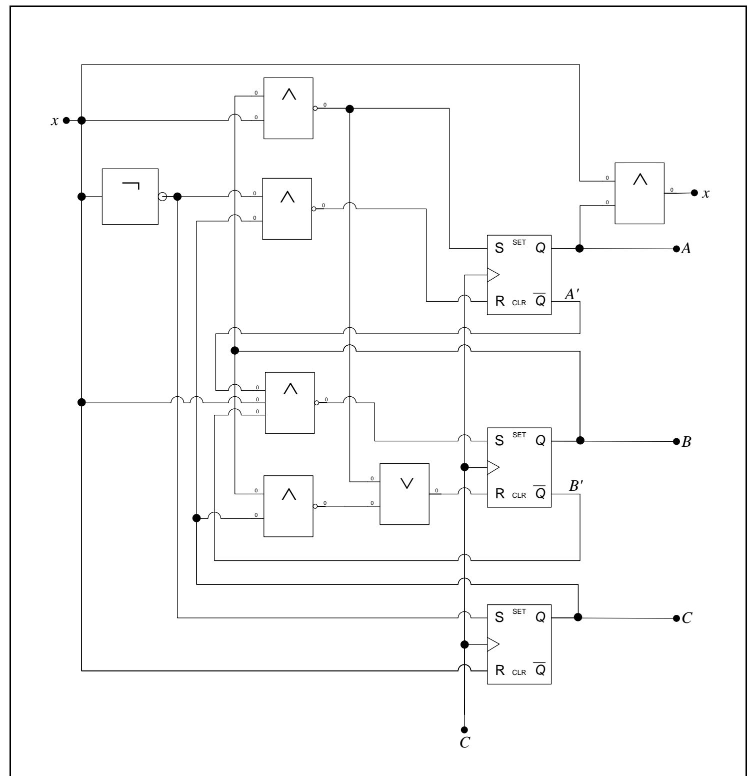
Los estados sin uso son **000, 110** y **110**. El análisis del sistema se realizar por el método delineado en la sección. Los mapas también pueden ayudar en el análisis. Lo que se necesita aquí es principiar con el diagrama del sistema en la figura 66 y derivar la tabla de estado o el diagrama. Si la tabla de estado es idéntica a la tabla (o a la tabla de estados que es parte de la tabla), entonces se sabe que el diseño es correcto. Además, debe determinarse los estado siguientes mediante los estados sin uso **000, 110** y **111**.

Tabla 7. Tabla de excitación para el sistema lógico secuencial

estado presente			entrada	estado siguiente			entradas del biestable						salida
A	B	C	x	A	B	C	SA	RA	SB	RB	SC	RC	y

0	0	1	0	0	0	1	0	X	0	X	X	0	0
0	0	1	1	0	1	0	0	X	1	0	0	1	0
0	1	0	0	0	1	1	0	X	X	0	1	0	0
0	1	0	1	1	0	0	1	0	0	1	0	X	0
0	1	1	0	0	0	1	0	X	0	1	X	0	0
0	1	1	1	1	0	0	1	0	0	1	0	1	0
1	0	0	0	1	0	1	X	0	0	X	1	0	0
1	0	0	1	1	0	0	X	0	0	X	0	X	1
1	0	1	0	0	0	1	0	1	0	X	X	0	0
1	0	1	1	1	0	0	X	0	0	X	0	1	1

Figura 33. (Árbol sintáctico) Diagrama lógico para el sistema lógico secuencial

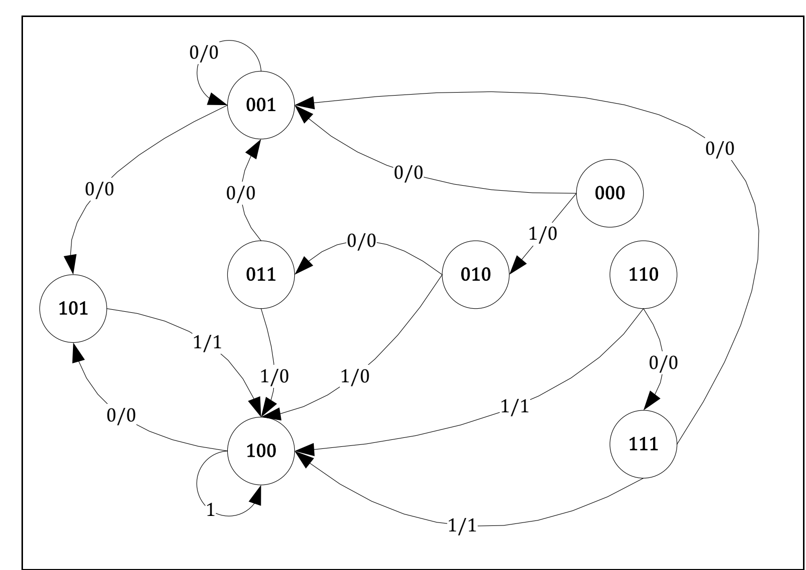


Fuente: Autor

Los mapas en la figura 67 pueden ayudar a encontrar el estado siguiente mediante cada uno de los estados sin uso. Tomar el estado sin uso. Si el sistema lógico, por alguna razón, está en el estado presente, una entrada $x = 0$ transferirá el sistema a algún estado siguiente y una entrada $x = 1$ la transferirá a otro (o el mismo) estado siguiente. Se investigara primero el mintermino $ABCx = 0000$. Mediante los mapas, se ve que este mintermino no esta incluido en alguna función excepto para SC, esto es, la entrada establecida del biestable C. Por tanto, los biestables A y B no cambiaran pero el biestable C se establecerá en **1**. Ya que el estado presente es $ABC = 000$, el estado siguiente será $ABC = 001$. Los mapas también muestran que el mintermino $ABCx = 0001$ está incluido en las funciones para SB y RC. De este modo, B se establecerá y C se despejara. Al principiara con $ABC = 000$ y establecer B, se obtienen el estado siguiente $ABC = 010$ (C esta ya despejada). La investigación en el mapa para la salida y muestra que esta será **0** para estos dos minterminos.

En el diagrama de estado en la figura 67 se muestra el resultado del procedimiento de análisis. El sistema opera como debe, mientras permanezca dentro de los estados **001 010, 111, 100** y **101**. Aun si se encuentra en uno de los estados inválidos **000, 110** o **111**, pasa a uno de los estados validos dentro de uno o dos pulsos de reloj. Así que, el sistema arranca y se corrige por sí mismo, ya que con el tiempo pasa a un estado valido desde el cual continua operando como se requiere.

Figura 34. Diagrama de estado para el sistema lógico combinacional



Fuente: Autor

Una situación indeseable puede haber ocurrido si el estado siguiente de **110** para $x = \mathbf{1}$ resulta ser **111** y el siguiente estado de **111** para $x = \mathbf{0}$ resulta que es **110**. Entonces, si el sistema principia desde **110** o **111**, circulara y permanecerá entre esos dos estados para siempre.

Deben evitarse los estados sin uso que provocan tal comportamiento indeseable; si se encuentra que existen, el sistema debe rediseñarse. Esto puede hacerse con mayor facilidad al especificar un estado siguiente valido para cualquier estado sin uso que se encuentre que circule entre los estados inválidos.

Diseño de contadores. Un sistema secuencial que pasa a través de una secuencia prescrita de estados bajo la aplicación de pulsos de entrada se denomina contador. Los pulsos de entrada, llamados pulsos de conteo, pueden ser pulsos de reloj, o pueden originarse en una fuente externa y pueden ocurrir a intervalos de tiempos prescritos o aleatorios. En un contador, la secuencia de estados puede seguir un conteo binario o cualquier otra secuencia de estados. Los contadores se encuentra en caso todo el equipo que contiene lógica digital. Se usa para contar el numero de ocurrencias de un evento y son útiles para generar secuencias de temporizado para controlar operaciones en un sistema digital.

De las diversas que puede seguir un contador, la secuencia binaria directa es la más simple y la más directa. Un contador que sigue la secuencia binaria se denomina contador binario. Un contador binario de n-bit consta de n biestables y puede contar binario desde **0** hasta $2^n - 1$. El diagrama de estados de un contador de 3-bit se muestra en la figura 68. Como se ve mediante los estados viarios indicados dentro de los círculos, las salidas biestable repiten la secuencia de conteo binario

con un retorno a **000** después de **111**. Las líneas dirigidas entre los círculos no se marcan con valores de entrada-salida como en los otros diagramas de estado.

Recuérdese que las transiciones de estado en los sistemas lógicos secuenciales temporizados ocurren durante un pulso de reloj; los biestables permanecen en sus estados presentes sino ocurre pulso. Por esta razón la variable de pulso de reloj CP no aparece en forma explícita como una variable de entrada en un diagrama de estado o tabla de estado. Desde este punto de vista, el diagrama de estado de un contador no tiene que mostrar los valores de entrada-salida junto con las líneas dirigidas. La única entrada al sistema es el pulso de conteo, y las salidas están especificadas de manera directa por los estados presentes de los biestables. El siguiente estado de un contador depende por completo de su estado presente, y la transición de estado ocurre cada vez que ocurre el pulso.

Debido a esta propiedad, un contador se especifica en forma completa por una lista de la secuencia de conteo, esto es, la secuencia de los estados binarios por los que pasa.

La secuencia de conteo de un contador binario de 3-bit se da en la tabla. El número siguiente en la secuencia representa el estado siguiente que alcanza el sistema bajo la aplicación de un pulso de conteo. La secuencia de conteo se repite después de que alcanza el último valor, de modo que el estado 000 es el estado siguientes después de **111.** La secuencia de conteo da toda la información necesaria para diseñar el sistema lógico.

No es necesario listar los estados siguientes en una columna separada porque pueden leerse en el número siguiente en la secuencia. El diseño de contadores sigue el mismo procedimiento que se delineo, excepto que la tabla de excitación puede obtenerse de manera directa mediante la secuencia de conteo.

La tabla es la tabla de excitación para el contador binario de 3-bit. Los tres biestables reciben las designaciones de variables A_2, A_1 y A_0. Los contadores binarios se construyen en la forma más eficiente con biestables T (o biestable JK con J y K unidas).

Figura 35. Diagrama de estado de un contador binario de 3-bit

Fuente: Autor

La excitación biestable para las entradas T se derivan mediante la tabla de excitación del biestable T y mediante la inspección de la transición de estado desde un conteo dado (estado presente) al siguiente bajo el (estado siguiente). Como ilustración, se consideran las entradas biestable para el renglón 001. El estado presente aquí es 001 y el estado siguiente es 010 que es el conteo siguiente en la secuencia. Al comparar estos dos conteos, se observa que A_2 pasa desde **0** a 0; de

modo que TA_2 se marca con un **0** porque el biestable A_2 debe permanecer sin cambio cuando ocurre el pulso de reloj. A_1 pasa de **0** a **1**; de modo que TA_1 se marca con un **1** porque este biestable debe complementarse en el siguiente pulso de reloj. En forma similar, A_0 pasa de **1** a **0**, indicando que den complementarse; de modo que TA_0 se marca con un **1**. El ultimo renglón con el estado presente **111** se compara con el primer conteo 000 del cual es su estado siguiente. El paso de todos los **1** a todos los **0** requiere que los tres biestables se complementen.

Tabla 8. Tabla de excitación para un contador binario de 3 bits

Secuencia de cuenta			Entradas de biestable		
A_2	A_1	A_0	TA_2	TA_1	TA_0
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	0	0	1
0	1	1	1	1	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	0	0	1
1	1	1	1	1	1

Las funciones biestable de entrada para las tablas de excitación se simplifican en los mapas de la figura. Las funciones booleanas que se listan bajo cada mapa especifican la parte del sistema combinacional del contador. Al incluir estas funciones con los tres biestable, se obtienen el diagrama lógico del contador como se muestra en la figura.

Un contador con n biestable puede tener una secuencia binaria de menos de 2^n números. Un contador BCD cuenta la secuencia binaria desde **0000** hasta **1001** y regresa a **0000** para repetir la secuencia. Otros contadores pueden seguir una secuencia arbitraria que es posible no sea la secuencia binaria directa. En cualquier caso el procedimiento de diseño es el mismo. La secuencia de conteo se lista en la tabla de excitación que se obtienen al comparar un conteo presente con el siguiente conteo que se lista bajo el. Una secuencia de conteo tabulada siempre supone una cuenta repetida, de modo que el siguiente estado de la última entrada es el primer conteo listado.

Diseño mediante las ecuaciones de estado. Un sistema secuencial puede diseñarse mediante las ecuaciones de estado en lugar de la tabla de excitación. Una ecuación de estado es una expresión algebraica que proporciona las condiciones para el siguiente estado como función del estado presente y las variables de entrada. Las ecuaciones de estado de un sistema secuencial se expresan en forma algebraica la misma información que se expresa en forma tabular en una tabla de estado.

Tabla de excitación para un contador binario de 3 bits

Secuencia de cuenta			Entradas de biestable		
A_2	A_1	A_0	TA_2	TA_1	TA_0
0	0	0	0	0	1

0	0	1	0	1	1
0	1	0	0	0	1
0	1	1	1	1	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	0	0	1
1	1	1	1	1	1

El método con ecuación de estado es conveniente cuando el sistema ya esta especificado en esta forma o cuando las ecuaciones de estado se derivan con facilidad de la tabla de estado. Este es el método preferido cuando se utilizan biestables D. Algunas veces puede ser conveniente usar este método con los biestables JK. La aplicación de este procedimiento a sistemas lógicos con biestables RS y T es posible pero implica una cantidad considerable de manipulación algebraica. Aquí se mostrara la aplicación de este método a sistemas lógicos secuenciales que emplean biestables D y JK. El punto de inicio en cada caso es la ecuación característica de biestable derivada en la sección.

Sistemas secuenciales con biestable D. La ecuación característica del biestable D se deriva es $Q(t+1) = D$

Esta ecuación establece que el siguiente estado del biestable es igual al valor presente de su entrada D y es independiente del valor del estado presente. Esto significa que las entradas para el siguiente estado en la tabla de estados son exactamente las mismas que las entradas D. Por tano, no es necesario derivar las condiciones de entrada biestable para la tabla de excitación, ya que esta información ya está disponible en las columnas de los estados siguientes.

Se toma, la tabla de excitación en la tabla. La siguiente columna de estado par*a* A tiene cuatro números **1**, y sucede lo mismo en la columna para el estado siguiente de B. Para diseñar este sistema con biestables D, se escriben las ecuaciones de estado y se igualan a las entradas D correspondientes:

$$A(t+1) = DA(A, B, x) = \Sigma(2, 4, 5, 6)$$
$$B(t+1) = DB(A, B, x) = \Sigma(1, 3, 5, 6)$$

Donde DA y DB son las funciones de entrada biestable para los biestables D denotados A y B respectivamente, y cada función se expresa como la suma de cuatro minterminos.
Las funciones simplificadas pueden obtenerse mediante dos mapas de tres variables. Las funciones de entrada biestable simplificada son:

$$DA = AB' + Bx'$$
$$DB = A'x + B'x + ABx'$$

Si hay estados sin usar en el sistema secuencial, deben considerarse junto con las entradas, o combinacionales no importa. Los minterminos no importa así obtenidos pueden utilizarse para simplificar las ecuaciones de estado de las funciones de entrada biestable D.

Diseñe un sistema secuencial con cuatro biestable $A, B, C, D.$ Los siguientes estados de B, C, D son iguales a los estados presente de A, B, C respectivamente. El siguiente estado de A es igual a la $\vee$-excluyente de los estados presentes C y D.
Mediante el planteamiento del problema, es conveniente escribir primero las ecuaciones de estado para sistema lógico:

$$A(t+1) = C \oplus D$$
$$B(t+1) = A$$
$$C(t+1) = B$$
$$D(t+1) = C$$

Este sistema especifica un registro de corrimiento con retroalimentación.
En un registro de corrimiento con retroalimentación, cada biestable transfiere o corre su contenido al siguiente biestable cuando el pulso de reloj, pero el estado siguiente del primer biestable (A en este caso) es cierta función presente de los otros biestables. Ya que las ecuaciones de estado son muy simples, el biestable de uso más conveniente es el tipo $D.$

Las funciones de entradas biestable para estos sistemas lógicos se toman de manera directa mediante las ecuaciones de estado, con la variable del estado siguiente reemplazada por la variable de entrada biestable:

$$DA = C \oplus D$$
$$DB = A$$
$$DC = B$$
$$DD = C$$

El sistema puede construirse con cuatro biestables D y una compuerta $\vee$-excluyente.

Ecuaciones de estado con biestables JK. La ecuación característica para el biestable JK se deriva

$$Q(t+1) = (J)Q' + (K')Q$$

Las variables de entrada J y K están encerradas entre paréntesis para no confundir los términos $\wedge$ de la ecuación característica con la convención de dos letras, que se ha usado para representar las variables de entrada biestable.

El sistema secuencial puede derivarse de manera directa mediante las ecuaciones de estado sin tener que dibujar la tabla de excitación. Esto se hace mediante un procedimiento de comparación entre las ecuaciones de estado para cada biestable y la ecuación característica general del biestable JK. El proceso de comparación consiste en manipular cada ecuación de estado hasta que esté en la forma de ecuación característica. Una vez que esto se ha hecho, las funciones para las entradas

J y K pueden extraerse y simplificarse. Esto debe hacerse para cada ecuación de estado que se liste y su nombre de variable biestable $A, B, C, ...$ debe reemplazar la letra Q en la ecuación característica.
Una ecuación dada de estado para $Q(t+1)$ ya puede expresarse como una función de Q y Q'. Con más frecuencia Q y Q' o ambas pueden estar ausentes en la expresión booleana. Entonces es necesario manipular en forma algebraica la expresión hasta que tanto Q como Q' estén incluidas en la expresión. El siguiente caso demuestra todas las posibilidades que puedan encontrarse.
Diseño de un sistema secuencial con biestables JK para satisfacer las siguientes ecuaciones de estado:

$$A(t+1) = A'B'CD + A'B'C + ACD + AC'D'$$
$$B(t+1) = A'C + CD' + A'BC'$$
$$C(t+1) = B$$
$$D(t+1) = D'$$

Las funciones de entrada para el biestable A se derivan por este método ordenando la ecuación de estado y comparándola con la ecuación característica como sigue:

$$A(t+1) = (B'CD + B'C)A' + (CD + C'D')A$$
$$= (J)A' + (K')A$$

Mediante la igualdad de las dos ecuaciones, se deduce que las funciones de entrada para el biestable A son:

$$J = B'CD + B'C = B'C$$
$$K = (CD + C'D') = CD' + C'D$$

La ecuación de estado para el biestable B puede ordenarse como sigue:

$$B(t+1) = (A'C + CD') + (A'C')B$$

No obstante, esta forma no es adecuada para compararla con la ecuación característica ya que falta la variable B'. Si la primera cantidad entre paréntesis se trata con el operador AND con $(B' + B)$, la ecuación permanece igual pero con la variable B' incluida. Por lo tanto:

$$B(t+1) = (A'C + CD')(B' + B) + (A'C')B$$
$$= (A'C + CD')B' + (A'C + CD' + A'C')B$$
$$= (J)B' + (K')B$$

Mediante la igualdad de las dos ecuaciones, se deducen las funciones de entrada para el biestable B:

$$J = A'C + CD'$$
$$K = (A'C + CD' + A'C')' = AC' + AD$$

Las ecuaciones de estado para biestable C pueden manipularse como sigue:

$$C(t+1) = B = B(C' + C) = BC' + BC$$
$$= (J)C' + (K')C$$

Las funciones de entrada para el biestable C son:

$$J = B$$
$$K = B'$$

Por último, la ecuación de estado para el biestable C puede manipularse con el objeto de comparación como sigue:

$$D(t+1) = D' = 1.D' + 0.D'$$
$$= (J)D' + (K')D$$

Lo cual da la función de entrada: $J = K = 1$

Las funciones derivadas de entrada pueden acumularse y listarse juntas. La convención de dos letras para denotar la variable de entrada biestable que no se usa en la derivación anterior esta a continuación:

$$JA = B'C \qquad KA = CD' + C'D$$
$$JB = A'C + CD' \qquad KB = AC' + AD$$
$$JC = B \qquad KC = B'$$
$$JD = 1 \qquad KD = 1$$

El procedimiento de diseño que aquí se introduce es un método alterno para determinar las funciones de entrada biestable de un sistema secuencial cuando se emplean biestables $JK.$ Para usar este procedimiento cuando inicialmente se especifica un diagrama de estado o una tabla de estado, es necesario que las ecuaciones de estado se deriven por el procedimiento delineado.. El método de las ecuaciones de estado para encontrar las funciones de entrada biestable puede ampliarse para cubrir los estados sin uso que se consideran como condiciones no importan. Los minterminos no importa se escriben en la forma de una ecuación de estado y se manipulan hasta que están en la forma de la ecuación característica para el biestable particular que se considere. Las funciones J y K en la ecuación de estado no importa se toman entonces como minterminos no importa cuando se simplifican las funciones de entradas para un biestable particular.

9. DISENO E IMPLEMENTACION

9.1. PASO 8. Autómata finito

Separar aquellos problemas que se podían resolver de forma eficiente mediante un ordenador de aquellos problemas que, en principio, pueden resolverse, pero que en la práctica consumen tanto tiempo que las computadoras resultan inútiles para todo excepto para casos muy simples del problema. Este último tipo de problemas se denominan *insolubles* o NP-difíciles.
En particular, la teoría de los problemas intratables nos permite deducir si podremos enfrentarnos a un problema y escribir un programa para resolverlo (porque no pertenece a la clase de problemas intratables) o si tenemos que hallar alguna forma de salvar dicho problema: hallar una aproximación, emplear un método heurístico o algún otro método para limitar el tiempo que el programa invertirá en resolver el problema.

La ventaja de disponer de sólo un número finito de estados es que podemos implementar el sistema mediante un conjunto fijo de recursos. Por ejemplo, podríamos implementarlo por hardware como un circuito, o como una forma simple de programa que puede tomar decisiones consultando sólo una cantidad limitada de datos o utilizando la posición del propio código para tomar la decisión. Acercar los resultados de autómatas a la práctica dentro de la metodología planteada, para efectuar una programación estructurada y simplificar las demostraciones apelando a la intuición del investigador al reconocer el potencial de la programación moderna.

Para conseguir iteraciones o recursiones correctas, es necesario establecer hipótesis inductivas, y resulta útil razonar, formal o informalmente, que la hipótesis es coherente con la iteración o recursión. La teoría de autómatas, quizá más que cualquier otra materia de las que conforman las ciencias de la computación, se presta a demostraciones naturales e interesantes, tanto de tipo deductivo (una secuencia de pasos justificados) como inductivo (demostraciones recursivas de una proposición parametrizada que emplea la propia proposición para valores *más bajos* del parámetro). (Rodríguez, 2010)

Inducciones mutuas. En ocasiones, no se puede probar una sola proposición por inducción, y es necesario probar un grupo de proposiciones $S_1(n), S_2(n), \ldots, S_k(n)$ por inducción sobre n. En la teoría de autómatas se pueden encontrar muchas situaciones así. En el Ejemplo 1.23 veremos la situación habitual en la que es necesario explicar qué hace un autómata probando un grupo de proposiciones, una para cada estado. Estas proposiciones establecen bajo qué secuencias de entrada el autómata entra en cada uno de los estados.
En términos estrictos, probar un grupo de proposiciones no es diferente a probar la conjunción (AND lógico) de todas las proposiciones. Por ejemplo, el grupo de proposiciones $S_1(n), S_2(n), \ldots, S_k(n)$ podría reemplazarse por una sola proposición $S_1(n)$ AND $S_2(n)$ AND $\cdots$ AND $S_k(n)$. Sin embargo, cuando hay que demostrar realmente varias

proposiciones independientes, generalmente, resulta menos confuso mantener las proposiciones separadas y demostrar cada de una ellas con su caso base y su paso de inducción. A este tipo de demostración se la denomina inducción mutua.

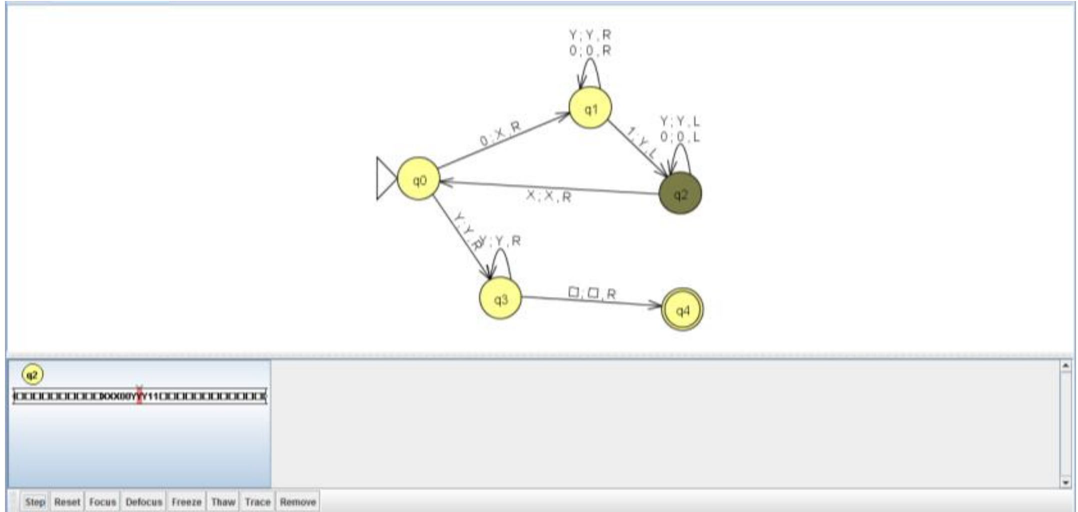
Se presentan las definiciones de los términos más importantes empleados en la teoría de autómatas. Estos conceptos incluyen el de *alfabeto* (un conjunto de símbolos), *cadenas de caracteres* (una lista de símbolos de un alfabeto) y *lenguaje* (un conjunto de cadenas de caracteres de un mismo alfabeto).

La demostración de que los AFD pueden hacer lo que hacen los AFN implica una *construcción* importante conocida como construcción de subconjuntos, porque exige construir todos los subconjuntos del conjunto de estados del AFN. En general, muchas de las demostraciones acerca de autómatas implican construir un autómata a partir de otro. Es importante para nosotros ver la construcción de subconjuntos como un ejemplo de cómo se describe formalmente un autómata en función de los estados y transiciones de otro, sin conocer las especificidades de este último.
Q_D es el conjunto de los subconjuntos de Q_N; es decir, Q_D es el conjunto potencia de Q_N. Observe que si Q_N tiene n estados, entonces Q_D tendrá 2^n estados. A menudo, no todos estos estados son accesibles a partir del estado inicial de Q_D. Los estados inaccesibles pueden ser *eliminados*, por lo que el número de estados de D puede ser mucho menor que 2^n.

9.2. PASO 9. Lenguajes recursivos

Un conjunto de cadenas, todas ellas seleccionadas de un Σ∗, donde Σ es un determinado alfabeto se denomina lenguaje. Si Σ es un alfabeto y L ⊆Σ∗, entonces L es un lenguaje de Σ. Observe que un lenguaje de Σ no necesita incluir cadenas con todos los símbolos de Σ, ya que una vez que hemos establecido que L es un lenguaje de Σ, también sabemos que es un lenguaje de cualquier alfabeto que sea un superconjunto de Σ.
La elección del término *lenguaje* puede parecer extraña. Sin embargo, los lenguajes habituales pueden interpretarse como conjuntos de cadenas.
Sin embargo, existen también otros muchos lenguajes que veremos a lo largo del estudio de los autómatas.
Algunos ejemplos son los siguientes:

1. El lenguaje de todas las cadenas que constan de n ceros seguidos de n unos para cualquier $n \geq 0$: $\{\varepsilon, 01, 0011, 000111, \ldots\}$.
2. El conjunto de cadenas formadas por el mismo número de ceros que de unos: $\{\varepsilon, 01, 10, 0011, 0101, 1001, \ldots\}$
3. El conjunto de números binarios cuyo valor es un número primo: $\{10, 11, 101, 111, 1011, \ldots\}$
4. Σ∗ es un lenguaje para cualquier alfabeto Σ.
5. $\emptyset$, el lenguaje vacío, es un lenguaje de cualquier alfabeto.
6. {ε }, el lenguaje que consta sólo de la cadena vacía, también es un lenguaje de cualquier alfabeto. Observe que $\emptyset \neq$ {ε }; el primero no contiene ninguna cadena y el segundo sólo tiene una cadena.



1. $\{0^n1^n \mid n \geq 1\}$. Esta expresión se lee: El conjunto de 0 a la n 1 a la n tal que n es mayor o igual que 1, este lenguaje consta de las cadenas {01,0011,000111, . . .}.Observe que, como con los alfabetos, podemos elevar un solo símbolo a una potencia n para representar n copias de dicho símbolo.
2. $\{0^j1^j \mid 0 \leq i \leq j\}$. Este lenguaje consta de cadenas formadas por ceros (puede ser ninguno) seguidos de al menos el mismo número de unos.

Problemas. En la teoría de autómatas, un problema es la cuestión de decidir si una determinada cadena es un elemento de un determinado lenguaje (salida del sistema lógico combinacional). Como veremos, cualquier cosa que coloquialmente denominamos *problema* podemos expresarlo como elemento de un lenguaje. De manera más precisa, si Σ es un alfabeto y L es un lenguaje de Σ, entonces el problema L es: Dada una cadena w de Σ∗, decidir si w pertenece o no a L.

¿Es un lenguaje o un problema? *Lenguajes y problemas son realmente la misma cosa.* El término que empleemos depende de nuestro punto de vista. Cuando sólo nos interesan las cadenas, por ejemplo, en el conjunto $\{0^n1^n \mid n \geq 1\}$, entonces tendemos a pensar en el conjunto de cadenas como en un lenguaje. Se tendrá a asignar semánticas a las cadenas, por ejemplo, pensaremos en ellas como en grafos codificados, expresiones lógicas o incluso enteros. En dichos casos, se estará más interesado en lo que representan las cadenas que en las mismas, tenderemos a *pensar que el conjunto de cadenas es un problema.*

La definición de *problema* como lenguaje ha resistido la prueba del tiempo como la forma apropiada de tratar con la importante cuestión de la teoría de la complejidad. En esta teoría, lo que nos interesa es determinar los límites inferiores de la complejidad de determinados problemas. Especialmente importantes son las técnicas que demuestran que ciertos problemas no se pueden resolver en un periodo de tiempo menor que una exponencial del tamaño de la entrada. Resulta que la versión sí/no, basada en el lenguaje, de problemas conocidos resulta ser igual de difícil en este sentido que las versiones *resuélvelo*. (Enrique Mandado Pérez, 2009)

Es decir, si podemos demostrar que es difícil decidir si una determinada cadena pertenece al lenguaje L_X de cadenas válidas del lenguaje de programación X, entonces será lógico que no sea fácil traducir los programas en el lenguaje X a código objeto.

Esta técnica que demuestra que un problema es difícil utilizando su algoritmo supuestamente eficiente, para resolver otro problema que se sabe que es difícil se conoce como *reducción* del segundo problema al primero. Es una herramienta fundamental en el estudio de la complejidad de los problemas y se aplica con mucha más facilidad a partir de la idea de que los problemas son cuestiones acerca de la pertenencia a un lenguaje, en lugar de cuestiones de tipo más general.

9.3. PASO 10: Máquina de estados finitos: Máquina de Turing MT

Las máquinas de estados finitos (conocidas como FMS *Finite State Machines* por su traducción al inglés), nos sirven para realizar procesos bien definidos en un tiempo discreto. Reciben una entrada, hacen un proceso y nos entregan una salida. Estas máquinas hacen una computación, en otras palabras, imaginar una máquina sea capaz de seguir una *secuencia finita* de pasos al introducir un conjunto de datos en ella, solo se puede leer un dato en cada paso que se realice, por tanto el numero de pasos a seguir esta dado por el numero de datos a introducir. Cada entrada diferente genera una salida diferente, pero siempre el mismo resultado con los mismos datos de entrada. Por lo tanto una computación es capaz de resolver un problema, si y solo si tiene una *solución algorítmica*, es decir, puede ser descrito mediante una *secuencia finita de pasos bien definidos*. Mediante una computación se puede encontrar soluciones a problemas que teóricamente tienen una representación algorítmica (para nuestro problema de tipo booleano), pero que pueden necesitar tal cantidad de recursos (factores como el tiempo y el espacio de almacenamiento) que desde el punto de vista práctico no se puede llegar a la solución.

Para resolver el problema se aplicara una secuencia finita de acciones no ambiguas sobre los datos de entrada para obtener unos datos de salida, es decir, consiste en aplicar un algoritmo que calcule la solución. Los datos de entrada (cadenas de símbolos) son generados por otra MT en diferente nivel y los sistemas lógicos combinacionales y secuenciales definidos.
Se determinara que el problema de secuenciación de operaciones es computable si el problema definido (la función) se puede calcular mediante un algoritmo. El modelo está definido por unas primitivas y unas reglas de aplicación de las mismas, que además se definirá su capacidad computacional.
Para poder utilizar un modelo computacional, lo primero que hay que hacer es representar simbólicamente los elementos del conjunto origen para que la máquina la entienda. En las máquinas que se van a considerar siempre se va a tomar como entrada una cadena de símbolos en el alfabeto de la máquina, la cual pertenece a un *lenguaje determinado*. Así el conjunto origen lo podemos definir como un lenguaje y cada uno de sus elementos se representa por las cadenas de entrada a la máquina de estados finitos (autómata finito).
El que se haga una u otra representación simbólica de la máquina es indiferente, pues siempre se va a poder convertir de una a otra representación. Lo importante es que, con la representación en cadena y lenguajes a los que pertenecen, podernos representar el problema de secuenciación de operaciones.

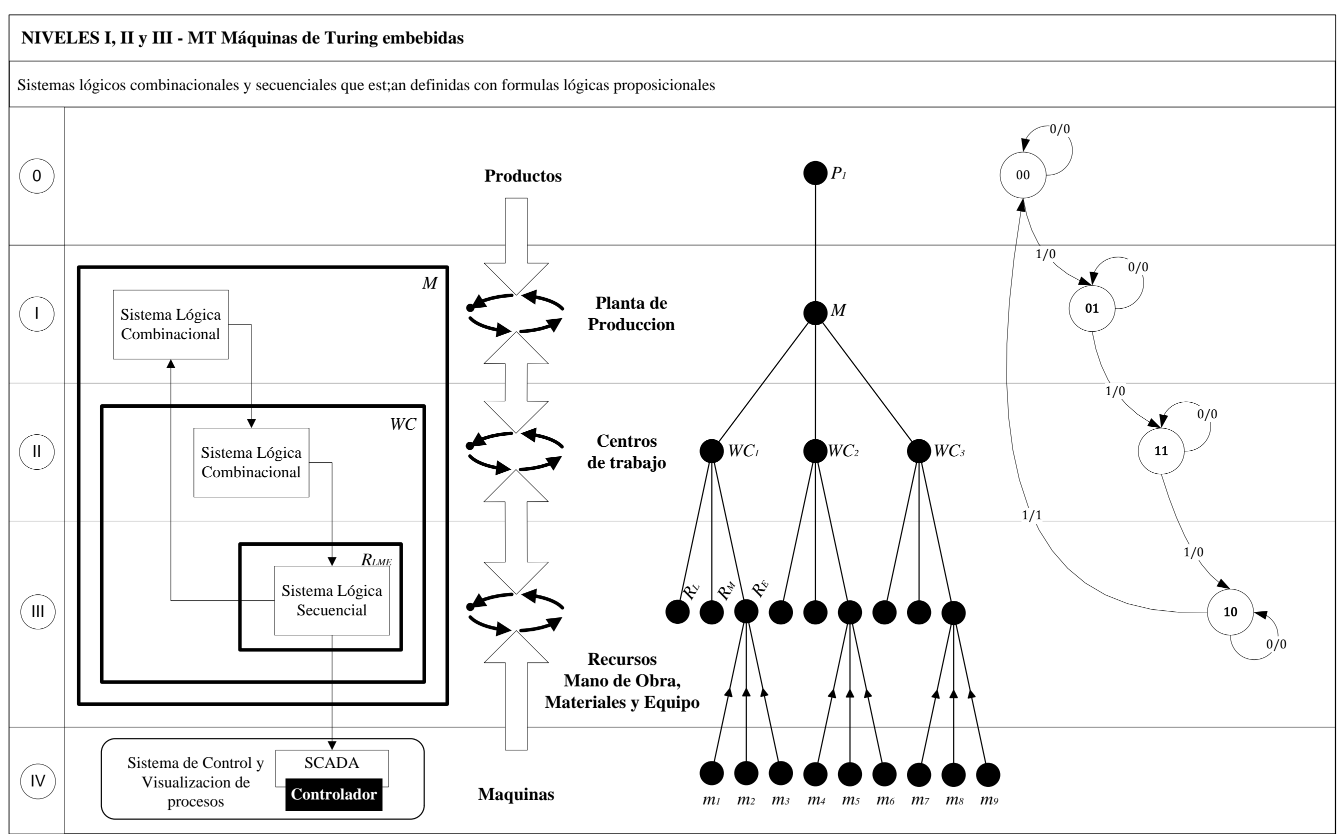
Se plantea el problema de secuenciación como un problema de decisión, y por ende el conjunto imagen (rango) solo tienen dos elementos dicotómicos posibles: $\{si, no\}$, $\{0,1\}$,… y se podrá demostrar que cualquier problema que no sea decisión se puede reformular para que así lo sea. Dado que se han codificado los elementos como cadenas que pertenecen a un lenguaje, el problema se puede reformular en términos de encontrar un algoritmo que sea capaz de decidir si una cadena a la entrada de la máquina pertenece o no al lenguaje.

