PONTIFICIA UNIVERSIDAD CATOLICA DE VALPARAISO
FACULTAD DE INGENIERIA
ESCUELA DE INGENIERIA INFORMATICA

MODELADO Y RESOLUCION DEL MANUFACTURING CELL DESIGN PROBLEM (MCDP) UTILIZANDO PROGRAMACION CON RESTRICCIONES

JUAN ANDRES GUTIERREZ QUEZADA
ALEXIS EFRAIN LOPEZ ESTAY

INFORME FINAL DEL PROYECTO
PARA OPTAR AL TITULO PROFESIONAL DE
INGENIERO DE EJECUCION EN INFORMATICA

OCTUBRE 2011

Pontificia Universidad Católica de Valparaíso
Facultad de Ingeniería
Escuela de Ingeniería Informática

MODELADO Y RESOLUCION DEL MANUFACTURING CELL DESIGN PROBLEM (MCDP) UTILIZANDO PROGRAMACION CON RESTRICCIONES

JUAN ANDRÉS GUTIÉRREZ QUEZADA

ALEXIS EFRAÍN LÓPEZ ESTAY

Profesor Guía: **Ricardo Soto De Giorgis**

Profesor Co-referente: **José Rubio León**

Carrera: **Ingeniería de Ejecución en Informática**

Octubre 2011

Dedicatoria

A nuestros padres que con esfuerzo y entereza
lograron incentivar el caminar hacia nuestros sueños.

Agradecimientos

Los agradecimientos no bastan para nuestros padres y familiares que con ilusión apoyaron e incentivaron el deseo de terminar con esta etapa en nuestra vida. Tantas noches en vela estudiando, riendo, aprendiendo y reconociendo que sin esfuerzo no hay recompensa que valga la pena recordar.

Gracias a Ricardo Soto, una de las personas que inspiran a un buen trabajo en equipo, el desarrollo de nuestras capacidades y aptitudes que no se enseñan en una sala de clases, sino que se transfieren como un valor importante en el desarrollo personal. Un gran saludo al que llamábamos encubiertamente "El Maestro" y lo recordaremos como nuestro profesor guía y el gestor del trabajo realizado.

Saludos cordiales a nuestros amigos y compañeros de clases, que siempre con buenos momentos podemos recordar como grandes amigos y que cada día compartieron las penas y alegrías de estar en la universidad, pero que dejan una cuota de anhelo y deseo de repetir los momentos vividos.

Gracias a todos.

Resumen

Este proyecto tiene por objetivo modelar y resolver el problema de la Manufactura de Celdas de Diseño, en inglés, *Manufacturing Cell Design Problem* (MCDP) utilizando programación con restricciones. El MCDP propone dividir una planta de producción industrial en un cierto número de celdas. Cada celda conteniendo máquinas con tipos de procesos similares o familias de piezas.

El objetivo es identificar una organización de celdas de manera tal que el transporte de las distintas piezas entre celdas sea minimizado. El presente problema se resuelve en dos *solvers* para programación con restricciones: ECL^{i}PS^e y Choco. Se presentan los resultados obtenidos alcanzando el óptimo global en todos los experimentos.

Palabras Claves: *Manufacturing cell design problem, Machine grouping, Constraint programming*

Abstract

The purpose of this project is to model and solve the Manufacturing Cell Design Problem (MCDP) by using constraint programming. The MCDP proposes to divide an industrial production plant into a certain number of cells. Each cell contains machines with similar types of processes or piece families.

The goal is to identify a cell organization in a way that the transportation of the different pieces between cells is minimized. The problem presented is solved with two solvers for constraint programming: ECL^{i}PS^e and Choco. The obtained results reach the global optima in all of the experiments.

Key Words: Manufacturing cell design problem, Machine grouping, Constraint programming

Índice

Lista de Figuras

Lista de Tablas

1. Introducción

Recientemente las situaciones económicas competitivas demandan suministros rápidos de nuevos productos con funcionalidades más innovadoras para satisfacer rápidamente las necesidades cambiantes de los clientes. Las organizaciones se encuentran constantemente buscando soluciones que generen una mayor productividad y eficiencia, pero sobre todo buscan diferenciarse y responder activamente a las exigencias diarias del mercado como son: entregas rápidas, alta calidad y precios competitivos; para ello buscan nuevas formas de producir, innovar y permanecer en el mercado.

En el afán de mantener competitividad, las empresas han incorporado a sus actividades el concepto de diseño de celdas de manufactura, el cual ha sido ampliamente empleado para sistemas de fabricación. El diseño de celdas de manufactura emerge principalmente como una estrategia de producción capaz de resolver un problema complejo de manufactura y minimizar la cantidad de tiempo de fabricación en sistemas de producción en serie.

El diseño de celdas de manufactura, en inglés *Manufacturing Cell Design Problem* (MCDP), es una estrategia que divide un sistema de producción en pequeños grupos o celdas, cada una de ellas enfocada a la producción de un conjunto o partes de componentes. Debido a lo anterior, existen variadas y diferentes restricciones de carácter operativo para cumplir este fin, como por ejemplo, la disposición física de máquinas en un sistema de producción de manera tal que los movimientos e intercambio de material entre grupos sea el mínimo.

1.1. Motivación

En la literatura moderna [9], existen variados y numerosos estudios en el campo del diseño de celdas de manufactura, algunos más eficientes, y otros menos eficientes en términos de resultados en el diseño de celdas. Sin embargo, técnicas, como por ejemplo, la programación con restricciones aún no han sido explotadas para cumplir esta finalidad. Dado lo anterior, se hace imperativa la motivación principal de generar una solución al problema de diseño de celdas de manufactura (MCDP) proponiendo la utilización de la programación con restricciones.

Este proyecto tiene por objetivo, modelar y resolver el MCDP utilizado programación con restricciones. Esta última considerada como una alternativa valiosa para determinar un

modelo que ayude, primero, a la descripción y luego, a la resolución de problemas típicamente combinatorios y de optimización bajo una complejidad *Non Polynomial* (NP).

La fase de modelamiento del proyecto consta de una formalización del problema como un *Constraint Optimization Problem* (COP) y la fase de resolución a cargo de 2 motores de búsqueda (*solvers*) distintos. En una primera etapa, la resolución se desarrolló en el *solver* ECL^{i}PS^e, para luego el mismo problema ser modelado y resuelto por el *solver Choco*; Finalmente, se analizarán y compararán los resultados de los dos *solvers* para determinar la mejor estrategia de resolución y enumeración.

1.2. Estructura de la Tesis

La disposición de contenidos de este documento es la siguiente, primeramente se presenta en el capítulo 2 el objetivo general y los objetivos específicos involucrados en la investigación.

Posteriormente en el capítulo 3, se entrega el estado del arte de la problemática, abarcando esencialmente las técnicas más significativas que han dado solución al MCDP. Seguidamente en el capítulo 4 se habla sobre la programación con restricciones, técnica que será utilizada para dar solución al MCDP, este capítulo rodea diferentes conceptos que van principalmente, por ejemplo, desde definiciones propias de la programación con restricciones, ejemplos de programación de CSP, hasta lenguajes y sistemas de programación que otorgan características para la resolución de un problema sujeto a restricciones.

En el capítulo 5 se otorga al lector una profundización conceptual acerca del *Manufacturing Cell Design Problem*, se engloban en este capítulo antecedentes históricos del diseño de celdas de manufactura, definiciones acerca de un Grupo Tecnológico (GT), descripción de una distribución de celdas en una planta utilizando GT proveniente de la industria mecánica. Luego se presenta el estado del problema entendiéndose este como la organización y reorganización de un sistema de producción representado por matrices de incidencia. Y para concluir el capítulo se formaliza el MCDP como un COP definiendo constantes, variables, dominios, restricciones y la función objetivo.

Consecutivamente en el capítulo 6, se dispone al lector, la definición de una introducción y posteriormente una ilustración de las características más significativas del *solver* ECL^{i}PS^e y del *solver* Choco; *solvers* a utilizar en la resolución del MCDP.

Más tarde en el capítulo 7, se define y explica la representación en ECL^{i}PS^e del MCDP en función de la formulación de este problema como un problema de satisfacción de restricciones. Algunas de las principales definiciones de este capítulo son las características y utilización de librerías en ECL^{i}PS^e, las reglas en términos de sintaxis lógica del modelo a implementar y la figuración en ECL^{i}PS^e de las restricciones definidas en capítulos anteriores.

Luego en el capítulo 8 se exponen las principales características, descripciones y representación del MCDP en el segundo *solver* de programación con restricciones Choco.

Posteriormente en el capítulo número 9, se desarrolla la experimentación del modelo implementado tanto en el solver ECL^{i}PS^e como en el *solver* Choco, con el objetivo que tales experimentos ayuden a visualizar el comportamiento del modelo para luego analizar tales datos con la finalidad de inferir y concluir la veracidad del modelo en base a diferentes patrones de prueba debidamente validados.

Finalmente se proporciona al lector en el anexo del documento, los códigos fuente de los modelos finales en ECL^{i}PS^e y Choco, para luego presentar en las conclusiones el trabajo realizado y los desafíos pertinentes para un trabajo futuro por alcanzar.

2. Definición de Objetivos

2.1. Objetivo General

Modelar y resolver el *Manufacturing Cell Design Problem* (MCDP) utilizando dos diferentes motores de búsqueda (*solver*) para programación con restricciones, con el fin de comparar y analizar resultados obtenidos para determinar la mejor estrategia de resolución.

2.2. Objetivos Específicos

- Modelar el MCDP utilizando programación con restricciones.
- Resolver el MCDP utilizando dos *solvers* distintos de programación con restricciones.
- Comparar y analizar los resultados utilizando distintas estrategias de resolución.

3. Estado del Arte

En la actualidad la mayoría de las empresas de todo el mundo están en una constante mejora de procesos de manufactura con el fin de aumentar su productividad, buscando principalmente disminuir los costes de operación en términos de tiempo y dinero. El diseño de celdas de manufactura busca desarrollar una estrategia de productividad que entregue lineamientos sobre técnicas y sistemas de producción en empresas para tener la suficiente competitividad y rentabilidad en el mercado.

En años recientes y en la actualidad diferentes métodos meta-heurísticos han sido usados para resolver problemas de agrupación de máquinas; Boctor [1]; Chen & Srivastava [2] y más recientemente Wu, Chang, y Chung, presentaron un algoritmo denominado SACF (*Simulated Annealing Cell Formation*), técnica fundada en *Simulated Annealing* (SA) [3], que buscaba esencialmente minimizar el número de elementos excepcionales (EEs), entendiéndose estos últimos como máquinas asignadas a una celda mientras al mismo tiempo son requeridas por otras celdas. El comportamiento eficiente de este método es esperable principalmente sólo en experimentos a gran escala.

Venugopal y Narendran [4] presentaron el intento de resolver un problema de diseño de celdas con la ayuda de un algoritmo de computación evolutiva, el cual considera tres funciones medulares; minimizar los movimientos totales interceldas así como los movimientos intraceldas, minimizar la variación de carga en una celda y por último minimizar las funciones objetivo de manera simultánea. Gupta, Kumar, y Sundaram [5] emplean la misma representación genética de soluciones que Venugopal y Nerendran [4] sobre la base de un algoritmo evolutivo, pero siguiendo un multi-objetivo diferente de optimización, donde el número total de movimientos y la variación de carga de la celda es aproximado.

Aljaber, Baek y Chen [6] y Lozano, Adenso, Salinas, y Gimenez [7] usan el algoritmo *Tabu Search* (TS). Aljaber et al. [6] entregaba un diseño del MCDP basado en la teoría de grafos y en un par de problemas del camino más corto, este modelo en general producía mejores soluciones en términos de calidad pero operaba con un alto tiempo computacional. Lozano et al. [7] presentó un enfoque gradual al problema de la agrupación de máquinas, el asume ciertos límites en el tamaño de celdas y familias de componentes para máquinas. El implementa un algoritmo basado en TS el cual fue un punto de referencia contra varias técnicas SA, heurísticas y otros métodos TS, además de entregar la propuesta de un modelo de programación cuadrático entero, basado en la suma ponderada de espacios intraceldas y movimientos interceldas, este método propuesto supera a otros procedimientos de la época teniendo menores tiempos de computación.

Recientemente un documento de Andres y Lozano [8], presentó la primera técnica *Particle Swarm Optimization* (PSO) para el problema del diseño de celdas de manufactura. La solución codificada corresponde a un vector de partículas de posición que es inicializado con una población de soluciones aleatorias en donde estas soluciones aleatorias envuelven otras generaciones de soluciones aleatorias para encontrar la solución óptima. En general el enfoque indicó que PSO tiene una mayor capacidad de descubrir una solución óptima en una cantidad razonable de tiempo.

Ming y Ponnambalam [9], propusieron un enfoque PSO híbrido combinando un algoritmo genético para el problema de diseño de celdas y PSO para un *layout* óptimo, la metodología considera 2 variables como las partículas generadas inicialmente de manera aleatoria y la velocidad de estas, este modelo se aplicó exitosamente para minimizar la variación de carga y movimientos totales de las celdas.

Mehdizadeth y Tavakkoli-Moghaddam [10] propusieron un *Fuzzy Particle Swarm Optimization* (FPSO); técnica para resolver problemas en el diseño de celdas de manufactura en un contexto de agrupación de partes de máquinas, donde cada partícula corresponde al vector centro de un *cluster* y el *swarm* representa un número de agrupaciones candidatas definida por el vector de datos actualizados. En definitiva, este método demostró que para un problema a gran escala podría producir una mejor solución.

Duran, Rodríguez y Consalter [11] direccionan el problema de fabricación y diseño de celdas de manufactura usando un algoritmo modificado de cúmulo de partículas (PSO), la principal modificación con el algoritmo original PSO consiste en no usar el vector de velocidades que el algoritmo estándar PSO entrega. El algoritmo propuesto utiliza el concepto de probabilidad proporcional, una técnica que es utilizada en minería de datos.

4. Programación con Restricciones

La programación con restricciones, *Constraint Programming* (CP) en inglés, es un paradigma de la programación informática y es una reciente tecnología de software, dedicada a la resolución de problemas de satisfacción de restricciones y problemas de optimización con restricciones.

Ciertos tipos de problemas pueden estar presentes en diversas áreas, incluyendo inteligencia artificial, investigación operativa, bases de datos y sistemas de recuperación de la información, etc., con aplicaciones con múltiples ámbitos en problemas de *scheduling*, diseño en la ingeniería, problemas de empaquetamiento, criptografía, diagnóstico, toma de decisiones, etc.

La vida diaria lleva a tomar ciertas decisiones, estas decisiones acarrean consigo restricciones que limitan o encausan la toma de una buena elección. Estas decisiones pueden tener un nivel de complejidad básico, como el tipo de locomoción a tomar, planificar un encuentro con amigos o, un nivel un poco más avanzado de complejidad como la compra de un vehículo, que pueden tener restricciones al menos, a primera vista.

Como lo mencionado anteriormente, existen restricciones simples, pero en la sociedad compleja y moderna que hoy se vive en el que el tiempo es dinero y el ahorro, ganancias, se necesita de procedimientos que sean óptimos y minimicen costos para la generación de utilidades a gran escala.

4.1. Problema de Satisfacción de Restricciones

El objetivo de la programación con restricciones es solucionar problemas por medio de la declaración de restricciones sobre el dominio del problema y encontrar soluciones a instancias de los problemas de dicho dominio que satisfagan todas las restricciones [12].

Un problema de satisfacción de restricciones (CSP), es una *n*-tupla de (*X*, *D*, *C*) donde:

- *X* es una *n*-tuplas variables $X = < X_1, X_2, ..., X_n >$.
- *D* corresponde a *n*-tuplas de dominios $D = < D_1, D_2, ..., D_n >$. Donde se encuentran todos los posibles valores que se pueden asignar a la variable X_i. Tal que $X_i \in Di$ es un conjunto de valores, para $i = 1,\ldots, n$.

- *C* es una *m*-tuplas finitas de restricciones $C = < C_1, C_2, \ldots, C_m >$ y una restricción C_j se define como un subconjunto del producto cartesiano de dominios $D_1 X \ldots X D_n$, para $j = 1, \ldots, m$.

A continuación, una restricción de *Cj* se satisface con una tupla de valores $(a_1, \ldots, a_n)$ si $(a_1, \ldots, a_n) \in C_j$. El CSP se resuelve cuando todas sus restricciones se han satisfecho. Si el CSP tiene al menos una solución, se dice que es un problema consistente, de lo contrario se dice que es inconsistente.

Existen diferentes clases de CSPs, por ejemplo:

- Un CSP con dominio finito, corresponde a un CSP en el que cada dominio es un subconjunto finito de Z (universo de variables). Las restricciones son generalmente definidas como un conjunto de expresiones aritméticas o lógicas.

- Un CSP numérico corresponde a un CSP en el que cada dominio es un intervalo continuo de valores de *R*. Las restricciones son generalmente definidas como ecuaciones lineales, no lineales o desigualdades.

4.2. Problema de Optimización de Restricciones

Muchas veces en los problemas de la vida real no basta con encontrar una solución, sino que se busca la mejor solución. La calidad de ésta solución se mide comparándola con la llamada función objetivo. La meta es encontrar una solución que satisfaga todas las restricciones y minimice o maximice la función objetivo según corresponda. Tales problemas son llamados, *Constraint Optimization Problem* (COP) en inglés, también son llamados problemas CSOP por R. Barták [13].

Un problema de optimización de restricciones consta de un CSP estándar y una función de optimización que asigna a cada solución numérica un valor.

Para los COPs se definen *n*-tuplas donde *X*, *D* y *C* de la misma manera que un CSP, adicionando la función de costos *f*, llamada función objetivo, también llamada función de coste global y es la suma de todas las funciones de costo individual, $F(X) = \sum_{i=1}^{m} fi(X)$. La solución es la completa asignación que minimiza/maximiza *F(X)* [14].

4.3. Resolución de un CSP

El resolver un problema de satisfacción de restricciones considera principalmente dos fases diferentes, donde la primera de ellas busca principalmente modelar el problema como CSP. Este modelamiento expresa el problema mediante sintaxis CSP, es decir, como un conjunto de variables, dominios y restricciones del mismo modo como se reveló en la sección anterior (ver capítulo 4.1).

El procedimiento de búsqueda para resolver un CSP se suele abordar mediante una estructura de árbol, intercalando las fases de enumeración y propagación. En la fase de enumeración, una variable y su valor son elegidos desde el dominio para crear una rama del árbol. En la fase de propagación, un nivel de consistencia se aplica para podar el árbol, es decir, los valores que no conducen a una solución son eliminados temporalmente de los dominios. De esta manera la exploración no inspecciona instancias inviables, acelerando todo el proceso. De todas maneras en la sección 4.5 Algoritmos de Búsqueda, se profundizarán dichos conceptos.

4.4. Ejemplos de CSP y su Modelización

El primer paso para la resolución de un CSP es su modelamiento, es decir, su representación en términos de variables, restricciones y dominios. Al igual como ocurre en el lenguaje natural, el modelamiento de un problema puede ser realizado de maneras diferentes.

Principalmente respecto a un problema de satisfacción de restricciones hay dos aspectos fundamentales:

- La potencia expresiva de las restricciones, es decir, la capacidad de modelar las restricciones verdaderamente presentes en el problema y que estas sean representativas en su totalidad.

- La eficiencia de la representación, es decir, dependiendo de la modelización del CSP, la solución será más o menos eficiente.

4.4.1. Criptografía

Dentro de la criptografía (entendiéndose esta última cómo, una técnica que permite cifrar mensajes o hacerlos ininteligibles), existe el típico problema denominado el "*send+more=money*", el cual consiste en asignar a cada letra {*s,e,n,d,m,o,r,y*} un digito diferente dentro del dominio {0,…,9} de tal forma que satisfaga la ecuación *send+more=money*.

$$\begin{array}{cccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

La manera más sencilla de modelar este problema es asignar una variable a cada una de las letras, todas estas con un dominio entre {0,…,9}, con la restricción que obligue a que todas las variables tomen valores distintos. Junto con las restricciones anteriores la restricción más importante, es la ecuación misma "*send+more=money*". De esta manera la representación de las restricciones es la siguiente:

- $10^3(s+m)+10^2(e+o)+10(n+r)+d+e=10^4m+10^3o+10^2n+10e+y$;
- *alldiferent* : (*s,e,n,d,m,o,r,y*), restricción de todas las letras diferentes

4.4.2. El problema de las N-Reinas

Este problema es uno de los más representativos de CP, principalmente consiste en colocar *N* reinas en un tablero de ajedrez de dimensión *NxN*, de manera tal que cada reina cuando efectúe un movimiento no interfiera en los desplazamientos de las otras reinas y las reinas no puedan ser capturadas entre sí. De esta forma no puede haber 2 reinas en la misma fila, misma columna, o misma diagonal. Si se asocia cada columna a una variable y su valor representa la fila donde se coloca una reina, el problema puede ser formulado como sigue:

- Variables: { Q_i }, i = 1..*N*
- Dominio: {*1,2,3,…,N*}, para todas las variables
- Restricciones: ($\forall Q_i,Q_j,i\neq j$):
  - $Q_i \neq Q_j$, para las filas

- $Q_i + \ i \neq Q_j + \ j$, diagonal 1
- $Q_i - \ i \neq Q_j - \ j$, diagonal 2

La figura 4.1 muestra una representación de satisfacción de restricciones para un problema de 4-Reinas utilizando la formulación anterior.

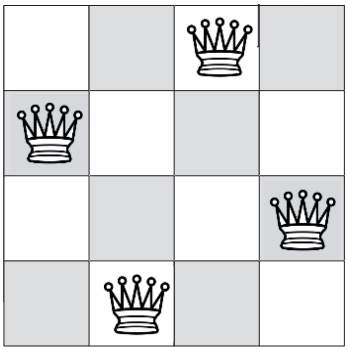


Figura 4.1 – Problema de las n-reinas

4.5. Algoritmos de Búsqueda

Los métodos de búsqueda o algoritmos *backtracking* se centran en explorar el espacio de estados o soluciones de un problema en cuestión. Estos métodos pueden ser *completos*, si exploran todo el espacio de soluciones o *incompletos*, si solamente exploran una parte del espacio de soluciones. Los algoritmos que exploran todo el espacio de soluciones garantizan de manera convincente encontrar una solución, si existe, en caso contrario estos algoritmos demuestran que el problema es irresoluble. Los algoritmos *incompletos*, que no garantizan encontrar una solución óptima, también son muy utilizados en problemas de satisfacción de restricciones y en problemas de optimización debido a su mayor eficiencia en términos de tiempos de búsqueda y posibilitan encontrar soluciones cercanas al óptimo, a diferencia del alto costo que requiere una búsqueda completa.

Las posibles combinaciones de la asignación de variables a un CSP generan un espacio de búsqueda que puede ser visualizado como un árbol de búsqueda. La búsqueda mediante *backtracking* en un CSP corresponde a la tradicional exploración en profundidad en el árbol de búsqueda (figura 4.2). En cada nivel del árbol se instancia una variable y los sucesores de un nodo son todos los valores de una variable asociada a ese nivel. Cada camino del nodo raíz a un nodo terminal denominado rama, representa una asignación

completa. La raíz que corresponde al nivel 0, representa la asignación vacía, y cada camino de un nodo raíz a un nodo no terminal representa una asignación parcial. El número total de ramas es la cardinalidad del producto cartesiano de los dominios de todas las variables.

4.5.1. Generate and Test

La manera más sencilla, aunque poco eficiente, de encontrar todas las soluciones de un CSP es la que implementa el algoritmo *Generate and Test*. Este método genera las posibles n-tuplas de instanciación, de todas las variables de forma sistemática y después prueba sucesivamente sobre cada instanciación si se satisfacen todas las restricciones del problema. La primera combinación que satisfaga todas las restricciones, será la solución al problema.

Se ilustra el proceso del algoritmo GT (*Generate and Test*) mediante el problema de las 4 reinas, La figura 4.2 representa un extracto de un procedimiento hecho por el algoritmo de búsqueda GT para dar solución a este problema.

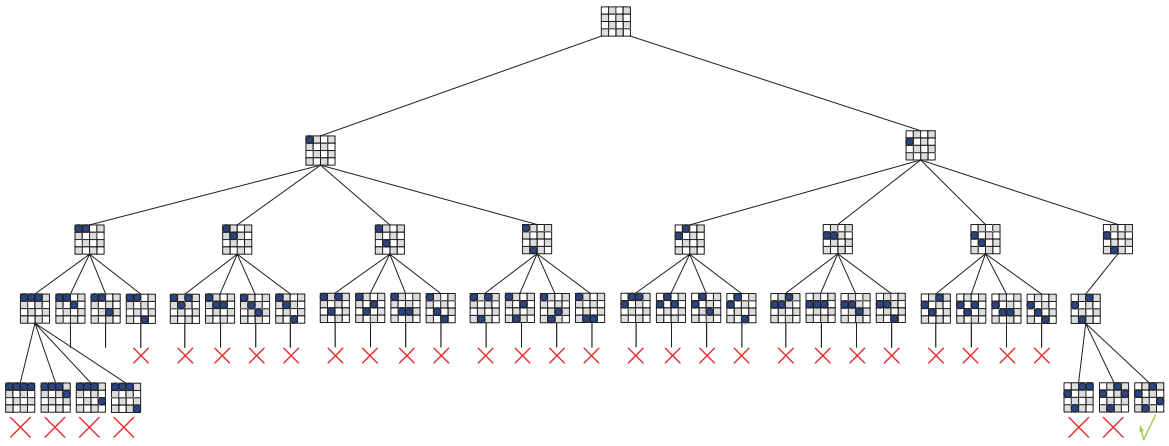


Figura 4.2 – Resolviendo el problema de las n-reinas con *Generate and Test*

4.5.2. Backtracking Cronológico

El algoritmo BT (*backtracking cronológico*) mejora el algoritmo *Generate and Test*. BT cada vez que asigna un nuevo valor a la variable actual (*Xi*), se comprueba si es consistente con los valores que se han asignado a las variables pasadas. Si no lo es, abandona esta asignación parcial, y se asigna un nuevo valor a *Xi*. Si ya se han agotado

todos los valores de *Di*, BT retrocede para probar otro valor para la variable *Xi*. Si se ha agotado *Di*, retrocede al nivel *i*, y así sucesivamente hasta encontrar una asignación de un valor a una variable que es consistente con las variables pasadas o hasta que se demuestra que no hay más soluciones.

En términos generales, BT recorre el árbol utilizando búsqueda primero en profundidad y, en cada nodo, comprueba si la variable actual es consistente con las variables pasadas. Si detecta inconsistencia, descarta la asignación parcial actual, puesto que no es parte de ninguna asignación completa que sea solución. De esta forma, se ahorra recorrer el subárbol que *cuelga* de esta asignación parcial.

La ilustración grafica a este procedimiento se puede ver en la figura 4.3, que representa el problema de las 4-reinas y la forma en que el algoritmo de *Backtracking* resuelve dicho problema.

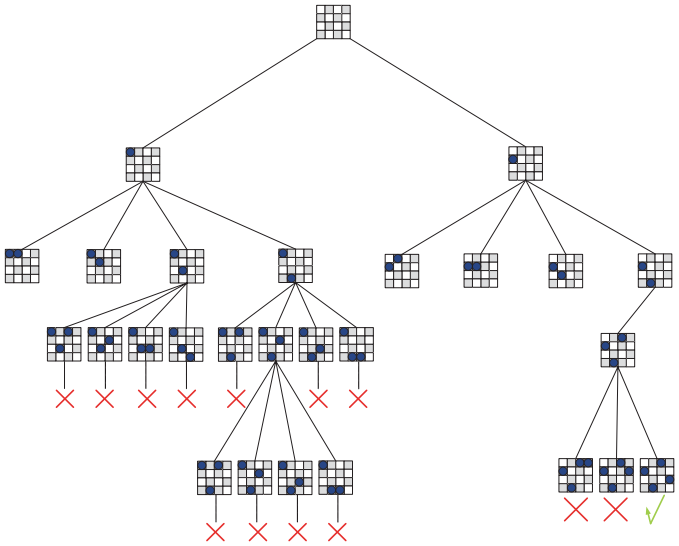


Figura 4.3 – Resolviendo el problema de las n-reinas con Backtracking Cronológico

4.5.3. Forward Checking

El algoritmo de *Forward Checking* (FC), es de los llamados algoritmos *look-ahead*, que son algoritmos que hacen una comprobación hacia adelante en cada etapa de la búsqueda, es decir, ellos llevan a cabo ciertas comprobaciones para determinar si los futuros caminos a tomar pueden llevar o no a obtener inconsistencias en las variables futuras y pasadas; a diferencia de los algoritmos *look-back* como BT, que comprueban la consistencia de las variables hacia atrás. La profundización de conceptos de consistencia será ilustrada acentuadamente en la sección 4.6.1.

El FC es uno de los algoritmos *look-ahead* más comunes, en cada etapa el FC comprueba hacia adelante la asignación actual con los valores futuros de las variables que están restringidas con la variable actual. Si hay inconsistencia en los valores futuros, con la asignación actual, éstos son eliminados temporalmente del dominio de las soluciones. Si el dominio de una variable futura queda vacío, la asignación de la variable actualmente evaluada se deshace y se prueba con nuevos valores. De tal forma si no quedan valores consistentes se lleva a cabo BT cronológico.

Se pueden identificar caminos inconsistentes en FC y hacer podas tempranas en el espacio de búsqueda. Esto quiere decir que, FC garantiza que la solución actual es consistente con todos los valores de las variables futuras ya que se comprueba solo hacia adelante las futuras variables.

El proceso realizado por FC puede verse como un simple paso de arco-consistencia, (véase en técnicas de consistencia, capítulo 4.6.1) sobre cada restricción de las variables actuales con las futuras después de cada asignación. Esto se muestra en la figura 4.4

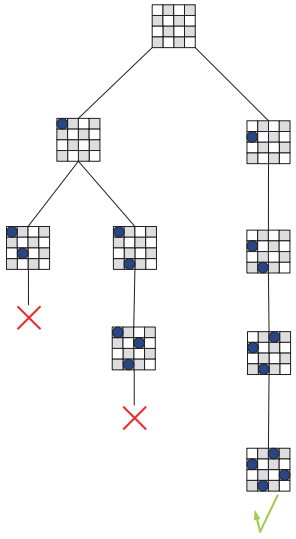


Figura 4.4 – Resolviendo el problema de las n-reinas con Forward Checking

4.5.4. Maintaining Arc Consistency

El *Maintaining Arc Consistency* (MAC) [15] o método también llamado *Full Look Ahead* (FLA), es un poderoso algoritmo de resolución de CSPs. Este revisa los conflictos entre las variables futuras y además prueba las variables actuales con las futuras variables en búsqueda de posibles conflictos. Funciona como un FC, pero en cada nodo del árbol de búsqueda se aplica el algoritmo *Arc Consistency* (AC) para alcanzar arco consistencia entre todas las restricciones del problema.

Mientras que en FC se exige la arco consistencia parcial, en MAC se exige arco consistencia total en cada nodo.

Existen otros algoritmos, como por ejemplo *Partial Look Ahead* (PLA) que en cada nodo, exige un grado de consistencia local intermedio entre FC y MAC. PLA y FLA hacen más esfuerzos por la consistencia que el FC [16]. La ilustración gráfica de proceso del algoritmo, se ve representada en la figura 4.5

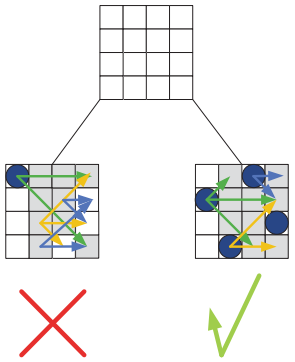


Figura 4.5 – Resolviendo el problema de las n-reinas con Maintaining Arc Consistency

4.5.5. Branch and Bound

Las tareas de un CSP envuelven la búsqueda de cualquier solución que satisfaga todas las restricciones. El trabajo de un COP es tomar la mejor solución relativa para uno o más criterios, usualmente representado por la función de costo o una función objetivo. Por ejemplo se podrían tomar las restricciones $X \leq Y$, $Y \leq 10$, con un dominio $D = \{1,\ldots,10\}$ y la función objetivo $f = X + Y$, para ser maximizada. Entonces la mejor solución (sólo una para éste ejemplo) es $X = 10$, $Y = 10$. Todas las otras soluciones, como por ejemplo $X = 5$, $Y = 6$, satisfacen las restricciones pero son de menor preferencia al ser la función objetivo maximizada. Frecuentemente la función de costo es especificada además, como una suma de componentes de costos, los cuales son definidos en subconjuntos de variables.

Branch and Bound (BnB) es una extensión de la búsqueda BT, pero este algoritmo es utilizado para la optimización en CP. Su tarea es ir seleccionando una solución que tenga un menor coste. Al igual que BT, BnB atraviesa el árbol de búsqueda de la manera del algoritmo de *Depth-First Search* (DFS), que poda no solamente las instanciaciones parciales que son inconsistentes, sino que también los que son evaluados como inferiores a la mejor solución actual. Es decir que cada valor del nodo de la actual solución parcial es estimado por medio de una evaluación de la función y comparación con la mejor solución

actual; si esta es inferior, la búsqueda por ese camino se termina. Cuando la evaluación de la función es correcta, BnB poda las ramas del árbol de búsqueda [17].

4.6. Estrategias de Resolución

Un algoritmo de búsqueda puede ser adecuado para ofrecer buenos resultados, ya sea para un CSP o COP, aportando tiempos de cómputo reducidos y ahorro de recursos computacionales. Pero todo esto puede mejorar si se combina con una buena técnica de consistencia y/o heurísticas. El algoritmo de búsqueda debe ser adecuado para el tipo de problema que se desea resolver y además de esto, podría agregarse un método o técnica de resolución inteligente y cómoda para resolver de manera más eficiente el problema.

Otra técnica que puede destacarse como estrategia de resolución, es la selección del orden correcto de la instanciación de las variables y los valores de ellas, que pueden mejorar notablemente la eficiencia al momento de buscar las soluciones o la verificación de la consistencia de las restricciones.

Las técnicas de consistencia son esenciales en la mejora de los algoritmos de búsqueda, ya que podan los dominios, para que el tiempo utilizado por los algoritmos sea menor.

4.6.1. Técnicas de consistencia

Como se mencionó en el párrafo anterior, el rendimiento de algoritmos de búsqueda para CSP puede ser mejorado mediante la reducción de los dominios de las variables de cada sub-problema generado. Esto es posible mediante el cálculo de las propiedades de consistencia en las restricciones. A esto se le denomina Arco Consistencia, en inglés *Arc Consistency* (AC) que puede ser ejecutada antes o durante la búsqueda [18].

Una restricción se dice ser arco consistente sí, es arco consistente con respecto a todas las variables. Un CSP se dice ser arco consistente si todas las restricciones son arco consistentes.

La AC, permite verificar que para cada valor de un dominio, exista al menos un valor en el dominio de otra variable, de tal manera que la restricción en cuestión sea satisfecha. Esta propiedad puede ser calculada por medio de un algoritmo de propagación de restricciones a fin de reducir los dominios de las variables. Como un ejemplo, se puede considerar el problema de las 4 reinas (véase en la sección 4.4.3). El lugar de la primera reina es en la celda (1,1) del tablero (Figura 4.1). Tres celdas han sido eliminadas para hacer el sub-problema arco consistente. El valor 1 ha sido eliminado desde el dominio de

Q_2 desde que este no tenga valores correspondientes a el dominio de Q_2 de tal manera la restricción $Q_1 \neq Q_2$ es satisfecha (considerando que el dominio de Q_1 se convirtió {1} antes de la instanciación). En el mismo sentido, el valor 1 ha sido eliminado desde el dominio de Q_3 y Q_4. Este proceso es seguido por cada restricción del problema, permitiendo evitar las posibles instanciaciones erróneas. Nótese que existen diferentes algoritmos para hacer cumplir la arco consistencia, por ejemplo AC-3 o Consistencia de Caminos, AC-4 y AC-5 entre otros.

Hay más de un método para buscar la consistencia, pero también existe una fuerte noción de consistencia y como ésta se puede aplicar, pero el problema es que a pesar que puede eliminar un gran número de valores conflictivos desde los dominios, ésta incurre en altos costos computacionales, lo que desincentiva al momento de implementar técnicas mucho más complejas de acotamiento de dominios o de consistencia.

4.6.2. Heurísticas de ordenación de variables

Las investigaciones respaldan que el orden en el que las variables son asignadas durante la búsqueda incide fuertemente en el tamaño del espacio explorado en búsqueda de soluciones, demostrando que el orden de las variables es crucial para el procesamiento de una eficiente solución. Aquí se presentan algunas heurísticas para la selección de una variable de ordenamiento que ayudará a disminuir el tiempo en el proceso de búsqueda de soluciones.

Antes de seleccionar una variable para enumerarla o dividirla, los algoritmos tienen que seleccionar un valor desde el dominio de las variables. Esta selección es llamada, *"ordenamiento de valor"* y puede tener también un impacto considerable. Por ejemplo, si el correcto valor es elegido en el primer intento por cada variable, una solución puede ser encontrada sin el proceso de BT. Sin embargo si el CSP es inconsistente, todo el conjunto de soluciones es requerido, entonces el valor de ordenamiento es irrelevante.

La literatura presenta diferentes caminos para el desarrollo de ésta selección, la cual depende de la naturaleza del problema, para dirigir una eficiente propagación de restricciones. Se deben tomar en cuenta dos tipos de heurísticas de ordenación de variables.

- Las heurísticas de ordenación de variables estáticas: que generan un orden fijo de las variables antes de iniciar la búsqueda, basado en información global derivada del grafo de restricciones inicial.

- Las heurísticas de ordenación de variables dinámicas: que pueden cambiar el orden de las variables dinámicamente basándose en información local que se genera durante la búsqueda.

4.6.2.1. Heurísticas de Ordenación de Variables Estáticas

Las heurísticas de ordenación de variables se dice que son estáticas justamente por no cambiar su forma de analizar los posibles caminos de una manera total. Estas heurísticas se basan en la información global que deriva de la topología del grafo de restricciones original que representa el CSP.

- *Minimum Width* (MW) según Freuder [19], se impone un orden total en las variables, desde la última hasta la primera de manera decreciente, por lo que el orden tiene la mínima anchura. Se utiliza anchura como medida de ordenación. La anchura de la variable *x* es el número de variables que están antes de *x*, de acuerdo a un orden dado y que son adyacentes a *x*. La anchura de un orden es la máxima anchura de todas las variables bajo ese orden. La anchura de un grafo de restricciones es la anchura mínima de todos los posibles órdenes. Con esto en cuenta, se ordena en base a la variable que tiene mínima anchura el orden de las variables, para que las variables más restringidas queden en el principio de la ordenación.

- *Maximum Cardinality* (MC) selecciona la primera variable arbitrariamente y luego selecciona en cada paso la variable que es adyacente al conjunto más grande de las variables seleccionadas previamente [20].

- *Maximum Degree* (MD) ordena las variables en un orden decreciente de su grado, en el grafo de restricciones. El grado de un nodo depende del número de nodos adyacentes a él. También puede encontrar un orden de anchura mínimo, aunque como no es su objetivo y tampoco lo garantiza.

4.6.2.2. Heurísticas de Ordenación de Variables Dinámicas

El problema de los algoritmos de búsqueda estáticos es que no toman en cuenta los cambios que sufren los dominios de las variables que se van modificando a medida que la búsqueda a través de la propagación de las restricciones se lleva a cabo. O simplemente esto no se realiza por la densidad de las restricciones. Esto se debe a que mayoritariamente los tipos de heurísticas que se utilizan llevan una comprobación hacia atrás, donde la propagación de las restricciones no se lleva a cabo.

- *First-fail*: Se selecciona la variable con el dominio más pequeño. Esta elección es inspirada por la suposición que el suceso puede ser logrado por probar primeramente las variables que tienen una gran posibilidad de fallar, en éste caso, los valores con un pequeño número de alternativas disponibles. Esta heurística es conocida por su aplicación mucho más adaptable en dominios discretos.

- *Most-constrained variable*: esta alternativa puede ser justificada por el hecho que la instanciación de dicha variable debe conducir a un árbol de dominios más grande a través de la propagación de la restricción.

- *Reduce-first*: la selección de la variable con el dominio más grande. Esta heurística es conocida por ser la más adoptada para los dominios continuos.

- *Round robin*: es usada para seleccionar alguna variable en orden racional y equitativo, para instanciar desde la primera variable definida en el modelo hasta la última.

4.6.3. Heurísticas de Ordenación de Valores

La idea principal de las heurísticas de ordenación de valores es que una vez que se tengan acercamientos a las mejores estrategias de resolución, se logre determinar qué valor de la variable actual posee mayor probabilidad de llevar a una solución, es decir, identificar la rama del árbol de búsqueda que sea más probable que obtenga dicha solución.

La mayoría de las heurísticas propuestas tratan de seleccionar el valor menos restringido de la variable actual, es decir, el valor que menos reduce el número de valores útiles para las futuras variables.

- *Min-conflicts*: esta heurística ordena los valores de acuerdo a los conflictos en los que están involucradas las variables no instanciadas. Esta heurística asocia a cada valor *a* de la variable actual, el número total de valores en los dominios de las futuras variables adyacentes que son incompatibles con *a*. El valor seleccionado es el asociado a la suma más baja. Esta heurística se puede generalizar para CSPs no binarios de forma directa. Cada valor *a* de la variable x_i se asocia con el número total de tuplas que son incompatibles con *a* en las restricciones en las que está involucrada la variable x_i. De nuevo se selecciona el valor con la menor suma.

  Keng y Yun [21] proponen una variación de la idea anterior. De acuerdo a su heurística, cuando se cuenta el número de valores incompatibles para una futura variable x_k, si éste se divide por la talla del dominio de x_k. Esto entrega el porcentaje de los valores útiles que pierde x_k debido al valor *a* que actualmente se está examinando. De nuevo los porcentajes se añaden para todas las variables futuras y se selecciona el valor más bajo que se obtiene en todas las sumas.

- *Promise*: para cada valor de *a* de la variable *x* se cuenta el número de valores que soporta *a* en cada variable adyacente futura y se toma el producto de las cantidades contadas [22]. Este producto se llama la promesa de un valor. De esta manera se selecciona el valor con la máxima promesa. La heurística de Geelen trata de seleccionar el valor que deja un mayor número de soluciones posibles después de que este valor se haya asignado a la variable actual. La promesa de cada valor representa una cota superior del número de soluciones diferentes que pueden existir después de que el valor se asigne a la variable.

5. Manufacturing Cell Design Problem

Las celdas de manufactura (*Manufacturing Cell*) es una estrategia de manufactura que divide un sistema de producción en pequeños grupos o células cada una de ellas enfocada a la producción de un conjunto de partes o componentes. Esta estrategia de producción es una clara consecuencia de la aplicación de la filosofía de la tecnología de grupos, en inglés *Group Technology* (GT), la cual plantea que "*cosas parecidas deberían de ser fabricadas de manera parecida"*.

Las celdas de manufactura buscan distribuir la totalidad de piezas fabricadas y máquinas necesarias para su fabricación en un conjunto de grupos, concentrado cada uno a la fabricación de un subconjunto de estas partes que forman familias y que son determinadas de acuerdo a las similitudes entre ellas, similitudes que pueden ser de tipo geométrico, por peso, materiales de fabricación, volúmenes de fabricación, operaciones necesarias, entre otras.

5.1. Antecedentes de las Celdas de Manufactura

La idea de la manufactura de celdas, tiene su origen en las metodologías de agrupamiento (Tecnología de Grupos), de estas metodologías de desprenden los sistemas de producción dentro de los cuales se ubica la producción celular o de celdas.

Sin embargo existe la teoría que la agrupación de familias de componentes surgió desde hace mucho tiempo, por ejemplo, F. Koenigsberger explaya en su artículo que alrededor del año 2500 A.C. [23], el hombre de las cavernas ya fabricaba herramientas de corte como flechas, cuchillos, hachas, etc., que eran agrupadas y clasificadas de acuerdo a la geometría y se llegó a la conclusión de que el proceso de fabricación de todas ellas era en esencia el mismo. Como particularidad la variación en el proceso de fabricación de tales piezas, era sólo el ángulo en el cual se tenían que desprender las esquirlas para obtener la geometría deseada.

5.1.1. Tecnología de Grupo

La Tecnología de Grupos, es una metodología aceptada hasta el día de hoy para resolver muchos de los problemas que las organizaciones de manufactura enfrentan en una planta de producción. La implementación de los principios de GT en una planta es con

frecuencia referida como manufactura celular o manufactura de celdas, esto quiere decir que partes similares son agrupadas en familias y las máquinas asociadas en el proceso dentro de grupos, así una o más familias de partes se pueden procesar dentro de un sólo grupo de máquinas. El procedimiento mencionado que consiste en agrupar familias de partes y grupos de máquinas se refiere explícitamente al MCDP. La ilustración grafica de una familia de partes similares se presenta en la figura 5.1 a continuación:

Figura 5.1 – Ejemplo familia de partes similares

Un indicio de una línea flexible de manufactura no fue sino hasta el año 1952, en un taller de ingeniería eléctrica de Francia, donde se implantó por primera vez; Basado en la idea anterior Jacques Schaffran replantea la línea de producción lineal por una producción tipo circular (ver la figura 5.2) con esta distribución de máquinas Schaffran plantea las siguientes ventajas: Los productos que utilizaban algunas de las máquinas agrupadas en círculo, la mayor distancia que deberían recorrer sería el diámetro del círculo, caso contrario de una distribución lineal. Otra de las ventajas que manejaba la producción celular era una mayor ocupación del tiempo de uso de las máquinas y el desahogo de algunas máquinas dependiendo del lote de producción.

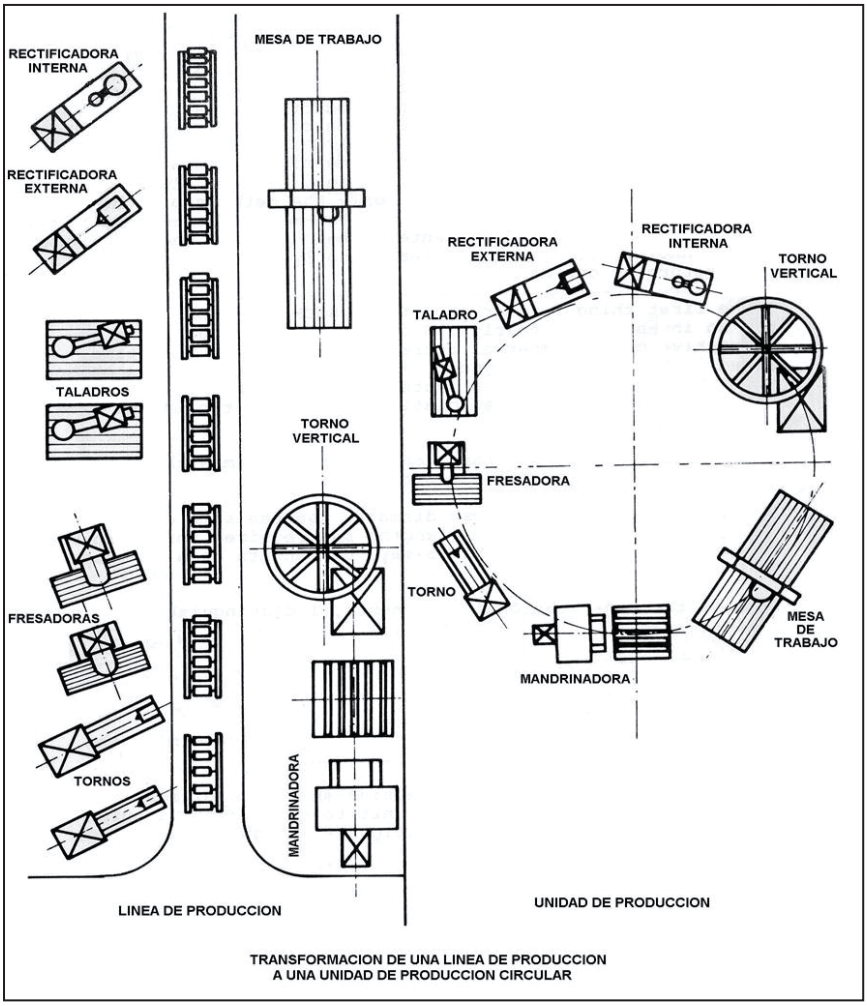


Figura 5.2 – Ejemplo de transformación de línea de producción a producción circular

En sus inicios este sistema de producción estuvo limitado por los requerimientos tecnológicos de la época; es decir, para que dicho sistema fuera autosuficiente y flexible se requería:

- Alto costo de inversión para la implementar nuevos sistemas.
- Que el equipo utilizado dentro de la celda tuviese una mayor automatización y autonomía.
- Un sistema de codificación de partes que tomara en cuenta las variantes que se presentaban al formar las familias de piezas y grupos de máquinas.
- Mayor utilización de algunas máquinas y desocupación de otras.