Si el sistema planteado es consistente, es decir, probar si hay un procedimiento para determinar la verdad o falsedad de una proposición (*decidir si es verdadera o falsa*) ya que no existe un método general para resolver este problema.
A partir de la tesis de Church-Turing se desarrolla la teoría de computación y decibilidad (un problema que una máquina de Turing no es capaz de decidir, es un problema que no es *computable*). Para el problema de secuenciación es un algoritmo basados en expresiones lógicas proposicionales (definidas en clausulas) que es equivalente a una máquina de Turing. Al ser una máquina de Turing, el modelo computacional más poderoso que existe (no debemos olvidar que es conceptual, o real) se puede convertir computacionalmente en una máquina de Turing, y por lo tanto se puede describir el comportamiento del algoritmo que en él se ejecuta de una manera formal.

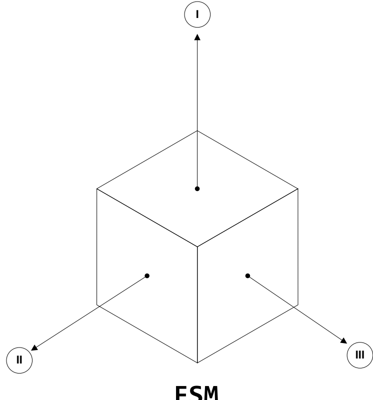
Conjunto recursivos y conjuntos recursivos enumerables. Como se sabe $\mathbb{N}$ es un conjunto infinito contable. Un subconjunto $S \subseteq X$ se dice que es recursivo si existe una función en $\mathbb{N}$ tal que
$f(x) = 1$ si $x \in S$ *y* $f(x) = 0$ si $x \notin S$
(en este caso se dice que la función es recursiva primitiva) es decir, el conjunto generado de los sistemas combinacionales y secuenciales es recursivo ya que se tiene el medio para determinar si el elemento pertenece o no a dicho conjunto. Como consecuencia, cualquier conjunto contable (finito o infinito) es recursivo. Tanto S como $\bar{S}$ (complementario de S, elementos que no pertenecen a S, teniendo en cuenta aquí que la complementación es una operación cerrada) son recursivos.
El conjunto S es *recursivamente enumerable* (RE) si existe una función recursiva por la cual los elementos de S son el conjunto imagen de la función. En otras palabras, el conjunto generado también es recursivamente enumerable si tenemos un medio de *enumerar los elementos* del mismo. Como se puede observar, este segundo tipo de conjuntos es menos restrictivo que el recursivo, porque se puede afirmar que el conjunto S (cadena de bits) es recursivo, también es recursivamente enumerable. De hecho, para que S sea recursivo, S y $\bar{S}$ han de ser recursivamente enumerables.

¿Qué tiene que ver esto con la máquina de Turing? Turing llego a la conclusión de que no se puede decidir la verdad o falsedad de una proposición matemática. Esto es lo mismo que decir que el conjunto de las proposiciones matemática (basada en un sistema axiomático) no es recursivo. Una máquina de Turing solo es capaz de reconocer *conjuntos recursivos enumerables.* En término de teoría de computación y lenguajes formales, el conjunto generado aquí es equivalente a lenguaje aceptado (conjunto de cadena que acepta la máquina de estados).

La descripción de las máquinas de estados está funcionalmente especificada por niveles; donde se encuentran jerárquicamente embebidas una dentro de otra de menor nivel. Para cada una de ellas se tiene un número de estados internos en los que se recuerda cierto tipo de información (propia para cada máquina) que se encontrará en un estado en concreto. La finalidad de evaluación de estados se efectuó partir de los conceptos desarrollados sobre conjuntos y funciones bases para la definición de los sistemas lógicos tanto combinacionales como secuenciales.



La representación para la máquina de estados, se expresara en los siguientes niveles definidos así:

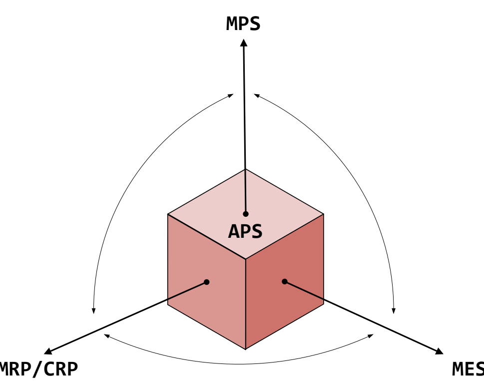


Nivel I. Se presenta el programa maestro de producción MPS (*master production schedule* por sus siglas en ingles), donde se toman decisiones de tipo estratégico, táctico y operativo, respecto a los artículos y cantidades que deben ser fabricados en el tiempo (horizonte de programación), determinando que debe hacerse y cuando. La validación para cada artículo se

presenta en términos de requerimiento del mismo para la planta de producción, y por lo tanto en caso afirmativo será **1** y en caso negativo será **0**.

Nivel 2. Se presenta el plan de recursos de fabricación (materiales) de y de capacidades (mano de obra y equipo) (MPS *manufacturing resource planning* y CRP *capacity requirements planning* por sus siglas en ingles), donde se toman decisiones de tipo táctico y operativo, respecto a la orden de fabricación planeada a través del MRP o explosión de piezas y suele ser presentada a la división de administración de la planta por la administración que control de operaciones de manera integral la división de fabricación y se determina primero la capacidad y carga a través de una serie de procesos. Los requisitos de capacidad la planificación se refiere a la planificación en la que se toma la carga de cada proceso de acuerdo con la orden de fabricación, se realizan los ajustes y luego se planifica el trabajo (secuenciación de cada proceso). La validación para cada artículo se presentara en términos del requerimientos en la planta de producción para los centros de trabajo necesarios en cada proceso, y por lo tanto en caso afirmativo será **1** y en caso negativo será **0**. La definición de este nivel se definió en detalle con los sistemas lógicos combinaciones definidos por la representación de la arquitectura del productos, a través de la *lista de materiales* y la *ruta de fabricación* extraídos de la gestión de datos para cada uno de los productos (PDM *product data management* por sus siglas en ingles).

Nivel 3. Se presenta el sistema de ejecución de manufactura MES (*manufacturing execution systems* por sus siglas en ingles), donde se toman decisiones de tipo operativo, respecto a la orden de fabricación planeada a través de los sistemas informatizados utilizados en la fabricación, para rastrear y documentar la transformación de materias primas en productos terminados. MES proporciona información que ayuda a los responsables de la toma de decisiones de fabricación a entender cómo se pueden optimizar las condiciones actuales en el control de piso en la planta para mejorar la producción. MES trabaja en tiempo real para permitir el control de múltiples elementos del proceso de producción (por ejemplo, insumos, personal, máquinas y servicios de apoyo en detalle). MES puede operar a través de múltiples áreas de funciones, por ejemplo: administración de definiciones de productos a lo largo del ciclo de vida del producto PDM, programación de recursos MRP/CRP, ejecución y despacho de pedidos, análisis de producción y gestión de tiempo de inactividad para la efectividad general del equipo (OEE) rastro básico para secuenciación de operaciones. MES crea el registro *as-built*, capturando los datos, procesos y resultados del proceso de fabricación. Esto puede ser especialmente importante en las industrias reguladas, tales como la fabricación de neumáticos - llantas, donde la documentación y la prueba de los procesos, eventos y acciones pueden ser necesarios. La idea de MES puede considerarse como un paso intermedio entre, por un lado, un sistema de sistemas de planificación de recursos empresariales ERP (*enterprise resource planning*, por sus siglas en inglés) y un control de supervisión y adquisición de datos (SCADA) o un sistema de control de procesos, por otro; Aunque históricamente, los límites exactos han fluctuado.

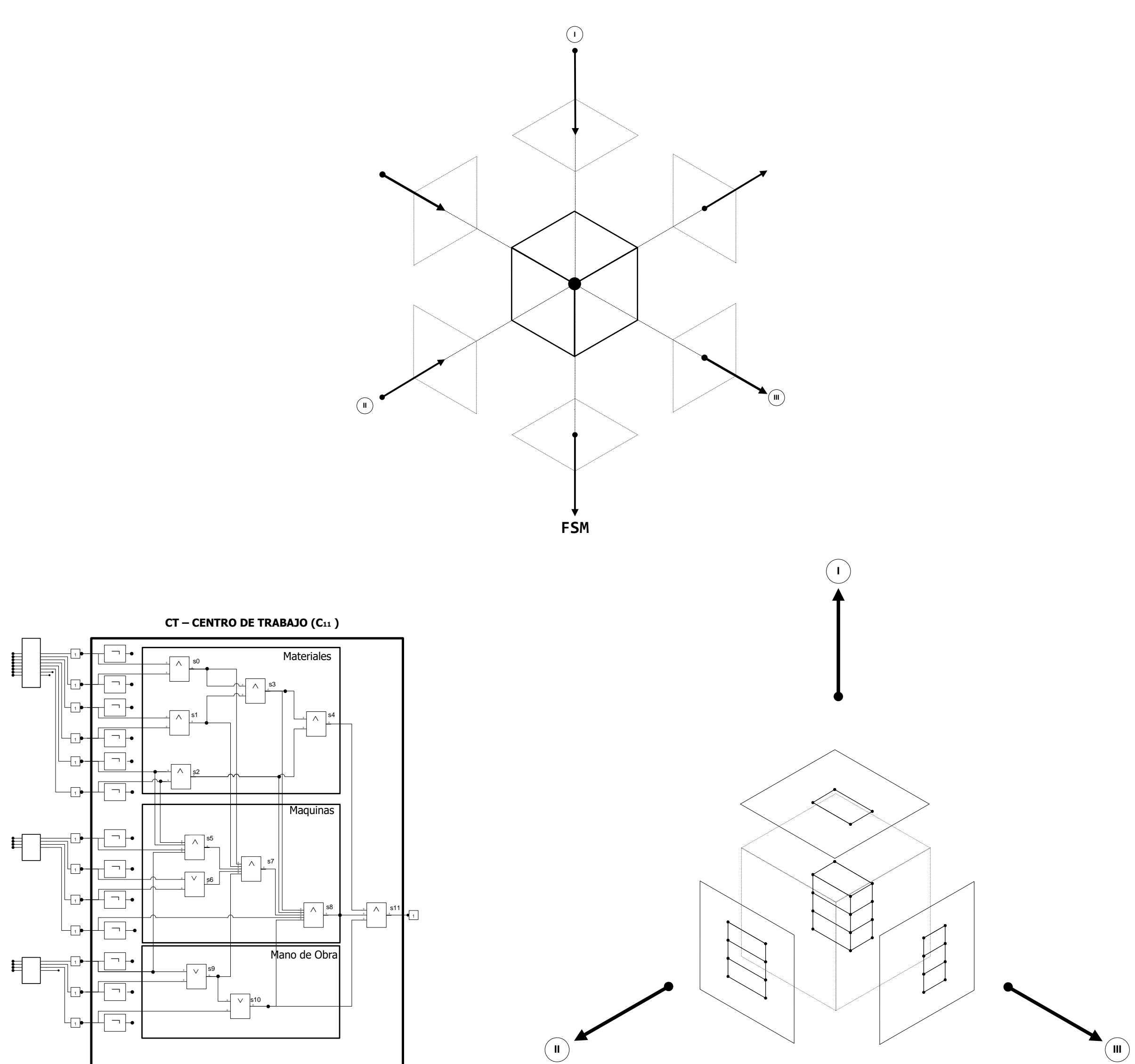


La FMS hará parte de la Planificación avanzada y programación APS (advanced planning and scheduling, por sus siglas en ingles), también conocida como fabricación avanzada y se refiere a un proceso de gestión de fabricación por el cual las materias primas y la capacidad de producción se asignan de la forma más confiable para satisfacer la demanda. APS es especialmente adecuado para entornos en los que los métodos de planificación más sencillos no pueden abordar adecuadamente los complejos equilibrios entre prioridades competitivas. La programación de la producción y secuenciación es intrínsecamente muy difícil debido a la dependencia (aproximadamente) del tamaño factorial del espacio solución sobre el número de artículos / productos a fabricar. Los sistemas tradicionales de planificación y planificación de la producción MRP/CRP (como la planificación de los recursos de fabricación) utilizan un procedimiento escalonado para asignar material y capacidad de producción. Este enfoque es simple pero engorroso, y *no se adapta fácilmente* a los cambios en la demanda, la capacidad de recursos o la disponibilidad de material. Los materiales y la capacidad se planifican por separado, y muchos sistemas no consideran la disponibilidad limitada de material o limitaciones de capacidad. Por lo tanto, este enfoque a menudo resulta en planes que no se pueden ejecutar. Sin embargo, a pesar de los intentos de cambiar al nuevo sistema, los intentos no siempre han tenido éxito, lo que ha llamado a la combinación de la filosofía de gestión con la fabricación.

A diferencia de los sistemas anteriores, se pretende incluir dentro del APS la máquina de estados finitos definida para planear y programar simultáneamente la producción basada en materiales disponibles, mano de obra y capacidad de la planta. APS se ha aplicado comúnmente donde una o más de las condiciones siguientes están presentes:

El software *advanced planning & scheduling* permite la programación de la fabricación y la optimización avanzada de la programación en estos entornos y se deja la puerta abierta a integrar la metodología propuesta a este tipo de solución. Ver recomendación Oracle PS.

La descripción del centro de trabajo como un sistema lógico combinacional, compuesto de tres subsistemas lógicos combinacionales y secucuenciales que representan los recursos descritos (definición de las clausulas de Horn para el problema de satisfacibilidad 3SAT de secuenciación), se procesan a través de las máquinas de estados finitos en cada nivel. La perspectiva para el análisis de mismo sistema lógico, se visualiza (proyección ortogonal) según el nivel indicado en cada una de las dimensiones representadas para cada una de las MT con respecto al centro de trabajo C_i, como se identifica en la figura.



Las principales características para dichas máquinas son las siguientes:

La máquina solo puede estar en un número finito de estados durante un periodo determinado. Tales estados se denominan estados internos de la máquina y en cada tiempo dado, el total de memoria disponible para la máquina es el conocimiento del estado interno en que se encuentra en ese momento.

La máquina aceptara como entrada solo un numero finito de símbolos, que se conocen colectivamente, como alfabeto ξ. En la descripción de la máquina, el alfabeto de entrada es $\{0,1\}$; cada estado interno reconoce a cada una de estas unidades según la cadena de entrada que represente la cinta.

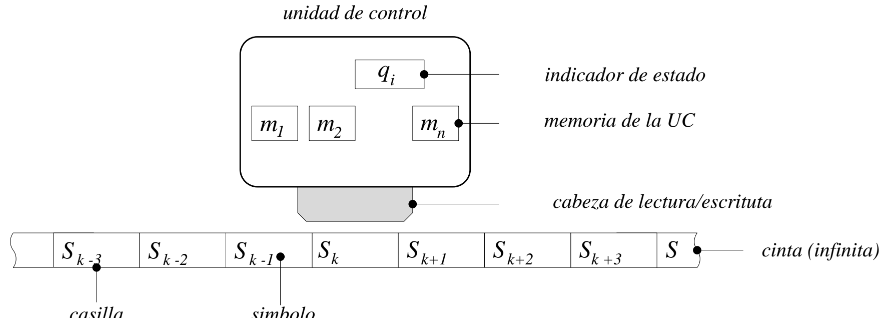
Se determina una salida o un estado siguiente por cada combinación de entradas y estados internos. El conjunto finito de todas las salidas posibles constituye el alfabeto de salidaζ.

Se supone que los procesos secuenciales de la máquina están sincronizados por impulsos de reloj independientes y distintos, y que la máquina opera de *forma determinista*, ya que la salida está determinada en todo momento por el total de entradas proporcionadas y por el estado inicial de la máquina.

Definición de la máquina de estados finitos – Máquina de Turing (MT). La máquina de Turing es conceptual, y se ideo antes de la aparición de los ordenadores modernos. En la metodología se incorpora ya que está basada en como un ser humano es capaz de realizar *computaciones* (razonamientos lógicos). Al realizar el supuesto cálculo de secuenciación de operaciones con lápiz y papel. El papel va a ser el soporte de almacenamiento y el lápiz es el mecanismo que realiza los cambios en aquel. Al principio el papel estará el blanco, donde se irá escribiendo sucesivamente el enunciado de lo que vamos a computar (secuenciar). En un momento del cálculo, la mente estará en una condición determinada (un estado determinado) de entre un conjunto de ellas (que se puede considerar finito). Es esta condición (estado) la atención estará centrara en uno de los símbolos escritos en el papel (este símbolo no tiene porque ser solo un carácter). Para avanzar en nuestro calculo, se hará alguna acción (leer o escribir otros símbolos) basándola en el estado y el símbolo en los que se está centrado, lo que llevará a una condición nueva (un nuevo estado). Cuando se considere que lo que se ha plasmado en el papel es la solución, se deja el lápiz en la mesa, es decir, paramos de calcular. Además, en nuestro avance también podremos memoriza algún valor intermedio de nuestro calculo para posterior uso durante la computación.

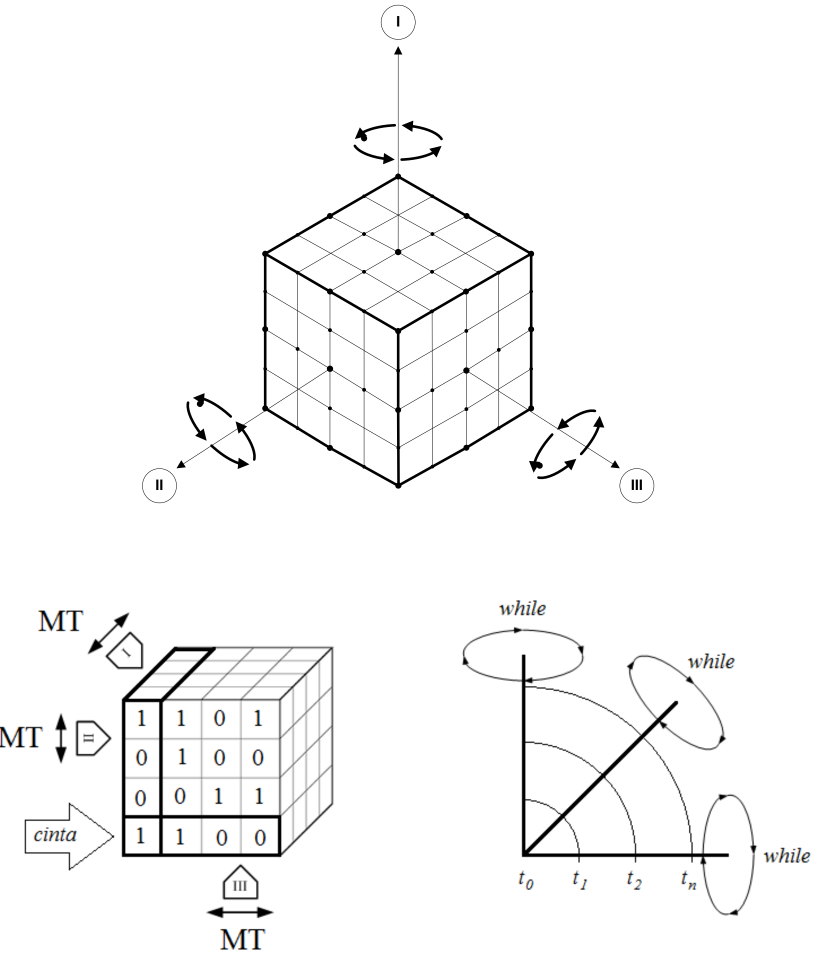
Aunque está máquina es descrita para realizar cálculos sobre números computables de manera que acepta la solución cuando está escrita, se va definir la máquina de Turing (MT) en los términos descritos: aceptación de una cadena que pertenece a un lenguaje (es otra forma de decir que puede representar el problema de secuenciación de operaciones).

Así la MT va a estar compuesta por una unidad de control (la mente – como un sistema experto) que va estar en uno de entre un conjunto finito de estados (q_i) y que va a tener asociada una cabeza de lectura/escritura (el lápiz). Como soporte de almacenamiento va a tener una cinta (para hacernos una idea podemos pensar en una cinta de papel, sobre la que apunta la cabeza de lectura/escritura. Esta cinta, que tendrá una longitud infinita por ambos lados (lo que hace que la MT tenga una capacidad infinita de almacenamiento), va a estar dividida en casillas o celdas (a una de ellas, y solo una, apunta la cabeza) y cada una de está casilla va a contener un símbolo (que puede ser más de un carácter). La unidad de control solo puede considerar, en cada paso de cálculo, el símbolo de la casilla a la que apunta la cabeza y el estado en el que se encuentre la unidad de control (UC). Si está unidad tiene la capacidad de almacenar un conjunto finito de símbolo de entre los que ya se visitaron durante el proceso de computación, entonces tendrá que considerar, además del estado en el que se encuentra, tanto los símbolos como al que apunta la cabeza.



¿Cómo opera la MT?. Primero hay que inicializar la máquina. Al principio, la cinta está en blanco. A continuación se introduce la cadena de entrada en la cinta, que será una secuencia de símbolos del alfabeto que entiende (salida de los sistemas lógicos combinacionales y secuenciales en cada nivel), se pone la unidad de control en el estado inicial y se posiciona la cabeza sobre la primera letra de la cadena introducida.

El cambio de condición (de estado), es decir el avance en el cálculo, es lo que vamos a llamar transición). Este, como ya hemos dicho, depende del estado actual y del símbolo al que apunta la cabeza. La transición se producirá en el siguiente orden:



La cabeza lee el símbolo de la casilla a la que está señalando. Teniendo en cuenta este símbolo y el estado en que se encuentra:

Cambiamos al siguiente estado. Si resulta que la operación es repetitiva, el estado será el mismo.
Escribirá un símbolo en la cinta (los símbolos que se pueden son los del alfabeto y algunos mas). Este también puede ser el mismo (esto es equivalente a uno operación de solo lectura).
Mover la cabeza a la izquierda o a la derecha (según se necesite) uno y solo una casilla.
Durante el cálculo se pueden producir dos situaciones:
Que se llegue a una solución, en cuyo caso la máquina se parara.
Que el problema no tenga solución, en cuyo caso la máquina puede pararse o entrar en un bucle infinito y no para nunca. Este comportamiento dependerá exclusivamente del problema que se esté computando.

En este punto el problema de secuenciación se considera ya como un problema de decisión, y pertenece a un conjunto recursivo, se le denominará decidible, o efectivamente solucionable.

Definición formal del la MT. El modelo de MT que se va a definir no incluye la posibilidad de almacenar un número finito de símbolos en la unidad de control y los símbolos van a estar compuestos por un solo carácter. Además, se utilizará la MT como una máquina que acepta una cadena perteneciente a un lenguaje dado (en el mismo sentido de los autómatas finitos y de pila). Ver definición de Máquina de Turing Anexo C.

MT no deterministica (MTN). Como ya se sabe, el no determinismo de una máquina puede ser de dos tipos: que la máquina no esté totalmente definida y que, para un estado y un símbolo de cinta, la máquina pueda tener más de un opción posible.
Ya se ha dicho que el modelo de la máquina original podría incurrir en el primer tipo de no determinismo y que la consideramos deterministica ya que se define con una función (solo una posible transición) y sin estados de rechazo para simplificar su representación. Por tanto, se consideran no deterministicas a las de segundo caso, en las que cada transición, para un estado y un símbolo de cinta, será un conjunto de movimientos posibles, en lugar de un solo movimiento. Así, una máquina de no determinista, MTN, para un mismo estado y mismo símbolo de cinta puede no tener transición definida, tener un único movimiento o tener la posibilidad de varios movimiento. En los dos primeros casos se comporta como la MT ya vista, mientras que cuando ocurre el tercero, la máquina sigue todos los posibles caminos en paralelo hasta un estado de aceptación. En paralelo significa que no se coge un camino y se sigue hasta que se rechace o se acepte, sino que se producen las configuraciones sucesoras de cada camino a la vez. Si uno de los caminos llega a un esfuerzo de rechazo, este camino se desecha y si es el último posible, entonces se rechaza la cadena.

Se dice que una MTN acepta un lenguaje, si existe algún camino que lleve desde la configuración inicial a una configuración de aceptación. Observar que se habla de posibilidad de existencia de un camino, no de producción por la función de transición.
La definición formal de la MTN, M será $M = (Q, \Sigma, \Gamma, \sigma, q_0, B, F)$
Donde σ es el conjunto de transiciones $\sigma \subset (Q' \text{x}\, \Gamma \;\rightarrow (Q \text{x}\, \Gamma \text{x}\, S)/$ para todo $Q \text{x}\, \Gamma \text{x}\, S$

Para representar el funcionamiento de M, se representan los posibles movimiento y los posibles caminos mediante el recorrido en amplitud de un grafo: los nodos serán las configuración y las aristas las transiciones. EL nodo raíz va a ser la configuración de inicio. Cuando la transición tenga varios posibles movimientos, cada una de las configuraciones resultantes será in nodo hijo del anterior. Los distintos caminos se seguirán en paralelo, desechándose aquel el que no haya transición posible. Se aceptara la cadena si llegamos por algún camino a un estado de aceptación. Esta aceptación será la de menor profundidad en el grafo.

Para obtener la MT determinista (MTD) M', que simule la MTN, M, se construye una MT de tres cintas (ya sabemos que es equivalente a una MT de una cinta) que va a simular el recorrido en anchura del grafo implícito asociado a las transiciones de M.

Definimos $\mathrm{M}' = (Q', \Sigma, \Gamma', \delta', q_0, B, F)$ donde

Q'es el conjunto de estados que resulte de la conversión

$\Sigma, q_0, B\ y\ F$ igual que en M.

$\Gamma' = \Gamma \cup Q \cup \{\#, @\}$ donde # es un separador de configuraciones y @ es un separador de configuraciones que indica que es la actual.

$\delta' \in \{Q' \text{x } \Gamma \ \rightarrow (\text{Q x } \Gamma \text{ x S}) /$ /para todo Q x Γ x S

La primera cinta contendrá la cadena de entrada y no se va a modificar. La segunda cinta contendrá una cola FIFO de configuraciones de la máquina que a uno no se han evaluado. La tercera cinta servirá para evaluar la configuración actual de M. El algoritmo de M' realizará la siguiente secuencia:

Copia en la cinta 1 la cadena de entrada y en las cintas 2 y 3 la configuración inicial de M, esto es, el contenido de la cinta junto con el estado inicial de M. En la cinta 2 se copiara precedida por la izquierda con @ (configuración actual).

(2) Para la configuración de la cinta 3, se comprueba si es de aceptación, en cuyo caso M' se detiene aceptando la cadena. Si no es de aceptación obtiene de la unidad de control las transiciones para está y guarda las configuraciones resultantes a; final del contenido de la cinta2, precedidas por # como separador de configuraciones.

En la cinta 2 desmarcamos la configuración actual (sustituimos @ por #) y marcamos como actual la siguiente configuración. Si no hay una configuración al final de la cinta, entonces es que la máquina debe parar rechazando la cadena.

Se borra el contenido de la cinta 3, y se copia en ella la configuración actual y se pasa al segundo punto anterior (2).

Como se ha dicho, el recorrido en anchura simula el funcionamiento de M, cuyo cas M' aceptara con la misma configuración que aceptaba M, por lo que hemos demostrado que los lenguajes aceptados por un MTN también lo serán por una MT determinista.

Una consecuencia inmediata de la definición de la MTN es que la MT determinista es un caso especial de está (conjuntos de transiciones de solo un elemento), luego los lenguajes aceptados por la MT determinista, también lo serán por la MTN. Por tanto, MTN y MTD aceptaran exactamente las mismas cadenas.

Para representar una MTN en una tabla de transiciones habrá que modificarla para que en cada entrada de la tabla, en lugar de una tupla con la transición, puede contener varias tuplas. Para hacerlo con el diagrama de transiciones, la diferencia estará en que un arista podrá tener más de una transición en su etiqueta (hay que etiquetarlas de forma distinta a la máquina de varias cintas, por ejemplo en líneas distintas). A la hora de simular el funcionamiento, lo haremos con el grafo antes descrito y realizando el recorrido en anchura.

MT con cinta semi infinitas. Esta máquina es una restricción a la MT original, de manera que ahora la cinta es infinita solo por el lado derecho, y no por los lados. La cadena se introducirá en las casillas más a la izquierda de la cinta y la posición inicial de la cabeza será la primera casilla de la izquierda. Otra restricción de esta máquina es que no está permitido escribir símbolos de casilla en blanco.

Para convertir una MT original M, en una MT de cinta semi infinita M', que describa formalmente es $\mathrm{M}' = (\mathrm{Q}', \Sigma, \Gamma', \delta', q_0', (\mathrm{B}, \mathrm{B}), \mathrm{F}')$
Lo primero que hará que hacer es partir la cinta de M en dos mitades por la posición indicada por la configuración inicial de M. Después se convierten las dos mitades en una cinta semi infinita de dos pistas, de manera que, si asociamos las posiciones de la cinta de M con un numero entero, la posta superior serán las posiciones positivas incluido el cero y el resto en la segunda pista, pero en orden invertido y precedidas por * como marcador de comienzo de cinta. Así, para la cinta de M con los subíndices como posiciones, X como símbolos de cintra y B como símbolo de casilla en blanco.

$$\ldots B_{-m} B_{-(m-1)} \ldots B_{-3} B_{-2} B_{-1} \mathrm{X}_0 \mathrm{X}_1 \mathrm{X}_2 \mathrm{X}_3 \ldots \mathrm{X}_{\mathrm{m}-1} \mathrm{X}_{\mathrm{m}} \ldots B_{(m+1)}$$

Dara como resultado, para M', una contad de dos pistas semi infinitas de las siguiente forma: El segundo paso es modificar M para convertirla en una máquina que no escriba casilla de espacio en blanco. Esto se hará añadiéndole un símbolo de cinta B' para simular los símbolos de casilla vacía, de manera que una transición de $M\ (p, X) \rightarrow (q, B, S)$, se convertirá en $(p, X) \rightarrow (q, B', S)$, lo cual no afectará al funcionamiento de M.

El alfabeto de M', será $\Sigma' = \Sigma$ x B, dado que los símbolos de cinta ahora son duplas: para cualquier elemento de $a\ \in \Sigma$, el equivalente en Σ'es (a, B).
Al principio la cinta en M' está vacía (las dos pistas). Se introduce la entrada a partir de la primera casilla, que serán pares del tipo (a, B) y se posiciona la cabeza sobre la primera columna. Ante de simular los estados de M, se debe marcar el principio de cinta con * en la primera casilla de la segunda pista. Para ello se define Q' como $\{q_0', q_1'\} \cup \{Q\mathrm{x}P\}$, donde q_0' y q_1' son los estados que marcan el principio de cinta, en la segunda pista, con *, Q es el conjunto de estados de M, y P es una memoria que indica si la máquina se encuentra en la pista superior o la inferiores; así el contenido de $P = \{S, I\}$ o, si son las iniciales de las palabras en ingles Up y Down , $P = \{U, D\}$.

El alfabeto de cinta de M', serán pares de símbolos de cinta de M al que se le añaden pare de la forma (X,*) para todo X de X de Γ. Formalmente $\Gamma' = (\Gamma \text{ x } \Gamma) \cup (\Gamma \text{ x } *)$

Una vez definidos los estados y los símbolos de cinta de M', se puede especificar q_0' y q_1' de manera que, al iniciar la ejecución, las dos primeras transiciones sean

$\big((q_0', B), (a, B)\big) \rightarrow (q_1', B), (a,*), D)$ que marca la primera casilla de la cinta inferior con *.

$\big((q_1', B), (X, B)\big) \rightarrow (q_0', S), (X, B), I)$ para volver a la primera casilla de la cinta superior.

Ahora ya se pueden simular las transiciones de M en estas dos pistas de M', de manera que, para una transición $(p, X) \rightarrow (q, Y, S) \in \delta$:

Todas las transiciones de M que se realicen a la derecha de la posición inicial de la configuración de inicio serán iguales a las de M', marcando la memoria como S y modificando solo el símbolo correspondiente a la primera pista. $\big((p, S), (Y, X)\big) \rightarrow ((q, S), (Z, X), S)$ para cualquier símbolo de cinta de M.

Todas las transiciones de M que se realicen a la izquierda de la posición inicial de la configuración de inicio lo serán en dirección contrario, marcando la memoria como I y modificando solo el símbolo correspondiente a la segunda pista $\big((p, I), (X, Y)\big) \rightarrow ((q, S), (X, Z), \bar{S})$ para cualquier símbolo de cinta de M, siendo $\bar{S}$ el complemento de S (si la transición de M es a la izquierda, en M' es a la derecha y viceversa).

El punto crítico de M' está cuando pasamos de una pista a otra, y esto lo hacemos utilizando el marcador*. Esta situación se da cuando M ha vuelto a la posición inicial, ya sea desde la izquierda (por la segunda pista) o desde la derecha (por la primera pista).

Para una transición de M de la forma $(p, X) \rightarrow (q, Y, D)$, en la posición inicial en la que se viene desde izquierda de la cinta M' estará en un estado (p, I) con símbolo actual $(X,*)$ para cualquier X de símbolos de cinta de M, y la transición cambiara el estado de la memoria a S y realiza la sustitución de la transición de M.

$$((p, I), (X,*) \rightarrow ((q, S), (Y,*), D)$$

Para la transición de M de la forma $(p, X) \rightarrow (q, Y, I)$, en la posición inicial en la que se viene desde la derecha de la cinta, M' estará en un estado (p, S) con símbolo actual $(X,*)$ para cualquier X de símbolos de cinta de M, y la transición cambiara el estado de la memoria a I y realiza la sustitución de la transición de M.

$$((p, S), (X,*) \rightarrow ((q, I), (Y,*), D)$$

Por último el conjunto de estados de aceptación F' de M' serán duplas del producto cartesiano F x P.

Podemos decir que M' simula el funcionamiento de M en todo momento, aceptando la cadena en los mismos estados que lo haría M y no escribiendo en ningún momento el símbolo de casilla en blanco (B, B) que es una de las restricciones que teníamos. Por tanto M' aceptara los mismos lenguajes que acepta M. Al ser una restricción sobre M, cualquier lenguaje aceptado por M' lo será también por M y por tanto aceptaran exactamente los mismos lenguajes.

Lenguajes decidibles y reconocibles por una MT. El hecho de que la MT que decida (que se pare) si la cadena de entrada pertenece o no al lenguaje depende de este mismo y no de cómo se defina el modelo de la máquina. Si identificamos las características del lenguaje del planteamiento del principio.

$L = \{\omega\$\omega / \omega \in \{a + b\} *\}$

Se puede llegar fácilmente a la conclusión de que el conjunto de cadenas aceptadas es un conjunto infinito contable (se puede definir una aplicación sobre el conjunto ℕ de los números naturales). Así mismo, si se analiza el lenguaje complementario al anterior (las cadenas infinito que no pertenecen al lenguaje), también se podrá demostrar que se trata de un conjunto infinito contable. Por tanto, se puede decir que este lenguaje es un conjunto recursivo o, simplemente, que el lenguaje es recursivo. En el planteamiento realizado, la MT siempre va a parar: siempre identifica si la cadena pertenece o no al lenguaje. Este comportamiento será siempre el mismo al margen de la variante de MT utilizada.

Pero, se recuerda, una MT también puede tener otro comportamiento: entrar en un bucle sin fin. Así, según el comportamiento que tenga una MT se puede afirmar que

El lenguaje es decidible si la MT que lo acepta siempre se para: en estado de aceptación si la cadena es válida y en cualquier otro estado si no lo es (se rechaza la cadena). Los lenguajes decidibles por MT son precisamente los lenguajes recursivos.

El lenguaje es reconocible si la MT que lo acepta se para cuando la cadena pertenece al lenguaje, pero si la cadena no pertenece al lenguaje, no se sabe si la máquina parara o si entrar en un bucle infinito. Los lenguajes reconocibles por MT son precisamente los lenguajes enumerados recursivamente (RE).

Dentro de la jerarquía de lenguajes, los aceptados por las MT se denominan estructurados por frases. Esto es así porque se su estructura gramatical no tiene restricciones (excepto que en el antecedente debe tener algún no terminal). Estos pueden ser decidible o solo reconocibles por la MT. Existen lenguajes que no son reconocidos por las MT, los cuales no tienen ninguna estructura gramatical. El resto de lenguajes inferiores en la jerarquía (los lenguajes y los independientes del contexto) son todos decidibles. Para demostrar esto último se tienen que construir una MT que simule el funcionamiento de autómata finito o uno de pila respectivamente.

Codificación de las MT. Para demostrar que existen lenguajes que no son ni siquiera aceptados por las MT, se va a proceder primero a codificar la entrada para que las cadena sean binarias, esto es que $\Sigma = \{0,1\}$ (ya se ha dicho que el formato que tenga la entrada es irrelevante, pues se puede codificar como se precise). Luego se codificara también la descripción de la máquina como una cadena binaria. Esta codificación no es única, pudiendo haber otras variantes. Un detalle que servirá más adelanta es que la longitud del código resultante de la máquina será finita, pues su descripción tiene un numero finito de elementos.

Se vuelve a la MT que acepta el lenguaje $L = \{\omega\$\omega / \omega \in \{a + b\} *\}$. Lo primero que se hace es listar los elementos del alfabeto de la máquina y codificarlos en binario: Cada símbolo se codificara con una cadena i ceros, donde i es la posición dentro de la lista ordenada, así, si $\Sigma = \{a, b, \$\}$.

Orden	símbolo	código
1	a	0
2	b	00
3	$	000

Una cadena se codificara concatenando los códigos de cada símbolo separados por un uno. está codificación es válida porque cada símbolo tiene al menos algún cero. Así la cadena $abaa\$abaa$ se codificara como 0100101010001010010 10
Ahora habrá que construir una MT, M’, cuya entrada fuese binaria, y que determinase que hay el mismo número de grupos de ceros en el mismo orden y a su vez con el mismo número de ceros (como se ve, el problema es totalmente distinto, aunque equivalente).

Una vez obtenida la MT con alfabeto de entrada binaria, que además solo tendrá un estado de aceptación (si tuviera más de uno, se realizarían los cambios en la máquina que correspondan) se va a codificar en binario, el modelo de MT definido por $M = (Q, \Sigma, \Gamma, \delta, q_0, B, q_a)$
Código de los estados. Poner los estados en una lista ordenada, de forma que el primero sea el estado de inicio y el segundo el de aceptación. Cada estado se codificara con una cadena de i ceros, donde i es la posición dentro de la lista ordenada.
Símbolos de cinta. Poner los símbolos de cinta en una lista ordenada, de forma que los tres primeros símbolos de la lista serán 0,1 y B, si estos existen en la máquina, y a continuación los demás. Cada símbolo de cinta se codificara con una cadena de i ceros, donde i es la posición dentro de la lista ordenada.
Sentido de movimiento. Codificar con un cero a la izquierda y con dos ceros a la derecha.
Transiciones. Con la codificación de sus elementos, su código resultante se obtendrá concatenando la representación de cada elemento e intercalando un 1 entre cada uno de ellos. está codificación es válida para cada elemento está representado al menos con un cero.
Función de transición. Se ordenan las transiciones (el orden puede ser arbitrario), y se concatenan, intercalando dos unos consecutivos entre cada una de ellas. está codificación es válida pues en cada transición nunca va a haber dos unos consecutivos.

La siguiente máquina, que acepta cadenas solo unos y cadenas con solo ceros:
$M = (\{q_{0,} q_{1,} q_{2,} q_a\}, \{0,1\}, \{0,1, B\}, \{[q_{0,} 0) = (q_1, 0, D)], [(q_{0,} 1) = (q_2, 1, D)], [(q_1, 0) = (q_{1,}, 0, D)], [(q_2, 1) =$
(q2,1,D)],[q1,B=(qa,B,D)],q2, B=qa,B,D,q0,B,qa)
Primero hacemos una lista ordenada tanto de los estados como de los símbolos de cinta (el orden de los estados no es único):

Estado	código
q_0	0
q_a	00
q_1	000
q_2	0000

Símbolo	
0	0
1	00
B	000

Dada está codificación de elementos, la máquina quedara codificada:

$$M = 010100010100\ 11\ 010010000100100\ 11\ 00010100010100\ 11\ 000010010000100100$$
$$11\ 00010001001000100\ 11\ 0001001001000100\ 11\ 000010001001000100$$

Con este planteamiento se puede observar que está es una de las muchas posibles codificaciones para la misma máquina M. De hecho, el numero de codificaciones para una máquina M que acepte alguna cadena será, si está tiene un estado de aceptación y en la cinta tiene al menos los símbolos 0,1 y B.

(N_o estados -2)! × (N_o símbolos de cinta-3)! × (N_o de transiciones)!

¿Cómo se identifica cada codificación? Porque plasmar cada cadena binaria es tedioso además de ilegible (ya sea de entrada o la codificación de una máquinas). Dado un alfabeto $\Sigma = \{0,1\}$, el conjunto de todas las cadena binarias posibles (Σ^*) es infinito contable ya que podemos asociar un numero natural a cada una de las cadenas, es decir, existe la función ($f: \Sigma^* \to \mathbb{N}$). Así, se ordenan todas las cadenas de $\Sigma *$ primero por longitud y dentro de está por orden binario, asignado en orden, a cada una, un numero natural. Así dada la cadena ω_i esta será la i-esima cadena de Σ^*. Las primeras cadenas corresponderán al número:

λ^9 =1; 0=2, 1=3; 00=4; 01=5; 10=6; 11=7; 000=8; 001=9

Para obtener la cadena binaria ω a partir del valor de i, decimos que la cadena i-esima es el valor decimal de la cadena obtenida de la concatenación de 1 y ω

$$i_{(10} = 1\omega_{(2}$$

Dado cualquier MT se codifica con una cadena binaria ω_i, entonces se puede asociara a cada codificación un numero natural, de manera que podamos hablar de la MT i-esima, M_i. Así mismo se puede asociara un numero natural a cada una de las cadena de entradas de la máquina. Se puede observar que, al ser cadenas binarias, una cadena de entrada cualquiera con número de orden n, será la codificación de la máquina M_n.

Límite de las máquinas de Turing. Si se comparan todas las cadenas de Σ^* con las codificaciones de MT, se puede observar que hay muchas cadena que no se corresponden con ningún código correcto de la descripción de una MT. En este caso se considera que la cadena es el código de una MT que no hace nada (ya se dijo que era un estado sin transiciones), es decir $L(MT) = \emptyset$. De esta forma tenemos tantas MT posible como números naturales, luego podemos decir que el número de MT es infinito contable.

Dado que conjunto de todos los posibles subconjunto de cadenas (todos los posibles lenguajes) es el conjunto potencia de Σ^*, del que se sabe que es infinito incontable, se llega a la conclusión de que hay más lenguajes que MT posibles, es decir, habrá lenguajes que una MT no podrá reconocer (aceptar).

En conclusión, se va a tener un problema decidible (recursivo), problemas que no son decidibles, aunque aceptables por MT (enumerables recursivamente, RE) y problemas indecidibles para los que no existe una MT capaz de aceptarlos (no RE).

Para demostrar la afirmación de que más lenguajes que máquinas de Turing, se va a utilizar el método de diagonalizacion de Cantor. Este matemático lo utilizado para demostrar que el cuerpo de los números reales era un conjunto infinito incontable. Un sencillo ejemplo: Sean tres cadenas de números decimales de tres dígitos, las cuales se organizan en una lista. Para encontrar una cadena de tres dígitos que no esté en la lista utilizamos el método de la diagonal como se describe a continuación.

Suponer el conjunto $X = \{0,1,2,3,4,5,6,7,8,9\}$ y dado un digito y_i, el conjunto Y_i ; el conjunto complementario de Y_i será $Y_i' = X - Y_i$

Se coge el primer digito de la primera cadena, y_1 para la nueva cadena no listada se elige un digito cualquiera de Y_1'. De esta manera se asegura que la nueva cadena no puede ser igual a la primera.

Se coge el segundo digito de la segunda cadena y_2: para la nueva cadena no listada se elige un digito cualquiera de $Y_2{}'$. De esta manera se asegura que la nueva cadena no puede ser igual a la primera.

Se coge el digito i de la cadena i-esima para la nueva cadena no listada se elige un digito cualquiera de $Y_i{}'$. De esta manera se asegura que la nueva cadena no puede ser igual a la i-esima.

De esta forma se ha obtenido una cadena de 3 dígitos que no es igual a ninguna de las existentes en la lista.

Cantor demostró que el intervalo (0,1) es incontable, extrapolado, posteriormente, su resultado a todo el conjunto R: Al suponer cierto para el intercalo (0,1) se tienen la función $f: (0,1) \to \mathbb{N}$, se tendrán n números con n decimales, todos distintos entre sí. Si se aplica el método de diagonalización a la parte fraccionaria, encontraremos al menos un numero en el intervalo (0,1) que no estaban en la lista; de esta manera la afirmación de que existe f es falsa, y por lo tanto el conjunto R es infinito incontable.

Utilizando la misma técnica, se va a encontrar un lenguaje, denominado de diagonalización L_d, que no pueden ser aceptados por ninguna MT, esto es, que no es RE, de manera que se demostrara que hay más lenguajes que máquinas de Turing.

Para ello listaremos en una columna todas la MT, que hacemos asociado con un número natural. El resto de columna (ω_j) son cada una de las cadenas de $\Sigma *$, asociadas también con un número natural. Así una porción de la lista podría quedar:

ω_j \ M_i	…	44	45	46	47	…
…	…	…	…	…	…	…
44	…	0	0	1	0	…
45	…	1	1	1	0	…
46	…	0	0	1	1	…
47	…	0	0	0	0	…
…	…	…	…	…	…	…

En este momento se está en condiciones de definir lo que es una secuencia o vector característico de un lenguaje: será una cadena de unos y ceros, donde la posición j de la cadena resultante, iniciando desde la izquierda, será:

1, si la cadena ω_j pertenece al lenguaje $L(M_i)$

0, si la cadena ω_j no pertenece al lenguaje $L(M_i)$

Así, cada fila de la tabla el vector característico del lenguaje aceptado por la máquina correspondiente, donde se reflejan las cadenas que pertenecen al lenguaje de cada máquina.

Suponer el lenguaje compuesto por cada una de las cadenas que representan el código de la M_i cuya codificación es aceptada por ellas mismas. Como se pueda observar, el vector característico de este lenguaje se corresponde con la diagonal de la tabla (la M_{45} y M_{47} aceptan ω_{45} y ω_{46} es decir, aceptan su propia codificación, mientras que M_{45} y M_{47} no). Se puede demostrar que la máquina que acepta este lenguaje existe, siendo el vector característico de alguna M_k de la lista.

Para obtener el lenguaje de diagonalizacion, L_d se procede con el método descrito anteriormente (obtiene el complemento del anterior), complementando cada una de las entradas en la diagonal de la lista. De esta forma obtendremos el vector característico de L_d.

Si ω_j pertenece a $L(M_j)$ entonces ω_j no pertenece a L_d

Si ω_j no pertenece a $L(M_j)$ entonces ω_j pertenece a L_d

Es decir L_d es el lenguaje compuesto por cada una de las cadenas que representan el código de la M_i cuya codificación no es aceptada por ella misma.

Con este método se ha encontrado un conjunto de cadena que no es el vector característico de ninguna MT, es decir, no hay ninguna MT que acepte L_d. Para aclarar esta afirmación, suponer que existe una MT, M_k que acepta e lenguaje L_d. Entonces está máquina debería estar en la lista. Sin la cadena ω_k es el código de M_k pueden ocurrir dos cosas:

Que ω_k pertenezca a L_d. En este caso xxx debería rechazar la cadena. Sin embargo, se ha demostrado que L_d contiene ω_k si existe alguna máquina que no acepte su propia codificación, debiendo M_k aceptarla. Luego se produce una contradicción de nuevo.

Como se produce la paradoja de que ω_k debe pertenecer y no pertenecer a la vez a L_d se concluye que no hay ninguna MT que acepte este lenguaje.

Conjuntos recursivos y recursivos enumerables. Ya se ha comentado que, para los lenguajes recursivos, la complementación es una operación cerrada. También se ha dicho Que si en lenguaje L es recursivo también lo es L. Para demostrar esto último, basta con construir una MT que decida L:

Sea $L(M)$ un lenguaje recursivo decidido por M, que será una máquina con una cinta, un estado de aceptación y otro de rechazo. Se puede construir M' que decida $\overline{L}(M)$ a partir de M, cambiando la característica de aceptación a rechazo y viceversa. De esta forma, cuando M acepta la cadena, M' siempre parara rechazándola y como M siempre para rechazando la cadena, M' parará aceptándola.

Anteriormente también se afirmo que para que un lenguaje L sea recursivo, tanto L como L han de ser recursivamente enumerables. Para demostrar esto, se construirá una MT, M_r de dos cintas, donde la primera sirve para simular M, que acepta $L(M)$, y la segunda sirva para simular M', que acepta $L(M)$. M_r simulara en paralelo M y M' (ejecutara alternativamente transiciones de una y otra máquina). Si M acepta la entrada, entonces M_r acepta la entras. Sin M' acepta la entrada, entonces M_r rechaza la entrada. Así M_r siempre parará ante cualquier entrada al problema. Esta demostración también es válida si se parte del hecho de que si $\overline{L}$ es recursivo, tanto L como $\overline{L}$ han de ser recursivamente enumerables.

Hasta ahora se han demostrado los casos en los que el lenguaje y su complementario son simétricos (decidibles por una MT). En este caso se puede considerar que el problema es el mismo.
Sin embargo, también hay casos en los que no lo son, como se demostró en con el lenguaje de diagonalizacion L_d y su complementario. En este caso cada lenguaje es un problema diferente. Es más, el complemento a L_d es RE (es aceptado por alguna M_k), mientras que L_d, de lo que se demuestra que el complemento para RE no es cerrada.
En resumen, podemos relacionar un lenguaje y su complementario de la siguiente manera:
L y $\overline{L}$ son recursivos
L y $\overline{L}$ son no RE. Por tanto no se pueden aceptar con una MT.
L es RE pero $\overline{L}$ es no RE
$\overline{L}$ es RE pero L es no RE

L_d y su complementario están en el tercer caso (recordar que se ha complementado la diagonal de la tabla).

Máquina universal de Turing. Al determinar si la MT cualquiera acepta una cadena de entrada. Para ello se va a utilizar como entras a la MT, que resuelve este problema, el par (M, ω): la codificación de la máquina M que acepta la cadena ω. Este par, codificado como se ha hecho antes, es la concatenación del código de M y la cadena ω separados por tres unos consecutivos. Esta aceptación es correcta dado que la codificación de la máquina terminara siempre con algún cero, y además no va a tener de dos unos consecutivos; así mismo la cadena ω no va a tener nunca dos unos consecutivos.

El lenguaje que acepta está máquina será el conjunto de todas las codificaciones de máquinas de Turing junto con todas las cadenas que aceptan; así, el problema se reduce a resolver si un par (M, ω): pertenece a este nuevo lenguaje, que llamamos universal.
$L_u = \{(\text{M}, \omega) /$ M es una MT que acepta ω $\}$

La MT, U, que acepta L_u se la llama máquina de Turing Universal, y recibe este nombre debido a que es una máquina que es capaz de simular cualquier otra MT, M, con la cadena entrada, a partir de la descripción codificada de la misma. Este es el motivo por el que dice que U es una máquina programable, pues a la entrada tiene el código de cualquier MT con sus datos de entrada.

Esta máquina, podrá tener el siguiente comportamiento, mientras simula M:
Parara aceptando si M para aceptando o
Parara rechazando si M para rechazando o
Entrara en un bucle si M entra en u bucle. Este hecho, el de no saber si la máquina parara con una entrad determinada, a veces se le llama problema de la parada.
Por este comportamiento ultimo, es por el que, intuitivamente, se puede observar que L_u es indecidible. Sin embargo, este lenguaje si es RE ya que L_u es precisamente lo que representa toda la lista de máquinas con todas las cadena que acepta cada una. Luego se puede construir una MT, U, de tres cintas que simule cualquier otra MT, M, con una cadena de entrada determinada:
Cinta 1. Se almacena el par (M, ω)
Cinta 2. Simula el contenido de la cinta de M (en el mismo código binario utilizado para codificar la máquina, es decir, un cero para el cero, dos ceros para el 1, tres ceros para B,….). El código de cada símbolo se precede con un uno. Esto es correcto porque ningún símbolo de cinta de M se va a codificar con un uno.
Cinta 3. Contendrá el estado en que se encuentra M. Al igual que antes, con el mismo código que la máquina.

Esta máquina tendrá el siguiente comportamiento:

Primero se comprueba el tipo de máquina que se a simular. Si el condigo introducido no se corresponde con el código de alguna máquina, la máquina parar rechazando la cadena (ya se dijo que se considera que un código mal firmado representa una MT con un estado y ninguna transición).

Se codifica la cadena de entrada y se introduce en la cinta 2. Por ejemplo, si la cadena de entrada fuera 01101, en la cinta 2 se almacenaría 1010010010100. Un caso especial será el espacio en blanco: U considerará que la casilla en blanco de la cinta 2 es el código de casilla en blanco de M. De esta manera en la cinta nunca habrá tres ceros seguidos, pero si una casilla en blanco. Colocamos la cabeza de la segunda cinta en la primera casilla en la empieza el código de ω.

Se almacena en la cinta 3 el estado inicial de M.

Se busca en la cinta 1 la transición correspondiente al estado de la cinta 3 y el símbolo actual de la cinta 2, y se aplica:

Se sustituye el estado de la cinta 3 por la codificación del siguiente estado (el que marca la transición).

Se sustituye el código del símbolo de cinta actual de M en U por el que marque la transición. Cuando hay una sustitución puede ocurrir que el código del símbolo nuevo tenga una mayor longitud que el código del símbolo a sustituir, en cuyo caso hay que desplazar, en la cinta 2, el resto de la cadena a la derecha las casillas que sean necesarias (igualmente, si el código del símbolo nuevo es de menor longitud, habrá que desplaza a la izquierda el resto de la cadena, y evitar así espacios en blanco). Estos desplazamientos se podrían evitar si se codifican los estados y los símbolos como cadenas binarias de longitud fija (si x es el número de estados/simbolos, $x \geq 2^n$ donde n es el número de bits para un estado/símbolo).

Se realiza el movimiento que marque la transición, posicionando la cabeza de la cinta 2 al principio del código del siguiente símbolo (en el 1 que lo marca).

Si no hay transición definida en M, está para, luego U se para sin aceptar.

Si M acepta ω, entonces U terminara en su estado de aceptación.

Una vez que se ha demostrado que hay una MT que simula cualquier MT que acepta una cadena, por tanto que es RE, ahora hay que demostrar que L_u es indecidible, es decir que no hay una MT para el lenguaje $\overline{L}_u$. De forma intuitiva se puede observar que $\overline{L}_d$ es un subconjunto de L_u. Por esto mismo, L_d estará incluido en $\overline{L}_u$; sabemos que no hay ninguna MT que acepte L_d, por lo tanto no habrá ninguna máquina que acepte $\overline{L}_u$.

Para demostrar está afirmación, se supone que L_u es recursivo; esto es lo mismo que suponer que existe una MT, M_k que con una entrada (M, ω) acepta L_u, aceptando si la cadena esa ene le lenguaje y rechazado si no lo está.

Tomando como base M_k, se va a construir una MT $\overline{M}_k$ que acepte $\overline{L}_u$ con una entrada (M, ω) igal que hicimos antes: Cuando M_k acepta la cadena de entrada, $\overline{M}_k$ la rechazara y cuando M_k rechace la cadena, $\overline{M}_k$ la aceptara. A continuación se modifica $\overline{M}_k$, para obtener $\overline{M}_k$, que en lugar de tener como entrada el par (M, ω) tenga el par (M, M), es decir, una M que acepte su propia codificación. Así $\overline{M}_k$ rechazara el par (M, M) si M acepta su propia codificación y aceptara el par (M, M) si M rechaza su propia codificación. Como se puede observar, $\overline{M}_k$ es la máquina que acepta L_d la cual ya se demostró que no existía. Es fácil ver que si se introduce en $\overline{M}_k$, su propia codificación como entrada, debería rechazar el par $(\overline{M}_k{}', \overline{M}_k{}')$ si $\overline{M}_k{}'$ acepta su propia codificación (lo que es una contradicción} y por otro lado, debería aceptar el par $(\overline{M}_k{}', \overline{M}_k{}')$ si $\overline{M}_k$ rechaza su propia codificación (lo que también es una contradicción). Al demostrar que $\overline{M}_k{}'$ no existe, se está demostrando que $\overline{M}_k$ tampoco existe y por tanto la afirmación de que existe $\overline{M}_k$ es falsa. Es decir, el par $(\overline{M}_k{}', \overline{M}_k{}')$ debe pertenecer y no pertenecer a $\overline{L}_u$, con lo que concluye que no hay una MT que acepte $\overline{L}_u$.
Como corolario a toda está demostración, y al ser $\overline{L}_d$ un subconjunto de L_u, se pude afirmar que, como L_u es RE, cualquier subconjunto suyo también lo es, entre ellos $\overline{L}_d$.

Limite de los computadores actuales. Se ha demostrado que hay infinitos problemas para los que no se va a tener una MT que los resuelva (ni siquiera los reconozca). También se ha formulado la tesis de Church-Turing, que determina el límite de los computadores actuales. Si esto es así ¿Qué tipo de máquina es un ordenador actual? Como ya se habrá supuesto, es una máquina programable en el sentido de la máquina Universal de Turing. Para demostrar está afirmación, se procederá como siempre: primero simular una MT con un computador y segundo simulara un computador con una MT.

Para el primer paso, si se tiene una MT con cinta ser infinita, la unidad de control se simula por el procesador, cada uno de los símbolos de cinta se codifican como la información que se permiten guardar en la memoria del computador. Los estados, al ser finitos, se pueden guardar en una tabla (en realidad seria el estado del procesador, que incluye el contador de programa) y las transiciones se simulan mediante un programa. La cinta se simularía con la memoria. El inconveniente de que un computador tiene una memoria finita se soluciona suponiendo que siempre es posible añadirle, de forma indefinida, más memoria. De esta forma un computador simularía el funcionamiento de una MT cualquiera.

El segundo paso, pasar de un computador a una MT de varias cintas, es también casi inmediato. El programa y los datos de entrada al programa que se introducen en la memoria antes de ejecutarse en el ordenador se pueden codificar como un par (M, ω) donde M es la secuencia de instrucciones que hay que computar y ω son los datos de entrada al programa, almacenándose en primera cinta. En la segunda cinta almacenaremos las posiciones de memoria a las que va a acceder el programa para su lectura/escritura. En la tercera cinta almacenaremos el

contador de programa (el estado), que es el determina la dirección de la computación. De esta forma la MT simularía el funcionamiento de un computador muy básico. La memoria cache, los registros, la UAL son aplicaciones que se han hecho para mejorar la eficiencia.
Sin embargo, un computador admitirá solo un subconjunto de L_u exactamente el conjunto de aquellas para la que el par (M, ω) sea la codificación de la MT que acepta un lenguaje recursivo, esto es, solo aquellos problemas para los que pueda existir un algoritmo.

Un problema real. La verificación de una programa ¿es decidible?. Podemos enunciar este problema de otra manera ¿existe alguna MT que decida si su propia definición es válida o no? Por la demostración de la indecibilidad de L_u podemos decir que no.
Otro problema que corrobora está afirmación es que la lógica de predicados en la que se basa la verificación formal es decidible.

Determinar la indecibilidad. Método de reducción. Hasta ahora se ha utilizado un método laborioso para demostrar que un lenguaje es indicidible. Una manera más simple de hacer esta operación es utilizando el método de reducción, el cual está implícito en nuestra manera de pensar a la hora de solucionar ciertos problemas: dado un problema P1, este se reduce a solucionar P2. Es decir, si solucionamos P2, tenemos solucionado P1. De esta manera hemos convertido un problema en otro.
Un ejemplo:
P1. Hay hambre en el mundo
P2. Hay que redistribuir parte de la riqueza

El problema del hambre en el mundo se reduce a redistribuir riqueza. P2 es un problema mucho más general, pues además de solucionar el hambre también solucionaría otros más. Hay que hacer que este método no hace referencia a la manera en la que se soluciona P1 o P2, sino que determina como la solución de P2 conduce a solucionar P1. También se puede ver que P2 es un problema más general que P1. De hecho este método no funcionaria en el sentido inverso: siempre se reduce un problema a otro más general.

10. REPRESENTACION GRAFICA

10.1. PASO 11. Secuenciación

Un sistema lógico proposicional de computación en tiempo real propuesto dará a los planeadores la flexibilidad adicional de replantear fácil y rápido y hacer modificaciones de último minuto en la programación, lo cual los capacita para trabajar con órdenes reales (si están disponibles), más que con datos pronosticados.

Figura 36. Diagrama de operaciones – Salida del sistema lógico proposicional

Fuente: Autor

La siguiente tabla es una representación a tres niveles de la unidad de control en el formato de salida por terminal *.txt archivo de texto, donde se presenta la secuencia de operaciones por centro de trabajo.

```
<1> 1    10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 2    10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 3    10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 4    10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 5    10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 6    10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 7    10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 8    10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 9    10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 10   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 11   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 12   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 13   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 14   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 15   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 16   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 17   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 18   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
```

```
<1> 19   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 20   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 21   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 22   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 23   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 24   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 25   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 26   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 27   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 28   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 29   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 30   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 31   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 32   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 33   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 34   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 35   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 36   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 37   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 38   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 39   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 40   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 41   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 42   10000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 43   00000000 10000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 44   00000000 10000000 01000000 00000000 00000000 00000000 00000000 00000000
<1> 45   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 46   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 47   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 48   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 49   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 50   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 51   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 52   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 53   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 54   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 55   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 56   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 57   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 58   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 59   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 60   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 61   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 62   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 63   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 64   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 65   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 66   00000000 10000000 00010000 00000000 01000000 00000000 00000000 00000000
<1> 67   00000000 10000000 00000010 00000000 01000000 00000000 00000000 00000000
<1> 68   00000000 10000000 00000010 00000000 01000000 00000000 00000000 00000000
<1> 69   00000000 10000000 00000010 00000000 00010000 00000000 01000000 00000000
<1> 70   00000000 10000000 00000010 00000000 00010000 00000000 01000000 00000000
<1> 71   00000000 10000000 00000010 00000000 00010000 00000000 01000000 00000000
```

Sobre la línea de tiempo, se considera que la disponibilidad para cada centro de trabajo tendrá que ser asignada a través de las diferentes órdenes de productos en proceso - WIP. Estos procesos intermedios que se realizan en la medida que avanza el proceso de producción; generan productos que igualmente son considerados en la programación a corto plazo, dado que pueden ser resultado de centros de trabajo que funcionan en serie o en paralelo, y deberán ser sincronizados en función del tiempo t con las salidas y entradas necesarias para mantener el flujo de materiales a través del proceso para cada uno de los productos finales $p_1, p_2, \ldots., p_{10}$ definidos según lo los requerimientos de la ingeniería de producto.

El planteamiento nace de la necesidad de contar con información lo más actualizada posible, para modificar desde la administración de planta el escenario de planeación contando con cada uno de los recursos disponibles (mano de obra, equipo y material) disponibles por centro de trabajo en la medida que se desarrollan las actividades de producción.

Los supervisores y planeadores de producción se apoyan en diagrama de Gantt para detectar el *avance* de los centros de trabajo en comparación con su programa. Se desarrollará dentro de la metodología de investigación aplicada el planteamiento de una salida del sistema lógico proposicional para obtener este diagrama a través de funciones booleanas, para cada centro de trabajos c_i relacionados a cada producto p_i para luego ser combinados en uno solo o individual y efectuar el análisis de los trabajos en curso en forma continua a través del tiempo, y mitigar la generación de informes con información desactualizada como lo expresa la teoría de programación de operaciones.

Aplicaciones tecnológicas. A través de la integración sin fisuras con soluciones de planificación de producción (PS) de Oracle para este desarrollo y sistemas de Gestión empresarial (ERP) con soluciones MES.

10.1.1. Algoritmo

El hecho de que un problema decidible (un lenguaje decidible) se pueda resolver con una MT que siempre se para, constituye un punto importante en la teoría de la decibilidad. Es más, una MT que resuelve un problema decidible es la definición formal de algoritmo. De esta forma se puede afirmar que si, para un problema existe un algoritmo que lo resuelve, entonces se dice que el problema (el lenguaje que lo representa) es decidible por una MT.
Precisamente aquí es donde empiezan los problemas, ya que hay problemas (lenguajes) para los que no hay un algoritmo que los solucione (que los decida) e incluso hay otros para los que no existe ni siquiera una MT que los

reconozca (luego se verá porque). Una MT que solo reconozca un lenguaje (un lenguaje RE), soluciona un problema a medias, por lo que realmente no será útil en la resolución de un problema.

Un algoritmo es el procedimiento seguido para resolver un problema por una MT que siempre se para, y ya se ha visto que estos lenguajes son llamados decidibles. Todo lenguaje para el que no haya está MT, se dice que indecidible (ya sea un lenguaje reconocible a o no una MT). Cualquier programa no estructurado puede implementarse en un lenguaje estrictamente secuencial que contenga las siguientes instrucciones: *secuencia (;), condicional (if)* e *iteración (while).*

Muestra formal del funcionamiento de la máquina de Turing. Se muestra con el planteamiento propuesto de secuenciación y representado en el algoritmo desarrollado en lenguaje C. Se diseña una MT que compruebe (que se pare) si la cadena de entrada pertenece al siguiente lenguaje.

$$L = \{\omega\$\omega / \omega \ \in \{a + b\} \star\}$$

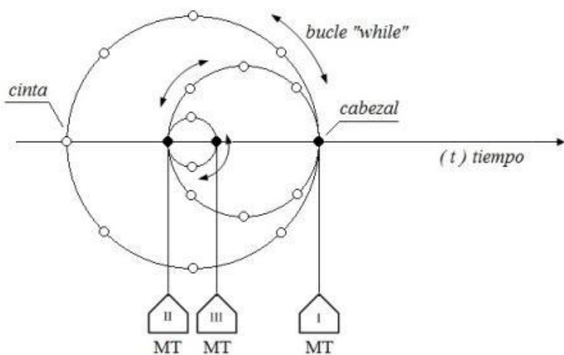
Es decir, comprobar si la cadena de $a, s,$ y b a la izquierda de $ es igual a la cadena de la derecha. Así se puede definir

$$\sum = \{a, b, \$\}$$

$$\Gamma = \{a, b, \$, \Delta, \mathrm{X}\}$$

Se ha utilizado el símbolo Δ para no confundir el símbolo de casilla en blanco con la b del alfabeto de la máquina, y X un símbolo de cinta adicional necesario para la operación.

Inicialización de la máquina: Al principio, la cinta está en blanco. A continuación se introduce la cadena de entrada, por ejemplo $\Delta abaa\$abaa\Delta$, se pone la unidad de control en el estado inicial y se posiciona la cabeza sobre la primera letra de la cadena introducida.



Puesta en marcha de la máquina: Solo se puede mover de celda en celda, y lo hará con arreglo a lo que marque la función de transición. La estrategia utilizada en este planteamiento consistirá en realizar un movimiento de ida y

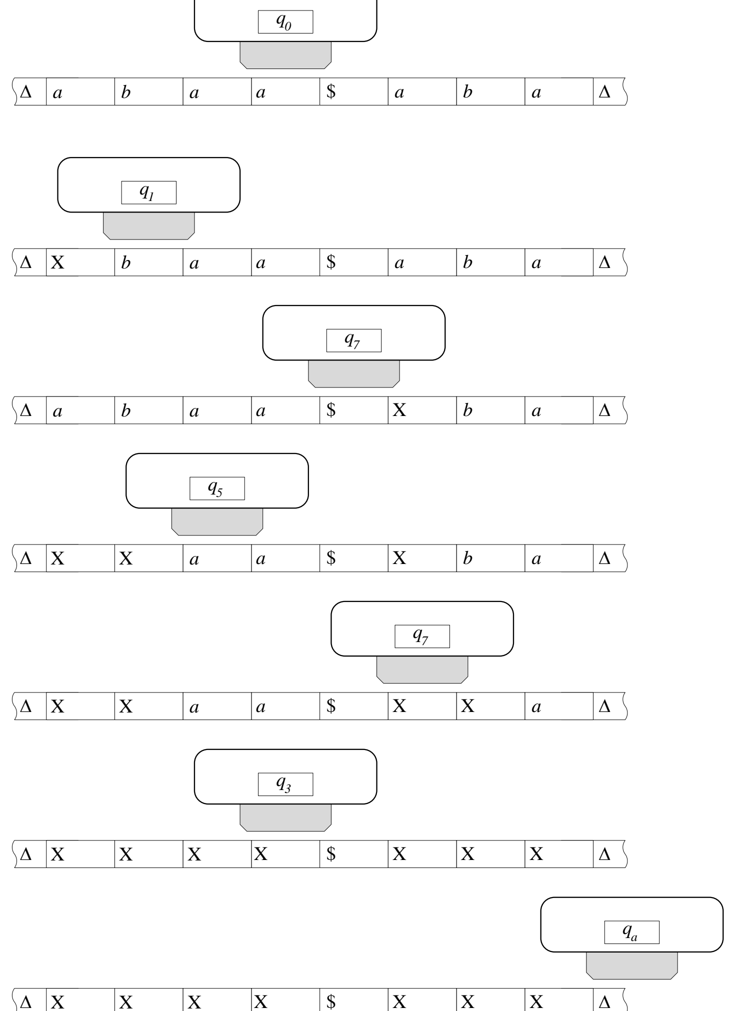
vuelta, de manera que vamos comprobando, símbolo a símbolo de la sub cadena ω a la izquierda de $ con su correspondiente, en la misma posición, de la subcadena ω de la derecha. Esto lo conseguiremos mediante la misma posición, de la sub cadena ω de la derecha. Esto se conseguirá mediante la técnica de marcado. En el planteamiento, se marca el primer símbolo (la primera a con una X), luego se recorre la cinta hasta la primera letra después de $ y se comrueba si es una a; si no lo es es la máquina se para en un estado que no es de aceptación, y si lo es, se la marca con una X y se vuelve a la primera letra de la subcadena ω de la izquierda.

La máquina se parara:

Aceptando la cadena, si se han marcado todas las letras y las sub cadenas son iguales, es decir, parándose en un estado de aceptación.

Rechazando la cadena si se produjo una discrepancia entre cadena (letras distintas, longitudes distintas), es decir, parándose en cualquier estado que no es de aceptación).

Algunas instantáneas de la máquina a lo largo de la computación son:



El marcado y el movimiento hacia adelante y hacia atrás y viceversa, caracterizan el funcionamiento de una MT.

Configuraciones. En la descripción anterior se muestran unas cuantas instantáneas de la MT. A cada una de estas se la llama configuración. En está tenemos representadas tres cosas: el contenido de la cinta, el símbolo que contiene la casilla a la que apunta la cabeza de lectura/escritura y el estado en el que se encuentra la unidad de control de la máquina en ese instante.