Sin embargo con el paso del tiempo todas estas variantes anteriores se transformarían en logros y ventajas obtenidas por el diseño de celdas de manufactura.

5.2. Distribución de Celdas en una Planta

La identificación de familias de piezas es una parte fundamental en el desarrollo de un sistema de fabricación a través de celdas cuando éste se diseña con base en la semejanza de piezas o componentes. Principalmente para lograr una eficiente distribución de las máquinas y sus familias de componentes, se debe obedecer a tres grandes tópicos. Tópicos que se ilustran a continuación:

- Agrupar las partes en familias que siguen una secuencia de pasos comunes.
- Identificar los patrones de flujo dominantes de las familias de partes como base para la ubicación o reubicación de los procesos.
- Agrupar físicamente las máquinas y los procesos en las celdas.

A modo de ilustración gráfica, se presenta en la figura 5.3 un esquema que integra las tareas anteriores con los conceptos de GT.

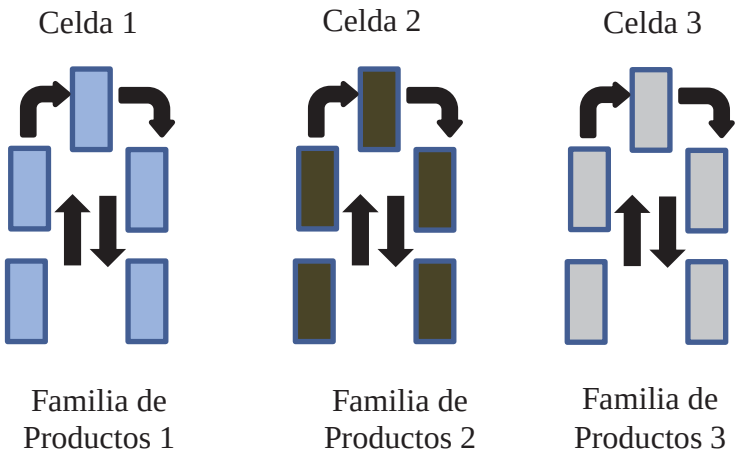


Figura 5.3 – Ejemplo de agrupación de máquinas en GT

5.2.1. Descripción Algorítmica de un Esquema de Formación de Celdas

En la figura 5.4 se entrega una visión algorítmica del proceso de implantación de una/s celda/s de manufactura [24], la descripción detallada es la siguiente:

1.- Identificación del problema: Etapa de tremenda importancia para el desarrollo de celdas de manufactura a implantar en la industria metalmecánica, con la finalidad de ser eficiente y mejorar el sistema de producción actual; ya que éste tiende a ser insuficiente para cubrir las nuevas necesidades que impone el mercado en el cual compite la compañía.

2.- Familiarización del problema: Para poder resolver un problema, después de ser conocido se necesita realizar las siguientes preguntas que llevan a la búsqueda de datos:

- ¿Qué elementos lo componen?
- ¿Por qué se presenta?
- ¿Cómo se presenta el problema?

3.- Búsqueda de métodos para la solución del problema: En esta etapa es necesario realizar una investigación documental acerca de problemas similares en el mismo sector industrial, con el fin de elaborar una idea o marcador que indique de qué manera se resolvió el problema y lleve a la mejor alternativa de solución.

4.- Definición de las características relevantes de las piezas: En esta etapa se identifican las piezas que se producen en la planta y se definen las características principales, así como los procesos de producción que éstas necesitan para obtenerlas como producto final.

5.- Identificación de las familias de piezas: En este punto se hace una búsqueda minuciosa de las piezas que se fabrican en la empresa, es decir, se hace una primera agrupación de piezas con base en las características geométricas, procesos de producción y la máquina que más se utiliza (máquina clave) para la manufactura de dichas piezas.

6.- Demanda de producción: En este punto se hace un análisis de la demanda de cada pieza que tiene la empresa para darle prioridad a la solución.

7.- Análisis: Dentro de este punto se encuentran ciertos criterios que sirven para dar una mejor agrupación a las familias de piezas y al posible arreglo de máquinas que conformarán las celdas de manufactura. Los criterios que se toman son los siguientes:

- **Análisis de Máquina Clave**: Este análisis se fundamenta en el estudio de las hojas de rutas de procesos de maquinado de las piezas que se fabrican.

- **Análisis de Flujo de Producción**: Este estudio consiste en tomar las hojas de ruta y ver cuáles son las piezas que tienen operaciones de maquinado en las mismas máquinas, creándose así una primera familia por tipos de operaciones similares o parecidas en la mayor medida posible.

- **Análisis de aglomeración (agrupamiento)**: Este análisis consiste en piezas semejantes en geometría y/o procesos de manufactura, creándose una matriz en donde en las columnas se ubica el Ítem de piezas y en las filas las máquinas, evaluándose dichos datos se obtendrá como resultado un arreglo de piezas agrupadas por proceso (matriz pieza- maquina).

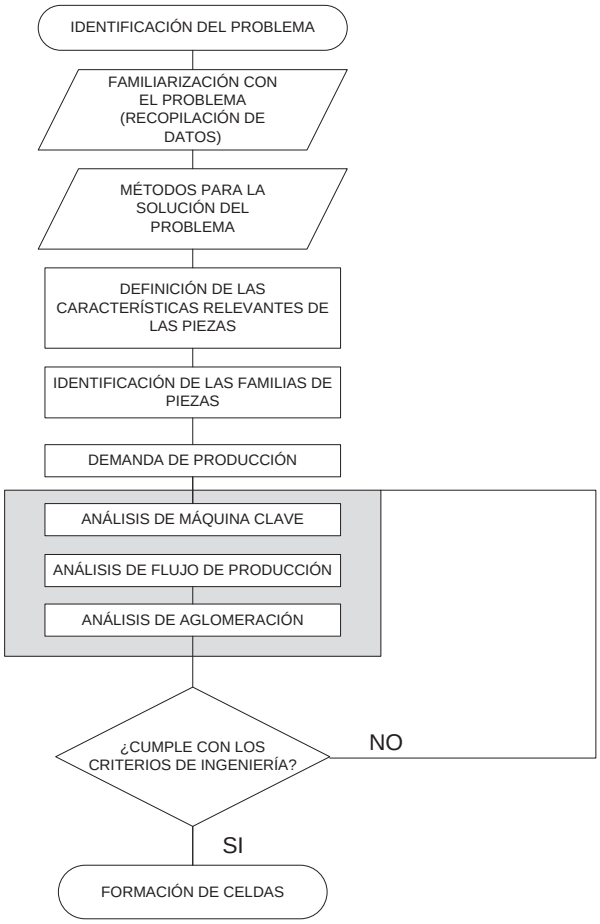


Figura 5.4 – Diagrama de flujo del algoritmo de formación de celdas

5.2.2. Ventajas y Desventajas de las Celdas de Manufactura en una Planta

Las principales ventajas y desventajas de un proceso de manufactura por celdas se ilustran en la tabla 5.1 [24]:

Tabla 5.1 – Ventajas y desventajas del uso de la manufactura de celdas

VENTAJAS	DESVENTAJAS
• Reducido manejo de material y tiempo de tránsito. • Reducido inventario de trabajo en proceso. • Buen uso del recurso humano. • Mayor facilidad de control. • Mayor facilidad de automatización.	• Familias de partes inadecuadas. • Celdas pobremente balanceadas. • Extendido entrenamiento y programación de operarios. • Aumento en la inversión de capital. • Elevada cantidad de datos.

5.3. Estado del Problema

Uno de los objetivos principales de la manufactura celular o de celdas es el de minimizar los movimientos e intercambio de material entre los grupos, objetivo que es logrado si se generan celdas que garanticen la fabricación completa de los productos asignados. Si esto no es posible entonces habría piezas que deberán visitar diferentes grupos en su fabricación.

Para lograr la reorganización del sistema de producción es necesario conocer las rutas de fabricación de cada una de las piezas, lo cual permite determinar las máquinas visitadas por cada una de las partes producidas en su ruta de fabricación Una herramienta que permite de manera ordenada resumir esta información es la matriz pieza-máquina en la cual se puede determinar con unos o ceros las máquinas que son necesarias para la fabricación de cada una de las piezas. En esta matriz de manera general se representan las máquinas en las filas y las piezas en las columnas, esta representación no presenta en ningún momento

información referente a la secuencia de las operaciones o el número de máquinas existentes de cada tipo en las instalaciones de manufactura.

En la Tabla 5.2 se muestra un ejemplo de una matriz pieza – máquina, en la cual se observa que los únicos componentes son 1 o 0 teniendo los siguientes significados: si la componente $a_{ij} = 0$, la máquina *i* no procesa la pieza *j*, de lo contrario si la componente $a_{ij} = 1$ la máquina *i* si procesa la pieza *j*

Tabla 5.2 – Ejemplo matriz pieza-máquina

Máquina	**Pieza**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
A	1	0	1	0	1	0	0	0	0	0
B	0	1	0	1	1	0	0	0	0	0
C	0	1	0	1	0	0	1	0	0	0
D	0	1	0	1	0	0	0	0	0	0
E	1	0	0	1	0	1	0	0	0	1
F	0	0	0	0	1	1	0	0	1	0
G	0	0	1	0	1	0	0	0	1	0
H	1	0	0	0	0	0	0	0	0	0
I	0	0	0	1	0	0	0	1	0	0
J	0	0	1	0	0	1	1	1	0	1

A partir de la matriz pieza - máquina se podría determinar una agrupación natural de piezas y partes en celdas. Para ello se hace necesario generar un procedimiento lógico que permita transformar una matriz de la forma observada en la Tabla 5.2 en una matriz pieza – máquina como la mostrada en la Tabla 5.3 [25].

Tabla 5.3 – Ejemplo matriz pieza-máquina agrupada

Máquina	**Pieza**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
A	1	1	1	0	0	0	0	0	0	0
B	1	1	1	0	0	0	0	0	0	0
C	1	1	1	0	0	0	0	0	0	0
D	1	0	0	0	0	0	0	0	0	0
E	0	0	0	1	1	0	0	0	0	0
F	0	0	0	1	0	1	0	0	0	0
G	0	0	0	1	1	1	0	0	0	0
H	0	0	0	1	0	1	0	0	0	0
I	0	0	0	0	0	0	1	1	1	1
J	0	0	0	0	0	0	1	1	0	0

En la Tabla 5.3 se observa una matriz-pieza máquina en la que la diagonal principal está formada preferentemente por números 1, mientras que las componentes en otro sector de la matriz son 0. En esta tabla se observa que los 10 tipos de piezas y los 10 tipos de máquinas se agrupan principalmente en tres grupos totalmente independientes entre sí. Se podría resumir la solución usando un diagrama como el mostrado en la Figura 5.5, en el que se muestra de forma esquemática los resultados plasmados en la Tabla 5.3.

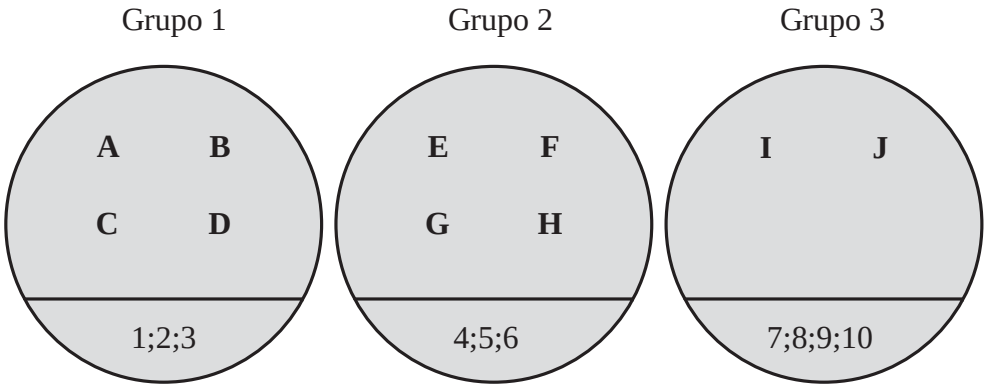


Figura 5.5 – Ejemplo grupo de máquinas

La solución de este ejemplo tiene la característica de que en ella no se hace necesario el intercambio de piezas entre grupos minimizándose de esta manera sus costos asociados. Hay ocasiones en que la solución no muestra el estado ideal de la Tabla 5.3, y en lugar a esto muestra un contexto como el referido en la Tabla 5.4.

Tabla 5.4 – Estructura diagonal de la matriz pieza-máquina

Máquina	**Pieza**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
A	1	1	1	0	0	0	0	0	0	0
B	1	1	1	0	0	0	0	0	0	0
C	1	1	1	0	0	0	0	0	0	0
D	1	0	0	1	0	0	0	0	0	0
E	0	0	0	1	1	0	0	0	0	0
F	0	0	0	1	0	1	0	0	0	0
G	0	0	0	1	1	1	0	0	0	0
H	0	0	0	1	0	1	0	0	0	0
I	0	0	0	0	0	0	1	1	1	1
J	0	0	0	0	0	0	1	1	0	0

En la tabla anterior se muestra un caso típico en el que se logra una solución que requiere manejar de alguna manera el intercambio de material entre dos grupos, se observa pues que la pieza 4 aunque es agregada al grupo 2, requiere la máquina D que haría parte del grupo 1. Este hecho generaría buscar una alternativa para el manejo del intercambio de la pieza 4 entre las celdas de manufactura 1 y 2. La Figura 5.6, muestra un esquema de la solución mostrada en la Tabla 5.4.

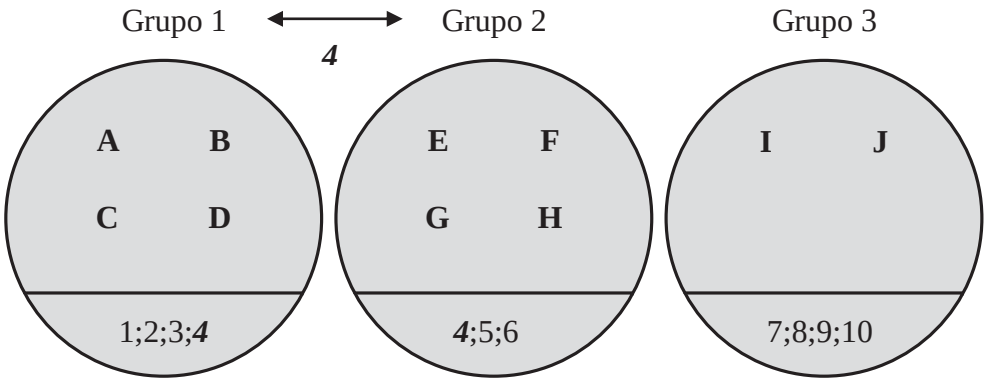


Figura 5.6 –Ejemplo de la representación grupal de una matriz pieza-máquina

5.4. Representación del MCDP como un COP

El principal objetivo para la solución del MCDP en una agrupación de máquinas es la formación de un conjunto de máquinas y piezas de trabajo en grupos dentro de celdas, de manera tal, que el número de piezas transportadas entre celdas sea minimizado. Por lo tanto se busca primordialmente la transformación de una matriz de incidencia (pieza-maquina) en una matriz con estructura diagonal (véase tabla 5.4). [11].

En este proyecto, el problema de agrupación de máquinas es considerado como un COP, dada esa condición, se declara a continuación el modelo de optimización sujeto a constantes, variables, dominios, función objetivo y restricciones:

A. Constantes

- M, es el número de máquinas.
- P, el número de piezas (partes).
- C, el número de celdas.
- M_{max}, el número máximo de máquinas por celda.

B. <u>Variables y dominios</u>

- A= [a_{ij}], matriz binaria de piezas-máquina de dimensiones *MxP*, con dominio [0,1] (véase figura 6.5)
  - *i*, el índice de máquinas, con dominio (*i*=1,…,M).
  - *j*, el índice de piezas, con dominio (*j*=1,…,P).
  - *k*, el índice de celdas, con dominio (*k*=1,…,C).
- B= [y_{ik}], matriz binaria máquina-celda de dimensiones *CxM* con dominio [0,1], (véase tabla 6.6), sujeta a la siguiente representación:

$$y_{ik} = \begin{cases} 1 & \text{si la máquina } i \in \text{a la celda } k \\ 0 & \text{otros casos} \end{cases}$$

- C= [z_{jk}], matriz binaria pieza-celda de dimensiones *PxC* con dominio [0,1], (véase tabla 6.7), sujeta a la siguiente representación:

$$z_{jk} = \begin{cases} 1 & \text{si la pieza } j \in \text{a la familia } k \\ 0 & \text{otros casos} \end{cases}$$

C. Función objetivo

La función objetivo para el MCDP es reducir al mínimo el número de veces que una determinada parte debe ser procesada por una máquina que no pertenece a una celda por sobre una parte que si pertenece a ella, la siguiente representación matemática ilustra la situación anterior.

Minimizar: $$\sum_{k=1}^{C}\sum_{i=1}^{M}\sum_{j=1}^{P} a_{ij} z_{jk}(1 - y_{ik})$$

D. Restricciones

$$\sum_{k=1}^{C} y_{ik} = 1 \quad \forall i,$$

$$\sum_{k=1}^{C} z_{jik} = 1 \quad \forall j,$$

$$\sum_{i=1}^{C} y_{ik} \leq M_{max} \quad \forall k.$$

La primera de las restricciones asegura si una maquina pertenece a una celda. La segunda de las restricciones muestra si una pieza pertenece a una familia de componentes (celdas), y la última de las restricciones muestra que el número de máquinas no exceda el número máximo de máquinas por celda. Las ilustraciones de las restricciones anteriores se ven en las tablas, 5.6, 5.7 respectivamente.

Tabla 5.5 – Ejemplo de la matriz pieza-maquina

Máquina	**Pieza**				
	1	**2**	**3**	**4**	**5**
A	1	0	0	0	0
B	0	0	0	0	1
C	0	1	0	0	0
D	0	0	0	1	0
E	0	0	1	0	0

Tabla 5.6 – Ejemplo de la matriz máquina-celda

Máquina	**Celda**				
	1	**2**	**3**	**4**	**5**
A	0	1	0	0	0
B	1	0	0	0	0
C	0	0	1	0	0
D	0	0	0	0	1
E	0	0	0	1	0

Tabla 5.7 – Ejemplo de la matriz pieza-celda

Pieza	**Celda**				
	1	**2**	**3**	**4**	**5**
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	1	0	0
4	0	1	0	0	0
5	1	0	0	0	0

6. Solvers Utilizados en la Resolución del MCDP

6.1. ECL^{i}PS^e

La primera generación de lenguajes de programación con restricciones se centraba principalmente en una simple técnica: propagación de restricciones [28]. El *solver* ECL^{i}PS^e fue diseñado para ser más que un programa de desarrollo e implementación de *Constraint Programming*. ECL^{i}PS^e también tiene soporte para la programación matemática y las técnicas de programación estocástica. La ventaja crucial de ECL^{i}PS^e es que permite a los programadores el uso de algoritmos combinatorios.

El comienzo de ECL^{i}PS^e se puede ver primeramente desde la creación de la *European Computer-Industry Research Center* (ECRC) en Múnich, entidad que nace en 1984 por tres compañías europeas para la exploración y desarrollo de técnicas de razonamiento avanzadas y aplicables a los problemas prácticos.

La primera interfaz puesta en liberada utilizando técnicas lineales fue mezclada con un paquete de programación entera en 1997. La integración de los *solvers* de dominios finitos para solucionar programas de programación lineales de apoyo a algoritmos híbridos, llegó el año 2000. En el 2001, la colección *"ic"* fue liberada, compatibilizando con las restricciones booleanas, enteras y reales satisfaciendo las demandas más importantes [26].

En 1999, los derechos comerciales de ECL^{i}PS^e fueron transferidos a IC-Parc´s *spin-off* de la compañía *Parc Technologies*, la cual usa ECL^{i}PS^e en su optimización de productos y provee la financiación para su mantenimiento y continuo desarrollo. En agosto de 2004, *Parc Technologies* y la plataforma ECL^{i}PS^e fueron adquiridas por *Cisco System.*

ECL^{i}PS^e está en cientos de instituciones de enseñanza e investigación alrededor del mundo y sigue siendo libre para la educación y con fines de investigación. Se ha explotado en una variedad de aplicaciones académicas de todo el mundo, incluida la planificación de la producción, programación de transporte, bioinformática, optimización entre otras. Es también utilizada para desarrollar software de optimización comercial de *Cisco Systems*.

6.1.1. Sintaxis de ECL^{i}PS^e

ECL^{i}PS^e basa su sintaxis en Prolog, este último utilizado para resolver problemas en los que existen objetos y relaciones entre objetos. La programación en Prolog, por consiguiente la de ECL^{i}PS^e consiste simplemente en:

- declarar hechos sobre los objetos y sus relaciones,
- definir reglas sobre dichos objetos y relaciones, y
- hacer preguntas.

6.1.1.1. Sistema Lógico

Un sistema lógico es un conjunto de hechos y reglas, las cuales se describen a continuación:

• Hechos: Conjunto de elementos que son lógicamente verdad.

• Reglas: Conjunto de leyes que cumplen todos los hechos que pertenecen al sistema.

Al sistema lógico comúnmente se le es llamado base de conocimientos.

6.1.1.1.1. Hechos

Un *hecho* es una relación entre objetos [42]. Su sintaxis en ECL^{i}PS^e es:

```
relación (objeto, objeto,…).
```

La relación se conoce como el predicado y los objetos como argumentos. Dada la significancia de estos dos últimos conceptos, los siguientes puntos son importantes:

- Los nombres de las relaciones deben de empezar con letra minúscula.
- Los objetos se escriben separados por comas y encerrados entre paréntesis.
- Al final del hecho debe de haber un punto.

Por ejemplo, un hecho puede ser: `persona (juan, 27).`

6.1.1.1.2. Reglas

Cuando la verdad de un hecho depende de la verdad de otro hecho o de un grupo de hechos se una *regla*. Una regla consiste en una cabeza y un cuerpo. El cuerpo puede estar formado por varios hechos u objetivos [42]. Su sintaxis es:

```
cabeza:- objetivo 1, objetivo 2,…, objetivo n
```

Los objetivos deben ir separados por comas, especificando conjunción y al final debe ir un punto. Ejemplo de una regla es el siguiente:

```
mayor_de_edad(X):- persona(X,E), E>18
```

6.1.1.1.3. Variables

Las variables se utilizan para describir hechos y reglas generales (ver sección 7.3.2).

6.1.1.2. Convenciones Sintácticas de Prolog

Las convenciones sintácticas siempre juegan un papel importante en la discusión de cualquier paradigma de la programación y la programación en lógica no es una excepción en esta materia. La sintaxis de Prolog está llena de convenciones que son altamente originales y muy poderosas. Su impacto es poco conocido fuera de la comunidad de la programación pero no lo hace menos importante en materias como *Constraint Programming*.

Por *"objects of computation"* o en español objetos de computación se referirá a cualquier cosa que pueda ser denotada como una variable Prolog. Estos no son solo números, listas y demás, sino también estructuras compuestas e incluso otras variables o programas. Formalmente los objetos de computación son *"base terms"* los cuales consisten en:

- Variables, denotadas por medio de *strings* que comienzan con una letra mayúscula o "_" (el *underscore*, en inglés o guión bajo en español), por ejemplo X3 es una variable.
- Números, denotados como dígitos por ejemplo 0,1,2,3,4,5,6,7,8,9 y la combinación de éstos. Soportando a su vez números reales, enteros, etc.
- Átomos, denotados por secuencias de caracteres comenzando con una minúscula, por ejemplo x4 es un átomo. Cualquier secuencia de caracteres puestos entre comillas simples, aun admitiendo espacios, también es un átomo, por ejemplo, "Mi nombre".

En la sintaxis estándar de ECL^{i}PS^e para los términos compuestos, existe lo que son los términos *functor*, estos últimos definidos para tratamiento de funciones matemáticas. La sintaxis de estos son con los argumentos separados por comas y cerrando todo entre paréntesis. Por ejemplo el término con *functor f* y argumentos a, b y c es escrito como *f (a, b, c)*. Similarmente se tiene también un ejemplo más complejo con *functor h* y tres argumentos, como:

- La variable A,
- El término compuesto `f (g, ´Twenty´, X),`
- La variable X, es escrita `h (A, f (g, ´Twenty´, X), X).`

Algunos términos compuestos puede ser escritos también usando la notación *infix notation* (es la notación aritmética común y la notación lógica de fórmulas).

Las expresiones aritméticas y de evaluaciones son principalmente importantes para la definición de condiciones de términos de las sentencias a utilizar. En Prolog se tienen muchas representaciones de constantes enteras, como lo son:

0, -1, 1, -2, 2,…

Al igual que las representaciones de números de puntos flotantes, como lo son:

0.0, -1.72, 3.141, -2.0,…

Los números del tipo punto flotante son representados como "reales" y los dígitos permitidos después del punto decimal obligatorio "." dependen de la implementación que sea establecida.

6.1.2. Ejemplos de ECL^{i}PS^e

Una de las características de ECL^{i}PS^e es que es un sistema de programación lógica con restricciones que consiste según sus creadores en, un sistema de código abierto para el eficaz manejo de costos y programación con restricciones [26]. Puede ser aplicado en planificación, programación, asignación de recursos, horarios, transporte, etc., Y es nombrado como ideal para la enseñanza de la mayoría de los aspectos de la resolución de problemas combinatorios, por ejemplo, programación con restricciones, programación matemática y técnicas de búsqueda. Contiene varias bibliotecas para *solvers* que utilizan restricciones, con un alto nivel de modelado y lenguaje de control, interfaces con otros fabricantes que dan un entorno de desarrollo integrado e interfaces para integrarse en entornos *host*.

6.1.2.1. El Problema de las N-Reinas

Como se describe en la sección 4.4.2, el problema de las n-reinas tiene restricciones las cuales debes ser satisfechas para encontrar la solución al problema. El siguiente algoritmo desarrollado en ECL^{i}PS^e presenta la resolución del problema de las n-reinas.

```
:-lib(ic).

queens(N,Board):-

dim(Board, [N]),
Board[1..N] :: 1..N,

( for(I,1,N), param(Board,N) do
      ( for(J,I+1,N), param(Board,I) do
            Board[I] #\= Board[J],
            Board[I] #\= Board[J]+J-I,
            Board[I] #\= Board[J]+I-J
      )
),

labeling(Board).
```

Figura 6.1 Ejemplo del problema de las n-reinas en ECL^{i}PS^e

6.1.2.2. Send + More = Money

Un clásico ejemplo se puede ver aquí. El problema criptográfico denominado "*send + more = money*", descrito anteriormente en la sección 4.4.1, está resuelto en ECL^{i}PS^e, como se presenta a continuación.

```
:-lib(ic).

send(L):-
L = [S,E,N,D,M,O,R,Y],
L :: [0..9],

S #\= 0,
M #\= 0,
alldifferent(L),
1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E
#= 10000*M + 1000*O + 100*N + 10*E + Y,

labeling(L).
```

Figura 6.2 Ejemplo problema criptográfico en ECL^{i}PS^e

6.1.3. Heurísticas de ECL^{i}PS^e

Las heurísticas de selección de valor según el Manual de Referencia online de ECL^{i}PS^e CLP [27] y las heurísticas de selección de variables que pueden ser implementadas sobre el *solver* ECL^{i}PS^e son variadas y se presentan a continuación [29].

6.1.3.1. Heurísticas de Selección de Variables Predefinidas

- **input_order:** La primera entrada es la seleccionada.
- **first_fail:** La entrada con el dominio más pequeño es seleccionada.
- **anti_first_fail:** La entrada con el dominio más grande es seleccionada.
- **smallest:** La entrada con el valor más pequeño en el domino es seleccionada.
- **largest:** La entrada con el valor más grande en el dominio es seleccionada.
- **occurrence:** La entrada con el mayor número de restricciones asignadas es seleccionada.
- **most_constrained:** la entrada con el domino más pequeño es seleccionada. Si varias entradas tienen el mismo tamaño del dominio, la entrada con el mayor número de restricciones es seleccionada.

- **max_regret:** la entrada con la mayor diferencia entre el menor y el segundo menor valor en el domino es seleccionada. Este método es típicamente usado si la variable representa un costo y se interesan en la elección de lo que podría aumentar los costos. En general, para la mayoría de los casos, la mejor posibilidad no se toma. Desafortunadamente, la implementación no siempre funciona. Si dos variables de decisión incurren en los mismos costos mínimos, el *regret* (arrepentimiento) no se calcula como cero, sino como la diferencia desde el mínimo valor al siguiente valor más alto.

6.1.3.2. Heurísticas de Selección de Valor Predefinidas

- **indomain:** Utiliza el predicado (*built-in*) **indomain/1**. Los valores son probados en orden creciente. En caso de fallo los valores previos no son borrados.
- **indomain_min:** los valores son probados en orden creciente. En caso de fallo los valores probados previamente son borrados. Los valores son probados en el mismo orden de "*indomain*", pero los *backtracks* (vueltas atrás) pueden ocurrir antes.
- **indomain_max:** los valores son probados en orden decreciente. En caso de fallo el valor probado previamente es eliminado.
- **indomain_reverse_min:** igual que *indomain_min*, pero las alternativas son juzgadas en orden inverso. Por ejemplo, el menor valor es el primero en ser removido desde el dominio y solo si estos falla el valor es asignado.
- **indomain_reverse_max:** el mismo que *indomain_max*, pero las alternativas son juzgadas en orden inverso. Es decir, el mayor valor es el primero en ser removido desde el dominio y solo si este falla, el valor es asignado
- **indomain_middle:** Los valores son evaluados desde la mitad del dominio. En caso de falla, el valor probado previamente es eliminado.
- **indomain_median:** Los valores son juzgados a partir del valor medio del dominio. En caso de falla el valor probado previamente se elimina.
- **indomain_split:** Los valores son probados dividiendo sucesivas veces el dominio, probando la mitad inferior del primer dominio. En caso de fallar, el intervalo probado es removido. Con esto se enumeran los valores en el mismo orden que *indomain* o *indomain_min*, pero puede fallar tempranamente.

- **indomain_reverse_split:** Los valores son probados por medio de sucesivas divisiones de los dominios, probando desde la mitad superior del primer dominio. En caso de fallar el intervalo probado es eliminado. Con esto la enumeración de valores es el mismo que *indomain* o *indomain_max*, pero puede fallar tempranamente.

- **indomain_random:** Los valores son evaluados en orden aleatorio. En *backtracking*, el valor previamente evaluado es eliminado. Usando esta rutina los resultados pueden ser irreproducibles, con otra llamada pueden crearse números aleatorios en una secuencia diferente. Este método usa un predicado *built-in*, **random/1** para crear números aleatorios. También se tiene el predicado **seed/1** que es capaz de obligar a la generación de una secuencia de números en otra ejecución.

- **indomain_interval:** Si el dominio se compone de varios intervalos, se elige la primera rama del intervalo. Para un intervalo se utiliza la división del dominio.

6.2. Choco

El *solver* Choco originado en 1999 con el proyecto OCRE, una iniciativa francesa de un *solver* abierto para la enseñanza y la investigación, con participación de investigadores y enseñar e investigar con participaciones de investigadores y profesionales (École des Mines), Montpellier (LIRMM), Toulouse (INRA y ONERA) y Bouygues SA. Esta primera implementación fue en CLAIRE [30], Yves Caseau's lenguaje compilado en C++. Un gran cambio en el año 2003 fue la implementación en lenguaje de programación Java, con el objetivo de asegurar la gran portabilidad y una rápida adaptación de los nuevos usuarios.

En 2008 Choco dío un paso más allá, contratando un ingeniero de *solvers* tiempo completo (financiado por la École des Mines de Nantes, Bouygues SA and Amadeus SA) para el *solver*, con lo que Choco entra a una nueva etapa de desarrollo. Con su versión 2 el 10 de Septiembre del 2008, se presenta una clara separación entre el modelado y los mecanismos de resolución, proveyendo a ambos herramientas de modelado y herramientas de resolución. Un completo reacondicionamiento aportando a un mejor desempeño y una interfaz más usable para los usuarios expertos y principiantes.

A diferencia de algunos *solvers* de programación con restricciones como por ejemplo *Sicstus Prolog*, Choco *solver* es una herramienta de código abierto en inglés conocido como *open-source* distribuido bajo licencia BSD y auspiciado y alojado en sourceforge.net. Fue especialmente diseñado para la investigación y el aprendizaje académico.

6.2.1. Diseño de Choco

Choco es una librería de Java, lo que provee una clara separación entre el modelado y la resolución. La figura 6.3 presenta la arquitectura de la librería Choco [30].

La primera parte, desde el punto de vista del usuario, es dedicada para expresar el problema. La idea de esto es manipular las variables y las restricciones de estas últimas con el objeto de abordar un problema de la manera más fácil posible.

La segunda parte es dedicada para resolver realmente el problema. Gestión que reside, en la solución del problema además de administración de la memoria involucrada en la búsqueda basada en árbol. En la figura 6.3 se muestran las situaciones anteriores:

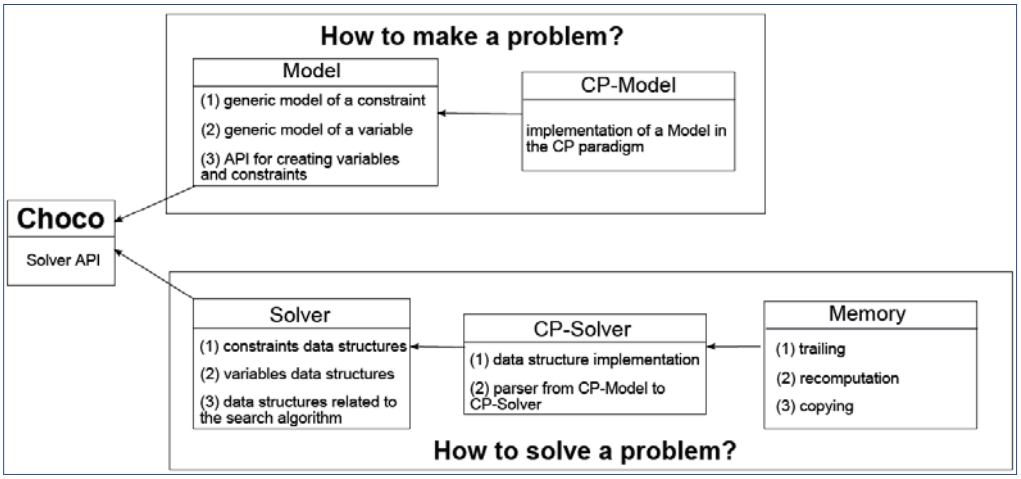


Figura 6.3 – Diseño del *solver* Choco.

6.2.2. Sintaxis de Choco

Dentro de las características de Choco como *solver*, se encuentran la de modelador de problemas y también la de un *solver* de programación con restricciones habilitado como una librería Java. Sin embargo, esta arquitectura permite aun adicionar complementos de *solvers* no necesariamente basados en CP.

Choco es un modelador de problemas que permite manipular una amplia variedad de tipos de datos como son:

- Variables enteras
- Conjuntos de variables representadas como variables enteras
- Variables reales, representando variables que toman ciertos valores en intervalos de números flotantes.
- Expresiones que representan un entero o real basado en una expresión que utiliza operadores como *plus, mult, minus, scalar, sum,* etc.

Los modelos de Choco aceptan más de 70 restricciones, siempre que la llamada del *solver* sea un programa basado en CP:

- Restricciones clásicas aritméticas en reales o enteros como: *equal, les sor equal, greater or equal,* etc.
- Operación de números booleanos.
- Tablas de restricciones definidas como un conjunto de tuplas.
- Un extenso número de restricciones clásicas como por ejemplo: *alldifferent, global cardinality, nvalue, element,* el *cumulative,* etc.

6.2.3. Ejemplos de Choco

Para ejemplificar de mejor manera el diseño del *solver* Choco, se presenta el siguiente código y posteriormente algunos ejemplos de códigos bajo una IDE Java.

Ejemplo de diseño del *solver* Choco.

```
//1- Create the model
Model m = new CPModel();
int n = 6;

//2- declaration of variables
IntegerVariable[] vars = makeIntVarArray("v", n, 0, 5, "cp:enum");
IntegerVariable obj = makeIntVar("obj",0,100,"cp:bound");

//3- add some constraints
String regexp = "(1|2)(3*)(1|4|5)";
m.addConstraint(regular(regexp, vars));
```

```
m.addConstraint(neq(vars[0], vars[5]));
m.addConstraint(eq(scalar(new int[]{2,3,1,-2,8,10}, vars), obj));

//4- Create the solver
Solver s = new CPSolver();

//5- read the model and solve it
s.read(m);
s.solve();
if (s.isFeasible()) {
       do {
             for (int i = 0; i < n; i++) {
                   System.out.print(s.getVar(vars[i]).getVal());
             }
             System.out.println("");
       } while (s.nextSolution());
 }

//6- Print the number of solutions found
System.out.println("Nb_sol : " + s.getNbSolutions());
```

Este código presenta los siguientes resultados:

133334 72
133335 82
233331 44
233334 74
233335 84
Nb_sol: 5

6.2.4. Heurísticas de Choco

Un ingrediente clave de cualquier enfoque de una estrategia de búsqueda inteligente es el uso de heurísticas de selección de variable y valor. En lo algoritmos *backtracking* y *Branch and Bound* los enfoques la búsqueda se organizan como un árbol de enumeración, en la que cada nodo corresponde a un sub-espacio de la búsqueda, y cada nodo hijo es una subdivisión del espacio de su nodo padre. El árbol de búsqueda se construye progresivamente mediante la aplicación de una serie de estrategias de ramificación que determinan cómo se subdivide el espacio en cada nodo creado.

Básicamente, según lo anterior, una estrategia de búsqueda en Choco es una composición de objetos en una estrategia de ramificación, cada uno definido en un conjunto de variables de decisión. Las estrategias más comunes de ramificación se basan en la asignación de una variable seleccionada por medio de una heurística de selección de

variable, a uno o varios valores seleccionados por medio de una heurística de selección de valor.

6.2.4.1. Heurísticas de Selección de Variable

Una heurística de selección de variable define básicamente, dependiendo de las restricciones involucradas, la manera de elegir una variable no instanciada como la siguiente variable de decisión.

Las heurísticas de selección de variable disponibles en la actualidad en Choco son las siguientes:

- **CompositiveIntVarSelector:** Básicamente es una composición de heurísticas de variable, que tiene por objeto seleccionar una restricción acorde a la primera de las heurísticas de selección de valor para luego asignar un valor entero involucrado en una restricción de la segunda de las heurísticas de variable. Esta composición de heurísticas es aplicable a las heurísticas de selección de dominios más pequeños, como también a las heurísticas de selección de dominios más amplios, todas estas explicitadas más adelante en este documento.

- **CyclicRealVarSelector:** Selecciona las variables reales en el orden en que aparecen en un arreglo *x*, a través de un camino cíclico hasta que todas ellas sean instanciadas.

- **LexIntVarSelector:** Selecciona una variable de tipo entero acorde a una primera heurística de variable, hasta que esa razón de selección se rompa con la elección de una segunda heurística de selección de variable.

- **MaxDomain:** Selecciona una variable de tipo entero con el dominio más grande.

- **MaxDomSet:** Selecciona un conjunto de variables de tipo entero con el dominio más grande.

- **MaxRegret:** Selecciona una variable de tipo entero con la diferencia más grande entre los dos valores más pequeños en esos dominios.

- **MaxRegretSet:** Selecciona un conjunto de variables con la diferencia más grande entre los dos valores más pequeños de ese conjunto.

- **MaxValueDomain:** Selecciona una variable de tipo entero con el valor más grande en su dominio.
- **MaxValueDomSet:** Selecciona un conjunto de variables con el valor más grande.
- **MinDomain:** Selecciona la variable de tipo entero con el dominio más pequeño.
- **MinDomSet:** Selecciona un conjunto de variables con el dominio más pequeño.
- **MinValueDomain:** Selecciona una variable de tipo entero con el valor más pequeño de su dominio.
- **MinValueDomSet:** Selecciona una variable de tipo entero con el valor más pequeño de un grupo seleccionado.
- **MostConstrained:** Selecciona una variable de tipo entero involucrada con el número más grande de restricciones posibles en el *solver*.
- **MostConstrainedSet:** Selecciona un conjunto de variables involucradas con el número más grande de restricciones en el *solver*.
- **RandomIntVarSelector:** Selecciona una variable de tipo entero de forma aleatoria.
- **RandomSetVarSelector:** Selecciona un conjunto de variables de tipo entero de manera aleatoria.
- **StaticSetVarOrder:** Selecciona un conjunto de variables en orden que aparezcan en un arreglo predefinido por el *solver*.
- **StaticVarOrder:** Selecciona una variable de tipo entero en orden que aparezcan en un arreglo predefinido por el *solver*.

6.2.4.2. Heurísticas de Selección de Valor

Básicamente las heurísticas de selección de valor se caracterizan, en una primera instancia, por ser parte de una estrategia de búsqueda y de computo, donde su principal tarea es instanciar las variables dependiendo de sus dominios conformes a los valores consultados en las llamadas a las variables. Actualmente las heurísticas de selección de valor disponibles en Choco son las siguientes:

- **MaxVal:** La heurística de selección de valor *MaxVal* selecciona el valor más grande en el dominio de una variable de tipo entero.
- **MidVal:** La heurística de selección de valor *MidVal* selecciona el valor más cercano (igual o mayor) al punto medio de una variable de tipo entero.
- **MinVal:** La Heurística de selección de valor *MinVal* selecciona el valor más pequeño en el dominio de una variable de tipo entero.

7. Modelado del MCDP en ECL^{i}PS^e

El modelado de un problema constituye un aspecto fundamental en la disciplina de la ingeniería informática. El modelado de sistemas y de procesos, entendido como el establecimiento de relaciones semánticas entre la teoría, los objetos y los fenómenos, es una herramienta básica en la explicación científica y es un proceso substancial en la resolución de problemas. A su vez, la conformación de problemas de optimización combinatoria, por la propia estructura del cuerpo de conocimientos que abarca, así como por la lógica de tratamiento de esos conocimientos, requiere para su comprensión y aprendizaje, trabajar con modelos y aplicar razonamiento hipotético deductivo.

Dada esa naturaleza, el modelamiento del MCDP no queda exento de esa lógica de tratamiento, es por esto que se requiere de un análisis hipotético (ver capítulo 5.4) y una transferencia desde este último a una representación codificable basada en sintaxis del *solver* ECL^{i}PS^e.

7.1. Representación del MCDP en ECL^{i}PS^e

La representación del MCDP consiste principalmente en hacer una transferencia en términos del modelamiento hipotético visto en el capítulo 5.4, en una representación codificable bajo términos y sintaxis del *solver* de programación con restricciones ECL^{i}PS^e, es decir, dada la formulación del MCDP como un problema de satisfacción de restricciones, establecer ese modelo a base de variables, constantes, dominios y restricciones bajo términos de ECL^{i}PS^e. Dado esto, primeramente se presenta el código sujeto a restricciones y sintaxis lógica; para posteriormente ilustrar y especificar cada uno de los componentes del ya mencionando código, dentro de lo cual se establecen las variables, dominios y restricciones del MCDP sujetos a ECL^{i}PS^e. El código descrito anteriormente se encuentra en el anexo 1.

7.1.1. La Librería ic en el MCDP

La librería *ic* de su sigla en inglés *Interval Constraint,* es principalmente utilizada en un principio por sobre la librería *sd,* ya que mejora considerablemente las capacidades proporcionadas por la librería *sd*, la cual posee soporte solo para dominios simbólicos. La librería *ic* al igual que le librería *sd* posee soporte para restricciones fundamentales [27], entendiéndose estas últimas como las declaraciones de variables (restricciones unarias),

restricciones booleanas y restricciones aritméticas. Algunas de las principales características de las librerías más utilizadas pueden ser vistas en la siguiente tabla:

Tabla 7.1 Tabla de librerías de ECL^{i}PS^e

Solver Lib	Var Domains	Constraints Class	Behaviour
suspend (sd)	Numeric, symbol	Arbitrary arithmetic in/dis/equalites	Passive test
Fd	Integer, symbol	Linear in/dis/equalities	Domain propagation
Ic	Real, integer	Arbitrary arithmetic in/dis/equalities	Bounds/domain propagation
ic_global	Integer	N-ary constraints over lists of integers	Bounds/domain propagation
ic_sets	Set of integer	Set operations (subset, cardinality, union,…)	Set-bounds propagation
ic_symbolic	Ordered symbols	Dis/equality, ordering, element,…	Bounds/domain propagation
Sd	Unordered symbols	Dis/equality, alldifferent	Domain propagation
propia	Inherited	Any	various

La librería *ic* también es compatible con un mayor alcance para las declaraciones de variables, esto referido a la mayor capacidad, por sobre la librería *sd*, para declarar variables en dominios más complejos. La representación de esta librería puede ser visualizada en la siguiente figura:

```
:-lib(ic).
:-lib(ic_global).
:-lib(ic_search).
:-lib(branch_and_bound).
```

Figura 7.1 – Representación de Librerías en ECL^{i}PS^e

7.1.2. Formalización del MCDP como un COP en ECL^{i}PS^e

Anteriormente se ha presentado la visualización de las diferentes reglas que componen el programa en ECL^{i}PS^e del MCDP, por lo tanto ahora corresponde esencialmente ahondar y explicitar la representación de las variables, constantes, dominios y función objetivo del MCDP en ECL^{i}PS^e. Para observar de mejor manera la representación, se clasifican los conceptos de la siguiente manera:

7.1.2.1. Constantes

- M, es el número de máquinas.
- P, el número de piezas (partes).
- C, el número de celdas.
- M_{max}, el número máximo de máquinas por celda.

Básicamente las constantes están presentes en 3 reglas del programa MCDP, una de ellas es la regla ***Matrix_Machine_Part***, en donde el número de máquinas y el número de partes están dados por la cantidad de filas y columnas de la data de test para resolver el problema. La representación si visualizada en la figura siguiente:

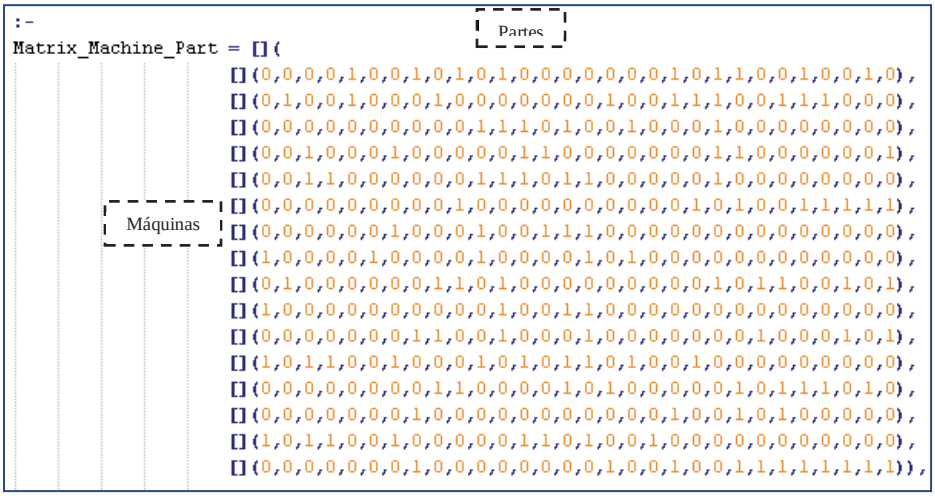


Figura 7.2 – Representación implícita de constantes en ECL^{i}PS^e

Las otras dos reglas representan a las constantes de manera explícita y definida. Las reglas que hacen esta representación de las constantes son, la regla ***machine_cell (Matrix_Machine_Cell)*** y la regla ***part_cell (Matrix_Part_Cell),*** en donde el número de máquinas, el número de celdas, el número de partes y el número máximo de máquinas por celda son declarados con mayúscula con su determinada dimensión. En la figura siguiente se observa esta situación:

```
36
37  machine_cell(Matrix_Machine_Cell):-
38
39      Machines = 12,
40      Cells= 2,
41      Sum is 1,
42      Mmax is 6, %Mmax = Maquinas - 1
43
44
45      dim(Matrix_Machine_Cell, [Machines,Cells]),              % make the variables
46      Matrix_Machine_Cell[1..Machines, 1..Cells] :: 0..1,      % set the domains
47
48      ( for(I,1,Machines),param(Cells,Matrix_Machine_Cell,Sum) do
49
50          sum(Matrix_Machine_Cell[I,1..Cells]) #=  Sum
51
52      ),
53
54      ( for(J,1,Cells),param(Machines,Matrix_Machine_Cell,Mmax) do
55
56              sum(Matrix_Machine_Cell[1..Machines,J]) #=<  Mmax
57
58      ),
59
60  labeling(Matrix_Machine_Cell).
```

Figura 7.3 – Representación de constantes en ECL^{i}PS^e

```
73
74  part_cell(Matrix_Part_Cell):-
75
76      Parts=12,
77      Cells=2,
78      Sum is 1,
79
80      dim(Matrix_Part_Cell, [Parts,Cells]),
81      Matrix_Part_Cell[1..Parts,1..Cells] :: 0..1,
82
83
84      ( for(I,1,Parts),param(Cells,Matrix_Part_Cell,Sum) do
85
86          sum(Matrix_Part_Cell[I,1..Cells]) #=  Sum
87
88      ),
89
90
91  labeling(Matrix_Part_Cell).
```

Figura 7.4 – Representación de constantes en ECL^{i}PS^e

7.1.2.2. Variables y Dominios

- A= [a_{ij}], matriz binaria de piezas-máquina de dimensiones *MxP*, con dominio [0,1]
  - *i*, el índice de máquinas, con dominio (*i*=1,…,M).
  - *j*, el índice de piezas, con dominio (*j*=1,…,P).
  - *k*, el índice de celdas, con dominio (*k*=1,…,C).