Para no tener que realizar un dibujo de la máquina con cada configuración que se pueden dar en la aceptación/rechazo de una cadena, se va a definir una notación más concisa para representarlas> con cadena de texto.

El primer problema que se plantea en esta nueva representación es ¿Cómo se puede representar el contenido de una cinta si su longitud es infinita? Como se ha dicho, al principio la cinta está en blanco y a continuación solo tendrá la cadena de entrada. En este caso los símbolos de casilla en blanco son irrelevantes para representa una configuración, y por lo tanto no se van a representar. Es decir, solo se escribirá desde el primer símbolo de la izquierda de la cadena de entrada hasta el última de esta.

Antes de continuar, es necesario fijar un convenio de notación para las configuraciones:

Símbolos de entrada/- a, b, c .... primera letras de abecedario en minúscula o una (minúscula) con subíndices.

Símbolos de cinta. – Además de B (símbolo de casilla en blanco),$\mathrm{Z}, \mathrm{Y}, \mathrm{X}$ .... ultimas letras de abecedario en mayúscula o una X (mayúscula) son subíndices.

Cadena del lenguaje.- z, y, x .... ultimas letras de abecedario en minúsculas o una x (minúscula) con subíndice.

Cadena de símbolos de cinta.- letras griegas minúsculas.

Estados.- p, q, r .... Cuando son pocos. Si son muchos, se suele denotar como q(minúscula) con subíndices (como el mostrado en el planteamiento anterior), empezando normalmente por el cero. No obstante, y para evitar ambigüedades, hay que tener cuidado con que la notación de los estado no pertenezca al alfabeto de la cinta.

Una configuración se representara de la siguiente manera: la cinta contiene una cadena de símbolos de cinta $\gamma = \alpha\beta$ donde α y β son dos subcadenas que, concatenadas, forman γ.

Dado un estado q_i y la cabeza de lectura/escritura posicionada sobre el primer símbolo de la sub cadena β, la configuración se representa como $\alpha q_i \beta$

Así, la primera instantánea del planteamiento anterior en texto estará representada por $\mathrm{X}q_i baa\$abaa$

Que es mucho más conciso que dibujar la máquina en cada configuración. Una configuración especial es la configuración de inicio. Si ω es la cadena de entrada, y q_0 el estado inicial, la posición de inicio será el primer símbolo de ω. Se representa por $q_0\omega$.

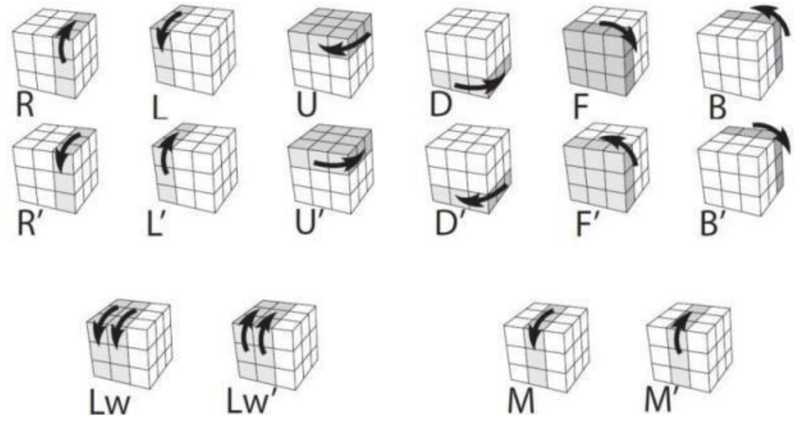
En el planteamiento, la configuración de inicio estará representada por $q_0 abaa\$abaa$

Una configuración de aceptación es aquella cuyo estado es de aceptación. En el planteamiento, es la última instantánea dibujada XXXX$XXXX$\Delta q_a \Delta$

Una configuración de rechazo es aquello cuyo estado es cualquiera que no sea de aceptación y para el que no hay definida una transición.

Transiciones. Una configuración C_i produce otra C_{i+1} cuando, por apliacion de la función de transición a C_i se obtiene C_{i+1}. En la máquina M, se define el operador transición con el símbolo $\vdash_{\mathrm{M}}$, donde el subíndice M es opcional si se sabe cuál es la máquina para la que se describe la transición. Se denota con cero, una o más transiciones aplicando la estrella de Kleene sobre el operador $\vdash_{\mathrm{M}}^{\star}$

Dado que el modelo de MT que se ha definido es obligatorio mover la cabeza a la derecha o izquierda, una producción (transición) será sinónimo de movimiento, el cual se efectuara de forma distinta, dependiendo de si es a la derecha o a la izquierda.



Dados p y q , dos estados para los que existen una transición $\mathrm{X_i}$ es cada uno de los símbolos de cinta en la configuración, Y es el símbolo de cinta que sustituye al que apunta la cabeza de lectura/escritura, $\mathrm{X_0}$ y la UC en el estado q , su configuración es $\mathrm{X_{-n}X_{-(n-1)}} \ldots \mathrm{X_{-2}X_{-1}}q\mathrm{X_0X_1X_2X_3} \ldots \mathrm{X_{m-1}X_m}$

Al realizar un movimiento:

A la izquierda.- Si la función de transición es $(q, \mathrm{X_0}) \rightarrow (p, Y, I)$, es decir, se está en el estado q señalado con el símbolo $\mathrm{X_0}$ debiendo sustituirlo por Y, al pasar al estado p y mover una casilla a la izquierda, la cabeza apuntara al símbolo de cinta anterior al sustituirlo.

$$\mathrm{X_{-n}X_{-(n-1)}} \ldots \mathrm{X_{-2}X_{-1}}q\mathrm{X_0X_1X_2X_3} \ldots \mathrm{X_{m-1}X_m}$$

$$\vdash$$

$$\mathrm{X_{-n}X_{-(n-1)}} \ldots \mathrm{X_{-2}}p\mathrm{X_{-1}}Y\mathrm{X_0X_1X_2X_3} \ldots \mathrm{X_{m-1}X_m}$$

A la derecha.- Si la función de transición es $(q, \mathrm{X}_0) \rightarrow (p, Y, D)$, es decir, se está en el estado q senalando el símbolo X_0, debiendo sustituirlo por Y, al pasar al estado p y mover una casilla a la derecha, la cabeza apuntara al siguiente símbolo de cinta al sustituirlo.

$$\mathrm{X}_{-\mathrm{n}}\mathrm{X}_{-(\mathrm{n}-1)} \dots \mathrm{X}_{-2}\mathrm{X}_{-1}q\mathrm{X}_0\mathrm{X}_1\mathrm{X}_2\mathrm{X}_3 \dots \mathrm{X}_{\mathrm{m}-1}\mathrm{X}_{\mathrm{m}}$$
$$\vdash$$
$$\mathrm{X}_{-\mathrm{n}}\mathrm{X}_{-(\mathrm{n}-1)} \dots \mathrm{X}_{-2}\mathrm{X}_{-1}Yp\mathrm{X}_1\mathrm{X}_2\mathrm{X}_3 \dots \mathrm{X}_{\mathrm{m}-1}\mathrm{X}_{\mathrm{m}}$$

Al tomar la configuración de inicio del planteamiento formal, se va a mostrar el movimiento desde el estado inicial hacia la primera instantánea. Así: $q_0\ abaa\$abaa\ \vdash$ X $q_1 baa\$abaa$

Y tomando la tercera y la cuarta instantánea, no siendo la segunda una producción de la primera, sino después de varias de ellas: XX $q_5\ aa\$abaa\ \vdash$ XX aa\$ q_7 XXaa

Cuando se termine de definir la máquina del planteamiento se vera de forma más clara la representación de los movimientos.

Cada tipo de movimiento (izquierda o derecha) puede dar lugar a dos situaciones especiales en notación: cuando la cabeza este el alguno de los extremos de la cadena representable de símbolos de cinta.

Situaciones para movimiento a la izquierda:

Que la cabeza apunte al primer símbolo de cinta que no es casilla en blanco. Habrá que representar la primera casilla en blanco en la configuración (suponer B como símbolo de casilla en blanco). Para la transición $(q, \mathrm{X}_\mathrm{n}) \rightarrow (p, Y, I)$, con la configuración mostrada, su producción será:

$$q\mathrm{X}_{-\mathrm{n}}\mathrm{X}_{-(\mathrm{n}-1)} \dots \mathrm{X}_{-2}\mathrm{X}_{-1}q\mathrm{X}_0\mathrm{X}_1\mathrm{X}_2\mathrm{X}_3 \dots \mathrm{X}_{\mathrm{m}-1}\mathrm{X}_{\mathrm{m}}$$
$$\vdash$$
$$pBY\mathrm{X}_{-(\mathrm{n}-1)} \dots \mathrm{X}_{-2}\mathrm{X}_{-1}Yp\mathrm{X}_1\mathrm{X}_2\mathrm{X}_3 \dots \mathrm{X}_{\mathrm{m}-1}\mathrm{X}_{\mathrm{m}}$$

Que la cabeza apunte al último símbolo de cinta que no es casilla en blanco y la transición sea sustituir el símbolo actual por la casilla en blanco, en cuyo caso está ultima la podemos eliminar de la representación, ya que se concatena con el resto de casillas en blanco de la parte derecha de la cinta. Para $(q, \mathrm{X}_\mathrm{m}) \rightarrow (p, B, I)$:

$$\mathrm{X}_{-\mathrm{n}}\mathrm{X}_{-(\mathrm{n}-1)} \dots \mathrm{X}_{-2}\mathrm{X}_{-1}q\mathrm{X}_0\mathrm{X}_1\mathrm{X}_2\mathrm{X}_3 \dots \mathrm{X}_{\mathrm{m}-1}q\mathrm{X}_{\mathrm{m}}$$
$$\vdash$$
$$\mathrm{X}_{-\mathrm{n}}\mathrm{X}_{-(\mathrm{n}-1)} \dots \mathrm{X}_{-2}\mathrm{X}_{-1}Yp\mathrm{X}_1\mathrm{X}_2\mathrm{X}_3 \dots p\mathrm{X}_{\mathrm{m}-1}\mathrm{X}_{\mathrm{m}}\mathrm{B}$$

Situaciones para movimiento a la derecha:

Que la cabeza apunte al primer símbolo de la cinta que no es casilla en blanco y se tenga que sustituir el símbolo actual por la casilla en blanco, en cuyo caso está ultima la podemos eliminar de la representación, ya que se concatena con el resto de casillas en blanco de la parte izquierda de la cinta. Para $(q, X_n) \rightarrow (p, B, D)$:

$$qX_{-n}X_{-(n-1)} \dots X_{-2}X_{-1}X_0X_1X_2X_3 \dots X_{m-1}qX_m$$
$$\vdash$$
$$pX_{-(n-1)} \dots X_{-2}X_{-1}YpX_1X_2X_3 \dots pX_{m-1}X_m$$

Que la cabeza apunte al último símbolo de cinta que no es casilla en blanco. Hay que incluir en la representación la primera casilla en blanco de la derecha. Para $(q, X_m) \rightarrow (p, B, I)$:

$$X_{-n}X_{-(n-1)} \dots X_{-2}X_{-1}qX_0X_1X_2X_3 \dots X_{m-1}qX_m$$
$$\vdash$$
$$X_{-n}X_{-(n-1)} \dots X_{-2}X_{-1}YpX_1X_2X_3 \dots pX_{m-1}X_mB$$

Situaciones para movimiento a la derecha:

Que la cabeza apunte al primer símbolo de la cinta que no es casilla en blanco y se tenga que sustituir el símbolo actual por la casilla en blanco, en cuyo caso está ultima la podemos eliminar de la representación, ya que se concatena con el resto de casillas en blanco de la parte izquierda de la cinta. Para $(q, X_m) \rightarrow (p, B, D)$:

$$X_{-n}X_{-(n-1)} \dots X_{-2}X_{-1}X_0X_1X_2X_3 \dots X_{m-1}qX_m$$
$$\vdash$$
$$X_{-(n-1)} \dots X_{-2}X_{-1}YpX_1X_2X_3 \dots pX_{m-1}YpB$$

Representando la MT. Cuando se ha definido el modelo de MT, se ha dicho que la función de transición es la que define el comportamiento (produce el conjunto de producciones-transiciones) de la máquina. No se ha dicho que, al ser una función de transición, por cada par (estado, símbolo de cinta) vamos a tener una sola transición (no un conjunto de transiciones posibles), con lo que se está definiendo una MT *determinística.* Al igual que los autómatas finitos y de pila, está definición no es estrictamente cierta, pues es necesitaría tener definidas las transiciones para todas las posibles combinaciones de símbolos de cinta-estado.

Al representar este conjunto función de transición es equivalente a representar la MT que describe, que puede ser mediante una tabla de transiciones o mediante un diagrama de transiciones. Como ocurre en los autómatas finitos y de pila, la representación mediante el diagrama de transiciones es adecuada mientras la máquina sea sencilla, pues en caso contrario puede llegar a ser ilegible.

Representación tabular.- Se tendrá tantas columnas como símbolos de cinta y tantas filas como estados posibles. Una entrada de la tabla podrá:

Estar en blanco, en cuyo caso corresponde a una situación con la función de transición sin definir (parada).

Contener la terna que describe la producción (transición). Si una transición dada se define por $(q, X_k) \rightarrow (p, Y, I)$, la entrada de la tabla para la columna X_k y la fila q, contendrá la terna (p, Y, I).

Representación mediante diagrama de transiciones. – Sera un pseudo-grafo dirigido donde cada nodo no corresponde con cada estado de la máquina, dibujándose para ello una pequeña circunferencia etiquetada en su interior con el nombre del estado. Uno de ellos será de inicio, estando señalado por una flecha etiquetada con nombre de INICIO, y uno o más serán de aceptación, dibujándose el nodo con dos circunferencias concéntricas etiquetado en su interior con el nombre del estado.

Cada una de las aristas dirigidas representara cada una de las transiciones definidas para la máquina. Se permiten transiciones sobre el mismo estado o desde un estado hacia otro. Estas aristas estarán etiquetadas con: símbolo de cinta actual, símbolo de cinta que lo sustituye y la dirección del movimiento. Si X es el símbolo actual de cinta, Y el que lo sustituye y S la dirección del movimiento, para la transición $(q, X) \rightarrow (p, Y, S)$, se podrá encontrar las siguientes notaciones, dependiendo de los autores $X/Y\ S$ $\quad X/Y, S\ \ X; Y, S\ \ X \rightarrow Y, S$

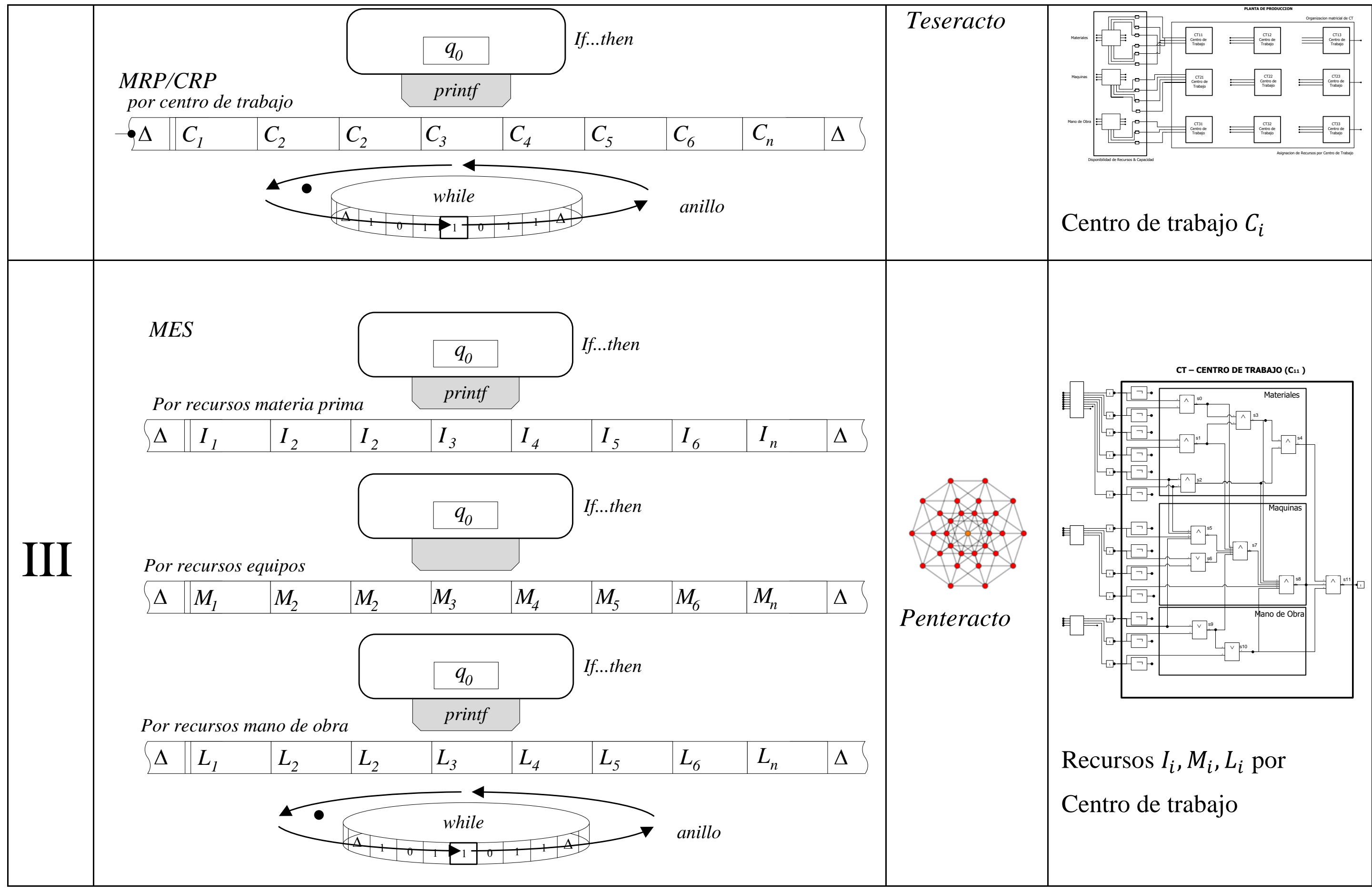
También se puede simplificar si X e Y son iguales (el símbolo se deja como esta)

$X/Y \equiv X/X, S \qquad X \rightarrow S \equiv X \rightarrow Y, S$

O la simplificación cuando hay varias transiciones del mismo tipo entre dos mismos estados.

$X_1, X_2/S \equiv \{X_1/S, X_2/S\} \qquad X_1, X_2 \rightarrow S \equiv \{X_1 \rightarrow S, X_2 \rightarrow S\}$

Nivel	Máquina de estados finitos FMS	Autómata	Sistema Lógico
I	MPS por articulos final; q_0; If...then; printf; Δ I_1 I_2 I_2 I_3 I_4 I_5 I_6 I_n Δ; while; anillo	*Cubo*	Artículos finales I_i
II			



Antes se ha definido el conjunto $S = \{I, D\}$ de las posibles direcciones de movimiento. Pues bien, en la etiqueta de la arista, podremos encontrarnos para izquierda con I (inicial en español), con L (inicial en ingles) o incluso en modo grafico con ← y para la derecha con D (inicial en español), con R (inicial en ingles) o en modo grafico→.

El símbolo de casilla en blanco, dependiendo del alfabeto de la cinta, se puede representar con B o con cualquiera de los símbolos $\Delta, \square$ o $\sqcup$.

A la secuencia finita de configuración que la máquina recorre en la aceptación o rechazo de una cadena, se le llama árbol historial de computo de la máquina. Si la máquina entra en un bucle infinito no se puede decir que exista un historial de computo (no sería finito).

Definición formal de lenguaje aceptado por la MT. Se acaba de ver como una MT acepta una cadena. Formalmente se dirá que una MT acepta la cadena ω si existe una secuencia de configuraciones en la que, si q_0 es el estado de inicio, la primera configuración es $q_0\omega$ cada C_{i+1} es una producción de C_i, y la configuración final

es de aceptación de la forma $\alpha q_a \beta$, donde q_a es un estado de aceptación, y α y β son suncadenas de cinta. Simbólicamente $q_0\omega \ \vdash_{M}^{\star} \ \alpha q_a \beta$

Dicho de otra formal, una MT, M acepta la cadena ω si la máquina para un estado de aceptación. El conjunto de cadenas aceptadas por la MT es el lenguaje $L(M)$ que acepta está MT. Se ha definido un lenguaje aceptado por una MT, no decidido.

Variaciones a la MT definida. La descripción mediante MT es un sistema muy robusto, pues admite múltiples variantes sin por ello perder su potencia de computación (aunque tampoco se ha conseguido ampliar). Se van a ver distintas variaciones de MT, las cuales, según la definición dada, serán un modelo computacional distinto. Como la entrada de un modelo computación se puede codificar para ser la entrada de otro modelo, el problema no cambiara. Se demuestra que las distintas variantes de MT son computacionalmente equivalentes.

Se dice que una MT M simula a otras MT M', si para cada transición de M' hay una secuencia de transiciones de M que realizan la misma operación. De esta manera M realiza lo mismo que M' y quizás alguna operación mas, es decir, el lenguaje aceptado por M' está incluido en el lenguaje aceptado por M.

Para demostrar que cualquier variante es equivalente en potencia computacional con la máquina definida originalmente, se procederá de manera que, dada dos máquinas M y M' primero demostramos que M simula a M' y segundo que M' simula a M. De esta manera demostramos que M y M' aceptan exactamente las mismas cadena y, por tanto, el mismo lenguaje.

Por la forma en que están construidas las MT, generalmente solo se va a tener que demostrar la simulación en una dirección, pues la otra está implícita en la definición de forma general:
Si M es una ampliación de M', M aceptara cualquier lenguaje de M'
Si M es una restricción de M', M' aceptara cualquier lenguaje que acepte M.

Retomando la MT que realiza cálculos. Cuando se definió la MT se hace una descripción a su concepto y posteriormente se definió como aceptora de un lenguaje. Hasta ahora lo que se ha hecho es comprobar que la cadena pertenece al lenguaje, y para ello el procedimiento es parar en un estado de aceptación, no importando el contenido final de la cinta. La MT calculadora original utiliza el concepto de aceptación por parada: no existe un estado de aceptación: parara siempre cuando termine el cálculo, pudiendo ser en cualquier estado. El resultado se evalúa por el contenido de la cinta cuando se para: la configuración de la cinta muestra la solución buscada (si existe).

Un detalle importante de esta máquina es que, como es un cálculo, la máquina siempre parara con el resultado en su cinta. Este comportamiento coincide con la definición de algoritmo basado en la MT, ya que un cálculo matemático no deja de ser un procedimiento (algoritmo) ya conocido de realizar algún computo.

La definición formal de está MT, donde se observa que no hay conjunto de estados de aceptación, será

$$C = \{Q, \Sigma, \Gamma, \delta, q_0, B\}$$

Para introducir en la máquina los números del cálculo primero se debe convertir (codificar). Un método muy utilizado es la notación unaria: dado un número natural con valor n símbolos consecutivos (n unos y n ceros) y se separa por otros símbolo de marcado (un cero o un uno respectivamente).

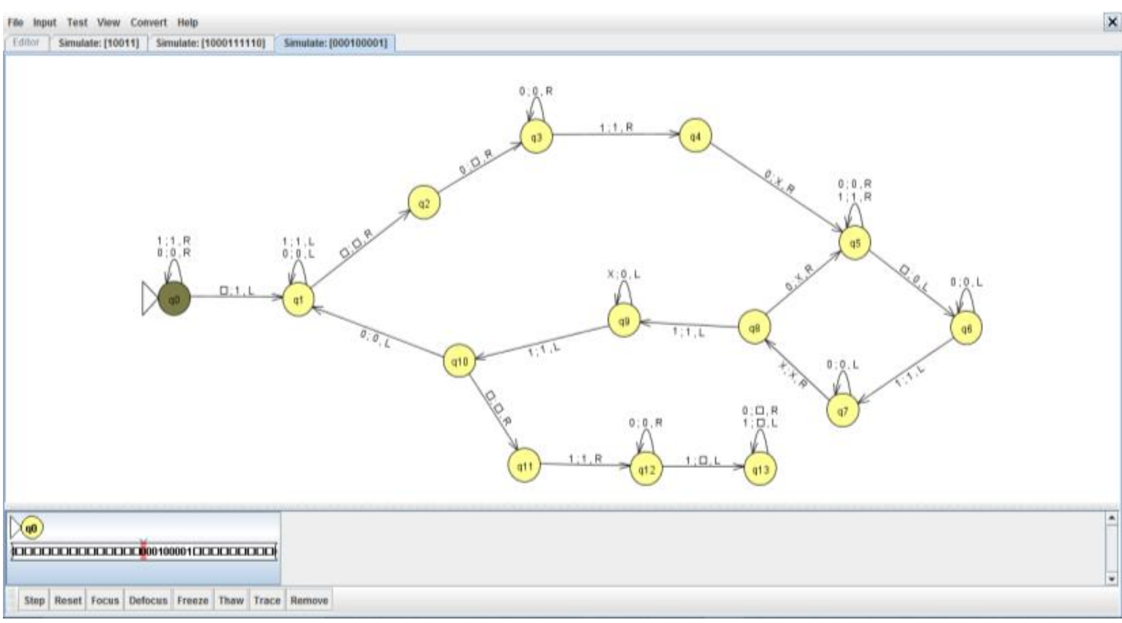
Al construir una máquina que sume números naturales n y m (mayores que cero). La máquina tendrá una cadena inicial de n ceros seguida de un 1 y seguido de m ceros (010), y se parara con la cinta conteniendo n veces m ceros (0). Los componentes de la definición formal de la máquina serán:

$Q = \{q_0, \ldots., q_{12}$

$\Sigma = \{0,1\}$

$\Gamma = \{0,1, \mathrm{B}, \mathrm{X}\}$, donde B es la casilla en blanco y X un marcador

$\delta =$



Supongamos que $n = 2$ y $m = 3$, la configuración inicial será $q_0 001000$

Marcamos el final de la cadena de entrada con un 1, que utilizaremos como marcador de fin de original. A partir de ese uno vamos a construir la solución (estados: de q_0 a q_1). $001000q_1 01$

Retrocedemos hasta la izquierda de la cadena y eliminamos el primer cero (estados: de q_1 a q_3). $q_3 010001$

Vamos al primer 1 de la cadena. Copinamos los m ceros que hay entre los dos unos, al final de la cadena (utilizando la técnica del mercado)(estados: de q_4 a q_5 y m veces el bucle de estados q_5-q_6-q_7-q_8). 01XXX10$q_6$00

Una vez copiado hay que desmarcarlos (estados: q_8, q_9, y q_{10}). $q_{10} 010001000$

Repetimos los pasos 2,3 y 4 ($m-1$) veces más, hasta que el principio de la cadena sea uno (estados: de q_{10} a q_1 y el bucle de los anteriores). $q_{10} 10001000000$

Cuando terminemos las m veces, nos posicionamos en el segundo 1 de la cadena y desde este incluido sustituimos el contenido de la cinta por símbolos de casilla en blanco (estados: q_{10} a q_{13}).

La máquina termina con el contenido en la cinta de 0^{nm}.

$$1000 q_{12} 1000000 \ \vdash \ q_{13} \text{BBBBB}000000$$

Para demostrar que está máquina tiene el mismo poder computacional que el modelo con estado de aceptación, se puede convertir esta máquina calculadora en una máquina aceptadora de cadenas añadiendo, primero, un estado de aceptación luego relacionando cada estado en el que paraba con la solución en la cinta con el de aceptación, añadiendo, para cada uno de aquellos tanta transiciones como símbolos de cinta tenga la máquina original. Si el cálculo termina siempre en el mismo estado, se marca este como de aceptación, no necesitándose añadir ninguno mas. De esta forma esta máquina simulara el funcionamiento de la máquina calculadora, aceptando cuando se pare con resultado sea positivo y rechazando cuando el resultado sea negativo.

Ahora hay que demostrar lo contrario: una máquina que acepta el lenguaje parándose en un estado de aceptación, con cualquier configuración en la cinta (se dio que no importaba lo que había en ella cuando se paraba), se puede convertir en un primera fase, en otra MT que acepte el lenguaje con un contenido de cinta determinado (solución), que precisamente es el resultado del cálculo. En una segunda fase se elimina el estado de aceptación, pues con el contenido en la cinta como solución, se da por valido el cálculo. Así, la máquina calculadora obtenida aceptara si tiene la solución en la cinta y rechazara si tiene cualquier otro contenido en la misma.

Por tanto, se ha encontrado la forma de simular, en una y otra dirección, las dos máquinas, de manera que se ha demostrado que las dos aceptan el mismo lenguaje. En conclusión, ambas máquinas tienen la misma potencia computacional. A partir de ahora se van a considerar todas las MT en los niveles I, II y III como aceptadoras de cadenas.

Otras variantes de MT. Todas las variantes van a tener en común una capacidad de almacenamiento infinito y se van a diferenciar por una notación distinta para la definición formal y/o la función de transición. Por ejemplo, se puede observar que la definición formal de la máquina calculadora no tenía conjunto de estados de aceptación.

Cada una de las variantes que se muestran se podrán utilizar tal cual o construir otras que sean una mezcla de varias de ellas según convenga, habida cuenta de que, con las debidas transformaciones, todas van a ser computacionalmente equivalentes entre sí.

Que solo tenga un estado de aceptación y solo uno. La definición formal con esta restricción seria $\mathrm{M}' = (Q, \Sigma, \Gamma, \delta, q_0, B, q_a)$
Es fácil convertir la MT original M en una que solo tenga un estado de aceptación, M', creando un estado de aceptación nuevo, quitando la característica de aceptación a los que antes la tenían y definir, desde cada uno de estos y para cada símbolo de cinta, una transición al nuevo estado de aceptación (para que se pueda mover a este, sea el que sea el contenido de la cinta) sin sustituir el símbolo actual, siendo la dirección del movimiento de la cabeza indiferente. Por tanto M' aceptara los lenguajes que acepte M.
Al ser una caso particular de la MT original en el que F solo tiene un estado de aceptación, tanto cualquier lenguaje aceptado por M' también lo es M. Por tanto M y M' aceptan exactamente los mismos lenguajes.

Que tenga un conjunto R de estados de rechazo. Si la máquina para, en lugar de hacerlo en cualquier estado que no sea de aceptación cuando la cadena no pertenezca al lenguaje, está lo hará en algún estado de rechazo. Así, la MT con esta ampliación se define formalmente: $\mathrm{M}' = (Q, \Sigma, \Gamma, \delta, q_0, B, F, R)$
Donde R es un conjunto al que pertenecen los estados de rechazo.

Con esta variación lo que realmente se está haciendo es definir completamente una máquina determinista, de manera que, para cada estado, (símbolo de cinta) hay una transición.
Para convertir la MT original M a está M' se añade a M un conjunto de estados R de rechazo y, para cada entrada de la tabla de transiciones vacía, se añade una transición a cualquiera de los estados de R (la dirección del movimiento de la cabeza es de nuevo irrelevante). M' aceptara o rechazará la cadena si M lo hacía originalmente, luego M' aceptara los lenguajes que acepte M.

Convertir M' en la original, bastaría con eliminar los estados de rechazo y las transiciones hacia ellos, de manera que M parara en cualquier estado que no es de aceptación si la máquina con R paraba en un estado de rechazo y continuara aceptando cuando M' acepte la cadena, luego M aceptara los lenguajes que acepte M'. Por tanto, M y M' aceptaran exactamente los mismos lenguajes.
Una variante basada en este misma, es aquella en la que la máquina tiene solo un estado de rechazo, donde la máquina se define como $M' = (Q, \Sigma, \Gamma, \delta, q_0, B, F, q_r)$
Es fácil demostrar que está es equivalente a la anterior utilizando el mismo método que para la variante de un solo estado de aceptación.

Una vez aceptada la cadena, el contenido de la cinta sea un mensaje de aceptación. Esta variante es la utilizada para convertir una MT que acepta un lenguaje en una MT que realiza un cálculo. Es posible convertir una MT, M, que se para con cualquier contenido en la cinta en otra MT, M', que acepta con un contenido especial, que se llama mensaje de aceptación.
Se acepta la cadena con un mensaje de aceptación de confirmación y la rechaza con cualquier contenido en la cinta. Convertir M en M' es fácil de hacer, sustituyendo el estado de aceptación original (si son varios se convierte primero para que solo tenga uno) por una ampliación de la MT que borre la cinta y escriba un mensaje de aceptación. M' aceptará el lenguaje si lo hacía M, es decir los lenguajes aceptados por M también lo serán por M'.

En cada uno de los niveles I, II y III, la MT efectúa una salida que de cierta forma se convierte en la entrada a la siguiente máquina, hasta llegar a la última que da la salida a la terminal y genera la respectiva secuencia de operaciones. Cada ciclo del anillo (cinta finita) se genera un cambio de estado de la máquina con las validaciones para cada uno de los símbolos que representan el lenguaje propio en cada MT.

En el planteamiento del lenguaje $L = (\omega\$\omega \;/\; \omega\epsilon\{a + b\} *)$ con una entrada $abaa\$abaa$, la máquina termina aceptándola, teniendo una configuración en la cinta XXXX\$XXXX$q_a\Delta$.
Se puede modificar la MT para que borre está configuración al final y la sustituya por un mensaje de aceptación, por ejemplo $\Delta Y \Delta$, donde Y es un nuevo símbolo de cinta.
La máquina original es una generalización de esta variación, por lo que cualquier lenguaje que acepta M' también será aceptado por M. Por tanto M y M' aceptan exactamente los mismos lenguajes.

Que se permita no mover la cabeza en una transición. Se permite a la cabeza no moverse cuando se cambia el valor de la casilla actual. En este caso, no es válido el términos movimiento para un transición (se dijo que se podía utilizar como sinónimo en algunos supuestos). Así la máquina queda definida formalmente con $M' = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ (igual que la original) y la función de transición cambia su notación por $\delta = Q' \text{x}\, \Gamma \rightarrow Q \text{ x } \Gamma \text{ x} S'$
Con $S' = \{I, D, E\}$ donde E es la inicial es español de Estacionario (también se puede ver S de Stationary en ingles).
Otra función de transición equivalente podría ser(o se realiza un movimiento con sustitución o se deja estacionaria la cabeza): $\delta = Q' \text{x}\, \Gamma \rightarrow Q \text{ x } (\Gamma \cup S')$
Como se observa en la tabla siguiente, la equivalencia de transiciones entre las funciones de transición mostradas y la de la MT es casi inmediata, por lo que M aceptara los lenguajes que acepte M', y viceversa. Una transición

sin movimiento se puede simular, en la máquina original, con una transición adicional que haga una vuelta atrás al símbolo anterior.

Una transición de la máquina original (sustitución y movimiento) se puede simular, en la segunda fila, con dos transiciones y un estado adicional simulado primero la sustitución de símbolo y luego el movimiento.

δ	***E***	***I***		***D***	
$\delta = Q'\text{x}\,\Gamma \rightarrow Q \text{ x } (\Gamma \cup S')$	$(q,a) \rightarrow (p,b,E)$	$(q,a) \rightarrow (p,a,I)$	$(q,a) \rightarrow (p,b,I)$	$(q,a) \rightarrow (p,a,D)$	$(q,a) \rightarrow (p,a,D)$
$\delta = Q'\text{x}\,\Gamma \rightarrow Q \text{ x } (\Gamma \cup S)$	$(q,a) \rightarrow (p,b)$	$(q,a) \rightarrow (p,I)$	$(q,a) \rightarrow (p',b)$ $(p',b) \rightarrow (p,I)$	$(q,a) \rightarrow (p,D)$	$(q,a) \rightarrow (p',b)$ $(p',b) \rightarrow (p,D)$
$\delta = Q'\text{x}\,\Gamma \rightarrow Q \text{ x } \Gamma \cup S$	$(q,a) \rightarrow (p,b,D)$ $(p,b) \rightarrow (p,b,I)$	$(q,a) \rightarrow (p,a,I)$	$(q,a) \rightarrow (p,b,I)$	$(q,a) \rightarrow (p,a,D)$	$(q,a) \rightarrow (p,a,D)$

Que la unidad de control tenga una memoria finita de símbolos. Se definió la MT con la unidad de control en el estado q_i, y con un número infinito de m celdas de almacenamiento temporal. Estas celdas podrán memorizar algún símbolo de cinta en algún momento de la computación para utilizarla en otro momento de la misma.

Antes de definir formalmente la MT que simula está unidad de control se debe primero redefinir el conjunto de estados.

Si $\mathrm{M_i}$ representa cada una de la celdas de almacenamiento, se define un estado como

$$\mathrm{Q_m} = \{\mathrm{Q} \text{ x } \mathrm{M_1} \text{ x } \mathrm{M_2} \text{ x } \mathrm{M_3} \text{ x } \mathrm{M_4}\text{x } \ldots \ \mathrm{M_m}\}$$

Donde $\mathrm{M_i}$ podrá contener cualquier símbolo de cinta de la máquina. Es decir, cada estado ahora será una tupla del producto cartesiano de cada estado original por el alfabeto de cinta para cada una de las celdas de memoria. Así, se define la máquina M' como M'= $(\mathrm{Q_m}, \Sigma, \Gamma, \delta, (q_0, B_1, B_2, B_3, \ldots, B_m), B, F_m)$

Donde $F_m \subset Q_m$ y cada B_i en el estado inicial (para $0 < i < m$), es el contenido de cada memoria, que en el estado inicial será el símbolo de casilla en blanco.

La función de transición será $\delta = Q'_m\text{x}\,\Gamma \rightarrow Q_m \text{ x } \Gamma \text{ x} S$ donde $Q'_m = Q_m - F_m$ y cada transición se podrá cambiar, o no, el contenido de alguna o todas las memorias.

Para mostrar el funcionamiento de esta máquina, se va a utilizar el planteamiento informal, el cual se va a modificar para que tenga una celda de memoria en su unidad de control. Si $L = \{\omega\$\omega \ / \ \omega \in \{a+b\}*\}$ donde $\Sigma = \{a, b, \$\}$ y donde $\Gamma = (a, b, \$, \Delta, \mathrm{X}\}$.

Se va a utilizar la memora para identificar si el elemento i-esimo de la subcadena ω de la izquierda es igual al elemento i-esimo de la subcadena ω de la derecha. Los símbolos de cinta para la memoria serán $\{a, b, \Delta\}$. La tabla

de transiciones será la siguiente (observar que no se han dibujado todas las filas correspondientes a Q_m ya que algunas no tienen transiciones definidas):

	a	b	$	Δ	X
q_0, Δ	$[q_1, a], X, D$	$[q_1, b], X, D$	$[q_8, \Delta], \$, D$	$-$	$-$
q_1, a	$[q_1, a], a, D$	$[q_1, a], b, D$	$[q_2, a], \$, D$	$-$	$-$
q_1, b	$[q_1, b], a, D$	$[q_1, b], b, D$	$[q_2, b], \$, D$	$-$	$-$
q_2, a	$[q_1, a], X, I$	$-$	$-$	$-$	$[q_2, a], X, D$
q_2, b	$-$	$[q_7, b], X, I$	$-$	$-$	$[q_2, b], X, D$
q_7, a	$-$	$-$	$[q_3, a], \$, I$	$-$	$[q_7, a], X, I$
q_7, b	$-$	$-$	$[q_3, b], \$, I$	$-$	$[q_7, b], X, I$
q_3, a	$[q_3, a], a, I$	$[q_3, a], b, I$	$-$	$-$	$[q_4, a], X, D$
q_3, b	$[q_3, b], a, I$	$[q_3, b], b, I$	$-$	$-$	$[q_4, b], X, D$
q_4, a	$[q_1, a], X, D$	$[q_1, b], X, D$	$[q_8, \Delta], \$, D$	$-$	$-$
q_4, b	$[q_1, a], X, D$	$[q_1, b], X, D$	$[q_8, \Delta], \$, D$		
q_8, Δ				$[q_a, \Delta], \Delta, D$	$[q_8, \Delta], X, D$
q_a, Δ	$-$	$-$	$-$	$-$	$-$

Las transiciones se pueden representar de una manera más concisa. Si Z e Y son la representación de un símbolo de cinta cualquiera, pudiendo ser en este caso a o b indistintamente, se pueden agrupar las transiciones. Por ejemplo, las transiciones $[q_0, \Delta], a = [q_1, \Delta], X, D$ y $[q_0, \Delta], b = [q_1, b], X, D$ tienen la misma estructura, diferenciándose solo por el símbolo en la cinta. Si se sustituye por Z, se podrán agrupar las dos transiciones en una: $[q_0, \Delta], a = [q_1, Z], X, D$

Para la tabla anterior las transiciones agrupadas podrán ser (se marca en negrilla las transiciones que realmente utilizan la memoria):

$[q_0, \Delta], Z = [q_1, Z], X, D$	$[q_0, \Delta], Z = [q_8, Z], \$, D$	$[q_1, \Delta], Z = [q_1, Z], Y, D$
$[q_1, \Delta], \$ = [q_2, Z], \$, D$	$[q_2, Z], X = [q_2, Z], X, D$	$[q_2, Z], Z = [q_7, Z], X, I$
$[q_7, Z], \$ = [q_3, Z], \$, I$	$[q_7, \Delta], Z = [q_7, Z], X, I$	$[q_3, Z], Y = [q_3, Z], Y, I$
$[q_3, Z], X = [q_4, Z], X, D$	$[q_4, Z], Z = [q_1, Y], X, D$	$[q_4, Z], \$ = [q_8, \Delta], \$, D$
$[q_8, \Delta], \Delta = [q_a, \Delta], \Delta, D$	$[q_8, \Delta], Z = [q_8, \Delta], X, D$	

Para convertir esta máquina M' con transiciones agrupadas en una que no tenga memoria, M, se procede de la siguiente forma:

Se expanden las representaciones de símbolos de cinta agrupados (en nuestro caso Z e Y). El resultado se plasma en una tabla de transiciones (como la que se ha mostrado en el planteamiento).

Para cada estado de Q_m se le asigna un nombre de estado de la máquina sin memoria y se sustituye está asignación en la tabla. El resultado es una tabla de transiciones sin memoria.
Se comprueban las filas de la tabla para eliminar aquellos estados que tengan las mismas transiciones.
Toda la cadena aceptada por M' será aceptado por M', luego M aceptará los lenguajes que acepte M'. La máquina original M es un caso particular de M' sin memoria, luego M' aceptara cualquier lenguaje que acepte M. Por tanto, M y M' aceptan exactamente los mismos lenguajes.
Continuando con el planteamiento, si se aplican estos pasos, el resultado es la tabla de transiciones de la máquina original (no tiene porque ser así).
La representación con el diagrama de transiciones será igual que son memoria, con la salvedad de que los estados estarán etiquetados ahora por la tupla estado $(q_1, Z_1, Z_2, Z_3, \dots Z_m)$ donde cada Z_i será el contenido de la memoria i.

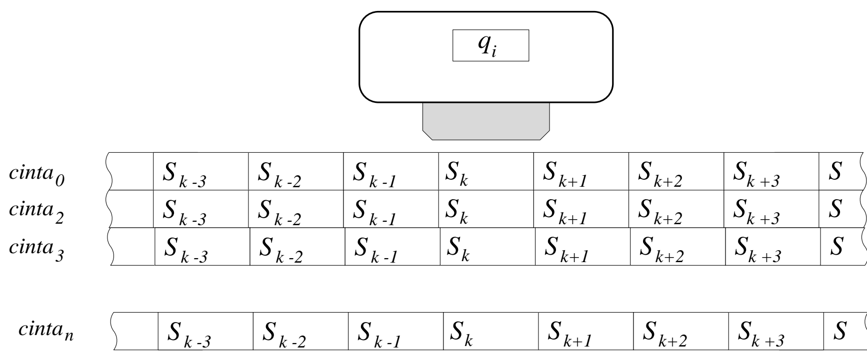
Cinta con varias pistas. Con anterioridad se dijo que un símbolo de cinta no tiene porque ser un solo carácter; pues bien, este es el caso. Se divide la cinta en bandas paralelas, que se van a llamar pistas y, a su vez, se van a dividir en casillas tal como se hizo con la cinta única, resultando en una matriz con tantas filas como pistas y con un número infinito de columnas a ambos lados de la cabeza.

Hay que tener en cuenta que no es lo mismo un símbolo de cinta y un símbolo de pista. Ahora un símbolo de cinta va a ser una tupla de i es el número de pistas. Para ello, la máquina accederá solidariamente al contenido de todas las casillas de la misma columna, es decir, accederá a la i-tupla de símbolos de pista.

El conjunto de símbolos, para cada pista, podrá ser el mismo o distinto y el conjunto de símbolos de cinta de la máquina serán la tuplas pertenecientes al producto cartesiano de los conjuntos de símbolos de todas las pistas. Si se tienen i pistas, y cada Γ_k' (para $1 < k < i$) es el conjunto de símbolos de la pista k, el alfabeto de cinta de M' será el conjunto de i-tuplas.

$$\Gamma' = \{\Gamma_1' \text{ x } \Gamma_2' \text{ x } \Gamma_3' \text{ x } \Gamma_4' \text{x} \dots \text{ x } \Gamma_i'\}$$

Y un símbolo de cinta de M' será, si estamos en la columna k, una tupla i-tupla de símbolos atómicos de la forma $(S_{k1}, S_{k1}, S_{k1}, \dots, S_{k1}) \in \Gamma'$
Una posible representación de la MT de varias pistas podría ser



Una forma típica de utilizar una máquina de este tipo es colocar la cadena de entrada en la primera pista y utilizar el resto para marcar lo que se necesite (que se ha visitado un símbolo, recordar una posición, dar un mensaje de aceptación…). Al principio, todas las pistas, a excepción de la primera, están vacías. Así cada símbolo del alfabeto de la máquina será una i-tupla del conjunto

$$\Sigma' = \{\Sigma \text{ x } \mathrm{B}_2 \text{x } \mathrm{B}_3 \text{x } \mathrm{B}_4 \text{x } \ldots . \text{x} \mathrm{B}_\mathrm{i}\}$$

Donde Σ es el alfabeto de la cadena de entrada en símbolos atómicos. La definición formal de la máquina de varias pistas será $\mathrm{M}' = (\mathrm{Q}, \Sigma', \Gamma', \delta, q_0, (\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \ldots, \mathrm{B}_i), \mathrm{F})$ donde $(\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \ldots, \mathrm{B}_i)$ es el símbolo de casilla en blanco. La función de transición seria $\delta = \mathrm{Q}' \text{x } \Gamma' \rightarrow \mathrm{Q} \text{ x } \Gamma' \text{ x S}$ que, desglosada en las partes descritas, queda como

$$\delta = \mathrm{Q}' \text{x } (\, \Gamma_1' \text{ x } \Gamma_2' \text{ x } \Gamma_3' \text{ x } \Gamma_4' \text{x } \ldots . \text{ x } \Gamma_i') \rightarrow \mathrm{Q} \text{ x } (\Gamma_1' \text{ x } \Gamma_2' \text{ x } \Gamma_3' \text{ x } \Gamma_4' \text{x } \ldots . \text{ x } \Gamma_i') \text{ x S}$$

Por ejemplo $\big(p, (Z_1, Z_2, Z_3, \ldots, Z_k)\big) \rightarrow (q, (Y_1, Y_2, Y_3, \ldots, Y_k), I)$

Significa que una MT en un estado p y símbolo de cinta $(Z_1, Z_2, Z_3, \ldots, Z_k)$ realizara un movimiento al estado q, escribiendo en la cinta el símbolo $(Y_1, Y_2, Y_3, \ldots, Y_k)$ y moviendo su cabeza a la izquierda.

Para simular una máquina de Turing M' de varias pistas con una M de una sola pista con símbolos de cinta atómicos, para M se definirá:

Si $\Sigma' = (\Sigma, B_2, B_3, \ldots, B_i)$, Σ es el alfabeto de máquina de M

$\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \cup \ldots \cup \Gamma_i \cup \{\#, @\}$ Estas ultima son símbolos de marca adicionales

La unidad de control va a tener i celdas de memoria. Esto es válido dado que el número de pistas es finito.

Primero se concatena el contenido de cada una de las pistas en la cinta de M, separadas por el delimitador #. El principio de la primera cadena y el final de la ultima también se delimitan con #. Para denotar la posición de la cabeza en la cinta de M', se introducirá como la marca @ y se hará en cada uno de los contenidos de las pistas introducidas en la cinta M'. Si ω está en la tupla de entrada $(\omega, B_2, B_3, \ldots, B_i,$ para M', la cinta de M quedara con $(i + 1)$ delimitadores e i cadenas entre ellos, cada una marcada con la posición de la cabeza. En el estado inicial la cinta de M será $\#@\omega\#B_2\# \ldots . \#@B_i\#$

Para cada transición de M' se simulara la siguiente secuencia de transiciones en M:

M recorrerá la cinta hasta el último delimitador #, procediendo en su recorrido a almacenar en la celda de memoria M_k (para $1 < k < i$) el símbolo actual de la cinta k (elemento k de la i-tupla de M'). De esta forma la unidad de control tiene acceso al símbolo de cinta actual de M' (i-tupla).

Aplicara la transición de M'. M recorrerá de nuevo la cinta y en cada subcadena que representa cada pista, en la posición de la marca de cabeza, se escribe, dependiendo del símbolo en la memoria y el estado, el símbolo que indica la transición, y realiza el movimiento que determina S. Utilizara la memoria para almacenar el símbolo de pista de M' después de la transición.

Si en una subcadena que representa una pista, después del movimiento de la cabeza, está apuntara al delimitador, a partir de este (inclusive) hay que desplazar una casilla a la derecha el resto de subcadena, y añadir un símbolo de casilla vacía en el hueco que hemos hecho. Se ha marcado en negrilla la situación y subrayada la subcadena desplazada a la derecha.

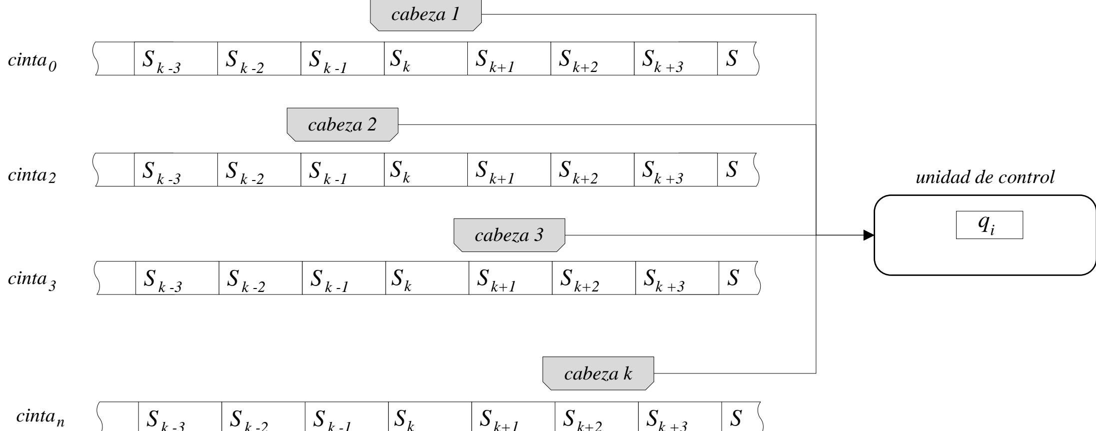
$$\#\alpha@\beta\#@\boldsymbol{Bq_x}\#\ ....\#@B\# \quad \vdash \quad \#\alpha@\beta\#@\boldsymbol{Bq_x B}\#\ ...\#@B\#$$

M acepta la cadena en el mismo estado en el que lo haría M', luego M aceptará los mismos lenguajes que aceptaría M'.

Se afirma con anterioridad que la codificación de la entrada a una MT es indiferente, ya que se puede convertir con otra MT. La ampliación de la cinta a varias pistas no deja de ser eso mismo: una codificación distinta para una misma entrada y por tanto está MT tampoco ofrece una mejora con respecto a la MT original: la máquina de varias pistas es una generalización de la máquina de una pista (tuplas de un solo elemento) y por tanto cualquier lenguaje aceptado por la máquina de pista única (ya se vio que la MT con memoria es equivalente a la MT sin memoria), lo será por una máquina de una cinta con varias pistas. Por tanto, aceptan exactamente los mismos lenguajes. En el tercer nivel, se encuentra una MT con tres cabezales de lectura/escritura con una sola unidad de control, la cual se retroalimenta en tiempo real de los estados de cada uno de los recursos que están asociados al centro de trabajo y contiene las reglas definidas por el planificador (usuario) para evaluar las alternativas con respecto a la disponibilidad, capacidad, cambios de la demanda (requerimientos nuevos a la planta de producción) y las salidas de inventario procesado en el centro de trabajo determinan el comportamiento del sistema lógico en un tiempo t actualizando la cadena (cinta en cada cabezal).

MT con varias cintas. Esta variante, que es una generalización de la MT con una cinta, es importante porque con ella se simula de manera directa el comportamiento de un ordenador. El hecho de que la MT de una cinta sea un caso particular de la de varias cintas hace que el lenguaje aceptado por la primero sea aceptado por la demanda por la segunda. Después de definir está máquina se procederá a demostrar que existe una MT de una cinta que simula el funcionamiento de una máquina con varias cintas, y por tanto cualquier lenguaje aceptado por una MT

de una cinta también lo será por la MT de varias cintas. Esto demuestra que una máquina con una cinta acepta exactamente los mismos lenguajes que una máquina con varias cintas.



Esta MT estará compuesta por varias cintas soporte de almacenamiento y una unidad de control que estará en uno de entre un conjunto finito de estados. Tendrá asociadas tantas cabezas de lectura/escritura como cintas, independientes todas ellas entre sí.

En cada paso del cálculo la unidad de control va a tener en cuenta el estado en el que se encuentra y el contenido de la casilla a la apunta cada una de las cabezas.

El estado inicial, la primera cinta contendrá la cadena de entrada, codificada en el alfabeto de entrada, y el resto de cintas estarán en blanco (se podrán utilizar durante la computación para comprobar que se ha visitado un símbolo, recordare una posición, dar un mensaje de aceptación…). La cabeza de la cinta 1 está apuntando al primer símbolo de la entrada y la cabeza en el resto de cintas podrá apuntar a cualquier casilla, dado que aquellas siguen en blanco.

Se define un único alfabeto de cinta para la máquina, de manera que cualquier cinta podrán contener cualquier símbolo de este. A este alfabeto de cinta, pertenece el alfabeto de la máquina (en el que estará codificada la entrada en la primera cinta). Se podría haber definido un alfabeto de cinta para cada una de ellas, pero esto, además de innecesario, solo complicaría la máquina, ya que el alfabeto de cinta para la máquina se podría obtener por la unión de todos los alfabetos independientes de cinta (como se hizo en la definición de MT de varias pistas).

La MT de varias cintas se define formalmente como $M = (Q, \Sigma, \Gamma, \delta', q_0, B, F)$

Se ha dicho que cada cabeza es independiente; esto nos lleva a que, para cada cinta, hay una función de transición de la forma $\delta = Q'$x $\Gamma \to Q$ x Γ x S' (es la misma que hemos definido para la máquina de una cinta, con la excepción de S', definida anteriormente con la transición estacionaria). Si la máquina tiene k cintas, se podrá generalizar la función de transición a $\delta' = Q'$x $\Gamma^k \to Q$ x Γ^k x S'^k

Así, la función de transición en una MT con k cintas pertenecerá al producto cartesiano de (la notación subíndice se debe identificar como número de cinta y no como un alfabeto de cinta distinto):

$$\delta' : Q \text{ x } \Gamma_1 \text{x} \Gamma_2 \text{x} \Gamma_3 \text{x} \dots \text{x} \Gamma_k \quad \to Q \ \Gamma_1 \text{x} \Gamma_1 \text{x} \Gamma_2 \text{x} \Gamma_3 \text{x} \dots \text{x} \Gamma_k \text{x } S_1' \text{x } S_2' \text{x} S_3' \text{x} \dots . \text{x} S_k'$$

Por ejemplo, la transición:

$$(p, Z_1, Z_2, Z_3, \dots, Z_k) \to \ (q, Y_1, Y_2, Y_3, \dots, Y_k, I_1, D_2, D_3, \dots, E_k)$$

Significa que una MT con k cintas que este en un estado p y con cada cabeza i de lectura/escritura apuntado al simbol Z_i de cinta, realizara un movimiento a un estado q, sustituyendo cada Z_i por el Y_i que corresponda y realizando el movimiento que corresponda a cada cabeza i.

Ya se ha mencionado que está máquina va a tener la misma potencia que la de una cinta con pista única. Para demostrarlo, se utiliza un paso intermedio: simular está MT con k cintas, con una MT, M', de una cinta dividida en $2k$ pistas y que además va a tener $k + 1$ memorias (antes ya se demostró que la MT de varias pistas es equivalente a una MT de una cinta con pista unitaria, y que la MT sin memoria es equivalente a la MT con ella).

Primero se visualizaron las k cintas en una cinta de $2k$ pistas, de manera que la pista $2i$ (para $1 < i < k$) es el contenido de la cinta i y la pista $2i - 1$ es un marcador a la posición actual de la cabeza (pistas impares marcadores, pistas pares contenido de la cinta).

El alfabeto de pista para las impares será $\{B, @\}$ y para las impares será el mismo que para las pistas de M, Γ. El alfabeto de cinta de M' será un conjunto de tuplas del producto cartesiano de la forma (la notación subíndice se debe identificar como numero de cinta y no como un alfabeto de cinta distinto).

$$\Gamma' = \{\{B, @\} \text{ x } \Gamma_1 \text{x } \{B, @\} \text{ x } \Gamma_2 \text{x } \{B, @\} \text{ x } \Gamma_3 \text{ x} \dots \text{x } \{B, @\} \text{ x } \Gamma_k\}$$

En la figura de M' con $2k$ pistas, el símbolo de cinta actual será la tupla $(@, S_{(1,0)}, B, S_{(2,0)}, \dots ., B, S_{(k,0)})$

Para cada movimiento de M, M' tendrá las siguientes transiciones:

La cabeza de M' buscara consecutivamente a los k marcadores de posición de las k cintas, y almacenara en cada memoria de unidad de control el contenido de la casilla que marca. Para saber en que con esta (pista $2i$ = cinta i) utiliza la ultima celda de memoria de la unidad de control. Una vez hecho esto, tenemos el estado y el símbolo actual de cada cabeza de M almacenado en M'.

Para las pistas correspondientes a cada cinta i, M' vuelve a leer el contenido del marcador de la pista $2i - 1$ y la casilla correspondiente de la pista $2i$: Escribe el símbolo que determina la transición correspondiente a la cinta i y mueve la marca (izquierda, derecha o estacionaria) a la casilla que determine la transición. Al final de las $2k$ pistas, M' tendrá varias tuplas que representan el contenido de las k cintas y la posición de cada cabeza sobre ellas después del movimiento. Utilizara la memoria para almacenar el símbolo de cada cinta de M después de cada transición.

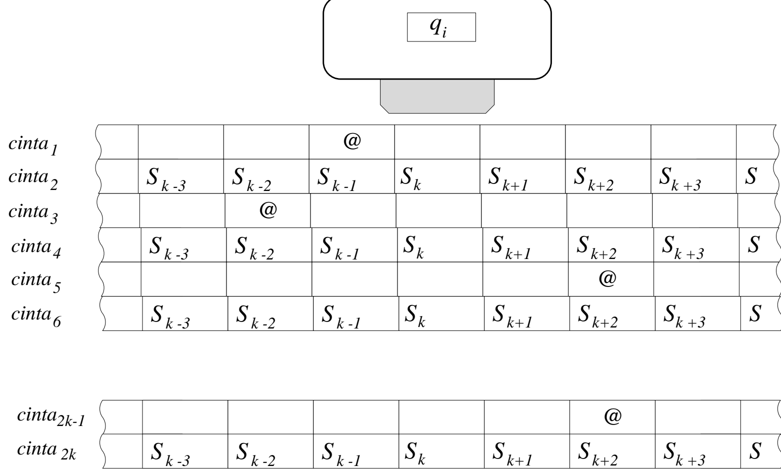
Los estados de aceptación de M' serán los estados de aceptación de M, luego cualquier lenguaje aceptado por M' también lo será por M.

En el planteamiento del dibujo, dad la transición de M:

$$\left(p, S_{(1,0)}, B, S_{(2,0)}, \dots., S_{(k,+2)}\right) \rightarrow (q, S_{(1,0)}, Z, S_{(3,1)}, \dots., S_{(k,+2)}, I, I, D, \dots, E)$$

M' buscara primero la tupla $S_{(1,0)}, \left(@, S_{(1,0)}, B, S_{(2,0)}, B, S_{(3,0)}, \dots., B, S_{(k,0)}\right)$ almacenando $S_{(1,0)}$ en la memoria 1, luego la tupla correspondiente a la segunda cinta ,$\left(B, S_{(1,-1)}, @, S_{(2,-1)}, B, S_{(3,-1)}, \dots., B, S_{(k,-1)}\right)$ almacenando $S_{(2,-1)}$ en la memoria 2 y así sucesivamente hasta tener en las k primeras memorias los símbolos actuales de M.

La transición que está computando modifica el contenido de la segunda cinta, y mueve cada uno de los marcadores. Así el contenido de la cinta en M' quedará:



Para reducir M' a una MT con una sola cinta, M', se aplicara la demostración antes dada para la MT de varias pistas: Las distintas pistas $2i$ se concatenan en una sola pista, introduciendo el marcado de posición de cabeza en cada pista $2i$. Así, en nuestro ejemplo partiríamos del contenido en la cinta de M" con

$$\# \, S_{(1,-3)}S_{(1,-2)}S_{(1,-1)}@S_{(1,0)}S_{(1,+1)}S_{(1,+2)}S_{(1,+3)}$$

$$\# \, S_{(2,-3)}S_{(2,-2)}@S_{(2,-1)}S_{(2,0)}S_{(2,+1)}S_{(2,+2)}S_{(2,+3)}$$

$$\# \, S_{(3,-3)}S_{(3,-2)}S_{(3,-1)}S_{(3,0)}@S_{(3,+1)}S_{(3,+2)}S_{(3,+3)}$$

$$\# \, S_{(k,-3)}S_{(k,-2)}S_{(k,-1)}S_{(k)}@S_{(k,+2)}S_{(k,+3)}\#$$

Y después de los sucesivos movimientos de M', que simulan un movimiento de M, M" tendrá un contenido en su cinta

$$\# \, S_{(1,-3)}S_{(1,-2)}@S_{(1,-1)}S_{(1,0)}S_{(1,+1)}S_{(1,+2)}S_{(1,+3)}$$

$$\# \, S_{(2,-3)}@S_{(2,-2)}S_{(2,-1)}S_{(2,0)}S_{(2,+1)}S_{(2,+2)}S_{(2,+3)}$$

$$\# \, S_{(3,-3)}S_{(3,-2)}S_{(3,-1)}S_{(3,0)},@S_{(3,+1)}@S_{(3,+2)}S_{(3,+3)}$$

$$\# \, S_{(k,-3)} S_{(k,-2)} S_{(k,-1)} S_{(k)} @ S_{(k,+2)}, S_{(k,+3)} \#$$

Para representar las transiciones de una MT de varias cintas también se utilizaran tablas y diagrama de transiciones:

Tabla de transiciones. Tendrá una fila por cada uno de los estados y tendrá una columna por cada tupla del producto cartesiano de los k alfabetos de cinta. Cada entrada contendrá la $2k$-tupla con los símbolos a sustituir y el desplazamiento de las cabezas correspondiente.

Diagrama de transiciones. El diagrama será el mismo que la MT con una cinta, con la excepción de que la relación entre estados (arista del grafo) se etiqueta con todas las transiciones posibles para esos dos estados en cada una de las cintas, separados por una barra.

10.1.2. El poder de los lenguajes de programación

Como aplicación práctica de la teoría presentada, regresamos a nuestra pregunta con respecto al poder expresivo de los lenguajes de programación. Lo que nos concierne ahora seria la cuestión de que aspectos deben incluirse en un lenguaje de programación, para garantizar que una vez diseñado e implementado no descubramos que existen problemas computables cuyas soluciones no pueden especificarse con dicho lenguaje. Al concebir el lenguaje a utilizar sea en código binario, y las instrucciones se ejecuten como un sistema lógico computable, con lo cual se optimizaría en parte la problemática de expresar un algoritmo que sea recursivo o decidible.

La estrategia consistirá en desarrollar un lenguaje de programación esencial, con el cual se pueda expresar un programa para calcular cualquier función recursiva parcial. Esto asegurara (suponiendo que la tesis de Church-Turing es verdadera) que, mientras un lenguaje de programación cuente con las características de nuestro lenguaje esencial, permitirá expresar una solución para cualquier problema que pueda resolverse de manera algorítmica.

10.1.3. El lenguaje de la lógica proposicional

El alfabeto de la lógica de proposiciones debe proporcionar los símbolos necesarios para representar proposiciones sobre el mundo. Como el número de proposiciones que pueden manejarse en un mismo razonamiento no está limitado, debe proveer un número infinito de letras proposicionales.

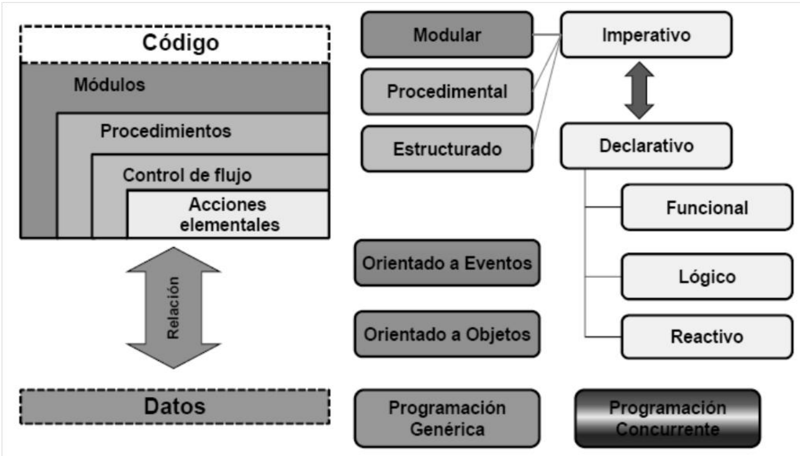
Alfabeto de la lógica de proposiciones. El alfabeto A de la Lógica de proposiciones consta de los siguientes elementos: infinitas letras proposicionales: p_0, p_1, p_2, p_3 ..., símbolos lógicos: constantes ($\top, \bot$), conectiva monaria ($\neg$) y conectivas binarias ($\wedge, \vee, \rightarrow, \leftrightarrow$) y dos símbolos auxiliares de puntuación: paréntesis izquierdo'(' y derecho ')'. Así, $A = \{p_0, p_1, p_2, p_3 \ldots, \top, \bot, \wedge, \vee, \rightarrow, \leftrightarrow, \neg (,)\}$

En las exposiciones teóricas, el número de letras proposicionales que se consideran simultáneamente es pequeño (por ejemplo, de p_0 a p_8). En estos casos se suelen notar informalmente con las últimas letras del alfabeto latino: $\{p, q, r, s, t, \ldots\}$. A las expresiones aceptables se las denominará *fórmulas o expresiones bien formadas*. El conjunto (infinito) que incluye todas estas fórmulas (y nada más) es el lenguaje de la lógica de proposiciones, que notaremos como $Form$.

El conjunto $Form$, que contiene todas las expresiones aceptables en nuestro lenguaje (y sólo éstas), es el menor de los conjuntos X que verifica que todas las fórmulas atómicas pertenecen a X; si la expresión $\varphi \in$ X entonces $(\neg\varphi) \in$ X; y si las expresiones $\varphi, \psi \in$ X entonces $(\varphi\wedge\psi), (\varphi\vee\psi), (\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi) \in$ X.

Paradigma de programación. Como una propuesta tecnológica adoptada cuyo núcleo central es incuestionable en cuanto que únicamente trata de resolver el problemas de secuenciación claramente delimitado; la resolución de este problema debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de software.

El paradigma de programación representa un enfoque particular o filosofía para diseñar soluciones. Los paradigmas difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema, en otras palabras, el *cómputo*. Tiene una estrecha relación con la formalización de determinados lenguajes en su momento de definición. Es un estilo de programación empleado. Un paradigma de programación está delimitado en el tiempo en cuanto a aceptación y uso, porque nuevos paradigmas aportan nuevas o mejores soluciones que la sustituyen parcial o totalmente.



Razones que sustenta el enfoque utilizado en la metodología con el uso de la *programación declarativa* (funcional, lógica y reactiva) dentro de los paradigmas de la programación, considerados en la jerarquía descrita en la figura.

- Un paradigma de programación indica un método de realizar cómputos y la manera en que se deben estructurar y organizar las tareas que debe llevar a cabo un programa.
- Los paradigmas fundamentales están asociados a determinados modelos de cómputo.
- También se asocian a un determinado estilo de programación
- Los lenguajes de programación suelen implementar, a menudo de forma parcial, varios paradigmas.
- Los paradigmas fundamentales están basados en diferentes modelos de cómputo y por lo tanto afectan a las construcciones más básicas de un programa.
- La división principal reside en el enfoque imperativo (indicar el *cómo* se debe calcular) y el enfoque declarativo (indicar el *qué* se debe calcular).
  - El enfoque declarativo tiene varias ramas diferenciadas: el paradigma funcional, *el paradigma lógico*, la programación reactiva y los lenguajes descriptivos.
- Otros paradigmas se centran en la estructura y organización de los programas, y son compatibles con los fundamentales: Ejemplos: Programación estructurada, modular, orientada a objetos, orientada a eventos, programación genérica.
- Por último, existen paradigmas asociados a la concurrencia y a los sistemas de tipado.

Paradigma Imperativo

- Describe cómo debe realizarse el cálculo, no el porqué.
- Un cómputo consiste en una serie de sentencias, ejecutadas según un control de flujo explícito, que modifican el estado del programa.
- Las variables son celdas de memoria que contienen datos (o referencias), pueden ser modificadas, y representan el estado del programa.
- La sentencia principal es la asignación.
  - Es el estándar 'de facto'.
  - Asociados al paradigma imperativo se encuentran los paradigmas procedural, modular, y la programación estructurada.
  - El lenguaje representativo sería FORTRAN-77, junto con COBOL, BASIC, PASCAL, C, ADA. También lo implementan Java, C++, C#, Eiffel, Python, ...

Paradigma Declarativo

- Describe que se debe calcular, sin explicitar el cómo.
- No existe un orden de evaluación prefijado.
- Las variables son nombres asociados a definiciones, y una vez instanciadas son inmutables.

- No existe sentencia de asignación.
- El control de flujo suele estar asociado a la composición funcional, la recursividad y/o técnicas de reescritura y unificación.
  - Existen distintos grados de pureza en las variantes del paradigma.
  - Las principales variantes son los paradigmas funcional, lógico, la programación reactiva y los lenguajes descriptivos.

Uso de la programación lógica. Una idea fundamental dentro de la metodológica es concebir que el problema de secuenciación de operaciones sea a través de la programación lógica y consistirá en emplear la misma lógica como *lenguaje de programación.* Hay que considerar que la lógica no es imperativa: no sirve para indicar *como* resolver el problema (*ordenes*); la lógica es declarativa: sirve para especificar *qué* problema resolver (*condiciones*), más sin embargo la programación lógica permite ambos enfoques.

a. El enfoque imperativo se diseña un algoritmo para resolver un problema, computar la solución ejecutando el algoritmo y el énfasis está en cómo resolver el problema.
b. El enfoque declarativo la lógica es capaz de describir *algoritmos* a un nivel de abstracción alto, donde se *especifican* las condiciones que satisfacen las soluciones, *deducir* las soluciones a partir de las condiciones y el énfasis está en *que* problema resolver. El problema se describe especificando *que caracteriza* a sus posibles soluciones.

El lenguajes lógicos se utilizo para especificaciones formales y pruebas de teoremas, en el estudio formal de la semántica del lenguaje y el diseño de sistemas lógicos (circuitos digitales) que se constituye en la base para expresar nuestro problema de secuenciación, pensando que el escenario de producción como un sistema lógico y podremos efectuar todas las operaciones.