La figuración de la variable matricial está dada por la matriz ***Matrix_Machine_Part*** en donde una matriz representa a las maquinas que fabrican una cierta parte de un producto. Esta matriz que se dimensiona de acuerdo a la cantidad de máquinas y cantidad de partes de la matriz de data inicial y posee un dominio que está configurado por la matriz de data inicial, es decir, corresponde a un dominio entre cero y uno. Esto se observa a continuación:

```
Matrix_Machine_Part = []([0,0,0,0,1,0,0,1,0,1,0,1,0,0,0,0,0,0,1,0,1,1,0,0,1,0,0,1,0),
                         [](0,1,0,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,1,1,1,0,0,1,1,1,0,0,0),
                         [](0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,0,0,1,0,0,0,1,0,0,0,0,0,0,0),
                         [](0,0,1,0,0,0,1,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,1),
                         [](0,0,1,1,0,0,0,0,0,0,0,1,1,1,0,1,1,0,0,0,0,0,1,0,0,0,0,0,0),
                         [](0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,1,1,1),
                         [](0,0,0,0,0,0,0,1,0,0,0,1,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0),
                         [](1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0),
                         [](0,1,0,0,0,0,0,0,1,1,0,1,0,0,0,0,0,0,0,1,0,1,1,0,0,1,0,1),
                         [](1,0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0),
                         [](0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,1,0,1),
                         [](1,0,1,1,0,0,1,0,0,0,1,0,1,0,1,1,0,1,0,0,1,0,0,0,0,0,0,0,0),
                         [](0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,1,1,0,1,0),
                         [](0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,1,0,0,0,0),
                         [](1,0,1,1,0,0,1,0,0,0,0,0,1,1,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0),
                         [](0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,1,1,1,1,1,1)),
```

Figura 7.5 – Representación de la matriz máquina-parte en ECL^{i}PS^e

- B= [y_{ik}], matriz binaria máquina-celda de dimensiones *CxM* con dominio [0,1],

$$y_{ik} = \begin{cases} 1 & \text{si la máquina } i \in \text{ a la celda } k \\ 1 & \text{otros casos} \end{cases}$$

- C= [z_{ik}], matriz binaria pieza-celda de dimensiones *PxC* con dominio [0,1],

$$z_{jk} = \begin{cases} 1 \text{ si la pieza } j \in \text{ a la familia } k \\ 0 \text{ otros casos} \end{cases}$$

La representación y definición en ECL^{i}PS^e de estas dos variables matriciales pueden ser observadas de forma explícita en las figuras 7.5 y 7.6 respectivamente, en donde, en la segunda de ellas se distingue el predicado ***dim (Matrix_Machine_Cell, [Machines,Cells]),*** encargado de crear y dimensionar las matrices en orden de las constantes detalladas anteriormente. También, se declaran los dominios en donde fluctuaran los valores de estas matrices, esto se define mediante el predicado ***Matrix_Machine_Cell[1..Machines, 1..Cells] :: 0..1,*** en donde los dominios de la matriz máquina-celda oscilan entre 0 y 1.

```
36
37  machine_cell(Matrix_Machine_Cell):-
38
39      Machines = 12,
40      Cells= 2,
41      Sum is 1,
42      Mmax is 6, %Mmax = Maquinas - 1
43
44
45      dim(Matrix_Machine_Cell, [Machines,Cells]),            % make the variables
46      Matrix_Machine_Cell[1..Machines, 1..Cells] :: 0..1,    % set the domains
47
48      ( for(I,1,Machines),param(Cells,Matrix_Machine_Cell,Sum) do
49
50          sum(Matrix_Machine_Cell[I,1..Cells]) #=  Sum
51
52      ),
53
54      ( for(J,1,Cells),param(Machines,Matrix_Machine_Cell,Mmax) do
55
56              sum(Matrix_Machine_Cell[1..Machines,J]) #=<  Mmax
57
58      ),
59
60  labeling(Matrix_Machine_Cell).
```

Figura 7.6 – Representación de la matriz máquina-celda en ECL^{i}PS^e

La representación de la matriz parte-celda mencionada anteriormente se define en ECL^{i}PS^e mediante el uso del predicado ***dim (Matrix_Part_Cell, [Parts,Cells]),*** encargado de la creación y dimensión de la matriz parte-celda de acuerdo a las constantes anteriores.

Por otra parte, el predicado ***Matrix_Part_Cell[1..Parts,1..Cells] :: 0..1,*** describe el dominio en donde fluctúan los valores de la matriz, que en este caso es cero y uno.

```
73
74  part_cell(Matrix_Part_Cell):-
75
76      Parts=12,
77      Cells=2,
78      Sum is 1,
79
80      dim(Matrix_Part_Cell, [Parts,Cells]),
81      Matrix_Part_Cell[1..Parts,1..Cells] :: 0..1,
82
83
84      ( for(I,1,Parts),param(Cells,Matrix_Part_Cell,Sum) do
85
86          sum(Matrix_Part_Cell[I,1..Cells]) #=  Sum
87
88      ),
89
90
91  labeling(Matrix_Part_Cell).
```

Figura 7.7 – Representación de la matriz parte-celda en ECL^{i}PS^e

7.1.2.3. Restricciones

La definición de las restricciones del MCDP en ECL^{i}PS^e y en cualquier otro *solver* de resolución merece un tratamiento especial, ya que es el eje medular de todo problema de que tenga relación con Programación con Restricciones. Por esto, se debe primeramente definir cierta sintaxis para razonar el tratamiento de restricciones en ECL^{i}PS^e. Esta sintaxis de representación se expone a continuación:

- “Menor que”, escrito como #<,
- “menor que o igual”, escrito como #=<,
- “igual”, escrito como #=,
- “no igual”, escrito como #\=,
- “Mayor que o igual”, escrito como #>=,
- “mayor que”, escrito como #>

Dada esta configuración, la matriz máquina-celda posee dos restricciones, las cuales son; ***"asegurar que una máquina esté asignada a una sola celda y que las máquinas asignadas"*** a una celda no sobrepasen al número máximo de máquinas por celda. Dado lo anterior, las restricciones son definidas y representadas de la siguiente manera:

$$\sum_{k=1}^{C} y_{ik} = 1 \qquad \forall i,$$

$$\sum_{i=1}^{C} y_{ik} \leq M_{max} \qquad \forall k.$$

Donde la primera de las restricciones es representada por la figura 7.8; la cual muestra a una constante local como ***Sum*** inicializado con el valor 1, propio de la restricción y también al predicado ***sum(Matrix_Machine_Cell[I,1..Cells]) #= Sum*** que asegura principalmente que la primera de las restricciones se cumpla a cabalidad.

La segunda de las restricciones que también es visualizada en la figura, es definida por una constante ***Mmax***, definida anteriormente en donde el predicado ***sum(Matrix_Machine_Cell[I,1..Cells]) #=< Mmax***, asegura el límite máximo de máquinas por una celda; en el caso de la figura este número es igual a 6 máquinas.

```
36
37  machine_cell(Matrix_Machine_Cell):-
38
39      Machines = 12,
40      Cells= 2,
41      Sum is 1,
42      Mmax is 6, %Mmax = Maquinas - 1
43
44
45      dim(Matrix_Machine_Cell, [Machines,Cells]),              % make the variables
46      Matrix_Machine_Cell[1..Machines, 1..Cells] :: 0..1,      % set the domains
47
48      ( for(I,1,Machines),param(Cells,Matrix_Machine_Cell,Sum) do
49
50          sum(Matrix_Machine_Cell[I,1..Cells]) #=  Sum
51
52      ),
53
54      ( for(J,1,Cells),param(Machines,Matrix_Machine_Cell,Mmax) do
55
56          sum(Matrix_Machine_Cell[1..Machines,J]) #=<  Mmax
57
58      ),
59
60  labeling(Matrix_Machine_Cell).
```

Figura 7.8 – Representación restricciones máquina-celda

La última de las restricciones es definida por el aseguramiento de una parte o una familia de partes pertenezca a una sola celda de producción. Esto se observa representado por el siguiente modelo matemático descrito en el capítulo 5.4:

$$\sum_{k=1}^{C} z_{jik} = 1 \quad \forall j,$$

Donde en la figura 7.9 visualiza a través de una sintaxis ECL^{i}PS^e el tratamiento de la restricción antes mencionada. El predicado ***sum(Matrix_Part_Cell[I,1..Cells]) #= Sum*** asegura el cumplimiento que una parte o una familia de estas sea asignada a una celda.

```
73
74  part_cell(Matrix_Part_Cell):-
75
76      Parts=12,
77      Cells=2,
78      Sum is 1,
79
80      dim(Matrix_Part_Cell, [Parts,Cells]),
81      Matrix_Part_Cell[1..Parts,1..Cells] :: 0..1,
82
83
84      ( for(I,1,Parts),param(Cells,Matrix_Part_Cell,Sum) do
85        ┌─────────────────────────────────────┐
86        │ sum(Matrix_Part_Cell[I,1..Cells]) #=  Sum      │
87        └─────────────────────────────────────┘
88      ),
89
90
91  labeling(Matrix_Part_Cell).
```

Figura 7.9 – Representación restricción parte-celda

7.1.2.4. Triple Sumatoria

Estas reglas representan a una de las secciones que aportan gran valor al modelo, ya que permiten a la función objetivo a cumplir su labor. Para el uso de ésta regla es necesario tener todas las matrices ya creadas y configuradas para tener los valores tentativos de las configuraciones que serán evaluadas en búsqueda de la configuración que preste el menor movimiento interceldario. En el caso de las matrices máquina-celda y parte-celda, éstas matrices son creadas en la ejecución de programa, pero la matriz máquina-parte, es entregada dentro del código fuente.

Luego de tener todas las matrices, éstas son llevadas a la triple sumatoria, para calcular el movimiento interceldario de las partes bajo esa configuración. Al satisfacer

todas las restricciones impuestas por el modelo se suma la cantidad de números "*1*" en las coordenadas *i,j,k* en las matrices, obteniendo como producto la cantidad que representa el movimiento interceldario de las partes dentro de las celdas configuradas. La siguiente figura sólo muestra la triple sumatoria que entrega una de las tantas configuraciones para encontrar la solución al MCDP, pero que no es suficiente para concluir con la tarea de encontrar la solución óptima al problema del MCDP.

```
(for(K,1,16) * for(I,1,16)* for(J,1,30),

    fromto(0,In,Out,TotalCost),
     param(Matrix_Machine_Part,Matrix_Machine_Cell,Matrix_Part_Cell) do

        A is Matrix_Machine_Part[I,J],
        Z is Matrix_Part_Cell[J,K],
        Y is Matrix_Machine_Cell[I,K],
        Out $= In + A*Z*(1-Y)

),
```

Figura 7.10 – Representación función objetivo MCDP

7.1.2.5. Función Objetivo

Estas reglas representan a la función objetivo del modelo, se sugiere visitar al capítulo 5.4 para visualizar el detalle de esta regla, que involucra a las tres matrices creadas por las reglas anteriores. En el caso de las matrices Máquina- Celda y Parte-Celda, éstas matrices son creadas en la ejecución de programa, pero la matriz máquina-parte, es entregada dentro del código fuente.

La regla "*minimize*" es la que se encarga de pedir y evaluar las distintas configuraciones posibles en búsqueda de la mejor. En éste caso se presentan las matrices máquina-parte, máquina-celda y parte-celda a las que se le aplicarán los predicados "*search"* para buscar posibles mejores soluciones dependiendo de la demanda del predicado *minimize*.

Luego al final de la regla se aprecia que existe una variable llamada "*TotalCost*", la cual almacena el número de movimientos interceldarios que producen cada una de las distintas configuraciones, por lo que, si el valor de *TotalCost* es igual al óptimo, entonces ha encontrado la configuración con menor número de movimientos interceldarios.

Minimizar: $$\sum_{k=1}^{C}\sum_{i=1}^{M}\sum_{j=1}^{P} a_{ij}z_{jk}(1-y_{ik})$$

La implementación de la función de minimización en ECL^{i}PS^e es la siguiente:

```
        A is Matrix_Machine_Part[I,J],
        Z is Matrix_Part_Cell[J,K],
        Y is Matrix_Machine_Cell[I,K],
        Out $= In + A*Z*(1-Y)

),


writeln(viajes:TotalCost),
minimize((labeling(Matrix_Machine_Part),labeling(Matrix_Machine_Cell),labeling(Matrix_Part_Cell)), TotalCost).
```

Figura 7.11 – Representación función objetivo MCDP en ECL^{i}PS^e

8. Modelado del MCDP en Choco

Como se explicitó en el capítulo 8, el modelado de un problema constituye un aspecto fundamental en la disciplina de la ingeniería informática. Un modelo de un sistema es básicamente una herramienta que permite responder interrogantes sobre este último sin tener que recurrir a la experimentación sobre el mismo. Este, siempre es una representación simplificada de la realidad, por lo que, utilizando este tipo de premisas y lógica de desarrollo se ha dispuesto de la reproducción de una nueva transferencia desde el análisis hipotético a una representación codificable, que a diferencia del capítulo 7, se ha desarrollado bajo una sintaxis y estructura del *solver* de programación con restricciones Choco.

8.1. Representación del MCDP en Choco

La representación del MCDP, al igual que en el capítulo 7, consiste principalmente en hacer una transferencia en términos metodológicos del modelamiento hipotético visto en el capítulo 5.4, en una representación codificable bajo términos y sintaxis de un *solver* en particular, para estos efectos el *solver* de programación con restricciones Choco. Todo esto, con el afán de suministrar al lector una nueva representación del MCDP pero con una plataforma de resolución diferente a la anterior.

8.1.1. Librerías Externas Choco

Choco utiliza las siguientes librerías externas presentes en el *solver*, las cuales se distribuyen dependiendo del propósito del problema, es decir, si se trata de un problema de optimización, variables de tipo entero o reales, como también librerías por defecto que indican el uso de sentencias para restricciones.

A continuación, la figura 8.1 ilustra la situación anterior:

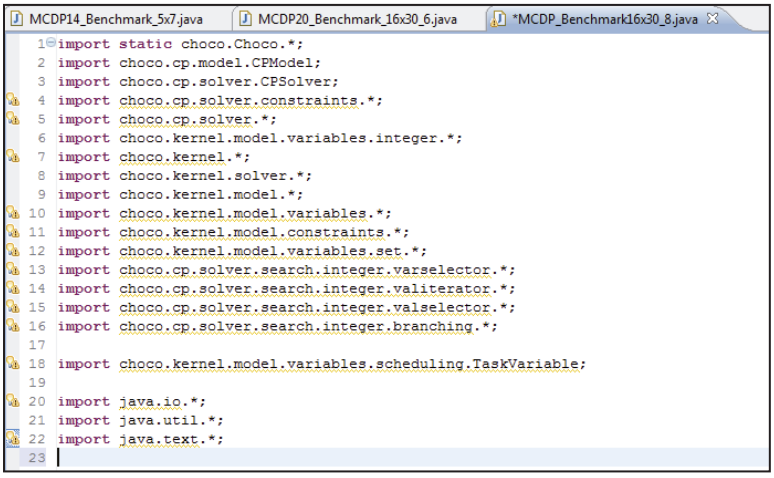


Figura 8.1 Librerías Externas Choco

8.1.2. Formalización del MCDP como un CSP en CHOCO

Anteriormente se ha observado el uso de las librerías externas de Choco, las cuales son primordiales para el correcto funcionamiento de un modelo de programación con restricciones. A continuación se precisará acerca del MCDP bajo una definición sintáctica y semántica de Choco, con el afán principalmente de entregar la estructura de solución del MCDP en Choco.

8.1.2.1. Constantes

- M, es el número de máquinas.
- P, el número de piezas (partes).
- C, el número de celdas.
- M_{max}, el número máximo de máquinas por celda.

Básicamente las constantes están presentes en el encabezado del modelo, donde estas últimas son utilizadas para denotar límites, en términos de dimensión, a las diferentes variables utilizadas en el modelo. Por ejemplo, en la variable ***matrix_machine_part***, que es

una variable de tipo entero con función de *Benchmark*, las dimensiones de esta variable matricial son definidas por medio de las constantes *máquinas y partes*. Ocurre la misma situación para las otras 2 variables matriciales del modelo, donde la cantidad de celdas y el número máximo de máquinas por celda, definen los límites de la variable matricial como también parámetros para las restricciones. La figuración de la definición de constantes es visualizada en la siguiente figura:

```
// constants of model

int maquinas=7; // invariable for the new model
int partes=11; // invariable for the new model
int suma=1; // invariable for the new model
int celdas=7;// invariable for the new model

int mmax=3;
```

Figura 8.2 Variables MCDP en Choco

8.1.2.2. Variables y Dominios

- A= [a_{ij}], matriz binaria de piezas-máquina de dimensiones *MxP*, con dominio [0,1]
  - *i*, el índice de máquinas, con dominio (*i*=1,…,M).
  - *j*, el índice de piezas, con dominio (*j*=1,…,P).
  - *k*, el índice de celdas, con dominio (*k*=1,…,C).

La figuración de la variable matricial está dada por la matriz ***matrix_machine_part*** en donde una matriz representa a las máquinas que fabrican una cierta parte de un producto. Esta matriz, que se dimensiona de acuerdo a la cantidad de máquinas y cantidad de partes de la matriz de data inicial, posee un dominio que está configurado por la matriz de data inicial, es decir, corresponde a un dominio entre cero y uno.

La ilustración de la situación anterior, se observa a continuación:

```
// matrix benchmark variable matrix_machine_part

int [][] matrix_machine_part = {{1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1},
                                {1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0},
                                {0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0},
                                {0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0},
                                {0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0},
                                {0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1},
                                {0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0}};
```

Figura 8.3 – Representación de la matriz máquina-parte en Choco

- B= [y_{ik}], matriz binaria máquina-celda de dimensiones *CxM* con dominio [0,1],

$$y_{ik} = \begin{cases} 1 & \text{si la máquina } i \in \text{ a la celda } k \\ 2 & \text{otros casos} \end{cases}$$

- C= [z_{ik}], matriz binaria pieza-celda de dimensiones *PxC* con dominio [0,1],

$$z_{jk} = \begin{cases} 1 & \text{si la pieza } j \in \text{ a la familia } k \\ 0 & \text{otros casos} \end{cases}$$

La representación y definición en Choco de estas dos variables matriciales pueden ser observadas de forma explícita en la figura 8.4, en donde, se distingue el objeto ***new IntegerVariable [maquinas][celdas],*** encargado de crear las matrices, en función, de las constantes detalladas anteriormente.

```
// start CP model

Model m = new CPModel();

// model CP variables

IntegerVariable [][] matrix_machine_cell = new IntegerVariable[maquinas][celdas];
IntegerVariable [][] matrix_part_cell = new IntegerVariable[partes][celdas];
```

Figura 8.4 – Representación Definición Variables Matriciales en Choco

También, se declaran los dominios en donde fluctuaran los valores de estas matrices, esto se define mediante un ciclo iterativo y la sentencia ***makeIntVar***, encargada de la definición de dominios para variables de tipo entero**,** donde los dominios de la matriz máquina-celda y la matriz parte-celda oscilan entre 0 y 1.

La ilustración de lo anterior se presenta en las siguientes figuras:

```
// define domain matrix variable matrix_machine_cell

for(int i = 0; i < maquinas; i++) {
        for(int j = 0; j < celdas; j++) {
            matrix_machine_cell[i][j]= makeIntVar("x_" + i + "_" + j, 0, 1 );
        }
}
```

Figura 8.5 – Representación Dominios Variable Matricial Máquina-Celda

```
//define domain matrix_part_cell

for(int i = 0; i < partes; i++) {
    for(int j = 0; j < celdas; j++) {
        matrix_part_cell[i][j]= makeIntVar("x_" + i + "_" + j, 0, 1 );
    }
}
```

Figura 8.6 – Representación Dominios Variable Matricial Parte-Celda

8.1.2.3. Restricciones

La definición de las restricciones del MCDP en Choco y en cualquier otro *solver* de resolución merece un tratamiento especial, ya que es el eje medular de todo problema de satisfacción de restricciones. Por esto, se debe primeramente definir cierta sintaxis para razonar el tratamiento de restricciones en Choco. Esta sintaxis de representación se expone a continuación:

- "Menor que", escrito como #<,
- "menor que o igual", escrito como #=<,
- "igual", escrito como #=,
- "no igual", escrito como #\=,
- "Mayor que o igual", escrito como #>=,
- "mayor que", escrito como #>

Dada esta configuración, la matriz máquina-celda posee dos restricciones, las cuales son; asegurar que una máquina esté asignada a una sola celda y que las máquinas asignadas a una celda no sobrepasen al número máximo de máquinas por celda. Dado lo anterior, las restricciones son definidas y representadas de la siguiente manera:

$$\sum_{k=1}^{C} y_{ik} = 1 \qquad \forall i,$$

$$\sum_{i=1}^{C} y_{ik} \leq M_{max} \qquad \forall k.$$

Donde la primera y la segunda de las restricciones es representada por la figura 8.7; la cual muestra a un operador como ***Sum*** , encargado de asegurar que la primera de las restricciones de la variable matricial ***matrix_machine_cell*** se cumpla a cabalidad.