Programación Lógica

- Basado en la lógica de predicados de primer orden
- Los programas se componen de hechos, predicados y relaciones.
- Evaluación basada en resolución SLD: *unificación + backtracking.*
- La ejecución consiste en la resolución de un problema de decisión, los resultados se obtienen mediante la instanciación de las variables libres.
- Lenguaje representativo: *ProLog*

La programación lógica es un paradigma de programación basado en la lógica. Los programas construidos un lenguaje lógico están construidos únicamente por expresiones lógicas, es decir, que son ciertas o falsas, en oposición a una expresión interrogativa (una pregunta) o expresiones imperativas (una orden). Un ejemplo de lenguaje lógico es Prolog (Programación lógica).

Prolog, proveniente del inglés Programming in Logic, es un lenguaje lógico bastante popular en el medio de investigación en inteligencia artificial. Prolog es un lenguaje muy diferente, tanto de los imperativos como Fortran, Pascal, C etc., como de los funcionales como Lisp. En todos los mencionados, las instrucciones se ejecutan normalmente en orden secuencial, es decir, una a continuación de otra, en el mismo orden en que están escritas, que sólo varía cuando se alcanza una instrucción de control (un bucle, una instrucción condicional o una transferencia).

En Prolog, las cosas son distintas: el orden de ejecución de las instrucciones no tiene nada que ver con el orden en que fueron escritas. Tampoco hay instrucciones de control propiamente dichas. Para trabajar con un lenguaje lógico, se debe acostumbrar a pensar de una manera muy diferente a la que se utiliza en los lenguajes clásicos.

Programación funcional. Es un paradigma de programación declarativa basado en el uso de funciones matemáticas. En la práctica, la diferencia entre una función matemática y la noción de una *función* utilizada en la programación imperativa, es que las funciones imperativas pueden tener efectos secundarios. Con código funcional, en contraste, el valor generado por una función depende exclusivamente de los argumentos alimentados a la función. Al eliminar los efectos secundarios se puede entender y predecir el comportamiento de un programa mucho más fácilmente. Ésta es una de las principales motivaciones para utilizar la programación funcional.

Los lenguajes de programación funcional, especialmente los puramente funcionales, han sido enfatizados en el ambiente académico y no tanto en el desarrollo comercial o industrial. La programación funcional también es utilizada en la industria a través de lenguajes de dominio específico como las matemáticas simbólicas.

Un lenguaje de uso específico usados comúnmente como SQL, utilizan algunos elementos de programación funcional, especialmente al procesar *valores mutables*. Las hojas de cálculo también pueden ser consideradas lenguajes de programación funcional. La programación funcional también puede ser desarrollada en lenguajes que no están diseñados específicamente para la programación funcional. En el caso de C, por ejemplo, que es un lenguaje de programación imperativo, existe una librería que describe como aplicar conceptos de programación funcional.

El objetivo es conseguir un lenguaje expresivo y matemáticamente elegante, que no sea necesario bajar al nivel de la máquina para describir el proceso llevado a cabo por el programa, y evitar el concepto de estado del cómputo.

Programación con restricciones. La Programación por restricciones es un paradigma de la programación en informática, donde las relaciones entre las variables son expresadas en términos de restricciones (ecuaciones). Actualmente es usada como una tecnología de software para la descripción y resolución de problemas combinatorios particularmente difíciles, especialmente en las áreas de planificación y programación de tareas (calendarización).

Este paradigma representa uno de los desarrollos más fascinantes en los lenguajes de programación desde 1990 y no es sorprendente que recientemente haya sido identificada por la ACM (*Asociación de Maquinaria Computacional*) como una dirección estratégica en la investigación en computación.Se trata de un paradigma de programación basado en la especificación de un conjunto de restricciones, las cuales deben ser satisfechas por cualquier solución del problema planteado, en lugar de especificar los pasos para obtener dicha solución.

La programación con restricciones se relaciona mucho con la *programación lógica* y con la *investigación operativa*.

De hecho cualquier programa lógico puede ser traducido en un programa con restricciones y viceversa. Muchas veces los programas lógicos son traducidos a programas con restricciones debido a que la solución es más eficiente que su contraparte.

La diferencia entre ambos radica principalmente en sus estilos y enfoques en el modelado del mundo. Para ciertos problemas es más natural (y por ende más simple) escribirlos como programas lógicos, mientras que en otros es más natural escribirlos como programas con restricciones.

El enfoque de la programación con restricciones se basa principalmente en buscar un estado en el cual una gran cantidad de restricciones sean satisfechas simultáneamente. Un problema se define típicamente como un estado de la realidad en el cual existe un número de variables con valor desconocido. Un programa basado en restricciones busca dichos valores para todas las variables.

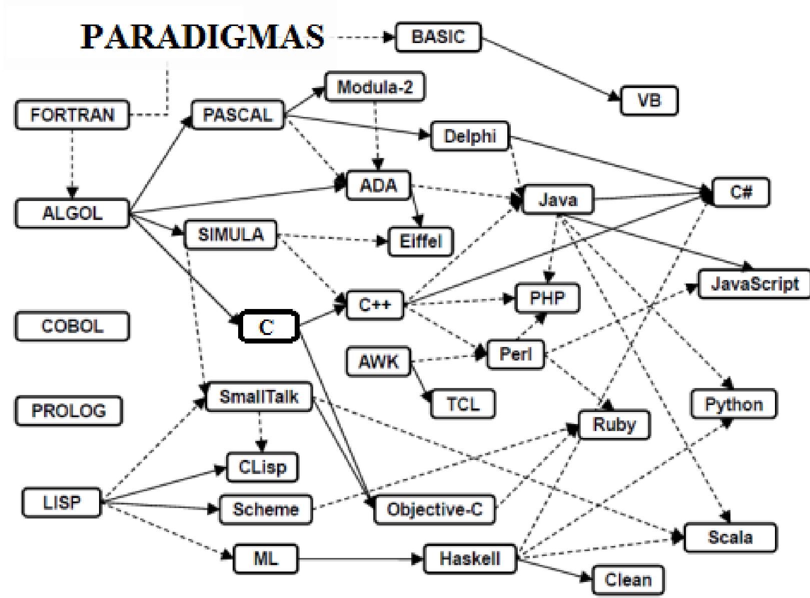
Algunos dominios de aplicación de este paradigma son:

- Dominios *booleanos,* donde solo existen restricciones del tipo *verdadero/falso*.
- Dominios en variables *enteras* y *racionales*.
- Dominios *lineales,* donde solo se describen y analizan *funciones lineales*.
- Dominios *finitos*, donde las restricciones son definidas en *conjuntos finitos.*
- Dominios *mixtos*, los cuales involucran dos o más de los anteriores.

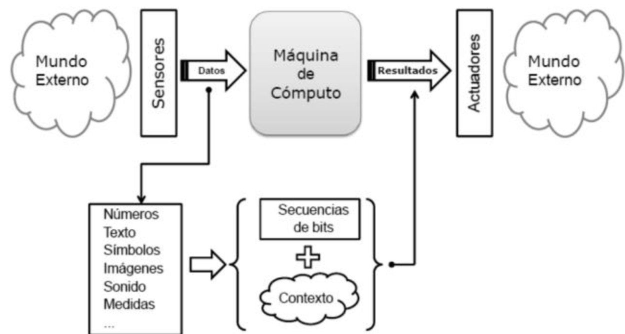
Los lenguajes de programación con restricciones son típicamente ampliaciones de otro lenguaje. El primer lenguaje utilizado a tal efecto fue Prolog. Por esta razón es que este campo fue llamado inicialmente

Programación Lógica con Restricciones. Ambos paradigmas comparten características muy similares, tales como las variables lógicas (una vez que una variable es asignada a un valor, no puede ser cambiado), o el *backtracking*. La programación con restricciones puede ser implementada como un lenguaje propio o como bibliotecas para ser usadas en algún lenguaje de programación imperativo. Algunos lenguajes populares de programación con restricciones son:

- CHIP V5 (Basado en Prolog, también existen bibliotecas en C y C++, propietario) siendo el elegido
- B-Prolog (Basado en Prolog, propietario)
- Ciao Prolog (Basado en Prolog, software libre: GPL/LGPL)
- ECLiPSe (Basado en Prolog, software libre)
- Mozart (Basado en Oz, software libre: X11)
- SICStus (Basado en Prolog, propietario)
- GNU Prolog (Basado en Prolog, software libre)
- SWI-Prolog Un entorno Prolog que contiene varias librerías para soluciones con restricciones (LGPL)



Modelo de cómputo. El concepto de cómputo puede modelizarse por el concepto matemático de función*: aplicación de un dominio de valores a un rango de resultados donde cada valor puede estar asociado como máximo a un resultado*. Se usa el modelo *caja de conexiones* para las funciones en la metodología planteada.



Máquinas y modelos de cómputo. Jerarquía de niveles según capacidad expresiva y poder de cómputo:

- *Circuitos combinacionales* (sistemas lógicos combinacionales y secuenciales)
- *Máquinas de estado finito / Autómatas secuenciales*
- *Máquinas de Turing* / Máquinas de registros (RAM)

Modelos formales

- Funciones parciales recursivas
- Cálculo lambda / Lógica combinatoria
- Lógica de predicados + unificación
- Sistemas de reescritura

Arquitecturas

- Modelo Von-Neumman
- Modelo Harvard
- Paralelismo

Máquinas de Turing

- Cinta de entrada:
  - Alfabeto de sólo dos símbolos: 0 (blanco) y 1 (punto)
  - Al comienzo la entrada se situa a la derecha de la cabeza
  - Al finalizar la salida se encuentra a la izquierda de la cabeza
- Controlador:
  - Cada máquina tiene n estados posibles
  - La máquina comienza siempre en el estado 0
  - Dispone de un único registro que almacena el estado actual
  - La lista de transiciones tiene n filas, una por cada estado, y dos columnas, una por cada valor posible de la celda actual {0 ó 1}

    - Cada transición indica lo siguiente:
        - Nuevo estado al que pasa la máquina
        - Símbolo {0 ó 1} que se escribe en la celda actual
        - Movimiento de la cabeza: I (izquierda), D (derecha), S (derecha y parada)

Computabilidad

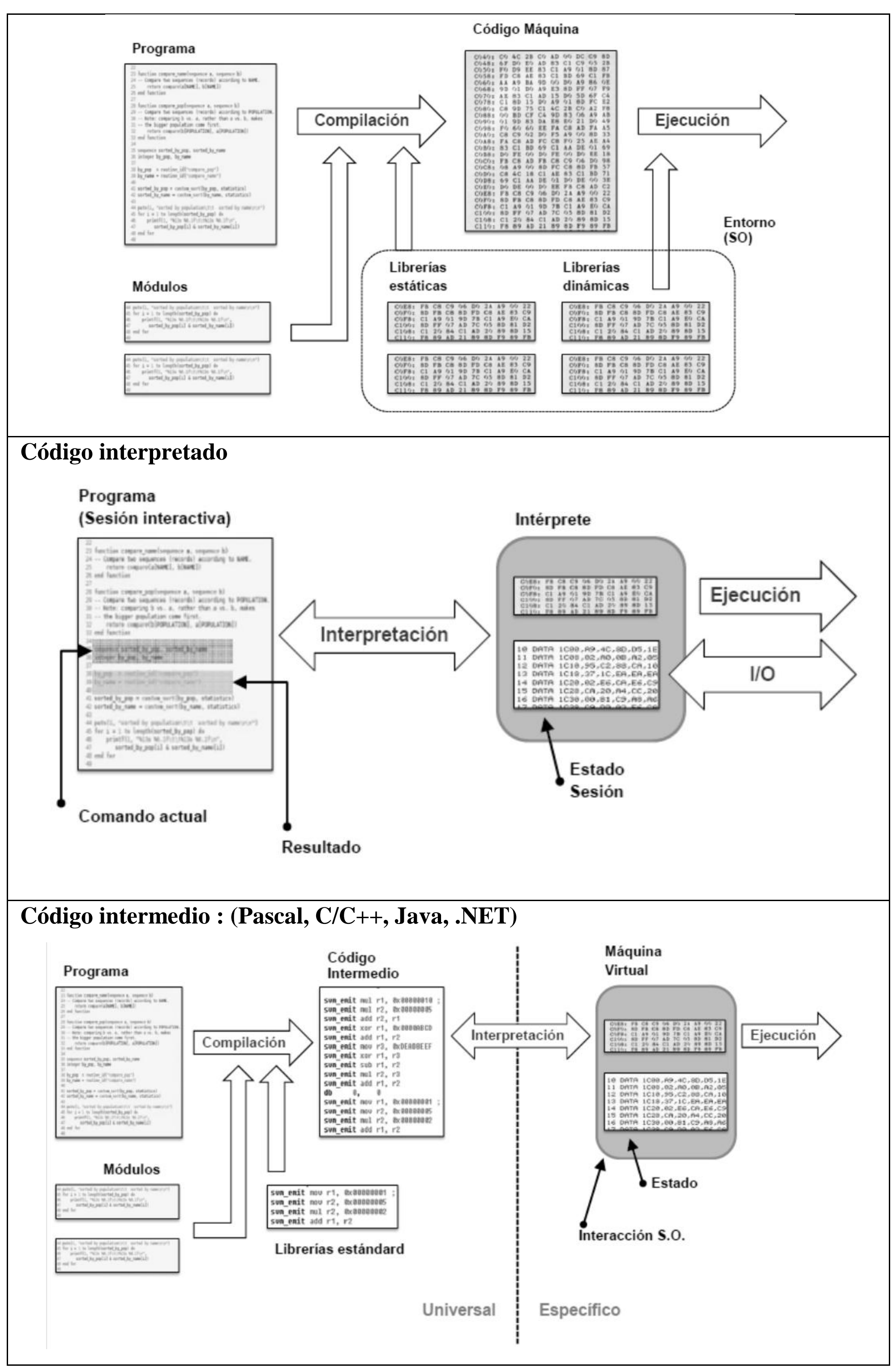
- *Algoritmo:* Procedimiento sistemático que permite resolver un problema en un número finito de pasos, cada uno de ellos especificado de manera efectiva y sin ambigüedad.
- *Función computable*: Aquella que puede ser calculada mediante un dispositivo mecánico dado un tiempo y espacio de almacenamiento ilimitado (pero finito)
- No importa la eficiencia, sino la posibilidad de ser calculada.
- ¿Existen funciones no computables?
- *Entscheidungsproblem*: Décima pregunta de Hilbert (Bolonia, 1928): ¿Existe un procedimiento mecánico (algorítmico) general para resolver toda cuestión matemática bien definida?

Lenguajes de Programación

- Lenguaje artificial diseñado para expresar cómputos que pueden ser llevados a cabo por una máquina.
    - Basado en un modelo de cómputo (que puede o no coincidir con el de la máquina en que se va a ejecutar)
    - Define un nivel de abstracción más elevado (más cercano al programador)
    - Debe traducirse a un código que pueda entender el procesador: el código máquina.
- Modos de traducción:
    - Lenguaje Compilado
    - Lenguaje Interpretado (Entorno interactivo)
    - Lenguaje traducido a Código Intermedio (Java - Bytecodes,.NET - IDL)

Estrategias de traducción

Código compilado



Es importante cumplir con las propiedades descritos en la matemática ya que son necesarios para efectuar las operaciones algebraicas en la lógica e igualmente afianzar el planteamiento formulado.

11. ANÁLISIS DE CASOS

Para el análisis de casos específico de producción bajo el escenario *flow shop* es indispensable conocer el proceso físico de producción y su distribución (*layout*) de centros de trabajo en la misma planta, asimismo de la gestión de datos de productos (PDM) donde se centra en la gestión y seguimiento de la creación, cambio y archivo de toda la información relacionada con cada producto, como la publicación de datos que usualmente involucran las especificaciones técnicas del producto, especificaciones para la fabricación y desarrollo (*ruta de fabricación*), y los tipos de materiales que se requerirán para producir, construir y manipular la estructura del producto *lista de materiales* (BOM).

11.1. CASO 1: Pasos del proceso de fabricación del neumático (llanta).

PASO 1. Descripción del proceso de producción

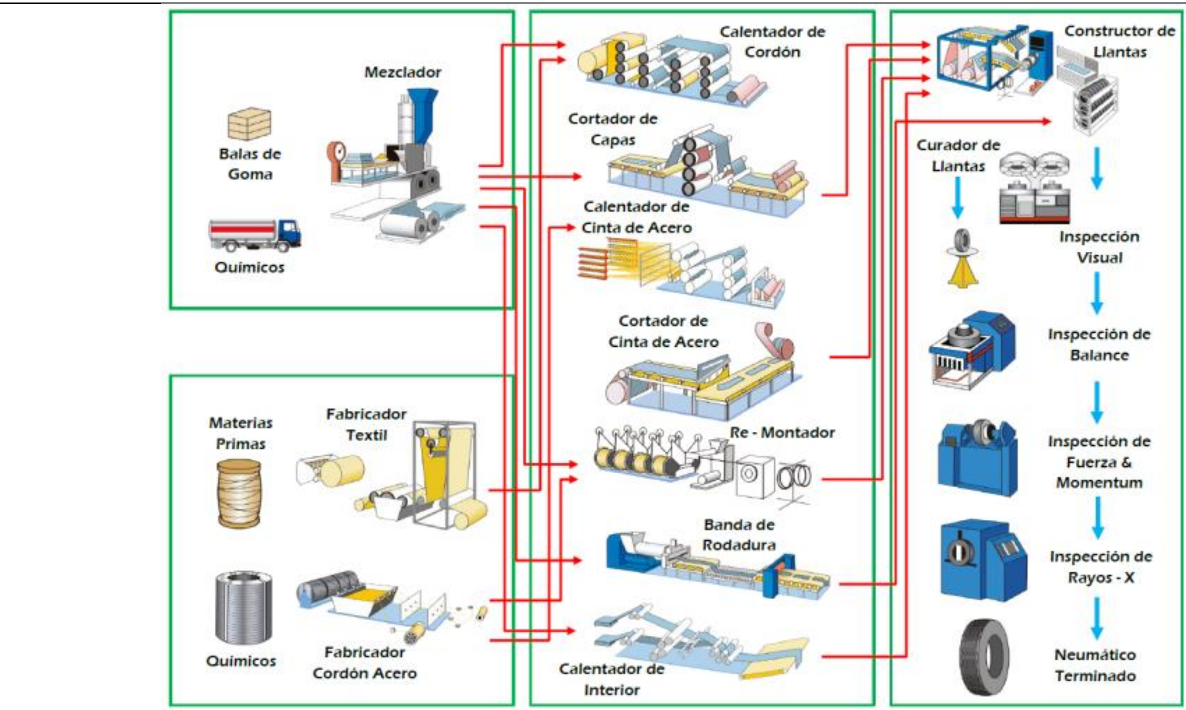
1- Conocimiento por medio de la investigación
Se estudian los hábitos de manejo y de uso de neumáticos de las personas para asegurar que los neumáticos cubran las necesidades de todos.

2- Desarrollo y combinación de materiales
Se utilizan más de 200 *componentes* en un neumático. Todos juegan un papel fundamental en la seguridad, el rendimiento del combustible, el funcionamiento y el cuidado del medio ambiente.
Estos componentes se dividen en cinco grupos:

- *Goma natural*: es el componente principal de las capas de la banda de rodamiento.
- *Goma sintética*: parte de las bandas de los neumáticos de autos, camionetas y 4x4.
- *Negro de carbón y sílice*: usado como agente de refuerzo para mejorar la durabilidad.
- *Cables de refuerzo metálicos y textiles*: el esqueleto del neumático, generando formas geométricas y ofreciendo rigidez.
- *Varios productos químicos:* para obtener cualidades únicas, como resistencia a la rodadura o alta adherencia.

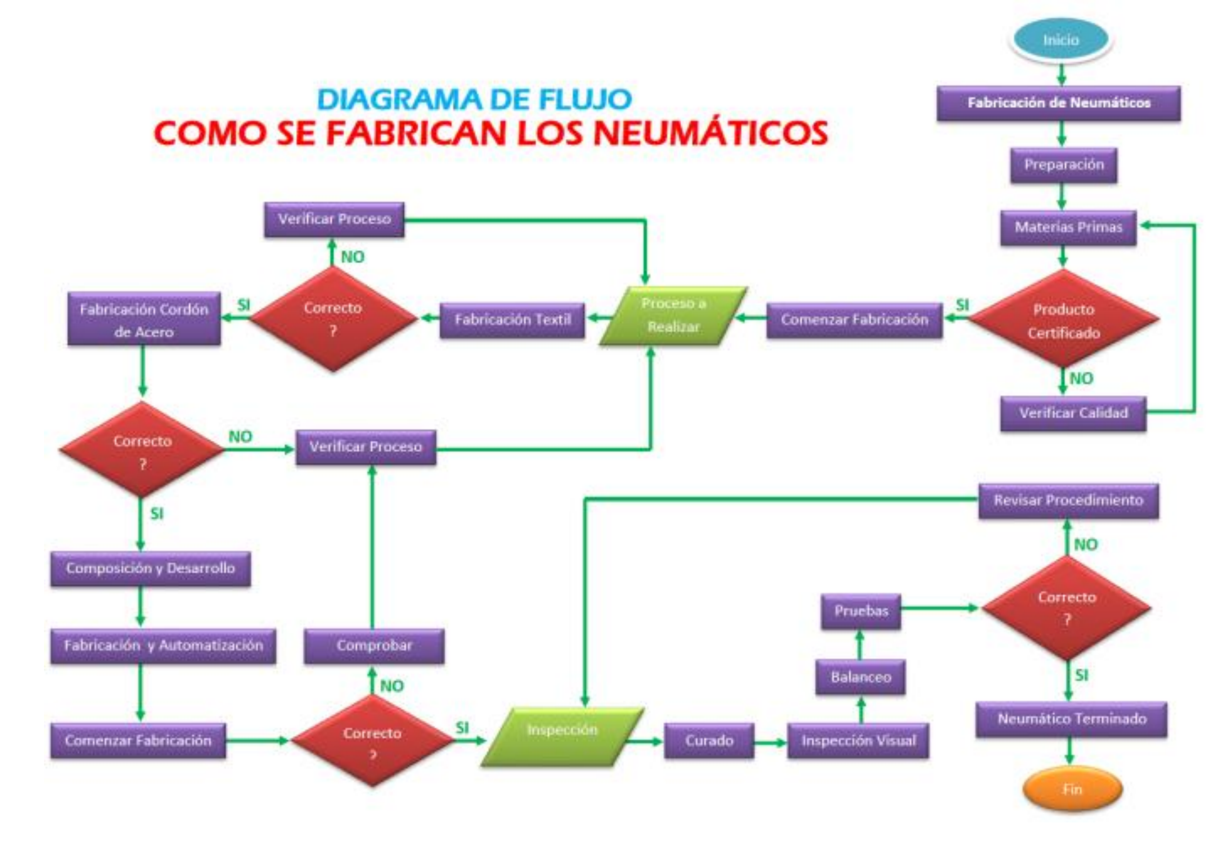
3- Diseño
Se crean varios diseños y se usan simulaciones para evaluar y seleccionar los mejores conceptos de neumático.

4- Fabricación
Se crea cada neumático por medio de procesos tanto manuales como mecánicos y el conocimiento de expertos. Cuando es necesario, se ha desarrollando maquinaria propia para lograr mejores exigencias.



5- *Control de calidad*

El control de calidad no es solo el último paso. Se mide la calidad a lo largo de todo el proceso. Se realizan más de mil millones de kilómetros de pruebas de neumáticos al año, el equivalente a manejar alrededor del planeta 40 veces.



Administración de datos de Producto (PDM)

1. ***Forro interno***: una capa de goma sintética hermética (es el equivalente moderno al tubo interno).
2. ***Capa de la carcasa:*** la capa que hay sobre el forro interno, hecho de cables de fibra textil, amalgamada en la goma. Estos cables determinan en gran medida la resistencia del neumático y le ayudan a resistir la presión. Los neumáticos estándar contienen cerca de 1.400 cables, y cada uno puede resistir una fuerza de 15 kg.
3. ***Área de talón inferior:*** es allí donde el neumático de goma se adhiere al aro de metal. La potencia del motor y del frenado se transmite desde el aro al neumático por medio del área de contacto con la superficie de la calle.
4. ***Talones:*** se ajustan con firmeza en contra del aro del neumático para garantizar un calce hermético y mantener el neumático ubicado correctamente en el aro. Cada cable puede resistir una carga de hasta 1.800 kg sin comprometer el frenado. Existen ocho en el auto, dos por neumático. Equivalen a un total de 14.400 kg de fuerza de resistencia. Un auto promedio pesa cerca de 1.500 kg.
5. ***Flanco:*** protege el costado del neumático contra impactos con las curvas y con la calle. Hay detalles importantes acerca del neumático escritos en el flanco, como el ancho del neumático y el índice de velocidad.
6. ***Capa de la cubierta***: determina en gran medida la resistencia del neumático. Está hecha de cables de acero muy finos y resistentes, amalgamados con la goma. Esto significa que el neumático puede resistir la tensión de los giros, y no se expande debido a la rotación del neumático. Es además lo suficientemente flexible para absorber las deformaciones causadas por lomos de burro, pozos, y otros obstáculos de la calle.
7. ***Capa de tapa (o lona de cima de cero grados):*** esta importante capa de seguridad reduce la fricción causada por el calor y ayuda a mantener la forma del neumático cuando se maneja a alta velocidad. Para evitar el estiramiento centrífugo del neumático, se utilizan cables reforzados a base de nailon incrustados en una capa de la goma y alrededor de la circunferencia del neumático.
8. ***Pliegues de corona (o lonas de cima):*** ofrecen una base rígida para la banda de rodamiento.
9. ***Banda de rodamiento:*** ofrece tracción y adherencia al doblar para el neumático, y está diseñada para resistir el desgaste, la abrasión y el calor.

Atributos que definen la referencia del neumático (Llanta)	
Sección Este número de tres dígitos representa el ancho de la llanta en milímetros (mm), medidos de flanco a flanco. Dado que es una medida condicionada por la anchura de la llanta, la medición se efectúa cuando la llanta está montado en la llanta de las dimensiones para las cuales fue diseñado.	
Relación de aspecto La relación de aspecto es la relación entre la altura del flanco de la llanta y el ancho de sección, expresada como porcentaje. En este ejemplo, la altura del flanco de la llanta es de aproximadamente un 55% de la sección de la llanta. Cuanto mayor sea la cifra, más alto será el flanco; cuanto menor sea la cifra, más bajo será el flanco.	
Tamaño de la llanta El diámetro de la llanta en pulgadas.	
Índice de carga Un número asignado, comprendido entre 0 y 130, que corresponde a la capacidad de carga de la llanta. Cuanto mayor sea el valor del índice de carga de la llanta, mayor será su capacidad de carga.	
Índice de velocidad El índice de velocidad indica la velocidad máxima a la que la llanta puede transportar carga en determinadas condiciones de uso. Los códigos de velocidad de la mayoría de llantas para automóviles y camiones ligeros van desde "L" (el más bajo) hasta "Z" e "Y" (los más altos).	

Pulgadas	13”	14”	15”	16”	17”	18”	19”	20”
Cantidad	3	11	22	38	39	31	25	8
$\boldsymbol{p_i}$	$\boldsymbol{p_1}$	$\boldsymbol{p_2}$	$\boldsymbol{p_3}$	$\boldsymbol{p_4}$	$\boldsymbol{p_5}$	$\boldsymbol{p_6}$	$\boldsymbol{p_7}$	$\boldsymbol{p_8}$
Dimensiones	165/65 R13 77T 165/70 R13 79T 175/70 R13 82T	165/65 R14 79T 165/70 R14 81T 165/70 R14 89R 175/65 R14 82H 175/70 R14 88T 185/60 R14 82H 185/65 R14 86H 185/70 R14 88H 195/60 R14 86H 195/80 R14 106R 205/75 R14 109Q	175/65 R15 84H 185/55 R15 86V 185/60 R15 88H 185/65 R15 88H 195/50 R15 82V 195/55 R15 85V 195/60 R15 88H 195/65 R15 91H 195/70 R15 104R 195/80 R15 106R 205/55 R16 91H 205/60 R15 91V 205/65 R15 94H 205/70 R15 106R 205/70 R15 96T 215/70 R15 109S 215/75 R15 100T 225/70 R15 112R 225/75 R15 108S 235/75 R15 105T 235/75 R15 108T 31X10.50R15LT 109S	185/55 R16 83V 195/45 R16 84V 195/55 R16 87H 195/55 R16 91V ZP 195/75 R16 107R 205/45 R16 87W 205/50 R16 87V 205/55 R16 91V 205/55 R16 91V ZP 205/55 R16 94W 205/60 R16 92H 205/60 R16 96V 205/65 R16 107T 205/75 R16 110R 215/45 R16 90V 215/55 R16 97W 215/60 R16 95H 215/60 R16 99V 215/65 R16 98H 215/65 R16 98T 215/70 R16 100H 215/70 R16 104H 215/75 R16 116R 225/50 R16 92V 225/55 R16 95W 225/60 R16 98W 225/70 R16 101T 225/75 R16 115S 225/75 R16 118R 235/60 R16 100H 235/70 R16 104T 235/70 R16 106T 245/70 R16 111T 255/65 R16 106T 255/70 R16 111T 265/70 R16 112T 265/75 R16 123R 275/70 R16 114H	205/45 R17 88Y 205/50 R17 89W ZP 205/50 R17 93W 205/55 R17 91W ZP 215/40 R17 87W 215/45 R17 91Y 215/50 R17 95W 215/55 R17 94V 215/55 R17 98W 215/60 R17 96H 215/60 R17 96V 225/45 R17 91W ZP 225/45 R17 94W 225/45 R17 94Y 225/50 R17 94W ZP 225/50 R17 98W 225/50 R17 98Y 225/55 R17 101W 225/60 R17 99V 225/65 R17 102H 235/45 R17 97W 235/55 R17 99H 235/55 R17 99V 235/60 R17 102V 235/65 R17 104V 235/65 R17 104W 235/65 R17 108H 245/40 R17 95Y 245/45 R17 99Y 245/65 R17 107T 245/65 R17 111T 245/70 R17 108S 255/40 R17 94Y 255/60 R17 106V 255/65 R17 114H 265/65 R17 112T 265/70 R17 121R 275/55 R17 109V LT245/75R17 121/118R	215/40 R18 89Y 225/40 R18 88Y ZP 225/40 R18 92W ZP 225/40 R18 92Y 225/45 R18 95Y ZP 225/55 R18 98V 225/60 R18 100H 235/40 ZR18 95Y 235/45 R18 98Y 235/50 R18 101Y 235/60 R18 103W 235/65 R18 104H 245/40 R18 97Y 245/40 R18 97Y ZP 245/45 R18 100W 245/50 R18 100Y ZP 245/60 R18 105H 255/35 R18 90Y ZP 255/35 R18 94Y 255/40 R18 99Y 255/45 R18 99Y 255/55 R18 105W 255/60 R18 112H 265/35 R18 97Y 265/40 R18 101Y 265/60 R18 110H 265/60 R18 110T 265/70 R18 124R 275/45 R18 103Y 285/35 R18 101Y 285/60 R18 120V	225/40 R19 93Y ZP 225/45 R19 96Y 235/35 R19 87Y 235/50 R19 99V 245/35 R19 93Y 245/40 R19 94Y ZP 245/40 R19 98Y 245/45 R19 98Y ZP 255/35 R19 96Y 255/40 R19 100Y 255/45 R19 100Y 255/50 R19 103y 255/50 R19 107V 255/50 R19 107W ZP 255/55 R19 111V 265/35 R19 94Y 265/50 R19 110Y 275/35 R19 100Y 275/35 R19 96Y ZP 275/40 R19 101Y ZP 275/45 R19 108Y 275/55 R19 111W 285/35 R19 99Y 285/45 R19 111W 295/30 R19 100Y	235/35 R20 92Y 255/40 R20 101Y 255/55 R20 110Y 275/40 R20 106Y 275/45 R20 110Y 275/55 R20 113H 295/30 R20 101Y 295/40 R20 106Y

PASO 2. Conjuntos

$\boldsymbol{c_i}$	**Descripción Centro de Trabajo**
c_1	Mezclador (*Mixing*)
	Fabricador Textil
	Fabricador cinta de Acero
c_2	Calentador de cordón (*Extruding*)
	Cortador de tapas
c_3	Calentador de cintas de acero (*Bead*)
c_4	Cortador Cinta de acero (*Cutting*)
	Re-Montador
c_5	Banda de Rodadura (*Calendering*)
	Calentador de interior
c_6	Constructor de llantas (*Building*)
c_7	Curador de llantas (*Curing*)
c_8	Inspección (*Inspection*)
	Neumático Terminado Embalaje (*Shipping*)

(a) demanda de productos (und) p_i – Pronostico semanal

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1400	600	1100	1500	1700	300	900	1400

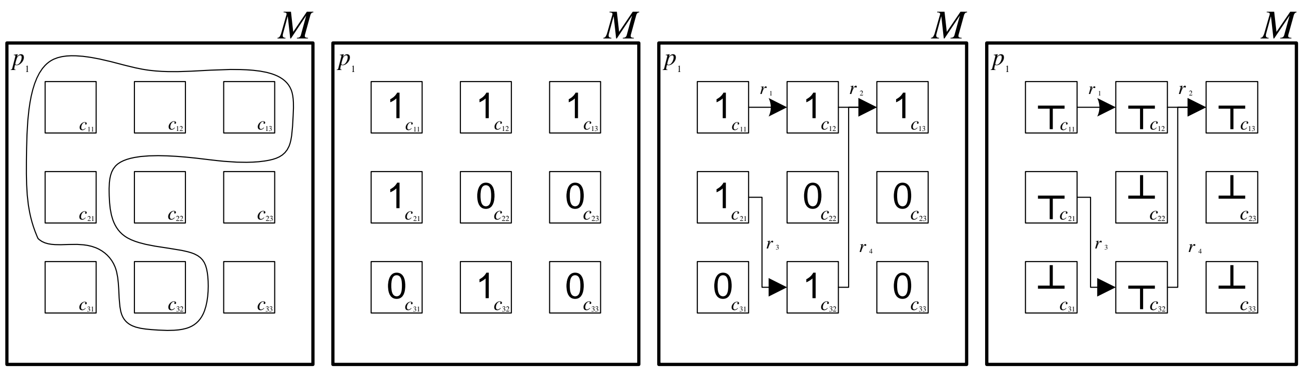
(b) tiempo de operación (min) c_i – centro de trabajo

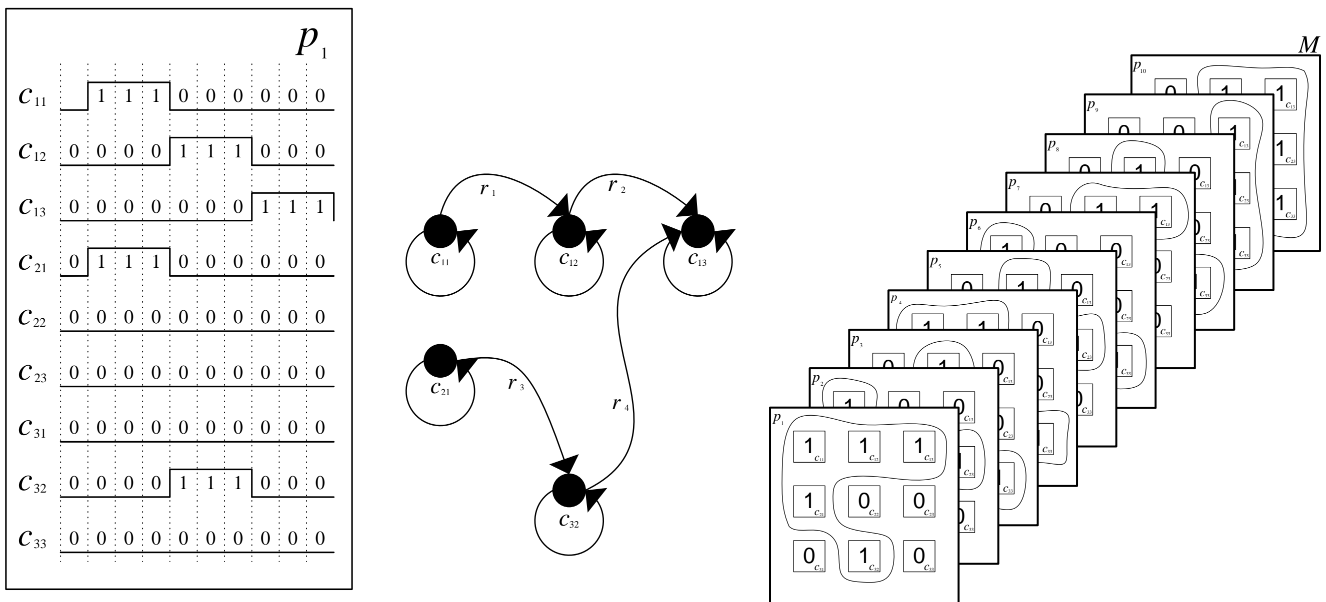
c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
4	3	3	2	5	3	20	3

(c) Matriz de indecencia (relaciones entre productos y centro de trabajo)

Pulgadas	13”	14”	15”	16”	17”	18”	19”	20”
	$\boldsymbol{p_1}$	$\boldsymbol{p_2}$	$\boldsymbol{p_3}$	$\boldsymbol{p_4}$	$\boldsymbol{p_5}$	$\boldsymbol{p_6}$	$\boldsymbol{p_7}$	$\boldsymbol{p_8}$
c_1	1	1	0	1	0	1	0	0
c_2	1	0	1	1	1	0	1	1
c_3	1	0	0	0	0	0	1	0
c_4	1	1	1	1	1	1	0	0
c_5	0	1	0	1	1	1	1	1
c_6	0	1	0	0	1	0	0	0
c_7	0	0	1	0	1	1	1	1
c_8	1	1	1	1	1	1	1	1

PASO 3. Relaciones





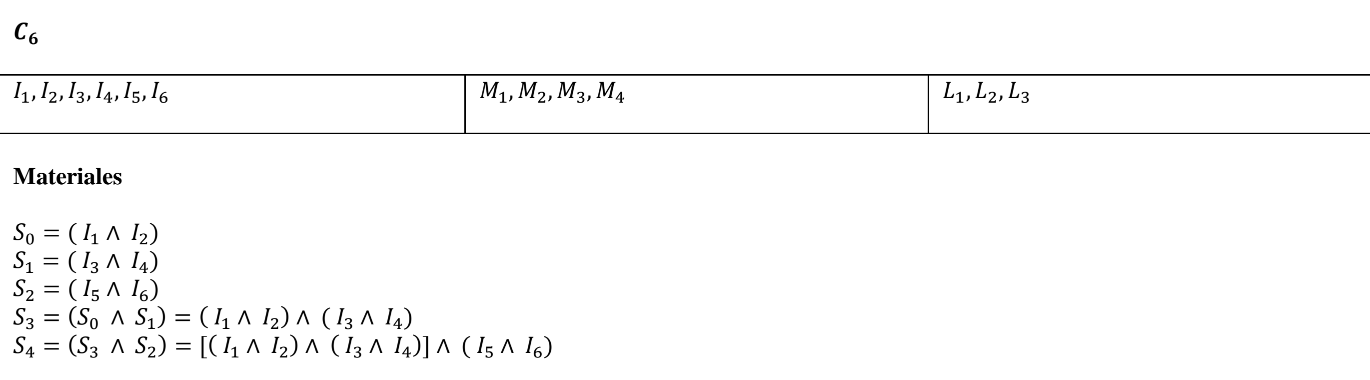
PASO 4. Funciones de recurrencia

PASO 5. Lógica proposicional

Formula lógica proposicional fbf de Productos $\boldsymbol{p_i}$ por Centro de Trabajo $\boldsymbol{C_{ij}}$

Producto	**Formula lógica combinacional fbf.** Centro de Trabajo	**Nivel FMS MT**
p_1	$(c_{11} \wedge c_{12} \wedge c_{13} \wedge c_{21} \wedge \neg c_{22} \wedge \neg c_{23} \wedge \neg c_{31} \wedge c_{32} \wedge \neg c_{33})$	I
p_2	$(c_{11} \wedge \neg c_{12} \wedge \neg c_{13} \wedge c_{21} \wedge c_{22} \wedge c_{23} \wedge \neg c_{31} \wedge c_{32} \wedge \neg c_{33})$	I
p_3	$(\neg c_{11} \wedge c_{12} \wedge \neg c_{13} \wedge c_{21} \wedge \neg c_{22} \wedge \neg c_{23} \wedge c_{31} \wedge c_{32} \wedge c_{33})$	I
p_4	$(c_{11} \wedge c_{12} \wedge \neg c_{13} \wedge c_{21} \wedge c_{22} \wedge \neg c_{23} \wedge \neg c_{31} \wedge c_{32} \wedge c_{33})$	I
p_5	$(\neg c_{11} \wedge c_{12} \wedge \neg c_{13} \wedge c_{21} \wedge c_{22} \wedge c_{23} \wedge c_{31} \wedge c_{32} \wedge \neg c_{33})$	I
p_6	$(c_{11} \wedge \neg c_{12} \wedge \neg c_{13} \wedge c_{21} \wedge c_{22} \wedge \neg c_{23} \wedge c_{31} \wedge c_{32} \wedge c_{33})$	I
p_7	$(\neg c_{11} \wedge c_{12} \wedge c_{13} \wedge \neg c_{21} \wedge c_{22} \wedge \neg c_{23} \wedge c_{31} \wedge c_{32} \wedge \neg c_{33})$	I
p_8	$(\neg c_{11} \wedge c_{12} \wedge \neg c_{13} \wedge \neg c_{21} \wedge c_{22} \wedge \neg c_{23} \wedge c_{31} \wedge c_{32} \wedge c_{33})$	I
p_9	$(\neg c_{11} \wedge \neg c_{12} \wedge c_{13} \wedge c_{21} \wedge c_{22} \wedge c_{23} \wedge \neg c_{31} \wedge c_{32} \wedge c_{33})$	I
p_{10}	$(\neg c_{11} \wedge c_{12} \wedge c_{13} \wedge \neg c_{21} \wedge c_{22} \wedge c_{23} \wedge c_{31} \wedge c_{32} \wedge c_{33})$	I

Formula lógica proposicional fbf de Recursos y la salida del Centro de Trabajo $\boldsymbol{C_{11}}$



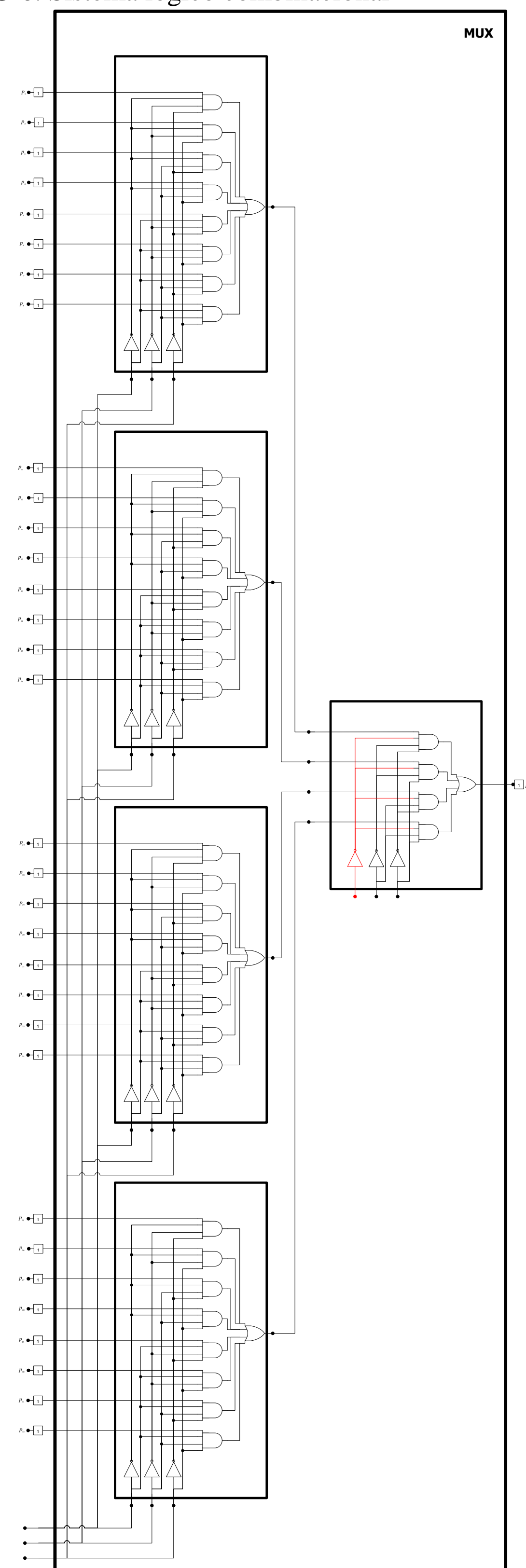
$\boldsymbol{C_6}$		
$I_1, I_2, I_3, I_4, I_5, I_6$	M_1, M_2, M_3, M_4	L_1, L_2, L_3

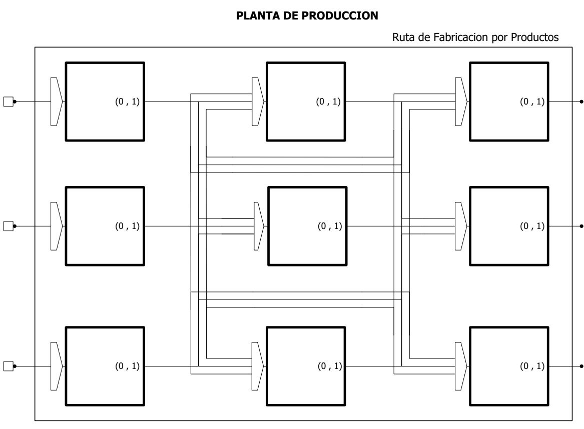
Materiales

$S_0 = (I_1 \wedge I_2)$
$S_1 = (I_3 \wedge I_4)$
$S_2 = (I_5 \wedge I_6)$
$S_3 = (S_0 \wedge S_1) = (I_1 \wedge I_2) \wedge (I_3 \wedge I_4)$
$S_4 = (S_3 \wedge S_2) = [(I_1 \wedge I_2) \wedge (I_3 \wedge I_4)] \wedge (I_5 \wedge I_6)$

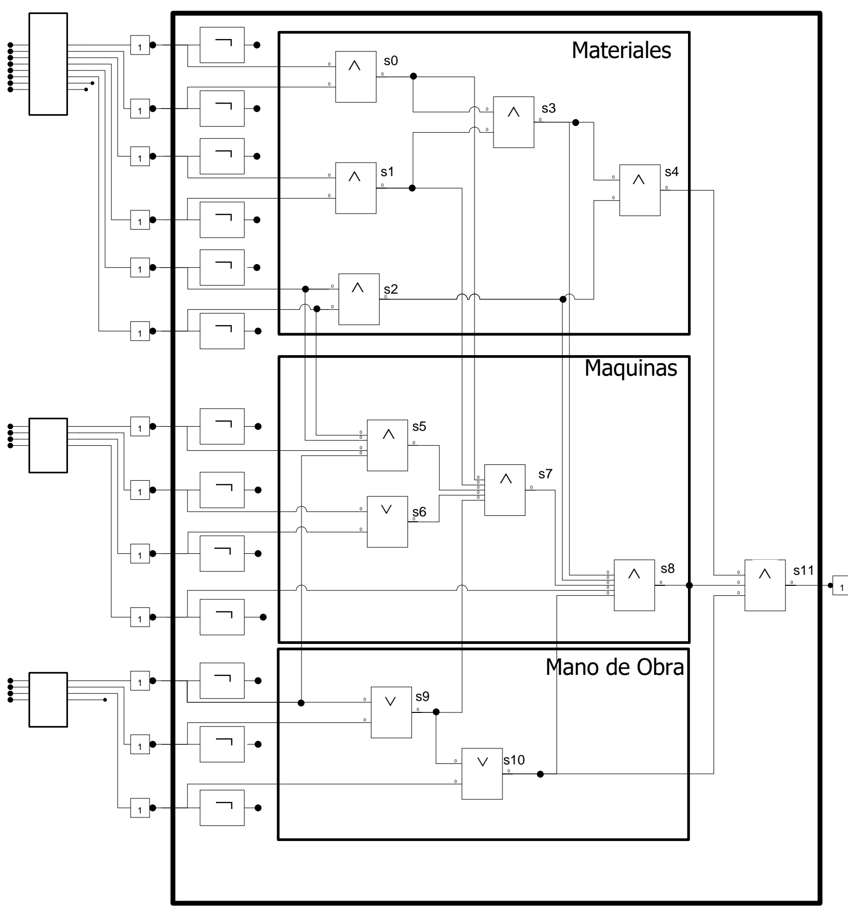
Equipos $S_5 = (M_1 \wedge I_5 \wedge I_6 \wedge L_1)$ $S_6 = (M_2 \vee M_3)$ $S_7 = (S_0 \wedge S_1 \wedge S_5 \wedge S_6 \wedge S_9) = (I_1 \wedge I_2) \wedge (I_3 \wedge I_4) \wedge (M_1 \wedge I_5 \wedge I_6 \wedge L_1) \wedge (M_2 \vee M_3) \wedge (L_1 \vee L_2)$ $S_8 = (M_4 \wedge S_7 \wedge S_2 \wedge S_3 \wedge S_{10})$ $= M_4 \wedge [(I_1 \wedge I_2) \wedge (I_3 \wedge I_4) \wedge (M_1 \wedge I_5 \wedge I_6 \wedge L_1) \wedge (M_2 \vee M_3) \wedge (L_1 \vee L_2)] \wedge (I_5 \wedge I_6)$ $\wedge [(I_1 \wedge I_2) \wedge (I_3 \wedge I_4)] \wedge [(L_1 \vee L_2) \vee L_3]$
Mano de obra $S_9 = (L_1 \vee L_2)$ $S_{10} = (S_9 \vee L_3) = (L_1 \vee L_2) \vee L_3$
$\boldsymbol{S_{11} = (S_4 \wedge S_8 \wedge S_{10}) = \{ [(I_1 \wedge I_2) \wedge (I_3 \wedge I_4)] \wedge (I_5 \wedge I_6)\} \wedge \{M_4 \wedge [(I_1 \wedge I_2) \wedge (I_3 \wedge I_4) \wedge (M_1 \wedge I_5 \wedge I_6 \wedge L_1) \wedge}$ ***M2∨M3∧ L1∨L2∧ I5∧I6∧I1∧I2∧ I3∧I4∧[L1∨L2 ∨L3]}∧{ [L1∨L2 ∨L3]}***

PASO 6. Sistema lógico combinacional



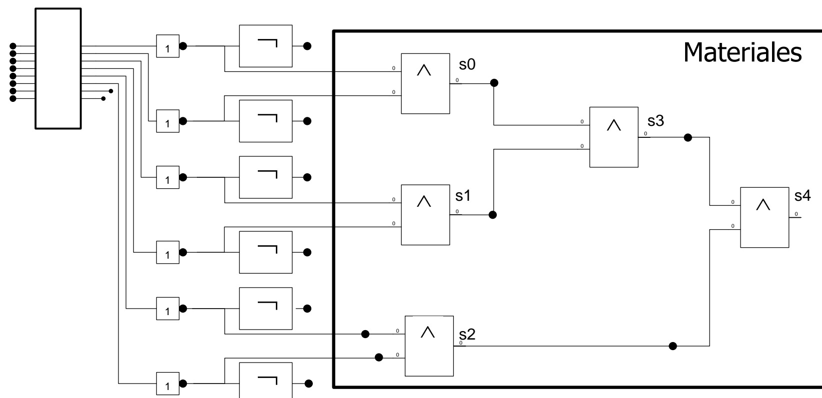


(Árbol sintáctico) Sistema lógico combinacional definido en centro de trabajo por recursos



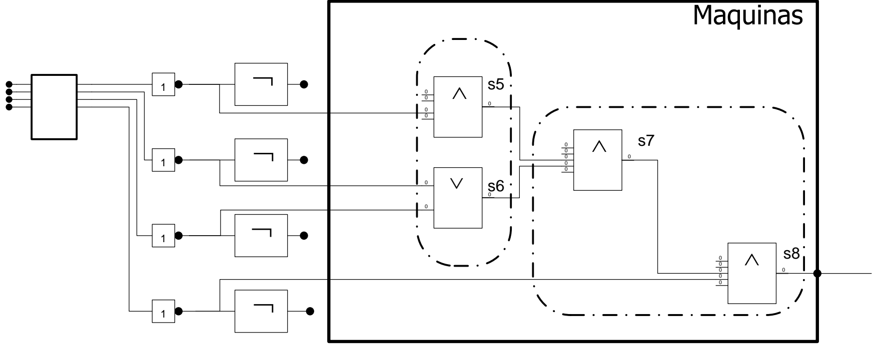
Fuente: Autor

(Árbol sintáctico) Sistema lógico combinacional para Materiales $\boldsymbol{m_i}$ por productos $\boldsymbol{p_i}$



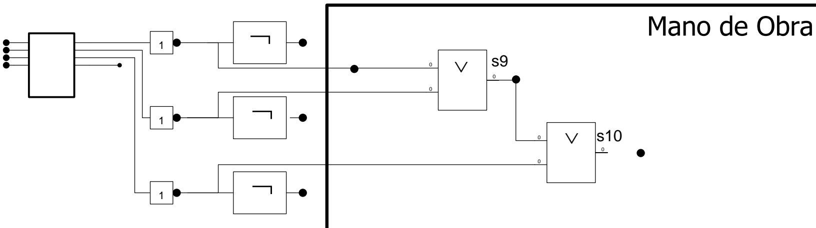
Fuente: Autor

(Árbol sintáctico) Sistema lógico combinacional para máquinas $\boldsymbol{e_i}$ por productos $\boldsymbol{p_i}$

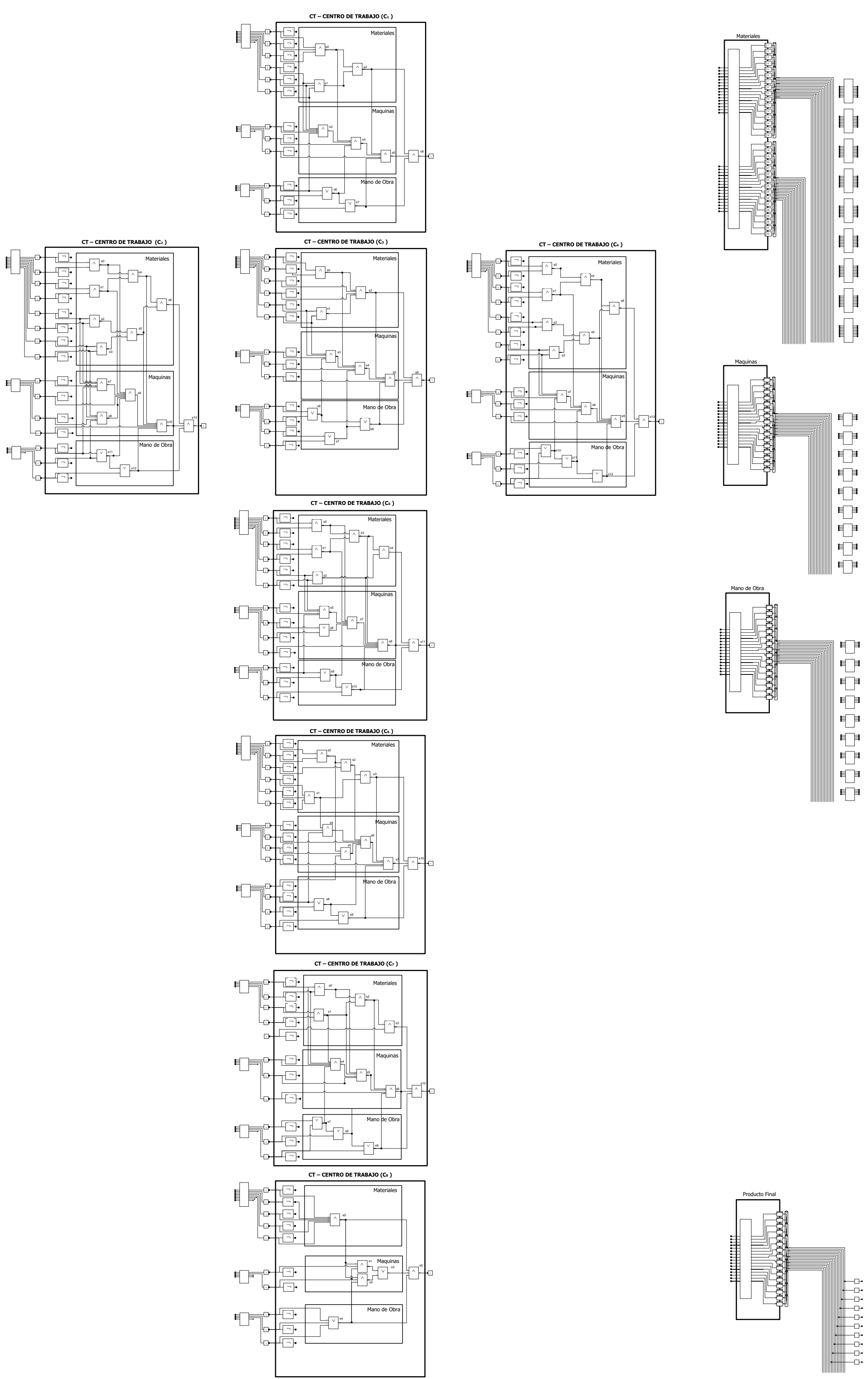


Fuente: Autor

(Árbol sintáctico) Sistema lógico combinacional para Mano de Obra $\boldsymbol{l_i}$ por productos $\boldsymbol{p_i}$



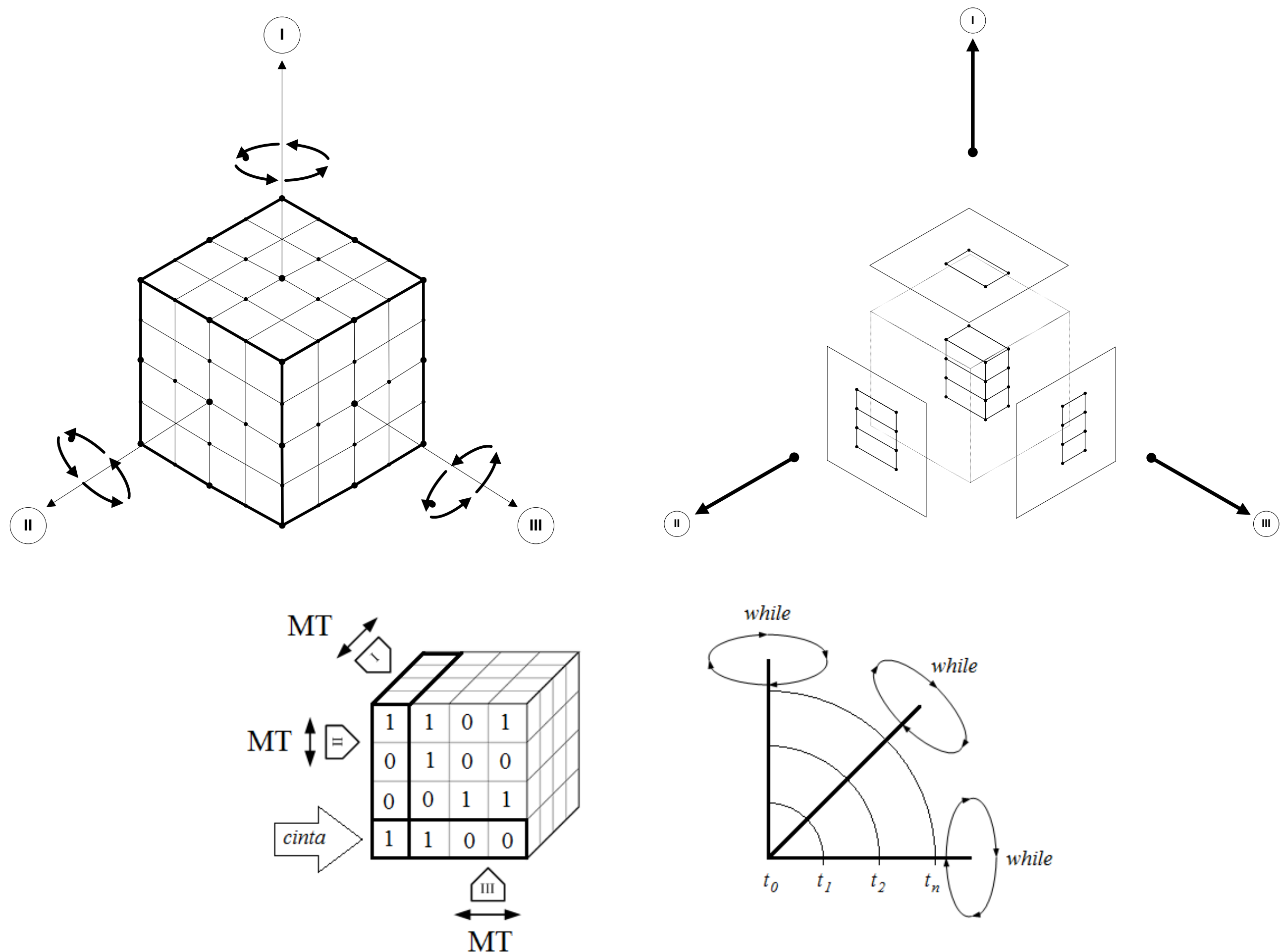
Fuente: Autor



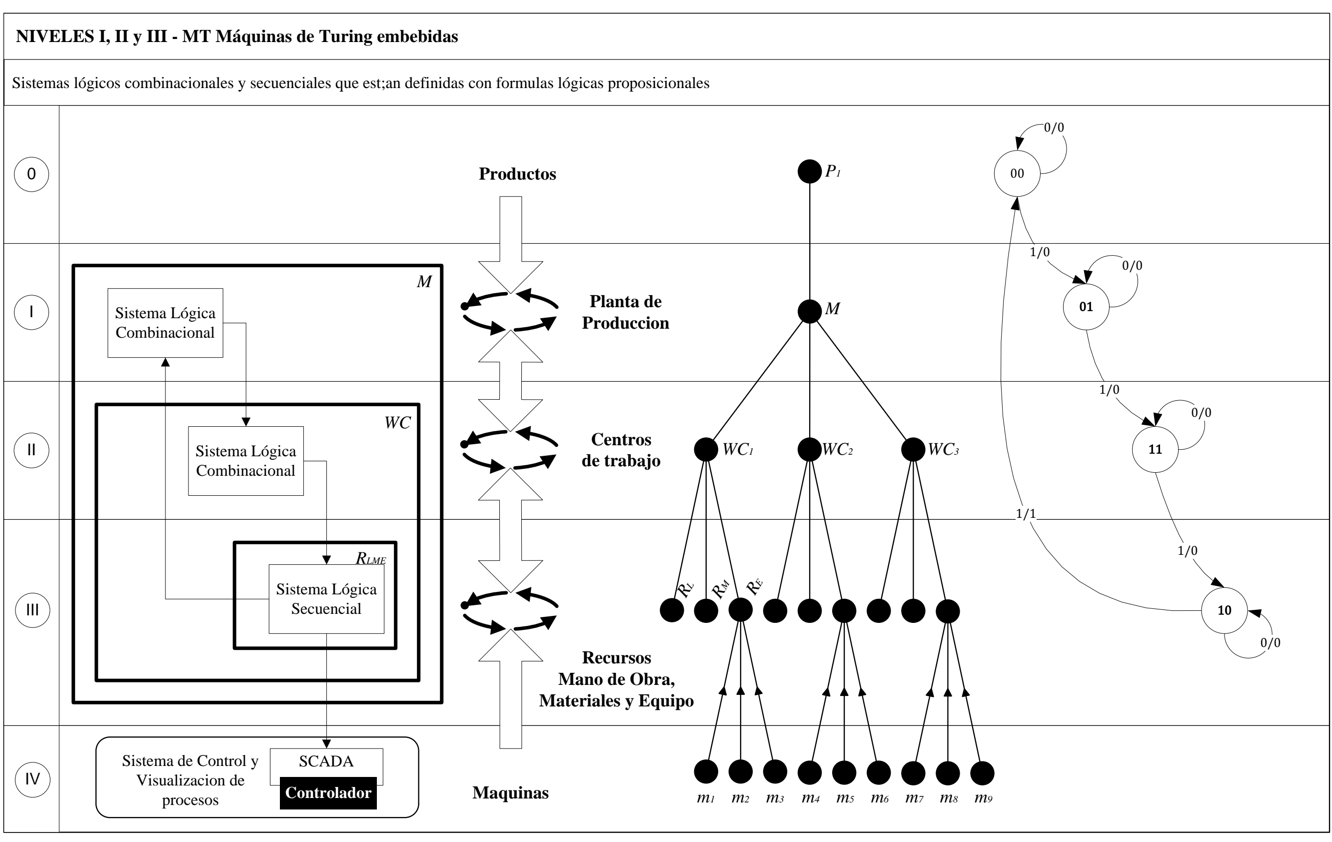
CT – CENTRO DE TRABAJO (C1)
Materiales
Maquinas
Mano de Obra
CT – CENTRO DE TRABAJO (C2)
CT – CENTRO DE TRABAJO (C3)
CT – CENTRO DE TRABAJO (C4)
CT – CENTRO DE TRABAJO (C5)
CT – CENTRO DE TRABAJO (C6)
CT – CENTRO DE TRABAJO (C7)
CT – CENTRO DE TRABAJO (C8)
Producto Final

PASO 7. Sistema lógico secuencial

PASO 8. Lenguajes recursivos



PASO 9. Autómata finito


NIVELES I, II y III - MT Máquinas de Turing embebidas
Sistemas lógicos combinacionales y secuenciales que est;an definidas con formulas lógicas proposicionales
0
I
II
III
IV
M
Sistema Lógica
Combinacional
WC
Sistema Lógica
Combinacional
RLME
Sistema Lógica
Secuencial
Sistema de Control y
Visualizacion de
procesos
SCADA
Controlador
Productos
Planta de
Produccion
Centros
de trabajo
Recursos
Mano de Obra,
Materiales y Equipo
Maquinas
P1
M
WC1
WC2
WC3
RL
RM
RE
m1
m2
m3
m4
m5
m6
m7
m8
m9
00
01
11
10
0/0
1/0
1/1

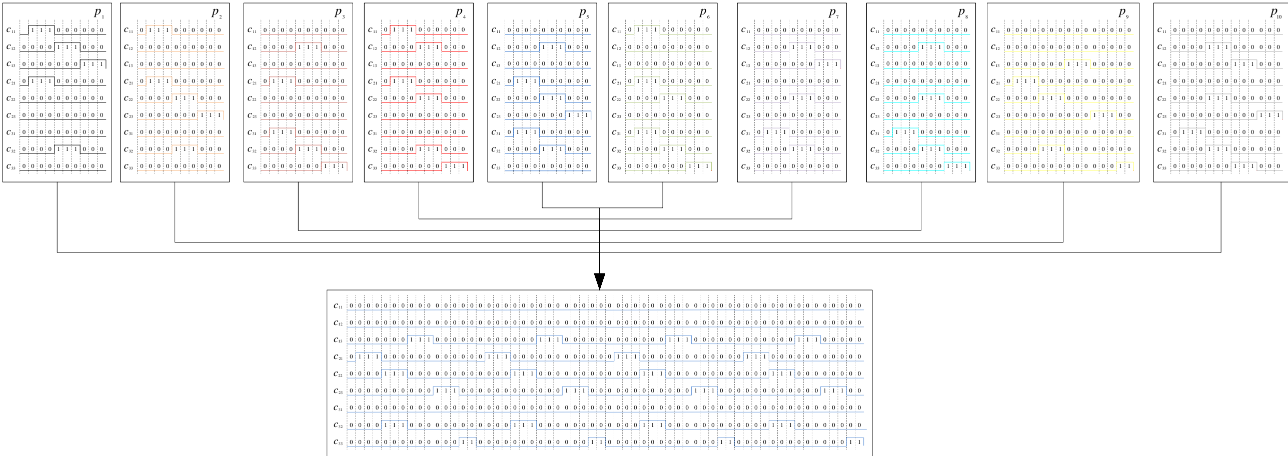
PASO 10. Maquina de estados finitos

Nivel	Máquina de estados finitos FMS	Autómata	Sistema Lógico
I	*If...then* q_0 *printf* *MPS* *por articulos final* Δ I_1 I_2 I_2 I_3 I_4 I_5 I_6 I_n Δ *while* *anillo*	*Cubo*	Artículos finales I_i Neumáticos (Llantas)
II	*If...then* q_0 *printf* *MRP/CRP* *por centro de trabajo* Δ C_1 C_2 C_2 C_3 C_4 C_5 C_6 C_n Δ *while* *anillo*	*Teseracto*	Centro de trabajo C_i
III	*MES* *If...then* q_0 *printf* *Por recursos materia prima* Δ I_1 I_2 I_2 I_3 I_4 I_5 I_6 I_n Δ *If...then* q_0 *Por recursos equipos* Δ M_1 M_2 M_2 M_3 M_4 M_5 M_6 M_n Δ *If...then* q_0 *printf* *Por recursos mano de obra* Δ L_1 L_2 L_2 L_3 L_4 L_5 L_6 L_n Δ *while* *anillo*	*Penteracto*	CT – CENTRO DE TRABAJO (C_{11}) Materiales Maquinas Mano de Obra Recursos I_i, M_i, L_i por Centro de trabajo

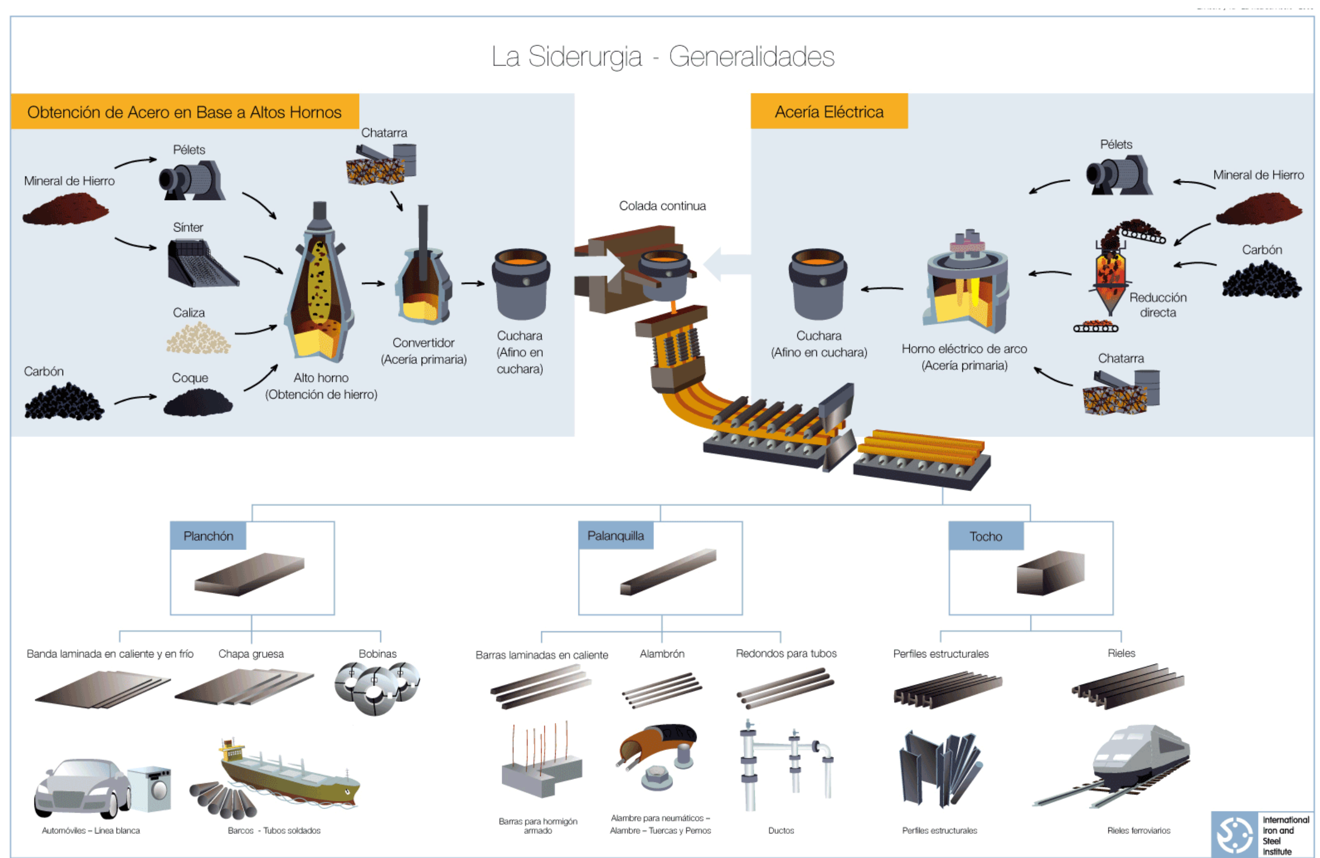
PASO 11. Secuenciación

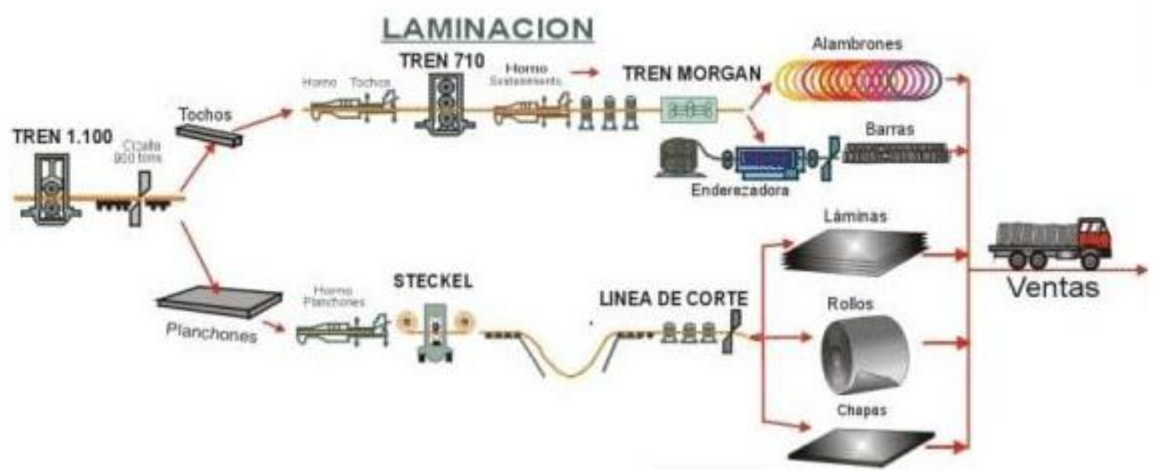
Se efectúan 1250 interacciones del algoritmo para el escenario de producción de neumáticos, donde los tiempos de operación están dados en función de densidad de probabilidad normal con $\bar{x} = 35$ min (según el proceso elegido con tiempos estándar) y una σ de ∓ 3 min, lo cual determina el comportamiento del sistema. Se consideran