La segunda de las restricciones que también es visualizada en la figura 8.7, es definida por una constante ***Mmax***, en donde se crea un nuevo objeto de tipo columna para asegurar la correcta definición de esta restricción. Este objeto se crea básicamente por la no existencia de una sentencia para la suma de columnas en Choco, he aquí la explicación del

uso de tal objeto que represente a una columna de la matriz, para poder posteriormente, sumar y asegurar que la cantidad máxima de máquinas por celda no sea excedida.

```
// first constraint: matrix_machine_cell sum rows = 1

for(int i=0;i<maquinas;i++){

    m.addConstraint(eq(sum(matrix_machine_cell[i]), 1));

}

// second constraint: max machines number per cell at matrix_machine_cell

 for(int i = 0; i < celdas; i++) {
        ArrayList<IntegerVariable> col = new ArrayList<IntegerVariable>();
        int j=0;
        //System.out.println("j: " +  j);
        for(j = 0; j < maquinas; j++) {
            //System.out.println("j: " +  j);

            col.add(matrix_machine_cell[j][i]);
        }
        m.addConstraint(leq(sum(col.toArray(new IntegerVariable[1])), mmax));
 }
```

Figura 8.7 – Representación Restricción Variable Matricial Máquina-Celda

La última de las restricciones es definida por el aseguramiento de que una parte o una familia de partes pertenezcan a una sola celda de manufactura. Esto se observa representado por el siguiente modelo matemático descrito en el capítulo 5.4:

$$\sum_{k=1}^{C} z_{jik} = 1 \quad \forall j,$$

Donde, la figura 8.8 visualiza a través de una sintaxis de Choco el tratamiento de la restricción antes mencionada. La sentencia ***Sum*** asegura el cumplimiento de que una parte o una familia de estas sean asignadas a solo una celda.

```
// third constraint: sum rows matrix_part_cell = 1

for(int i=0;i<partes;i++){

        m.addConstraint(eq(sum(matrix_part_cell[i]), suma));

}
```

Figura 8.8 – Representación Restricción Variable Matricial Parte-Celda

8.1.2.4. Función Objetivo

En esta sección lo primero que se hace es utilizar una variable auxiliar para almacenar los distintos valores de las soluciones encontradas por el modelo, es así como se define una variable de tipo *array* para almacenar los valores de la función objetivo. La ilustración de la situación anterior es visualizada por la siguiente figura:

```
/* objective function */


//array for save the solutions

IntegerVariable z = makeIntVar("z", 0, 10000);
IntegerVariable[] z_array_1 = new IntegerVariable[maquinas*maquinas*partes];

// define domain for auxiliary array in order to apply the sum

for(int i = 0; i < maquinas*maquinas*partes; i++) {

       z_array_1[i] = makeIntVar("z_array_1" + i, 0, 1);

}
```

Figura 8.9 – Representación Variable Array Auxiliar

Luego de tener todas las matrices instanciadas, éstas son llevadas a la triple sumatoria, para calcular el movimiento interceldario de las partes bajo esa configuración. Al satisfacer todas las restricciones impuestas por el modelo se suma la cantidad de números "*1*" en las coordenadas *i,j,k* en las matrices, obteniendo como producto la cantidad que representa el movimiento interceldario de las partes dentro de las celdas configuradas. La siguiente figura solo muestra la triple sumatoria que entrega una de las tantas configuraciones para encontrar la solución al MCDP, pero que no es suficiente para concluir con la tarea de encontrar la solución óptima al problema del MCDP. Las figuras 8.10, 8.11 y 8.12 grafican los procedimientos de búsqueda de soluciones y la selección de la mejor de entre estas soluciones al MCDP:

Minimizar: $$\sum_{k=1}^{C}\sum_{i=1}^{M}\sum_{j=1}^{P} a_{ij} z_{jk}(1 - y_{ik})$$

```
    // start objective function

int contador=0;

for(int k=0;k<celdas;k++){

  for(int i=0;i<maquinas;i++){

    for(int j=0;j<partes;j++){

        //z_array_1[contador] = makeIntVar("z_" + i +"_" + k, 0, 1);

        m.addConstraint(eq(mult(mult(matrix_machine_part[i][j],matrix_part_cell[j][k])
        ,minus(1,matrix_machine_cell[i][k])),z_array_1[contador]));

        contador= contador+1;


    }

  }

}

m.addConstraint(eq(sum(z_array_1), z));
```

Figura 8.10 – Representación Triple Sumatoria MCDP

La siguiente figura ilustra el hecho de solución y lectura del MCDP bajo el *solver* Choco, el cual es primordial para entregar valores íntegros a la sentencia *minimize* de optimización.

```
m.addConstraint(eq(sum(z_array_1), z));


/*Our solver */

Solver s = new CPSolver();


/* Read the Model */

s.read(m);

/* Statistics of Model */

s.monitorTimeLimit(true);
s.monitorBackTrackLimit(true);
s.monitorNodeLimit(true);
s.monitorFailLimit(true);

/* We do Statistics for the Solution */

CPSolver.setVerbosity(CPSolver.SOLUTION);
```

Figura 8.11 – Representación Lectura del MCDP en Choco

```
/* solve the problem */
s.minimize(s.getVar(z), true); // minimize

/* solve at least one solution*/

//s.solve();

s.printRuntimeStatistics();
System.out.println("Soluciones  Encontradas: " + s.getNbSolutions());
//System.out.println(s.pretty());



if (s.isFeasible()) {
    //System.out.println(s.pretty());
    System.out.println("Cost Exceptional Numbers: " + s.getVar(z).getVal());


}  else {
    System.out.println("No solution.");
}
```

Figura 8.12 – Representación Búsqueda Mejor Solución en Choco

Dada la figura anterior, la sentencia más importante de mencionar es la sentencia "*minimize*", la que se encarga de pedir y evaluar las distintas configuraciones posibles en la búsqueda de la mejor solución. En éste caso, se presentan los valores de las matrices máquina-parte, máquina-celda y parte-celda anteriormente guardados en el *array* auxiliar, donde se buscaran posibles soluciones y de estas el mejor resultado dependiendo de la demanda de la sentencia *minimize* con el objetivo de encontrar el menor número de movimientos interceldarios.

9. Experimentación del MCDP

Los experimentos computacionales forman parte de un área de investigación reciente y determinante. Las aplicaciones de tales simulaciones son muy amplias: desde diseño en ingeniería hasta experimentos de simulación de cambio climático. Usualmente, el interés del experimentador se dirige hacia la exploración eficiente de la región de datos más significativa, para luego tener la capacidad de construir en una segunda etapa, un modelo deductivo, y visualizar los resultados obtenidos con el objeto de analizar y deducir comportamientos y tendencias.

Principalmente en esta sección del documento, se definen y muestran los *benchmarks* utilizados en los experimentos con los *solvers* ECL^{i}PS^e y Choco, las configuraciones de experimentación, los resultados obtenidos y finalmente, el análisis de estos últimos en función de la racionalización de conclusiones.

9.1. Concepto de Benchmark

La elección de las condiciones bajo la cual dos o más sistemas distintos pueden compararse entre sí, es especialmente ardua, y la publicación de los resultados toma cierta validez cuando se establecen estructuras de comparación similares y congruentes. Es por esto que un buen *benchmark*, se puede considerar como un proceso útil de cara a lograr el impulso necesario para realizar mejoras y cambios.

Si se toma en cuenta la definición anterior, es sumamente importante declarar buenas estructuras de comparación, con lo cual se ha definido y trabajado a base de 10 matrices validadas como entrada de datos objetivos para realizar la experimentación del modelo MCDP en ECL^{i}PS^e y Choco por parte de F. Boctor [1]. Además, se estableció la utilización de otros *benchmarks*, también validados, para visualizar tendencias y realizar análisis deductivo ágil.

Ahora bien, en cuanto a la congruencia con las definiciones de carácter industrial-mecánico del modelo, se puede decir que se ha utilizado como matriz de datos, la matriz binaria máquina-parte, debido a que esta última entrega la relación de producción de una máquina con una determinada parte. La tabla 9.1 ejemplifica a una matriz utilizada en la experimentación.

Tabla 9.1 - Ejemplo matriz experimentación modelo ECL^{i}PS^e y Choco

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1					1			1		1		1								1		1	1			1			1	
2		1			1				1								1			1	1	1			1	1	1			
3											1	1	1		1			1				1								
4			1				1						1	1								1	1							1
5			1	1							1	1	1		1	1						1								
6										1											1		1			1	1	1	1	1
7							1				1			1	1	1														
8	1					1					1					1		1												
9		1							1	1		1										1		1	1			1		1
10	1											1			1	1														
11								1	1			1				1								1				1		1
12	1		1	1			1				1		1		1	1		1			1									
13									1	1					1		1						1		1	1	1		1	
14								1												1			1		1					
15	1		1	1			1				1		1	1		1			1											
16								1									1			1			1	1	1	1	1	1	1	1

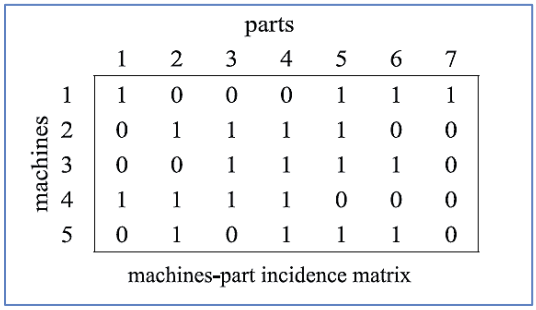
Las filas de la matriz anterior representan a la cantidad de máquinas utilizadas en una industria en el proceso de fabricación, las columnas representan a la cantidad de partes asociadas a un producto de fabricación y por último la cantidad de números 1 representa si efectivamente una máquina procesa a una parte o pieza en particular. Es decir, si a_{ij} = 1, la máquina 1 si procesa a la maquina j.

9.2. Benchmarks utilizados

Si se consideran las definiciones anteriores acerca de la conceptualización de *benchmarks*, esta no es más que una matriz de datos máquina-parte estructurada por un número de máquinas simbolizadas en las filas de una matriz y también por un número de partes simbolizadas en las columnas de una matriz en particular.

Según lo anterior, la experimentación del MCDP tanto en ECL^{i}PS^e como en Choco considera la utilización de 2 *benchmarks* de datos. El primero de ellos fruto de la investigación y posterior publicación de Xiangyong Li, M.F.Baki e Y.P.Aneja [31], este último compuesto por 5 máquinas y 7 partes para un proceso de manufactura celular.

La ilustración de la situación anterior, puede ser observada en la siguiente figura:



	1	2	3	4	5	6	7
1	1	0	0	0	1	1	1
2	0	1	1	1	1	0	0
3	0	0	1	1	1	1	0
4	1	1	1	1	0	0	0
5	0	1	0	1	1	1	0

Figura 9.1 – Ejemplo Matriz Benchmark 5x7

Fayes Boctor [54], investigó profundamente el problema de la formación de celdas para una planta de manufactura (5) y consecuencia de ese trabajo, propuso a través de métodos meta-heurísticos como SA un modelo que minimiza la cantidad de elementos excepcionales. Pero además, propuso junto a ese modelo, un estándar de 10 *benchmarks* para futuras comparaciones de técnicas de optimización. Estos, consideran la utilización de 16 máquinas y 30 partes para un problema de diseño de celdas de manufactura. Un ejemplo de *benchmark* propuesto por F. Boctor puede ser observado en la siguiente tabla:

Tabla 9.2 - Ejemplo matriz benchmark 16x30

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1					1			1		1		1								1		1	1			1			1	
2		1			1				1								1			1	1	1			1	1	1			
3											1	1	1		1			1				1								
4			1				1						1	1								1	1							1
5			1	1							1	1	1		1	1						1								
6										1											1		1			1	1	1	1	1
7							1				1			1	1	1														
8	1					1					1					1		1												
9		1							1	1		1										1		1	1			1		1
10	1											1			1	1														
11								1	1			1				1								1				1		1
12	1		1	1			1				1		1		1	1		1			1									
13									1	1					1		1						1		1	1	1		1	
14								1												1			1		1					
15	1		1	1			1				1		1	1		1			1											
16								1									1			1			1	1	1	1	1	1	1	1

9.3. Ambiente de Experimentación

El objetivo de cualquier experimento dentro de un modelo de investigación computacional, es ser el punto de decisión para que el investigador, pueda visualizar de manera objetiva, el comportamiento y tendencia de los resultados obtenidos. Para esto, es primordial diseñar el conjunto de casos y escenarios de experimentación que permitirán validar la completitud y la correcta implementación del modelo computacional con el que se está experimentando.

En el caso del MCDP, la experimentación enfatiza la evaluación de las funcionalidades requeridas y su adecuación al problema, las restricciones satisfechas, por tratarse de un modelo basado en programación con restricciones, y por último la ejecución de las funcionalidades según lo esperado.

9.3.1. Software de Experimentación y de Configuración de Parámetros

Se utilizaron diferentes programas para el desarrollo de la solución y posterior *testing* al MCDP. Entre los que destacan TkEclipse, Notepad++, Choco entre otros. A continuación se detallan las principales características de cada uno de ellos:

- **TkEclipse**: Es el compilador del código generado en un editor. Se utilizó por ser libre, de manejo más accesible y gratuito. Utilizando versión 6.0 #159 (i386_nt). Descargada directamente del sitio web de los desarrolladores [26].

- **Notepad++**: Se utilizó este editor de texto por su distribución libre y bajo licencia GNU. Es un editor que ayuda al desarrollador, destacando distintos colores o tipografías, sentencias, valores numéricos propios del lenguaje Prolog, lo que ayuda a mejorar la comprensión del código fuente.

- **Eclipse Galileo**: Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados (del inglés IDE), como el IDE de Java llamado *Java Development Toolkit* (JDT).

- **Choco Solver:** Choco es una librería de Java para programación con restricciones como también para la satisfacción y/u optimización de problemas complejos típicamente combinatoriales.

9.3.2. Parámetros de Experimentación

Como se explicitó anteriormente, el parámetro más significativo de inicio para los experimentos es el *benchmark*, pero también existen y se deben configurar otros parámetros de igual importancia en el modelo, la visualización de tales parámetros de muestra en la siguiente tabla:

Tabla 9.3 – Ejemplo Parámetros de Experimentación

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solutions Founds	Exceptional Numbers
1	16	2	30	8	392,66	104045	13	5
2	16	2	30	9	879,59	308556	29	4
3	16	2	30	10	4315,70	1521187	11	4
4	16	2	30	11	886,00	245120	7	3
5	16	2	30	12	748,32	209618	6	1

Donde, la definición de cada uno de los parámetros de la tabla anterior es la siguiente:

- **Config**: Representa el número de configuración de la prueba.
- **Machines**: Representa a la cantidad de máquinas de un proceso de producción en una industria. En el modelo, son las filas de la matriz máquina-parte.
- **Cells**: Representa a la cantidad de celdas mecánicas introducidas en el modelo.
- **Parts**: Representa a la cantidad de partes en un proceso de producción. En el modelo, simboliza a las columnas de la matriz máquina-parte.
- **Time**: Representa la cantidad de tiempo que el *solver* demora en entregar una o ninguna solución al modelo. Este se mide en segundos y tiene un *timeout* de 1 hora de procesamiento.
- **Backtracks (BT)**: Representa la cantidad de veces que el *solver* retrocede hacia instancias que conduzcan a una solución o en el caso de optimización, a una mejor solución.
- **Mmax**: Representa a la cantidad máxima de máquinas que se pueden introducir a una celda mecánica, siempre debe ser menor a la cantidad de máquinas utilizadas en un proceso productivo.
- **Exceptional Numbers (EN)**: Representa a la variable resultado en el modelo, bajo términos industriales y reales de una empresa de manufactura. Representa a la cantidad de viajes de una determinada parte por las distintas celdas mecánicas; cantidad de viajes que se quiere minimizar con la implementación del MCDP bajo programación con restricciones.

9.4. Experimentos Computacionales del MCDP en ECL^{i}PS^e

La implementación de las pruebas en el *solver* ECL^{i}PS^e se confeccionaron luego de haber realizado el modelo inicial y la mejora del algoritmo gradualmente. Sin embargo esta implementación no estuvo exenta de problemas evidentes en los tiempos de resolución en un principio, entregando resultados al cabo de horas de procesamiento y búsqueda por parte del *solver*.

Para direccionar los tiempos de búsqueda y analizar previamente la mejor estrategia de resolución, se realizó experimentación previa al *benchmark* de tamaño 16x30. Esto ultimo con el objeto de primero, trabajar utilizando las heurísticas por defecto y luego con estrategias de búsqueda compuestas por diferentes heurísticas para realizar pruebas y determinar la incidencia de las estrategias de resolución en cada uno de los experimentos. Esto se realizó, con la utilización de *benchmarks* pequeños debidamente validados en función, de beneficiar el empleo de las mejoras estrategias de resolución para los experimentos en el *benchmark* de 16 máquinas por 30 partes; esto último, con el propósito de realizar análisis comparativo en términos del número de viajes interceldarios encontrados frente a otras técnicas de optimización.

A raíz de esta investigación previa se puede determinar que ECL^{i}PS^e presenta grandes funcionalidades en la implementación de reglas o enunciados *built-in* de distintas heurísticas y estrategias, con capacidad de soportar matrices sin problemas y es ampliamente parametrizable para realizar pruebas versátiles y que marcan diferencias al momento de elegir la estrategia que tiene mayor y mejor desempeño en la resolución del MCDP.

Sin dejar de lado que siempre se utilizó el mismo compilador de ECL^{i}PS^e y versión anteriormente descrita en el capítulo 7.2.1.2 con el fin de homologar en todo momento las pruebas en ECL^{i}PS^e y determinar cuál o cuáles son los factores que inciden en los tiempos de resolución del problema analizado.

9.4.1. Experimentación Benchmark 5x7

Las pruebas en una primera etapa fueron realizadas con un *benchmark* pequeño de 5 máquinas por 7 partes a manufacturar. Con estas pruebas se logró determinar cuál o cuáles eran las tendencias a encontrar soluciones más rápidamente y comparar las estrategias de resolución desde temprano para tener antes de la realización de las pruebas con los *benchmarks* de 16 x30 proporcionados por F. Boctor, un acercamiento a las heurísticas de selección de valor y variable de mejor desempeño.

Los *benchmarks* que se utilizaron en una primera etapa se extrajeron de la investigación de Xiangyong Li, M.F.Baki e Y.P.Aneja [31] que fue un ejemplo usado para explicar el método de agrupación de máquinas. Este *benchmark* cuenta con una matriz de dimensiones 5 x 7 lo que hace que dichas pruebas con este modelo sean muy breves y permitan visualizar ciertas tendencias en el tiempo y cantidad de *backtracks* utilizados antes de llegar a la solución óptima.

Al momento de puntualizar el uso de heurísticas de selección de variables y heurísticas de selección de valor, las heurísticas que se utilizaron para estos casos fueron, *Input Order*, *First Fail*, *Anti First Fail*, *Most Constrained* y *Max Regret*. Dichas heurísticas de selección de variables fueron combinadas con heurísticas de selección de valor como; *MinVal*, *Middle* y *MaxVal*, que representan el mínimo valor, valor medio, y máximo valor respectivamente.

Para el análisis de los resultados se confeccionó un registro de tiempos de cómputo, cantidad de *backtracks* y números excepcionales encontrados. Con este registro se compararán los resultados al aplicar distintas estrategias de búsqueda. Las configuraciones para las pruebas con el *benchmark* 5x7 fueron las siguientes:

- Heurísticas de selección de valores probadas:
  - ✓ Valor mínimo (MIN)
  - ✓ Valor medio (MIDDLE)
  - ✓ Valor máximo (MAX)

- Heurísticas de selección de variables probadas:
  - ✓ Most Constrained (MC)
  - ✓ First Fail (FF)
  - ✓ Anti First Fail (AFF)
  - ✓ Max Regret (MR)
  - ✓ Input Order (IO)

- Cantidad máxima de máquinas por celda:
  - ✓ 4 máquinas por celda
  - ✓ 3 máquinas por celda
  - ✓ 2 máquinas por celda

- Cantidad de celdas:
  - ✓ 5 celdas
  - ✓ 3 celdas
  - ✓ 2 celdas

La ejecución de las pruebas arrojó resultados determinantes para etapas posteriores de análisis, identificando, heurísticas de pésimo desempeño para este problema, junto con combinaciones de heurísticas que socaban tiempos razonables de cómputo para la implementación de futuras pruebas y estrategias de resolución que pueden llevar a un desempeño sobresaliente al momento de usar estas en *benchmarks* de mayor tamaño y complejidad.

Cada una de las 5 pruebas de *benchmarks* testeados, constó de 9 configuraciones, combinando en una primera etapa la cantidad de celdas que se permitirían crear, con la cantidad máxima de máquinas por parte. Cada una de estas configuraciones fue realizada para cada heurística de selección de variables y una para cada heurística de selección de valor, ya sea de valor mínimo, valor medio o valor máximo. En total se ejecutaron 135 experimentos, todos con diferentes heurísticas y/o distintas restricciones.

A continuación se presentarán las tablas que muestran los tiempos de cómputo, cantidad de *backtracks* y cantidad de números excepcionales obtenidos de la ejecución de las pruebas.

Tabla 9.4 – Benchmark 5x7 – Most Constrained en ECL^{i}PS^e

					MC/MIN			MC/MID			MC/MAX		
Config	**Machines**	**Cells**	**Parts**	**Mmax**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**
1	5	5	7	4	1041,55	0	3	1051,28	0	3	1033,3	0	3
2	5	3	7	4	14	0	3	13,94	0	3	13,83	0	3
3	5	2	7	4	0,36	0	3	0,39	0	3	0,38	0	3
4	5	5	7	3	1586,95	30959	5	1581,19	30959	5	1588,41	33167	5
5	5	3	7	3	17,55	1447	5	17,14	1447	5	17,41	1548	5
6	5	2	7	3	0,36	20	5	0,39	20	5	0,38	19	5
7	5	5	7	2	2464,41	37444	8	2481,14	3744	8	2507,5	40739	8
8	5	3	7	2	11,59	624	8	11,56	624	8	11,55	795	8
9	5	2	7	2	0,05	-	-	0,05	-	-	0,05	-	-

La tabla 9.4 presenta los resultados de la heurística de selección de variables *Most Constrained*. Los resultados muestran una similitud muy estrecha en los tiempos de búsqueda (Time) y la cantidad de *backtracks* (BT). En general los resultados obtenidos por esta heurística son de mal rendimiento, ya que mantiene tiempos relativamente altos en comparación a heurísticas como *First Fail* o *Input Order*.

A pesar de la gran cantidad de tiempo que se consume en hallar soluciones posibles, una observación importante a destacar es la similitud en los resultados obtenidos por las heurísticas de valor mínimo (MC/MIN) y valor medio (MC/MID). Dichas heurísticas de selección de valor arrojan resultados casi iguales en todas las pruebas que se presentan en esta tabla y en las tablas siguientes. Esto se puede explicar debido a que el problema presentado consta de un dominio finito de 0 y 1, por lo que la existencia de una valor medio es imposible. Por esta razón en las siguientes tablas no se presentarán los resultados de la heurística de valor medio y únicamente se compararán y analizarán las heurísticas de valor mínimo (MIN) y las de valor máximo (MAX).

Tabla 9.5 – Benchmark 5x7 – First Fail en ECL^{i}PS^e

					FF/MIN			FF/MAX		
Config	**Machines**	**Cells**	**Parts**	**Mmax**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**
1	5	5	7	4	218,09	4810	3	222,69	5347	3
2	5	3	7	4	4,41	483	3	4,45	521	3
3	5	2	7	4	0,27	88	3	0,25	88	3
4	5	5	7	3	582,58	21420	5	585,88	21417	5
5	5	3	7	3	9,67	1393	5	9,41	1397	5
6	5	2	7	3	0,42	183	5	0,42	183	5
7	5	5	7	2	2087,03	3302	8	2098,97	3233	8
8	5	3	7	2	21,13	304	8	21,2	289	8
9	5	2	7	2	0,72	-	-	0,7	-	-

Nuevamente en la tabla 9.5 los resultados entre las heurísticas de selección de valor son muy similares, pero debido al cambio de heurística de selección de variable a *First Fail* los tiempos de cómputo para encontrar las soluciones óptimas disminuyeron considerablemente.

Al parecer para estos problemas la heurística *First Fail*, presenta excelentes resultados y bastante consistentes. Lo que aporta gran valor a la investigación, considerando que el tipo de problemas que se desean analizar son de la misma naturaleza pero con muchas más variables y problemas a resolver. En general la aplicación de esta heurística hizo bastante simple la resolución del problema, considerando que en todos los casos el Número de Elementos Excepcionales, en inglés *Exceptional Numbers* (EN), son los mismos que con la heurística *Most Constrained*, siendo así una heurística eficiente y sobre todo y más importante una heurística eficaz.

Tabla 9.6 – Benchmark 5x7 – Anti First Fail en ECL^{i}PS^e

					AFF/MIN			AFF/MAX		
Config	**Machines**	**Cells**	**Parts**	**Mmax**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**
1	5	5	7	4	217,58	4810	3	220,3	5347	3
2	5	3	7	4	4,44	483	3	4,39	521	3
3	5	2	7	4	0,25	88	3	0,26	88	3
4	5	5	7	3	589,52	21420	5	585,22	21417	5
5	5	3	7	3	9,69	1393	5	9,63	1397	5
6	5	2	7	3	0,44	183	5	0,42	183	5
7	5	5	7	2	2164,16	3302	8	2081,47	3233	8
8	5	3	7	2	21,86	304	8	20,63	289	8
9	5	2	7	2	0,73	-	-	0,69	-	-

Para las pruebas realizadas con la heurística *Anti First Fail*, prácticamente todos los resultados obtenidos son similares a los resultados que entregó la heurística de selección de variable *First Fail*, esto debido a la naturaleza de selección de las variables, por lo que resulta un método interesante para poder aplicar en los próximos experimentos, debido a la rapidez en hallar los óptimos esperados.

Tabla 9.7 – Benchmark 5x7 – Max Regret en ECL^{i}PS^e

					MR/MIN			MR/MAX		
Config	**Machines**	**Cells**	**Parts**	**Mmax**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**
1	5	5	7	4	223,44	4810	3	225,34	5347	3
2	5	3	7	4	4,59	483	3	4,59	521	3
3	5	2	7	4	0,27	88	3	0,27	88	3
4	5	5	7	3	602,16	21420	5	605,55	21417	5
5	5	3	7	3	10,03	1393	5	9,83	1397	5
6	5	2	7	3	0,44	183	5	0,44	183	5
7	5	5	7	2	2160,69	3302	8	2173,16	3233	8
8	5	3	7	2	21,67	304	8	21,16	289	8
9	5	2	7	2	0,72	-	-	0,72	-	-

Para la heurística *Max Regret* los resultados fueron bastante favorable aunque levemente mayores en cuanto a la cantidad de tiempos utilizado en resolver el problema (variable "*Time*" en la tabla). Con esto se concluye que esta heurística es un buen método de resolución pero su desempeño está generalmente por debajo de otras heurísticas como *First Fail* o *Anti First Fail*.

Tabla 9.8 – Benchmark 5x7 – Input Order en ECL^{i}PS^e

					OI/MIN			IO/MID		
Config	**Machines**	**Cells**	**Parts**	**Mmax**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**
1	5	5	7	4	216,5	4810	3	217,47	5347	3
2	5	3	7	4	4,36	483	3	4,42	521	3
3	5	2	7	4	0,25	88	3	0,27	88	3
4	5	5	7	3	581,84	21420	5	582,09	21417	5
5	5	3	7	3	9,83	1393	5	9,53	1397	5
6	5	2	7	3	0,42	183	5	0,42	183	5
7	5	5	7	2	2080,08	3302	8	2081	3233	8
8	5	3	7	2	21,23	304	8	21,08	289	8
9	5	2	7	2	0,73	-	-	0,75	-	-

La última heurística presentada es la heurística *Input Order* que presenta los mejores resultados ponderados de todas las pruebas que se realizaron con el *benchmark* 5 x 7. Aún, cuando los números son levemente inferiores en cantidad de tiempo utilizado en resolver el problema (Time), estas pequeñas diferencias pueden convertirse en tiempo considerablemente valioso en las pruebas con *benchmarks* de dimensiones 16 x 30.

Los resultados presentados anteriormente muestran las diferencias al utilizar las distintas estrategias de resolución. Cada heurística presenta comportamientos peculiares y hasta similitudes que a pesar de ser distintos métodos de propagación de las restricciones, los tiempos y la cantidad de *backtracks* utilizados en realizar las pruebas son los mismos o existen distintas heurísticas que presentan comportamientos, tiempos de cómputo, rapidez de convergencia de los resultados, todos de manera distinta.

En términos de gráfica cuantitativa a continuación se presenta un gráfico que entrega información acerca del tiempo de cómputo utilizado, este último presentando la heterogeneidad de algunas estrategias en términos de eficiencia de tiempos de computación

.

No se consideró para este análisis la cantidad de números excepcionales ya que todas las configuraciones llegan a la misma cantidad de números excepcionales, por lo que no hay problema ni tiene sentido comparar este parámetro, pero si es importante considerar la cantidad de tiempo utilizado en encontrar este óptimo que fue necesario para la búsqueda. Dichos parámetros son considerados y comentados a continuación de cada gráfico.

Básicamente el grafico mide la cantidad de tiempo utilizado para el proceso de búsqueda de soluciones. Por la selección del gráfico, claramente se puede ver que existen heurísticas, que dependiendo de la configuración, tienen mejor desempeño que otras; por ejemplo se puede observar el caso de *Input Order*, que posee un comportamiento tendencioso como la mejor de las estrategias de búsqueda para una gran cantidad de configuraciones.

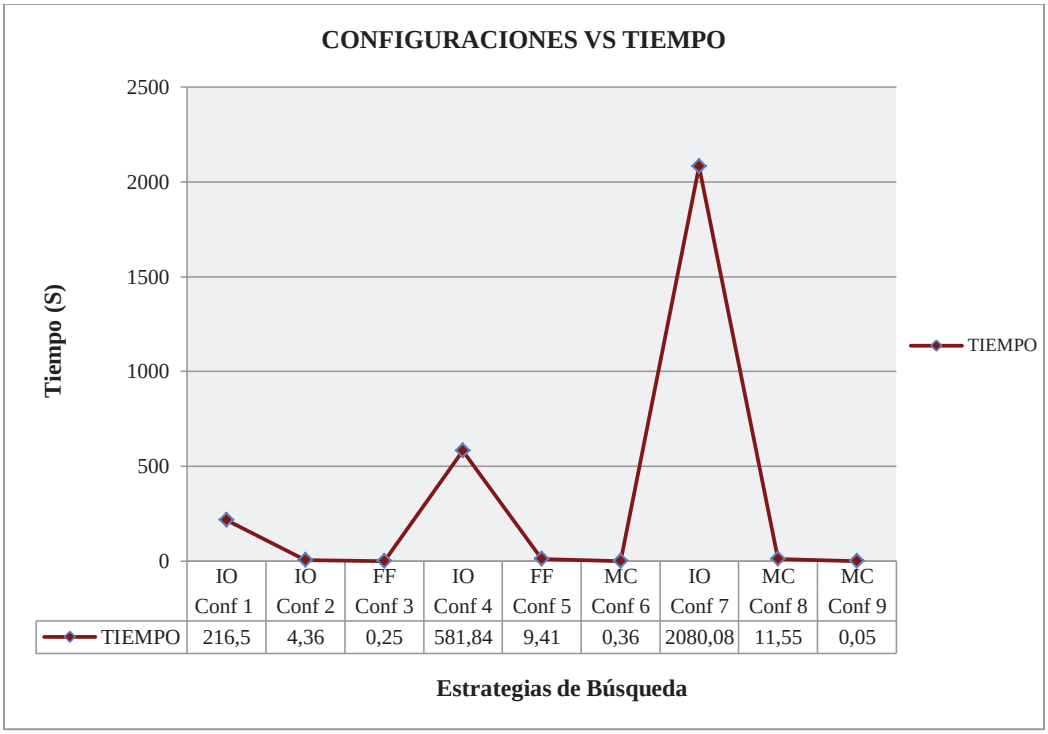


Figura 9.2 – Gráfico Mejores Soluciones por Configuración *Benchmark* 5x7 en ECL^{i}PS^e

9.4.2. Experimentación Benchmark 16x30

Esta matriz de datos máquina-parte, está estructurada por un número de 16 máquinas simbolizadas en las filas de la matriz y también por un número de 30 partes, simbolizadas en las columnas de la matriz.

Para realizar un análisis comparativo con otras técnicas de optimización del tipo incompletas, para este caso como lo es *Simulated Annealing* (SA), utilizada por Boctor, se hace necesario seleccionar la mejor estrategia de búsqueda operada en ECL^{i}PS^{e} como resultado de la experimentación preliminar con *benchmarks* más pequeños (5x7) y así generar más rápidamente, resultados; en función de la comparación de la cantidad de viajes interceldarios para este *benchmark* en particular.

Las pruebas con el *benchmark* de F. Boctor, son pruebas que entregarán gran cantidad de información para la resolución de todo tipo de problemas de manufactura de celdas de diseño, ya que por la naturaleza similar de todos los problemas presenta características interesantes a estudiar, además de ser un canal de comparación con distintas técnicas de optimización que han trabajado con esta matriz *benchmark.*

Para las pruebas con este *benchmark* se tienen las siguientes características del problema:

- Heurísticas de selección de valores probadas:
  - ✓ Heurística por Defecto (*MinDomain*)

- Heurísticas de selección de variables probadas:
  - ✓ Input Order (IO)

- Cantidad máxima de máquinas por celda:
  - ✓ 8 máquinas por celda
  - ✓ 9 máquinas por celda
  - ✓ 10 máquinas por celda
  - ✓ 11 máquinas por celda
  - ✓ 12 máquinas por celda

- Cantidad de celdas:
  - ✓ 2 celdas
  - ✓ 3 celdas

La culminación final del trabajo realizado se representa mediante la implementación de la problemática del MCDP en ECL^{i}PS^e. Según esto, los resultados se presentan en tablas para la posterior comparación de resultados, tomando en cuenta la cantidad de *backtracks*, el tiempo de cómputo y la cantidad de elementos excepcionales, siendo esta última finalmente la variable más importante del modelo por ser la variable resultado del MCDP .

Los resultados de las pruebas con los 10 *benchmarks* propuestos por Boctor tienen un desarrollo y comportamiento particular. La implementación del código en ECL^{i}PS^e, se mejoró gracias a la cooperación de un investigador "fanático", llamado Hakan Kjellerstrand, el cual posee gran conocimiento y es un investigador dedicado a la programación, y especialmente a la resolución de problemas mediante programación con restricciones. Su aporte fue valioso en muchas etapas de este desarrollo, descubriendo algunos errores cometidos por autores anteriores que resuelven el problema del diseño de celdas de manufactura y apoyando en la implementación y optimización de los modelos en ECL^{i}PS^e.

Lamentablemente, luego de aplicar algunas de las mejoras realizadas por Hakan, se encontró un problema en la implementación de las pruebas, que tiene su raíz en la optimización del diseño implementado para la resolución del problema del MCDP. Junto con esta mejora al algoritmo de resolución existen restricciones que evitan que se sobre evalúen datos más de una vez. Esta mejora hizo disminuir considerablemente los tiempos de cómputo, pero por otro lado se generó una discapacidad al momento de evaluar algunos *benchmarks*, como es el caso de los *benchmarks* 7 y 10 de F. Boctor. Arreglar dicho problema consta de eliminar el enunciado donde se omite la evaluación de la simetría en las matrices y no sobrevaluar los datos, pero genera un cambio en el modelo implementado con el que ya se realizaron el 80% de las pruebas, por lo que se optó por seguir realizando las pruebas normalmente pero sin implementar las soluciones no posibles de resolver. (*Benchmarks* 7 y 10)

Una vez planteada la problemática y omitiendo los resultados de los *benchmarks* 7 y 10 para presentar datos homogéneos los resultados de las pruebas son los siguientes:

Tabla 9.9 –Experimentación *Benchmark* 1 16x30 en ECL^{i}PS^e

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solutions Founds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	3231,52	1388677	38	11	11
2	16	2	30	9	2596,69	1022769	34	11	11
3	16	2	30	10	1780,86	554323	31	11	11
4	16	2	30	11	1555,53	422274	24	11	11
5	16	2	30	12	998,84	167843	16	11	11
6	16	3	30	6	30244,08	98191768,2	10	27	27
7	16	3	30	7	31036,88	117672338	9	18	18
8	16	3	30	8	39101,58	247229347	13	11	11
9	16	3	30	9	39335,64	142038228	5	11	11

La tabla 9.10 que es el *benchmark* número uno de las pruebas, muestra los resultados de los experimentos propuestos por F. Boctor y los resultados obtenidos utilizando el *solver* ECL^{i}PS^e en la resolución de este problema de optimización.

Los resultados obtenidos son importantes ya que consiguen la cantidad de Números Excepcionales *(Exceptional Numbers)*, que representan la cantidad de movimientos interceldarios que son necesarios para dicha configuración sea óptima. En cuanto a la cantidad de tiempo utilizado en alcanzar estos resultados, se puede apreciar que el tiempo medido en segundos es muy bajo permitiendo a través de las mejoras realizas al código implementado alcanzar los valores óptimos en tiempos computacionales muy bajos y por sobre todo con un nivel de asertividad muy alto, en este caso son con un nivel de asertividad absoluto.

Tabla 9.10 – Experimentación *Benchmark* 2 16x30 en ECL^{i}PS^e

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solutions Founds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	5671,86	4272773	38	7	7
2	16	2	30	9	5946,31	4322707	35	6	6
3	16	2	30	10	3174,52	2052798	30	6	4
4	16	2	30	11	1088,5	639305	25	4	3
5	16	2	30	12	283,13	138153	21	3	3
6	16	3	30	6	35467,57	605414263	7	7	7
7	16	3	30	7	16721,80	300658634	14	6	6
8	16	3	30	8	18658,23	787703603	7	6	6
9	16	3	30	9	18759,76	794520500	4	6	6

La tabla 9.10, presenta resultados que en tiempos de cómputo son mucho mayores que la configuración anterior pero con un nivel de asertividad mucho más bajo ya que se puede apreciar que en la configuración 3 y 4 no se encontró el valor óptimo para este *benchmark*. Aun considerando que no se alcanzó en este experimento la asertividad absoluta al encontrar los valores óptimos de la configuración, los resultados son significativamente cercanos al óptimo.

Tabla 9.11 – Experimentación *Benchmark* 3 16x30 en ECL^{i}PS^e

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solutions Founds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	291,02	155052	22	5	4
2	16	2	30	9	340,88	173037	23	4	4
3	16	2	30	10	290,61	152491	28	4	4
4	16	2	30	11	218,34	133175	21	3	3
5	16	2	30	12	49,3	30831	14	1	1
6	16	3	30	6	1568,15	90104937,7	9	9	9
7	16	3	30	7	3560,69	68825017,1	6	4	4
8	16	3	30	8	1616,27	80909726	3	4	4
9	16	3	30	9	1558,03	83762700,7	3	4	4

La siguiente configuración que se presenta, la tabla 9.11, presenta tiempos de cómputo muy disímiles con las configuraciones anteriormente expuestas, presentado tiempos bajos que permiten un rápido análisis, aunque existen diferencias en cuanto a la cantidad de números excepcionales encontrados, ya que la *Config* 1, de esta tabla presenta una cantidad de movimientos interceldarios de 5 cuando la referencia que presenta Boctor es de 4, aun así los resultados para las otras configuraciones presentan una asertividad total.

Los *benchmark* presentan diferencias sustanciales en cuanto al nivel de complejidad en la resolución de las distintas configuraciones, pero al parecer dicha dificultad o facilidad en resolver cada experimento está dado por cada *benchmark* y la posición que tienen los 1s y 0s dentro de las matrices y no por una configuración particular en cada configuración de un *benchmark*.

Tabla 9.12 – Experimentación *Benchmark* 4 16x30 en ECL^{i}PS^e

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solutions Founds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	15273,58	7694645	37	16	14
2	16	2	30	9	22375,05	10579464	35	16	13
3	16	2	30	10	21826,06	7900218	29	16	13
4	16	2	30	11	13677,92	3206783	27	16	13
5	16	2	30	12	7027,91	1031422	16	15	13
6	16	3	30	6	11467,22	607588457	10	30	27
7	16	3	30	7	12235,22	818873109	10	22	18
8	16	3	30	8	12624,09	503018106	4	14	14
9	16	3	30	9	15991,97	413755546	4	14	13

El hito más relevante de este experimento son las diferencias sustanciales y repetitivas al comprar la cantidad de números excepcionales obtenidos luego de la experimentación y al ser analizadas con la referencia entregada por Boctor son bastantes y disímiles, provocando que la asertividad del modelo en ECL^{i}PS^e sea muy baja pero cercana, pero junto con eso se puede apreciar que la cantidad de tiempo utilizado por este experimento es muy grande y supera en el mejor de los casos las 2 horas de cómputo.

Tabla 9.13 – Experimentación *Benchmark* 5 16x30 en ECL^{i}PS^e

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solutions Founds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	2012,86	1141622	35	10	9
2	16	2	30	9	2015,56	1098829	36	6	6
3	16	2	30	10	2233,69	1200001	30	6	6
4	16	2	30	11	1070,70	452750	20	5	5
5	16	2	30	12	441,16	88139	16	5	4
6	16	3	30	6	179,38	79647999,2	9	11	11
7	16	3	30	7	167,77	193577635	10	8	8
8	16	3	30	8	3064,37	19025149,4	11	8	8
9	16	3	30	9	180,65	96015289,6	4	7	6

Al igual que las configuraciones anteriores, los resultados obtenidos por el *solver* para la resolución de este *benchmark* son bastante interesantes ya que nuevamente la cercanía en casi todos los óptimos es realmente buena, exceptuando en la *Conf 1*, el *solver*

determina que el óptimo es 10, cuando el *benchmark* de referencia indica 9, al margen de esta diferencia, los tiempos y la cantidad de números excepcionales son muy buenos resultados para ver como alternativa de resolución el *solver* ECL^{i}PS^e.

Tabla 9.14 – Experimentación *Benchmark* 6 16x30 en ECL^{i}PS^e

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solutions Founds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	124,86	50591	31	5	5
2	16	2	30	9	192,97	108187	28	3	3
3	16	2	30	10	234,52	114600	22	3	3
4	16	2	30	11	179,03	68013	17	3	3
5	16	2	30	12	49,94	16765	14	2	2
6	16	3	30	6	9490,83	193773248	8	6	6
7	16	3	30	7	105137,61	80259790,4	10	4	4
8	16	3	30	8	70864,46	117712759	18	4	4
9	16	3	30	9	835478,19	104643793	15	3	3

El *benchmark* 6 es la configuración que tardó menos cantidad de tiempo en ser resuelta, dicha configuración al parecer es la más fácil de resolver y no presenta diferencias con respecto a la cantidad de números excepcionales. En promedio a cada configuración tardó menos de 3 minutos en ser resulta lo que hace cada vez más alentadora la resolución de este tipo de problemas en el *solver* ECL^{i}PS^e debido a la baja cantidad de recursos computacionales que este demanda y el tiempo de resolución relativamente bajo para todas sus configuraciones.

Tabla 9.15 – Experimentación *Benchmark* 8 16x30 en ECL^{i}PS^e

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solutions Founds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	4954,5	2213161	36	14	13
2	16	2	30	9	6260,05	3277057	40	10	10
3	16	2	30	10	5818,81	2884759	36	8	8
4	16	2	30	11	2086,22	904029	31	5	5
5	16	2	30	12	693,83	204221	23	5	5
6	16	3	30	6	407274,33	229750839	3	14	14
7	16	3	30	7	1292,82	647258302	17	12	11
8	16	3	30	8	924,17	210847627	3	11	11
9	16	3	30	9	80719,92	237093926	23	10	10

Si se omite el *benchmark* 7 debido a que la mejora implementada para disminuir los tiempos de cómputo provocó una incompatibilidad para resolver todas las configuraciones de los *benchmark* en ECL^{i}PS^e ahora se presentan los resultados del *benchmark* 8.

Los resultados que se pueden observar son bastante disímiles, con configuraciones que fueron resueltas tempranamente y de manera muy simple, alcanzando el valor óptimo esperado en la cantidad de números excepcionales (config. 5), pero también con complicaciones al encontrar y superar ampliamente el límite de tiempo de cómputo de 3 horas (10800 segundos) en las configuraciones 6, 7, 8 y 9 como ha sido la tónica en los experimentos del MCDP.

Tabla 9.16 – Experimentación *Benchmark* 9 16x30 en ECL^{i}PS^e

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solutions Founds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	2493,41	1045470	26	8	8
2	16	2	30	9	1222,53	448202	24	8	8
3	16	2	30	10	430,9	104413	20	8	8
4	16	2	30	11	201,53	52913	19	5	5
5	16	2	30	12	111,5	19896	15	5	5
6	16	3	30	6	1589,23	15491156,7	9	12	12
7	16	3	30	7	1316,10	1916932,51	13	12	12
8	16	3	30	8	1286,63	1800122,17	15	8	8
9	16	3	30	9	1301,16	1954233,88	12	8	8

Los resultados que se presentan en el último experimento (*benchmark 9*) dan cuenta de un hallazgo de todos los óptimos para el experimento. Junto con esto se puede apreciar que la cantidad de números excepcionales obtenidos como resultados son exactamente los mismos referenciados por F. Boctor, lo que señala nuevamente que el modelo implementado en esta ocasión a alcanzo todos los valores óptimos y la configuración óptima de diseño de celdas de manufactura, en tiempos muy bajos, 111 segundos, 201, segundos en algunos casos. Pero a pensar de los buenos resultados obtenidos en las configuraciones 1, 2, 3, 4, 5 en todos los casos las configuraciones 6, 7, 8, 9 presentan una cantidad de *backtracks*, tiempos de cómputo mucho mayores que las primeras configuraciones, haciendo mucho más difícil la resolución de estas configuraciones en general.

9.5. Experimentación del MCDP en Choco

La implementación de las pruebas en el *solver* Choco se realizaron al igual que el ECL^{i}PS^e con problemas validados y certificados por la literatura en términos de veracidad de datos; para ver, en una primera instancia la resolución al modelo y así conocer el patrón de comportamiento y no tanto los tiempos de resolución, donde esta última variable, será evaluada luego, para visualizar tanto el arquetipo del modelo como también la cantidad de soluciones encontradas bajo tiempos computacionales de resolución.

Básicamente el objetivo según lo anterior es; generar experimentación preliminar con un *benchmark* pequeño de tamaño 5 máquinas por 7 partes, para inferir las mejores estrategias de búsqueda y dada estas últimas, generar experimentación eficiente con un *benchmark* de mayor tamaño y complejidad como por ejemplo, el *benchmark* de F. Boctor, este ultimo de tamaño 16 máquinas por 30 partes. Por tanto, se podrá realizar análisis deductivo en una primera instancia, en cuanto a las estrategias de resolución empleadas, y luego, análisis comparativo con otras técnicas de optimización en términos de resultados de viajes interceldarios, además de análisis en tiempos de cómputo y cantidad de *backtracks* producidos por el modelo en comparación con los resultados del *solver* ECL^{i}PS^e .

Los *benchmarks* que se utilizaron, al igual que ECL^{i}PS^e, se extrajeron de la investigación de Xiangyong Li, M.F.Baki e Y.P.Aneja [31] que fue un ejemplo usado para explicar el método de agrupación de máquinas. Este *benchmark* cuenta con una matriz de 5 máquinas por 7 partes explicitado anteriormente, lo que hace que dichas pruebas con este *benchmark* sean muy breves y permitan visualizar tiempos computacionales y cantidad de *backtracks* utilizados antes de llegar a la solución óptima.

9.5.1. Experimentación Benchmark 5x7

Como se nombró anteriormente, tanto para ECL^{i}PS^e como para Choco, las pruebas que se realizaron tenían por objetivo marcar tendencias en el uso eficiente de heurísticas de selección de variables y heurísticas de selección de valor. Las heurísticas que se utilizaron para estos efectos en Choco fueron; *MinDomain, MaxDomain, MostConstrained, StaticVarOrder y MaxRegret*. Dichas heurísticas de selección de variables fueron combinadas con heurísticas de selección de valor como; *MinVal, Middle* y *MaxVal*, que simbolizan el mínimo valor, valor medio, y máximo valor respectivamente.

La finalidad de estas pruebas, como se explicó con anterioridad, es dirigir e orientar el camino para las pruebas con *benchmarks* más complejos, grandes y con más configuraciones posibles, evitando las pruebas con metodologías infructíferas de resolución.

Para el análisis de los resultados se confeccionó, al igual que para la experimentación en ECL^{i}PS^e, un registro de tiempos de cómputo, cantidad de *backtracks* y números excepcionales hallados. Con dichos datos, se compararán los resultados obtenidos al aplicar distintas estrategias de búsqueda.

Las configuraciones para las pruebas con el *benchmark* 5x7 fueron las siguientes:

- Heurísticas de selección de valores seleccionadas:
  - ✓ Valor mínimo (MIN)
  - ✓ Valor máximo (MAX)

- Heurísticas de selección de variables seleccionadas:
  - ✓ MostConstrained (MC)
  - ✓ MinDomain (MID)
  - ✓ MaxDomain (MAD)
  - ✓ StaticVarOrder (SVO)
  - ✓ MaxRegret (MR)

- Cantidad máxima de máquinas por celda:
  - ✓ 4 máquinas por celda
  - ✓ 3 máquinas por celda
  - ✓ 2 máquinas por celda

- Cantidad de celdas:
  - ✓ 5 celdas
  - ✓ 3 celdas
  - ✓ 2 celdas

En términos de la ejecución de las pruebas, al igual que el ECL^{i}PS^e, los resultados en este *benchmark* bajo el modelo en Choco, arrojó resultados determinantes para etapas posteriores de análisis, identificando primero, heurísticas con un desempeño no aceptable para este problema, y segundo, combinaciones de heurísticas que socaban tiempos razonables de cómputo para la implementación de futuras pruebas y estrategias de resolución que pueden llevar a un buen desempeño al momento de utilizar *benchmarks* de mayor capacidad.

Al igual que en ECL^{i}PS^e, la experimentación en el solver Choco para el *benchmark* de 5x7, constó de un total de 135 experimentos. Estos últimos, distribuidos en un total de 5 pruebas realizadas en función de la combinación de las heurísticas de variable y valor operadas por Choco.

El resultado de la experimentación, utilizando distintas estrategias de búsqueda, es plasmado mediante tablas de resultado; estas muestran, los resultados de la experimentación mediante la utilización de los parámetros de configuración detallados anteriormente como por ejemplo la cantidad de máquinas utilizada en el modelo, la cantidad de partes manufacturadas, también el tiempo de computo utilizado por el *solver*, cantidad de *backtracks* y por último la cantidad de elementos o números excepcionales encontrados al ejecutar el modelo en Choco.

La siguiente tabla (ver tabla 9.19) muestra la descripción anterior mediante el empleo de una estrategia de búsqueda combinada con la heurística de selección de variable *MostConstrained* en composición con las heurísticas de selección de valor *MinVal*, *MaxVal* y *Middle*.