8 productos (artículos) para 8 centros de trabajo, (matriz de incidencia de la tabla 17.a) y una demanda dinámica (pronostico semanal)



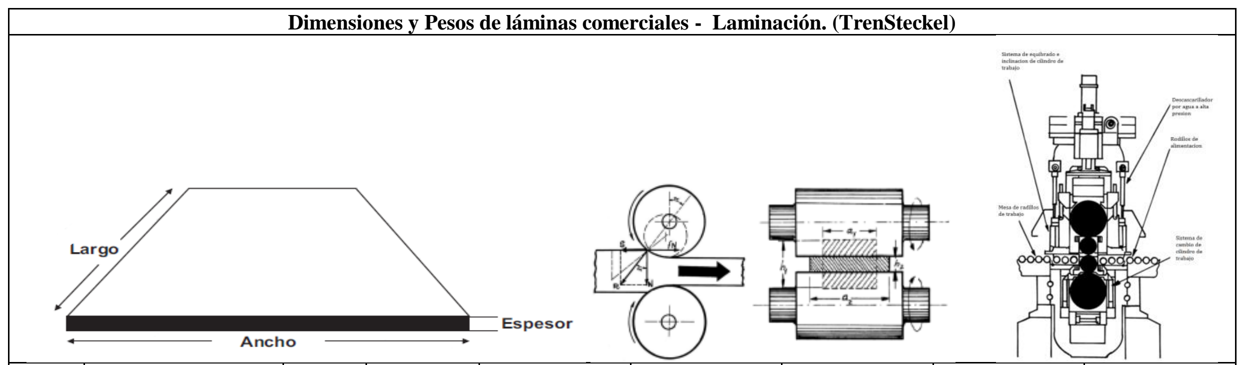
11.2. CASO 2: Pasos del proceso de fabricación de acero (Laminas)





LAMINACION
TREN 710
Horno
Tochos
Horno
TREN MORGAN
Alambrones
TREN 1.100
Tochos
Barras
Enderezadora
Láminas
Ventas
Planchones
STECKEL
LINEA DE CORTE
Rollos
Chapas

Dimensiones y Pesos de láminas comerciales - Laminación. (TrenSteckel)



Item	espesor			m^2	1.0x2.0 m	1.22x2.44 m	1.83x6.09 m	2.0x6.0 m	2.44x6.09 m
Ref	calibre	pulgada	mm	kg.	kg.2	kg.3	kg.4	kg.5	kg.6
1	16		1.5	11.78	23.56	35.01	131	141	175
2	14		1.9	14.92	29.85	44.35	166	179	222
3			2.5	19.63	39.27	58.35	219	236	292
4	12		2.66	20.89	41.78	62.09	233	251	311
5	11		3	23.56	47.12	70.02	263	283	350
6		1/8"	3.17	24.9	49.79	73.99	278	299	370
7			4	31.42	62.83	93.37	350	377	467
8			4.5	35.34	70.69	105.04	394	424	525
9		3/16"	4.76	37.38	74.77	111.11	417	449	556
10			5	39.27	78.54	116.71	438	471	584
11			6	47.12	94.25	140.05	525	565	700
12		1/4"	6.35	49.87	99.75	148.22	556	598	741
13		5/16"	7.94	62.36	124.72	185.33	695	748	927
14			8	62.83	125.66	186.73	700	754	934
15			9	70.69	141.37	210.07	788	848	1051
16		3/8"	9.53	74.85	149.7	222.44	834	898	1113
17			10	78.54	157.08	233.41	876	942	1167
18			12	94.25	188.5	280.1	1051	1131	1401
19		1/2"	12.7	99.75	199.49	296.44	1112	1197	1483
20			13	102.1	204.2	303.44	1138	1225	1518
21			15	117.81	235.62	350.12	1313	1414	1751
22		5/8"	15.88	124.72	249.44	370.66	1390	1497	1854
23			16	125.66	251.33	373.46	1401	1508	1868
24			19	149.23	298.45	443.49	1664	1791	2218
25		3/4"	19.05	149.62	299.24	444.65	1668	1795	2224
26			22	172.79	345.58	513.51	1926	2073	2568
27		7/8"	22.23	174.59	349.19	518.88	1964	2095	2595
28			25	196.35	392.7	583.54	2189	2356	2919
29		1"	25.4	199.49	398.98	592.87	2224	2394	2965
30		1 1/4"	31.75	249.36	498.73	741.09	2780	2992	3707
31			32	251.33	502.65	746.93	2802	3016	3736
32			38	298.45	596.9	886.97	3327	3581	4436
33		1 1/2"	38.1	299.24	598.47	889.31	3336	3591	4448
34			50	392.7	785.4	1167.07	4378	4712	5837
35		2"	50.8	398.98	797.96	1185.74	4448	4788	5931
36			62	486.95	973.89	1447.17	5429	5843	7238
37		2 1/2"	63.5	498.73	997.46	1482.18	5560	5985	7413
38			75	589.05	1178.1	1750.61	6567	7069	8756
39		3"	76.2	598.47	1196.95	1778.62	6672	7182	8896
40			100	785.4	1570.8	2334.14	8756	9425	11675
41		4"	101.6	797.96	1595.93	2371.49	8896	9576	11861

12. CONCLUSIONES Y RECOMENDACIONES

12.1. Conclusiones

- La planificación a corto plazo, se determina a lo sumo en un balde tiempo de una semana en condiciones normales de operación. Al basarse en los resultados obtenidos, las condiciones de una programación a corto plazo se extiende a trabajar en tiempo real exigiendo que el corto plazo sea inmediato y reactivo en la toma de decisiones en ambientes dinámicos, donde la cantidad de variables están en continuo cambio (tanto en la cantidad de estas mismas consideradas en el modelo como en valor que posea en el tiempo), lo que permite obtener una administración acorde a lo realmente ejecutado en la planta de producción y poder determinar una comparación con el planificado (en tiempos estándar).
- La medidas de desempeño contempladas igualmente se obtienen y retroalimentan de manera constante a través del tiempo, como lo determina la definición de un problema de secuenciación, $n =$ número de trabajos que serán procesados, $m =$ número de máquinas, $P_{ik} =$ tiempo de procesado del trabajo i en la máquina k (p_i si $m = 1$), $r_i =$ tiempo de liberación de la orden (o fecha de distribución) del trabajo i , $d_i =$ fecha de entrega del trabajo i ,$w_i =$ ponderación (importancia o valor) del trabajo i respecto a los otros trabajos
- Dado un programa especifico de fabricación de neumáticos para el cual se define para cada trabajo i, el $C_{max} = max_{i=1,n}\{ C_i \}$, tiempo máximo de terminación de todos los trabajos o lapso, $L_{max} = max_{i=1,n}\{ L_i \}$, retraso máximo de todos los trabajos y el $T_{max} = max_{i=1,n}\{ T_i \}$, tardanza máxima de todos los trabajos, se genera para periodos definidos en el tiempo (pueden ser horas, o fracción de la misma) con las opciones dispuestas de ordenes trabajo, luego de efectuar la visualización de las variables de estado, se deduce que no todo sucede al mismo tiempo, como se plantea en un problema de *schedule* para *flow shop*, y por lo tanto la planificación se desvirtúa de lo realmente ejecutado al momento de compararlo, debido a que cada vez que hay un suceso (terminación de una orden, falta de material, etc.) el C_{max}, L_{max} y T_{max} se recalculan nuevamente, generando una evaluación efectiva y acotada de alternativas de decisiones menos complejas, que incluso también se podrían trabajar para escenarios de 2 maquinas, con algoritmos clásicos luego de haber efectuado la reducción por el procedimiento de decisión. La base de conocimiento de quien planifica el proceso, lo convierte en mantener los indicadores de tiempo de manera eficiente, si los conoce constantemente, que por un proceso de inducción lo convierte un experto que tomara decisiones no solo basadas en la experiencia (recursión de procesos) sino en procesos de decisión basados en sistemas lógicos evaluados en tiempo real.
- Para la planificación de la producción es posible elaborar un calendario (cronograma) indicando en el tiempo cuando deben realizarse las operaciones correspondientes a cada pedido por centro de trabajo, de forma que se cumplan las fechas de entrega planificadas basados en salidas de los sistemas lógicos proposicionales para

generar la entrada al modelo computacional (autómata finito o máquina de estados finitos) que realiza cómputos para determinar la secuenciación de trabajos. Este modelo está conformado por un alfabeto, un conjunto de estados finito, una función de transición, un estado inicial y un conjunto de estados finales. Su funcionamiento se basa en una función de transición, que recibe a partir de un estado inicial una cadena de caracteres pertenecientes al alfabeto (la entrada), y que va leyendo dicha cadena a medida que el autómata se desplaza de un estado a otro, para finalmente detenerse en un estado final o de aceptación, que representa la salida binario (digital a un tablero abstracto del LED).

- La investigación del problema de secuenciación efectuado con programación lógica ha explorado el compromiso entre el uso de la pura lógica declarativa y las construcciones procedimentales que permitirá escribir programas de eficiencia razonable frente a la de los lenguajes procedimentales.
- Los problemas de decisión son interesantes dado que todos los problemas formales (que incluye tanto lógicos como matemáticos) pueden ser redactados (como lenguajes) para que tomen la forma de un problema de decisión, como las cadenas de longitud finita que tienen asociadas cadenas resultantes también de longitud finita que describen una máquina de Turing y una cinta de entrada para esta máquina tal que la máquina se detiene en un tiempo finito al procesar esa entrada.
- Un problema de decisión también se puede formalizar como el problema de decidir si una cierta frase (*clausulas de Horn*) pertenece a un conjunto dado de literales, también llamado lenguaje formal. El conjunto contiene exactamente las frases para las cuales la respuesta a la pregunta es positiva, es decir deducir el problema de secuenciación como un problema de *satisfacibilidad boolena*.
- Reconsiderando de nuevo todo la investigación presentada para la secuenciación de operaciones, vemos que el problema de la computabilidad ha sido atacado desde varias direcciones: máquinas computacionales, gramáticas generativas, teoría de funciones recursivas y lenguajes de programación. En cada caso, se ha descubierto un límite aparente para las capacidades computacionales, y se ha mostrado que estos límites coinciden unos con otros.
- Los lenguajes estructurados por frases son iguales que los lenguajes aceptados por máquinas de Turing, las funciones computables con máquinas de Turing son iguales que las funciones recursivas parciales, y las funciones recursivas parciales son iguales que las funciones computables con el lenguaje de programación esencial.
- Por lo tanto, parece que hemos identificado los límites de los procesos computacionales y, específicamente, los limites de los ordenadores. Es decir, hemos encontrado un firme apoyo para la tesis de Church-Turing: si una máquina de Turing no puede resolver un problema, entonces ningún ordenador podría hacerlo, ya que simplemente no existe un algoritmo para obtener su solución. En otras palabras, las limitaciones que hemos detectado corresponden a los procesos computacionales, no a la tecnología.

- La indiscutible ventaja de dejar el control de la ejecución al completo cuidado de un intérprete tiene, no obstante una importante contrapartida. Obviamente, por muy eficiente que sea un compilador de programas lógicos, su predeterminada estructura de control nunca podrá equipararse a una estructura específicamente concebida para una cierta computación. Dejar que un intérprete asuma exclusivamente el control de ejecución de un programa puede conducir a soluciones ineficientes, por ejemplo, cuando se entra en ramas de recursión infinita (hay que tener presente que el cálculo de predicados es indecidible).
- Los programas informáticos paralelos son más difíciles de escribir que los secuenciales, porque la concurrencia introduce nuevos tipos de errores de software, siendo las condiciones de carrera los más comunes. La comunicación y sincronización entre diferentes subtareas son algunos de los mayores obstáculos para obtener un buen rendimiento del programa paralelo.
- La funcionalidad del autómata finito como cubo-n o *hipercubo* n-dimensional, representando el diagrama de estados con las siguientes características: partiendo de un cubo, se compone de 12 aristas y $2^3 = 8$ vértices (tabla de verdad) ; donde $n = 3$ representa la cantidad de estados en la unidad de control para la FMS), 6 caras cuadradas. Sobre este mismo construir otros más complejos hasta obtener relaciones entre vértices de índole $2^{10} = 1024$.
- Establecer una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, operando sobre el principio de que problemas grandes, a menudo se pueden dividir en unos más pequeños, que luego son resueltos simultáneamente (*en paralelo*) para que las limitaciones físicas no impidan el aumento de la frecuencia (*interacciones*) y aplicar paralelismo a nivel de bit, paralelismo a nivel de instrucción, paralelismo de datos y paralelismo de tareas.
- El paralelismo se ha empleado durante muchos años, sobre todo en la computación de altas prestaciones, pero el interés en ella ha decrecido últimamente debido a las limitaciones físicas que impiden el aumento de la frecuencia (*interacciones*). Como el consumo de energía —y por consiguiente la generación de calor— de las computadoras constituye una preocupación en los últimos años, la computación en paralelo se ha convertido en el paradigma dominante en la arquitectura de computadores, principalmente en forma de procesadores multinúcleo.
- En estos sistemas de producción, el centro de trabajo es el controlador (que procesos que corren en el) y se tendrá una administración mucho más amplia de la ejecución del plan de producción. El proceso es capaz de especificar su prioridad; el mismo proceso es capaz de especificar el manejo que requiere (con que algoritmo lógico); el proceso especifica qué derechos tiene sobre el sistema en general. Esto aunque parece anárquico; pero no lo es, debido a que los sistemas de tiempo real usan tipos de procesos que ya incluyen estas características, y usualmente estos tipos de procedimientos y procesos son mencionados como requerimientos dentro del modelo de secuenciación de operaciones.

12.2. Recomendaciones

- Efectuar el montaje de la MFS (*controlador*) en un Sistema de Control Distribuido (por sus siglas en inglés DCS *Distributed Control System*), aplicado a procesos industriales complejos donde las tareas del controlador se distribuyen entre los módulos y estos módulos se distribuyen a lo largo de la red del sistema. El trabajo de ese sistema de control distribuido es buscar las instrucciones a la unidad de ejecución donde se decisión de donde una instrucción procesada se entrega a la parte correspondiente del sistema de control o módulo apropiado para que algunos se realiza y se logra una salida deseada. que controlaban procesos de hasta 5000 señales, y actualmente las capacidades pueden llegar hasta las 250.000 señales.
- Trabajar con una sola base de datos integrada para todas las señales, variables, objetos gráficos, alarmas y eventos del sistema. En sistema de planeación de operaciones como una herramienta de ingeniería para programar el sistema es sólo una y operará de forma centralizada para desarrollar la *lógica de sus controladores* o los objetos gráficos de la monitorización.
- El nuevo controlador resultante de la metodología a través del autómata programable, diseñado para lograr una aplicación especializada y confiable de secuenciación de operaciones directamente en la planta, que combinará lo mejor de las características PLC con lo mejor de las características de las PCs en una será una serie de dispositivos controladores de automatización programable, o PAC orientado al control automatizado avanzado para control de máquinas y procesos junto con la *administración de recursos*.
- Desde este puesto de ingeniería; cargar los programas de forma transparente a los equipos del sistema. La plataforma de programación es multiusuario de forma que varios programadores pueden trabajar simultáneamente sobre el sistema de forma segura sin conflictos de versiones.
- Todos los equipos del sistema (ordenadores, servidores, controladores) están sincronizados contra un mismo reloj patrón, de forma que todas las medidas, alarmas y eventos tienen una misma marca de tiempo. El software de control DCS dispone de herramientas para la gestión de la información de planta, integrándola verticalmente hacia la cadena de toma de decisiones y otros sistemas ubicados más arriba en la jerarquía de la producción.
- Desarrollar dentro de la metodología la programación concurrente ya que envuelve lenguajes de programación y algoritmos usados para implementar los sistemas concurrentes. La programación concurrente es considerada más general que la programación paralela porque puede envolver comunicación de patrones dinámicamente. Los sistemas paralelos siempre tienen un sistema de comunicación bien construido. La meta final de la programación concurrente incluye **correctness**, performance and *robustness.* Sistemas concurrentes como los *sistemas operativos* y *sistemas de manejo de bases de datos* están generalmente diseñados para operar indefinidamente, incluyendo recuperación automática de las fallas, y no terminar su ejecución inesperadamente. Algunos sistemas concurrentes implementan una concurrencia transparente, en el

cual las entidades de concurrencia computacional por un único recurso compartido, pero las complejidades de las mismas están escondidas del programador.

Concurrencia. Es una propiedad de los sistemas en la cual los procesos de un cómputo se hacen simultáneamente, y pueden interactuar entre ellos. Los cálculos (operaciones) pueden ser ejecutados en múltiples procesadores (PAC por centro de trabajo), o ejecutados en procesadores separados físicamente o virtualmente en distintos hilos de ejecución. Varios tipos de lógicas temporales pueden ser usados para ayudar a razonar sobre los sistemas concurrentes. Algunas lógicas como la lógica temporal lineal y la lógica de árboles computacionales permiten decidir por cuales estados puede pasar el sistema concurrente. La principal meta de estas lógicas es en la escritura de especificaciones para los *sistemas concurrentes*.

Niveles de control en un DCS. Un DCS aborda la complejidad de los procesos industriales dividiendo en cuatro niveles funcionales su alcance.

a. *Nivel de Operación*. Este nivel es el de interacción del sistema con los operadores de la planta y es donde se encuentran los sistemas informáticos para la monitorización del proceso y adquisición de la información en tiempo real, que se almacena en la base de datos transformándola en datos históricos para análisis posteriores. Este nivel gestiona además el intercambio de información con otros sistemas de mantenimiento y planificación de la producción.
b. *Nivel de control*. En un DCS la responsabilidad del control de las diferentes partes funcionales del proceso, se asignan a varios controladores locales distribuidos por la instalación, en lugar de centralizar estas funciones en un solo punto. Los controladores están conectados entre sí y con las estaciones de operación mediante redes de comunicación.
c. *Nivel módulos de Entrada/Salida*. Los módulos de entradas/salidas para señales cableadas, se distribuyen por la instalación, es lo que se denomina *periferia descentralizada*, esto ahorra tiradas de cables de señal aproximando la electrónica del control hasta los elementos de campo. Estos módulos de entrada/salida se comunican con los controladores mediante protocolos específicos o de bus de campo (en inglés *fieldbus*) para garantizar los tiempos de comunicación entre controlador y periferia en unos tiempos mínimos, del orden de milisegundos, adecuados a las necesidades del proceso. El bus de campo más extendido en Europa es el *Profibus* (tanto en sus variantes DP como PA) y en los países de influencia americana es el *Fieldbus Foundation* o FF.
d. *Nivel de elementos de campo.* Desde el año 2000, ha ido creciendo la necesidad de integrar directamente los instrumentos y los actuadores en los buses de campo del DCS, de forma que estos equipos son en realidad una extensión natural del nivel anterior. Estos equipos permiten funcionalidades adicionales como gestionar su mantenimiento o configurar sus parámetros de comportamiento de forma remota desde

el nivel de operación. Los instrumentos de este nivel deben ser compatibles con el bus de campo elegido, ya sea *Profibus, Fieldbus Foundation* u otro.

Redundancia. Todo DCS lleva implícitas las características de robustez y fiabilidad, por ello dispone de redundancia en todos los niveles antes descritos: equipos informáticos redundantes, controladores redundantes, redes de comunicación y buses redundantes, módulos de entrada/salida redundantes y así sucesivamente. Esta redundancia permiten alcanzar un factor de disponibilidad cercanos al 99,9999% muy superior a los sistemas de control convencionales. También este dispositivo nos va a permitir comunicar a grandes distancias sin que la señal sea dañada por el ruido o algún otro elemento.

13. LIMITACIONES DE LA INVESTIGACION

Determinar los parámetros de desempeño de un programa de computadora es una tarea difícil y depende de un número de factores como la computadora que se usa, la manera en que se representan los datos y cómo se traduce el programa en instrucciones de máquina.

Aunque las estimaciones precisas del tiempo de ejecución de un programa deben tener en cuenta esos factores, es posible obtener información útil si se analiza el tiempo del algoritmo subyacente.

El tiempo necesario para ejecutar un algoritmo es una función de la entrada. Por lo general, es difícil obtener una fórmula explícita para esta función, y conformarnos con menos. En lugar de manejar directamente los datos de entrada, se usan parámetros que caracterizan el tamaño de la entrada. Si la entrada es un conjunto que contiene n elementos, se diría que el tamaño de la entrada es n. Tal vez se quiera conocer el tiempo mínimo necesario para ejecutar el algoritmo entre todas las entradas de tamaño n. Este tiempo se llama tiempo del mejor caso para entradas de tamaño n. También puede pedirse el tiempo máximo necesario para ejecutar el algoritmo entre todas las entradas de tamaño n. Este tiempo se llama tiempo del peor caso para entradas de tamaño n. Otro caso importante es el tiempo del caso promedio, es decir, el tiempo necesario para ejecutar el algoritmo para un conjunto finito de entradas todas de tamaño n.

Como la preocupación principal se refiere a la estimación del tiempo de un algoritmo en lugar del cálculo del tiempo exacto, siempre que se cuenten algunos pasos fundamentales, dominantes del algoritmo, se obtendrán medidas útiles del tiempo. Si la actividad principal de un algoritmo es hacer comparaciones, como ocurre en una rutina para ordenar, se cuenta el número de comparaciones. Si un algoritmo consiste en un solo ciclo que se ejecuta cuando mucho en C pasos, para alguna constante C, se cuenta el número de iteraciones del ciclo.

Máquinas virtuales de proceso. Una característica esencial de las máquinas virtuales es que los procesos que ejecutan están limitados por los recursos y abstracciones proporcionados por ellas. Estos procesos no pueden escaparse de esta "computadora virtual".

Una máquina virtual de proceso, a veces llamada *"máquina virtual de aplicación",* se ejecuta como un proceso normal dentro de un sistema operativo y soporta un solo proceso. La máquina se inicia automáticamente cuando se lanza el proceso que se desea ejecutar y se detiene para cuando éste finaliza. Su objetivo es el de proporcionar un entorno de ejecución independiente de la plataforma de hardware y del sistema operativo, que oculte los detalles de la plataforma subyacente y permita que un programa se ejecute siempre de la misma forma sobre cualquier plataforma. El caso más conocido actualmente de este tipo de máquina virtual es la máquina virtual de Java.

Uno de los inconvenientes de las máquinas virtuales es que agregan gran complejidad al sistema en tiempo de ejecución. Esto tiene como efecto la ralentización del sistema, es decir, el programa no alcanzará la misma velocidad de ejecución que si se instalase directamente en el sistema operativo *"anfitrión" (host)* o directamente sobre la plataforma de hardware. Sin embargo, a menudo la flexibilidad que ofrecen compensa esta pérdida de eficiencia.

¿Qué hace que un problema sea difícil? Al restringir de alguna manera el conjunto de entradas para un problema NP-completo, el problema podría estar en P; de hecho, podría tener una solución muy rápida. Técnicamente, restringir las entradas implica modificar la parte de pregunta del problema de modo que más ejemplares de entrada tengan respuestas no fáciles (es decir, en tiempo polinómico). Esto se hace añadiendo una condición a la pregunta estándar del problema. Es más útil ver la condición adicional como una restricción del conjunto de entradas.

No obstante, incluso con restricciones, el problema podría seguir siendo NP-completo. Es importante conocer el efecto que restringir el conjunto de entradas de un problema que tiene sobre la complejidad porque, en muchas aplicaciones, las entradas que se dan en la realidad tienen propiedades especiales que podrían permitir una solución polinómicamente acotada. Lo malo es que los resultados no son alentadores; incluso con restricciones muy amplias de las entradas, muchos problemas NP-completos siguen siendo NP-completos.

Por otra parte, en muchas situaciones que se dan en ingeniería se dispone de cierta flexibilidad en cuanto a la forma de definir un problema y el criterio de optimización exacto. Si un criterio produce un problema NP-completo, es muy posible que un criterio alterno, totalmente aceptable, produzca un problema en P. Por ello, conocer bien las características de los problemas difíciles puede resultar muy útil en situaciones prácticas.

Una de las primeras restricciones que se planteadron fue la *3-satisfactibilidad*, que no permite a las fórmulas tener más de tres literales por cláusula. La *3-satisfactibilidad* es NP-completa. El problema de la *2-satisfactibilidad* prohíbe a las fórmulas tener más de dos literales por cláusula, y se puede resolver en tiempo polinómico.
Como ya dijimos, la 2-satisfactibilidad se CNF con cuando más dos literales por cláusula, y dado un entero k, ¿existe una asignación de verdad para las variables que satisfaga al menos k cláusulas? El problema es NP-completo.

Otro fenómeno interesante, es que dos problemas que al parecer tienen planteamientos muy parecidos podrían diferir mucho en cuanto a complejidad; uno podría estar en P mientras que el otro es NP-completo.

El problema de la calendarización de trabajos con castigos) es NP-completo, pero si se omiten los castigos y simplemente se pregunta si existe un calendario tal que no más de k trabajos terminen después de vencido su plazo, el problema está en P. (En otras palabras, si el castigo por no cumplir con un plazo es 1, se puede reducir al mínimo este castigo en un tiempo polinómicamente acotado.)

Estos casos no permiten hacer generalizaciones bonitas acerca de por qué un problema $es\,NP$-completo. Persisten muchas preguntas pendientes en este campo, siendo desde luego la principal ¿$P = NP$?

Algoritmos paralelos. Una forma de generar un procesamiento de alternativas de forma ágil, es asignar a cada centro de trabajo un procesador que evalué cada una de los indicadores que alimentan el sistema lógico proposicional, considerando que para cada centro de trabajo está asociado una máquina de estados finito, que evaluara la disponibilidad del mismo en términos de CRP y MRP. Esta arquitectura desplegara que los algoritmos paralelos que están expresados términos lógicos sean autónomo en la toma de decisiones para efectuar la secuenciación de trabajos propios del centro de trabajo, observado de igual manera los estados de los demás centros de trabajos que se relacionan con él, a través de la ingeniería de producto para cada uno de los productos p_i.

Al centralizar la administración de FMS por centro de trabajo presentes en la planta de producción, a una FMS de mayor jerarquía, se pueden establecer que la evaluación del algoritmo se efectué de manera evolutiva, regenerando estados en la medida que se tiene control de las entradas y salidas del sistema lógico.

La descripción de la operación se podrá expresar en una FMS central, la cual acopia todas las operaciones de las diferentes FMS definidas en la planta de producción por centro de trabajo. Esta misma relación se puede representar a través de diagrama de árbol de n niveles. Para nuestro caso $n = 3$, y describiría la forma en que el algoritmo recorre la planta de producción actualizando los estados. (Este procedimiento se describe dentro de la sección de algoritmos computacionales – grafos y recorridos de grafos).

14. REFERENCIAS BIBLIOGRAFICAS

Badesa C, J. I. (1998). Elementos de lógica formal. Ariel.

Baker. (1974). Introduction to Sequencing and Schedulig. Nueva York: Kohn Wiley & Sons.

Barnes, D. W. (1978). Una introducción algebraica a la lógica matemática. Eunibar.

Bridge, J. (1977). Beginning Model Theory. Oxford University Pres.

Brookshear, G. (1993). Teoria de la Computacion: Lenguajes Formales, Automatas y Complejidad. Addison-Wesley Iberoamericana.

Buyer's. (1995). Scheduling Software. IIE Solutions.

Carlos Guillén, D. A. (2005). Diseño de Algoritmos Combinatorios para # SAT y su Aplicación al Razonamiento Proposicional. Reporte Técnico No. CCC-05-005.

Chang. (1995). QS: Quantitative Systems . Prentice Hall Englewood Cliffs NJ.

Church, A. (1936). An undecidable problem in elementary number theory. American J. Math.

Cook, S. A. (2012). The Complexity of Theorem-Proving Procedures.

Cormen, T. H. (2010). Introduction to Algorithms,. (3, Ed.) MIT Press and McGraw-Hill.

Cormen, T. H., & Leiserson, C. E. (2010). Scheduling Theory, Algorithms, and Systmens. (5, Ed.) MIT Spring,.

Daniel Sipper, R. B. (1998). Planeacion y Control de la Produccion. Mexico: McGraw Hill.

Dewilde, V. &. (1994). Choosing a Scheduling Package. The Performance Advantage.

Enderton, H. B. (1972). Mathematical Introduction to Logic. Academic Press.

Enrique Mandado Pérez, J. M. (2009). Autómatas programables y sistemas de automatización. Marcombo, 2009.

Ershov, Y. P. (1990). Lógica matemática. Mir.

Gantt. (1911). How Scientific Management Is Applied. Easton PA: Hive Publishing.

Garey, M. R. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman,.

Garey, M. R. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness,. W. H. Freeman,.

Gilman. (1994). Interest in Finite Scheduling Is Growing. The performance advantage.

Gödel, K. (1931). Uber formal unentscheidbare Sätze der Principia Mathematica und verwander systeme",. 3. K. Gödel, "Uber formal unentscheidbare Sätze der Principia Mathematica und verwander systeme",.

Graham. (1969). Bounds on Multiprocessing Timing. Journal on Applied Mathematics.

Hamilton, A. G. (1981). Lógica para matemáticos. Paraningo.

Hofstadter, D. (1987). Gödel, Escher, Bach: un Eterno y Grácil Bucle. Tusquets Editores.

Huettle. (1993). Finite Capacity Scheduling. APICS : The performance adventage.

I-Hsuan Huang, S. F. (2011). A Fuzzy-based Multi-Term Genetic Algorith for Reentrant Flow Shop Scheduling Problem. Fukuoka, Japan: Waseda University,.

Infante, C. (2010). Guía para la presentación de proyectos de investigación. Bogotá, Colombia: Universidad Nacional de Colombia.

J.Chase Richard B, J. F. (2005.). Administración de la producción y operaciones para una ventaja competitiva. México D.F: Décima edición. Editorial Mc Graw Hill. .

Jackson. (1956). Extension of Johnson Results on Job Lot Scheduling. Naval Research Logistics Quarterly.

Jané, I. (1989). Álgebras de Boole y lógica. Publicaciones U.B.

John, H. R. (2008). Introducción a la teoria de automatas lenguajes y comutacion. Madrid: Pearson Educacion SA.

Johnson. (1954). Optimal Two-and Thee Stge Production Schedules with Setup Times Included. Naval Research Logistics Quarterly.

Johnson, G. &. (1979). Computers and Intratability. San Francisco: Freeman.

K.Ida, K. a. (2009). Two Proposed Algorithms for Re-Entrant Flow Shop Problem. IEEJ Transactions on Electronics, Information and Systems.

Kan, R. (1976). Machine Scheduling Problems : Classification and Computacions. Holanda: La Haya Martinus Nijhoff.

Kan, R. (1976). Machine Scheduling Problems: Classification, Complexity and Comtations. Holanda: La Haya .

Karp, R. (1972). Reducibility Among Combinational Problems. New York: Thatcer (editors).

Kleene, S. C. (1936). General recursive functions of natural numbers. Mathematische Annalen.

Lasserre. (1992). An Integrated Model for Jop-Shop Planning and Scheduling. Management Science.

Maxwell, C. (1967). Theory of scheduling. Reading MA: Addison Wesley.

McKay, S. (1988). Jop-Shop Scheduling Theory: What Is Relevant. Interfaces.

McNaughton. (1959). Sequencing with Deadlines and Loss Functions. Management Science.

Mendelson, E. (1997). Introduction to Mathematical Logic. (4. edición, Ed.) Chapman and May.

Monk, J. D. (1976). Mathematical Logic. Springer-Verlag.

Morris, M. (2003). Diseno Digital. Mexico: Pearson Educación de México.

Negrón, D. F. (2009). Administración de operaciones. Cengage Learning Editores, 2009.

Nidditch, P. H. (1978). El desarrollo de la lógica matemática. Cátedra.

Norman Gaither, G. F. (2000). Administracion de Produccion y Operaciones. Ediciones Paraninfo: Thomson Learning.

Panwalkar, S. (1973). Sequencinf Research and the Industrial Scheduling Problem. Nueva York: Springer Publishing.

Pentico, M. &. (1993). Hauristic Scheduling Sytems. Nueva York: John Wiley & Sons.

Pinedo, M. (1995). Scheduling: Theory, Algorithms, and Systems. Prentice-Hall, Englewood Cliffs.

Pla, J. (1991). Lliçons de lógica matemática. P.P.U.

Raposo, A. P. (2010). Logica, conjuntos, relaciones y funciones. Mexico: Publicaciones Electronicas.

Ricardo Rosenfeld, J. I. (2013). Computabilidad, complejdad computacional y verificacion de programas. Buenos Aires, Argentina: Universidad Nacional de la Plata.

Rodríguez, L. A. (2010). Practique la Teoría de Autómatas y Lenguajes Formales. ELIZCOM S.A.S, 2010.

Sara Baase, A. V. (2002). Algoritmos Computacionales (3 ed.). Mexico: Pearson Educación.

Sipper, D. (1998). Planeacion y Control de la Produccion. Mexico: McGraw Hill.

Smith. (1956). Various Optimizers for Single Stage Production. Naval Research Logistics Quarterly.

Thompson, M. &. (1963). Industrial Scheduling. NJ: Englewood Cliffs.

Torres, S. A. (2010). Lógica matemática para Ingeniería de Sistemas y Computación. ELIZCOM S.A.S.

Turing, A. M. (1936). On computable numberswith an application to the Entscheidungsproblem. Proc. London.

Van Dalen, D. (1983). Logic and Structure. (2, Ed.) Universitext, Springer-Verlag.

Villamizar, J. C. (2004). Tecnicas para resolver el problema de satisfactibilidad y sus aplicaciones. Bogota: Revista de Tecnologia.

15. ANEXOS

Archivo complementario (Anexos, Teoremas y Glosario)