Tabla 9.17 – Benchmark 5x7 – Most Constrained en Choco

					MC/MIN			MC/MID			MC/MAX		
Config	**Machines**	**Cells**	**Parts**	**Mmax**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**
1	5	5	7	4	14,259	54994	3	15,089	54994	3	10,827	81832	3
2	5	3	7	4	0,826	3084	3	1,098	3084	3	0,721	3857	3
3	5	2	7	4	0,138	359	3	0,309	359	3	0,529	359	3
4	5	5	7	3	32,43	253398	5	34,72	253398	5	46,327	381257	5
5	5	3	7	3	1,395	9814	5	1,743	9814	5	2,258	12295	5
6	5	2	7	3	0,19	684	5	0,897	684	5	0,473	684	5
7	5	5	7	2	173,43	1197842	8	182,44	1197842	8	140,911	1797104	8
8	5	3	7	2	2,039	13722	8	2,101	13722	8	2,346	17136	8
9	5	2	7	2	0,022	7	-	0,912	7	-	0,027	7	-

Es necesario observar de la tabla anterior la utilización de las tres heurísticas de selección de valor; pero también, es necesario notar que, la cantidad utilizada de tiempo computacional entre las heurísticas de selección de valor del minino valor, con la del valor medio tienen el mismo, con ínfimas diferencias, resultado, esto último debido básicamente a configuraciones propias del *solver* Choco. Por tanto, no se hace necesario seguir la experimentación con heurísticas de selección de valor medio, premisa que se utilizará para el resto de la experimentación en el *solver* Choco.

Tabla 9.18 – Benchmark 5x7 – MinDomain en Choco

					MID/MIN			MID/MAX		
Config	**Machines**	**Cells**	**Parts**	**Mmax**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**
1	5	5	7	4	12,169	54494	3	13,566	81833	3
2	5	3	7	4	0,56	3084	3	0,884	3857	3
3	5	2	7	4	0,187	359	3	0,524	359	3
4	5	5	7	3	51,869	253398	5	48,811	381257	5
5	5	3	7	3	1,317	9814	5	1,492	12295	5
6	5	2	7	3	0,202	684	5	0,401	684	5
7	5	5	7	2	159,552	1197842	8	167,141	1797104	8
8	5	3	7	2	2,175	13722	8	2,395	17136	8
9	5	2	7	2	0,027	7	-	0,025	7	-

La tabla anterior (ver tabla 9.20) muestra los experimentos basados en la utilización de una estrategia de búsqueda compuesta por una heurística de selección de variable *MinDomain* con heurísticas de selección del mínimo valor y máximo valor respectivamente. Se observa que ya no se realizaron experimentos con heurísticas de selección de valor medio, considerada esta última infructífera, de acuerdo a los experimentos efectuados con la estrategia de búsqueda anterior.

Consecutivamente a los experimentos anteriores, los resultados al ejecutar el modelo con una estrategia de búsqueda constituida por una heurística de selección de variable *StaticVarOrder* con heurísticas de selección de valor mínimo y máximo respectivamente, son visualizados en la figura 9.21. Se puede observar también las ínfimas diferencias que existen entre la elección de una heurística de selección de valor *MinVal* o *MaxVal*, situaciones tendenciosas que se observan en cada una de las configuraciones, como por ejemplo la configuración número 1 donde entre una elección y otra hay una diferencia de 2 segundos en tiempo computacional.

Tabla 9.19 – Benchmark 5x7 – StaticVarOrder en Choco

					SVO/MIN			SVO/MAX		
Config	**Machines**	**Cells**	**Parts**	**Mmax**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**
1	5	5	7	4	10,001	54494	3	12,926	81833	3
2	5	3	7	4	0,639	3084	3	0,737	3857	3
3	5	2	7	4	0,133	359	3	0,321	359	3
4	5	5	7	3	29,73	253398	5	29,843	381257	5
5	5	3	7	3	1,923	9814	5	1,472	12295	5
6	5	2	7	3	0,282	684	5	0,368	684	5
7	5	5	7	2	101,34	1197842	8	119,768	1797104	8
8	5	3	7	2	1,52	13722	8	1,781	17136	8
9	5	2	7	2	0,032	7	-	0,025	7	-

La siguiente tabla evidencia (ver tabla 9.22) los resultados obtenidos al ejecutar el MCDP bajo una estrategia de búsqueda formada por una heurística de selección de variable *MaxDomain* y heurísticas de selección de valor mínimo y máximo. A diferencia de las estrategias anteriores, se observan grandes diferencias en términos de tiempo computacional con respecto a estrategias de búsqueda anteriores. Esto debido principalmente a la elección de manera prioritaria, de la variable con el dominio más grande dentro del modelo.

Tabla 9.20 – Benchmark 5x7 – MaxDomain en Choco

					MAD/MIN			MAD/MAX		
Config	**Machines**	**Cells**	**Parts**	**Mmax**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**
1	5	5	7	4	9,313	76291	3	4357,094	93320964	3
2	5	3	7	4	0,736	4271	3	30,546	413478	3
3	5	2	7	4	0,123	475	3	0,46	1347	3
4	5	5	7	3	47,181	402558	5	3666,251	21031989	5
5	5	3	7	3	1,965	14702	5	29,266	373860	5
6	5	2	7	3	0,258	915	5	0,76	2415	5
7	5	5	7	2	197,307	2290999	8	3612,143	29670701	8
8	5	3	7	2	3,017	28959	8	13,095	235115	8
9	5	2	7	2	0,258	1118	-	0,277	1119	-

Por último, en la tabla 9.23 se muestran los resultados obtenidos al ejecutar el modelo bajo una composición de heurísticas de selección de variable *MaxRegret* y heurísticas de selección de valor como *MinVal* y *MaxVal*. Básicamente se percibe la tendencia con otras estrategias de búsqueda anteriores, donde, no se notan grandes diferencias en el tiempo computacional utilizado entre la elección de una heurística de selección de valor máximo o mínimo.

Tabla 9.21 – Benchmark 5x7 – MaxRegret en Choco

					MR/MIN			**MR/MAX**		
Config	**Machines**	**Cells**	**Parts**	**Mmax**	**Time**	**BT**	**EN**	**Time**	**BT**	**EN**
1	5	5	7	4	7,322	54494	3	7,721	818333	3
2	5	3	7	4	0,618	3084	3	0,806	3857	3
3	5	2	7	4	0,15	359	3	0,46	359	3
4	5	5	7	3	36,22	253398	5	33,693	381257	5
5	5	3	7	3	1,734	9814	5	1,925	12295	5
6	5	2	7	3	0,201	684	5	0,779	684	5
7	5	5	7	2	109,05	119782	8	117,067	179710	8
8	5	3	7	2	1,918	13722	8	1,911	17136	8
9	5	2	7	2	0,034	7	-	0,025	7	-

En términos de gráficos de resultados, a continuación se presenta un gráfico que muestra básicamente las mejores configuraciones por estrategia de búsqueda adoptada en cada una de las configuraciones del modelo. Los resultados presentados dan muestra de las diferencias al utilizar las distintas estrategias de resolución; donde, cada heurística presenta comportamientos singulares y heterogéneos en comparación, dependiendo de la configuración, con otras estrategias de resolución.

Básicamente en el gráfico presentado se distinguen por ejemplo las tendencias por el mejor comportamiento de las heurísticas *MinDomain* con la heurística *StaticVarOrder*: donde estas últimas presentan la mayor cantidad de buenos resultados, por configuración, de la experimentación.

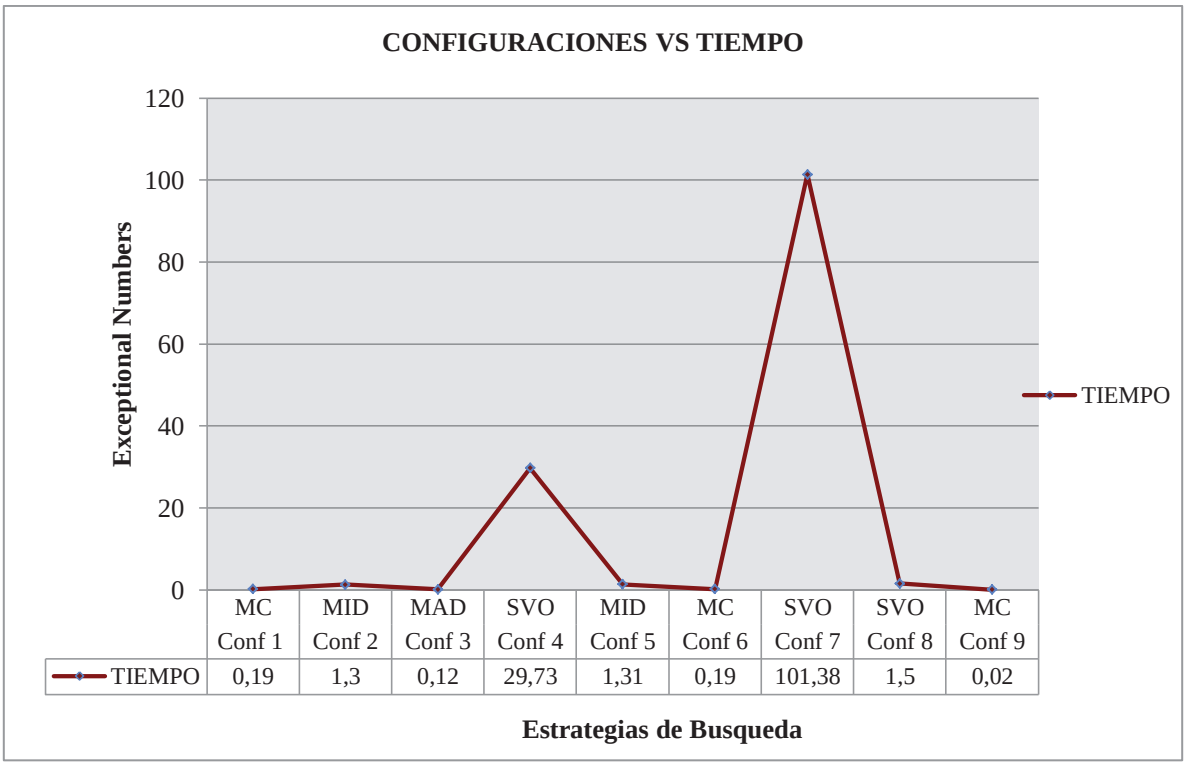


Figura 9.3 – Gráfico Mejores Soluciones por Configuración *Benchmark* 5x7 en Choco

Por otra parte, es importante señalar las relaciones involucradas entre los parámetros de configuración y los resultados del MCDP en Choco. Primero, se debe notar que la cantidad máxima de máquinas por celda tiene directa relación con el tiempo de cómputo empleado, situación visualizada en cada una de las estrategias de búsqueda graficadas en las tablas de resultado y en el gráfico anterior. Esto último se explica principalmente por el costo asociativo y combinatorio de la celda mecánica al disminuir la cantidad de máquinas a asignar.

A modo de conclusión general y particular para este tipo de *benchmark*, se puede mencionar el hecho de obtener similares resultados en base a un tipo de configuración en particular, con la legítima excepción claro, de una sola heurística de selección de variable y valor, en este caso *MaxDomain* y *MaxVal, (ver tabla 9.22)*, con la que se obtienen resultados inviables en términos de tiempos computacionales. A su vez, se manifiesta que la estrategia de búsqueda con mayor rendimiento bajo tiempos computacionales; es *MinDomain* con una heurística de selección de valor *MinVal*, esta última graficada en la tabla resultado 9.19.

9.5.2. Experimentación Benchmark 16x30

Esta matriz de datos máquina-parte, está estructurada por un número de 16 máquinas simbolizadas en las filas de la matriz y también por un número de 30 partes simbolizadas en columnas de la matriz máquina-parte.

Al igual que lo mencionado anteriormente con las pruebas en ECL^{i}PS^e, la experimentación en Choco del *benchmark* de 16 máquinas y 30 partes de Boctor, se basa en la elección de la mejor estrategia de búsqueda. Según las pruebas preliminares efectuadas en el *benchmark 5x7*, esta elección es una estrategia de búsqueda compuesta por una heurística de selección de variable como *MinDomain* y una heurística de selección del valor mínimo. Dada esta premisa, se genera experimentación con un total de 45 pruebas organizadas en 10 *benchmarks* validados y certificado, donde el detalle de la experimentación se presenta a continuación.

Para el análisis de los resultados se confeccionó, al igual que en el *benchmark* de 5x7, un registro de tiempos de cómputo, cantidad de *backtracks* y números excepcionales encontrados. Con estos datos, se compararán los resultados al aplicar la estrategia de búsqueda compuesta por una heurística de selección de variable *MinDomain* con una heurística de selección del mínimo valor. Una vez efectuado lo anterior, los resultados obtenidos serán materia de comparación, en términos de óptimos encontrados frente a otras técnicas de optimización como SA, como también fruto de comparación con otro *solver* de experimentación como ECL^{i}PS^e.

Las configuraciones para las pruebas con el *benchmark* de 16x30 son las siguientes:

- Heurísticas de selección de valor escogida:
  - ✓ Valor mínimo (MIN)

- Heurísticas de selección de variable escogida:
  - ✓ MinDomain (MID)

- Cantidad máxima de máquinas por celda:
  - ✓ 8 máquinas por celda
  - ✓ 9 máquinas por celda
  - ✓ 10 máquinas por celda
  - ✓ 11 máquinas por celda
  - ✓ 12 máquinas por celda
  - ✓ 6 máquinas por celda
  - ✓ 7 máquinas por celda.

- Cantidad de celdas:
  - ✓ 3 celdas
  - ✓ 2 celdas

Para cada uno de los *benchmarks* de 16x30 utilizados, las configuraciones son las mismas que las dispuestas por Boctor, es decir se mantienen la cantidad de celdas y cantidad máxima de máquinas, sólo se clasifican las distintas instancias a través de los parámetros de configuración. Es decir por ejemplo, la configuración de parámetros no se ve reflejada en la cantidad de celdas, pero si en las fluctuaciones del parámetro *Mmax,* este último; de importantísimo valor por ser directamente proporcional a los números excepcionales encontrados por el modelo.

A continuación (ver tabla 9.26) se presentan los resultados obtenidos al aplicar la estrategia de búsqueda compuesta por la heurística de selección de variable MinDomain con una estrategia de selección de valor Min en el *benchmark* 1 de dimensión 16x30.

A continuación (ver tabla 9.26) se presentan los resultados obtenidos al aplicar, en el *benchmark* 1, la estrategia de búsqueda compuesta por la heurística de selección de variable *MinDomain* con una estrategia de selección de valor *Min*. Se puede observar que el modelo halla el óptimo para la mayoría de las configuraciones dispuestas, salvo algunas excepciones, como por ejemplo la configuración número 7.

Tabla 9.22 – Experimentación Benchmark 1 16x30 en Choco

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solution Finds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	8.228,42	2547685	28	22	11
2	16	2	30	9	2.397,69	788697	23	11	11
3	16	2	30	10	5.903,96	1905848	24	11	11
4	16	2	30	11	11.433,48	3984698	14	11	11
5	16	2	30	12	4.485,44	1108125	11	11	11
6	16	3	30	6	110.928,00	3725847	10	50	27
7	16	3	30	7	121.441,00	4582689	9	19	18
8	16	3	30	8	128.033,00	9801448	13	11	11
9	16	3	30	9	131.973,00	4709582	5	37	11

En tabla 9.27 se exhiben los resultados obtenidos al ejecutar el modelo en el *benchmark* número 2, donde se observa que si bien, no se obtiene la solución óptima, los resultados son relativamente cercanos al optimo en la mayoría de los casos. Ahora bien, también se observa que para algunas configuraciones el modelo si logra la solución óptima, como por ejemplo en la configuración número 4.

Tabla 9.23 – Experimentación Benchmark 2 16x30 en Choco

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solution Finds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	6265,14	2368546	18	8	7
2	16	2	30	9	4835,57	1848793	18	7	6
3	16	2	30	10	76979,10	1715915	14	6	4
4	16	2	30	11	626,04	218740	13	4	4
5	16	2	30	12	453,55	160814	10	4	3
6	16	3	30	6	212180,00	4063221	7	36	7
7	16	3	30	7	103266,00	1992667	14	7	6
8	16	3	30	8	113947,00	4517876	7	22	6
9	16	3	30	9	114507,00	4776543	4	24	6

En tabla 9.28 se presentan los resultados obtenidos al ejecutar el modelo en el *benchmark* número 3. Es necesario notar que para configuraciones que utilizan 3 celdas el *solver* no encuentra la solución óptima en un tiempo computacional razonable, caso contrario a la utilización de 2 celdas, donde en la mayoría de los casos evaluados halla el óptimo.

Tabla 9.24 – Experimentación Benchmark 3 16x30 en Choco

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solution Finds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	392,66	104045	13	5	4
2	16	2	30	9	879,59	308556	29	4	4
3	16	2	30	10	4315,70	1521187	11	4	4
4	16	2	30	11	886,00	245120	7	3	3
5	16	2	30	12	748,32	209618	6	1	1
6	16	3	30	6	11336,60	3709408	9	23	9
7	16	3	30	7	24641,70	2846766	6	25	4
8	16	3	30	8	11021,80	3628064	3	25	4
9	16	3	30	9	10061,50	3252064	3	21	4

Consecutivamente en tabla 9.29 se exponen los resultados al aplicar el modelo en el *benchmark* 4, se observa el hecho de la cantidad de acierto en la solución óptima en gran parte de los casos con 2 celdas. Por otra parte, bajo una configuración de 3 celdas de manufactura, los resultados en tiempos de computación razonables no entregan la solución óptima, pero si una solución relativamente cercana a esta última.

Tabla 9.25 – Experimentación Benchmark 4 16x30 en Choco

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solution Finds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	6265,2	104045	13	14	14
2	16	2	30	9	4835,5	308556	29	15	13
3	16	2	30	10	76979,104	1521187	11	13	13
4	16	2	30	11	626,04	245120	7	14	13
5	16	2	30	12	748,32	209618	6	13	13
6	16	3	30	6	11336,60	3709408	9	23	27
7	16	3	30	7	24641,70	2846766	6	25	18
8	16	3	30	8	11021,80	3628064	3	25	14
9	16	3	30	9	10061,50	3252064	3	21	13

En la tabla 9.30 se exponen los resultados obtenidos en el *benchmark* 5, estos muestran que el desempeño del modelo en términos de resultados bajo un tiempo de computación consensuado no es el mejor, obteniendo, por ejemplo ninguna solución óptima para este *benchmark* en particular.

Tabla 9.26 – Experimentación Benchmark 5 16x30 en Choco

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solution Finds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	1085.75	4.475.954	30	17	9
2	16	2	30	9	421523,00	4.435.322	28	16	6
3	16	2	30	10	4215.57	3.456.890	25	16	6
4	16	2	30	11	31234,00	3.120.212	22	16	5
5	16	2	30	12	89788,90	2.908.120	20	15	4
6	16	3	30	6	114136,30	4.252.229	10	30	11
7	16	3	30	7	154545,70	5.521.937	10	25	8
8	16	3	30	8	106990,70	3.623.817	4	27	8
9	16	3	30	9	144009,00	2.847.018	4	18	6

Seguidamente en la tabla 9.31 se muestran los resultados en el *benchmark* número 6, donde básicamente no se obtiene la solución óptima bajo un tiempo de computación aceptable, pero si atisbos de mejores soluciones encontradas como es por ejemplo en la configuración número 5, donde se obtiene una aproximación razonable a la mejor solución.

Tabla 9.27 – Experimentación Benchmark 6 16x30 en Choco

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solution Finds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	11202114,00	3821584	19	31	5
2	16	2	30	9	10989.37	2274461	14	29	3
3	16	2	30	10	11435668,00	3367908	14	22	3
4	16	2	30	11	56779.09	2061630	16	16	3
5	16	2	30	12	6149426,00	1927471	14	11	2
6	16	3	30	6	1137948,80	2894068	9	43	6
7	16	3	30	7	1115658,00	7554746	10	20	4
8	16	3	30	8	18912428,00	811816	11	27	4
9	16	3	30	9	1108728,60	3582284	4	32	3

Por otra parte, en la tabla 9.32, se observa un comportamiento similar del modelo, en comparación con *benchmarks* anteriores, que se obtiene en una gran mayoría de configuraciones la solución óptima a los movimientos interceldarios, y en el resto de las configuraciones, atisbos muy cercanos a la mejor solución para esa configuración en particular.

Tabla 9.28 – Experimentación Benchmark 7 16x30 en Choco

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solution Finds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	4.668,45	4.668	11	7	7
2	16	2	30	9	990,84	307455	11	5	4
3	16	2	30	10	762,60	250999	12	4	4
4	16	2	30	11	325,56	85550	11	4	4
5	16	2	30	12	1.123,33	336145	13	4	4
6	16	3	30	6	105681.3	6977974	8	36	11
7	16	3	30	7	1159859,80	2756832	10	17	5
8	16	3	30	8	915339,80	3786672	18	5	5
9	16	3	30	9	10431079,00	3702285	15	6	4

Al igual que la tabla anterior, la tabla 9.33 muestra los resultados de un *benchmark*, en este caso el número 8, que es de similar estructura de parámetros a los anteriores, en donde se obtiene para una gran mayoría de configuraciones la solución óptima y para el resto, soluciones cercanas a esta última.

Tabla 9.29 – Experimentación Benchmark 8 16x30 en Choco

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solution Finds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	1132721,00	3910391	31	18	13
2	16	2	30	9	554359,20	1975737	29	10	10
3	16	2	30	10	2070,14	651376	20	9	8
4	16	2	30	11	3131,77	1163916	16	6	5
5	16	2	30	12	9017,57	291580	14	6	5
6	16	3	30	6	41840332,30	1936470	3	53	14
7	16	3	30	7	128182,40	5455459	17	23	11
8	16	3	30	8	113701,20	2030000	3	36	11
9	16	3	30	9	7961495,00	2166933	23	10	10

En la siguiente tabla (ver tabla 9.39), se observan los resultados del *benchmark* número 9 al ejecutar el modelo del MCDP en Choco. Se observa una vez más que se logra el objetivo de encontrar la solución óptima en la mayoría de las configuraciones, exceptuando solo algunos casos, que en su mayoría se tratan de la utilización de 3 celdas de manufactura.

Tabla 9.30 – Experimentación Benchmark 9 16x30 en Choco

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solution Finds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	5362,89	2173173	21	8	8
2	16	2	30	9	2734,90	1037179	21	8	8
3	16	2	30	10	10403,36	3745368	13	24	8
4	16	2	30	11	11885,54	3944614	4	30	5
5	16	2	30	12	20358,04	606865	11	5	5
6	16	3	30	6	20358,04	1204590	9	42	12
7	16	3	30	7	18388,10	145790	13	27	12
8	16	3	30	8	19190,56	158406	15	22	8
9	16	3	30	9	18218,12	189680	12	21	8

Por último, en la tabla 9.40, se pueden notar que en una gran mayoría de casos o configuraciones se logra la solución óptima, tendencia repetitiva en comparación con otros *benchmarks*, pero que demuestra fehacientemente el desempeño del modelo para *benchmarks* validados.

Tabla 9.31 – Experimentación Benchmark 10 16x30 en Choco

Config	Machines	Cells	Parts	Mmax	Time	Backtracks	Solution Finds	Exceptional Numbers	Reference Boctor
1	16	2	30	8	102182.32	2274492	26	8	8
2	16	2	30	9	322185,00	3307455	18	5	5
3	16	2	30	10	2155,02	745321	16	5	5
4	16	2	30	11	2028,67	1242621	14	5	5
5	16	2	30	12	1912,56	1021560	13	5	5
6	16	3	30	6	4721045,00	1782671	11	46	10
7	16	3	30	7	1243054,00	2308080	15	34	8
8	16	3	30	8	184832,00	6590788	24	23	8
9	16	3	30	9	171898,00	265980	21	18	5

9.6. Análisis de la Experimentación

Dentro de la experimentación efectuada, uno de los puntos principales a tomar en cuenta es la diferencia entre los *solver* con los que se realizaron las pruebas. Dichos *solver* uno ECL^{i}PS^e y otro Choco, son distintos por el modelo que utilizan para resolver los problemas y por los paradigmas que utilizan para llegar a ser los *solvers* que son ahora. ECL^{i}PS^e por un lado viene de una línea de la programación en lógica, utilizando reglas y enunciados lógicos, mientras que el *solver* Choco es mucho más actual y se complementa con la interfaz de los programas que usan la metodología de orientación a objeto principalmente y usando conceptos de la programación más actual.

Para el análisis de los resultados se tomó al igual que en los análisis individuales por *solver*, todas las configuraciones representadas y ejecutadas del modelo; es decir el análisis se efectúa en base al número de configuraciones aceptadas del modelo, además del número de elementos excepcionales utilizados como referencia de soluciones optimas, la cantidad de números excepcionales encontrados por Choco y por último, el número de elementos excepcionales encontrados por ECL^{i}PS^e.

En los siguientes gráficos, clasificados según los *benchmarks* experimentados, se compararan los números excepcionales encontrados por ambos *solvers*, pudiéndose analizar la tendencia en términos de efectividad y eficiencia entre ECL^{i}PS^e y Choco.

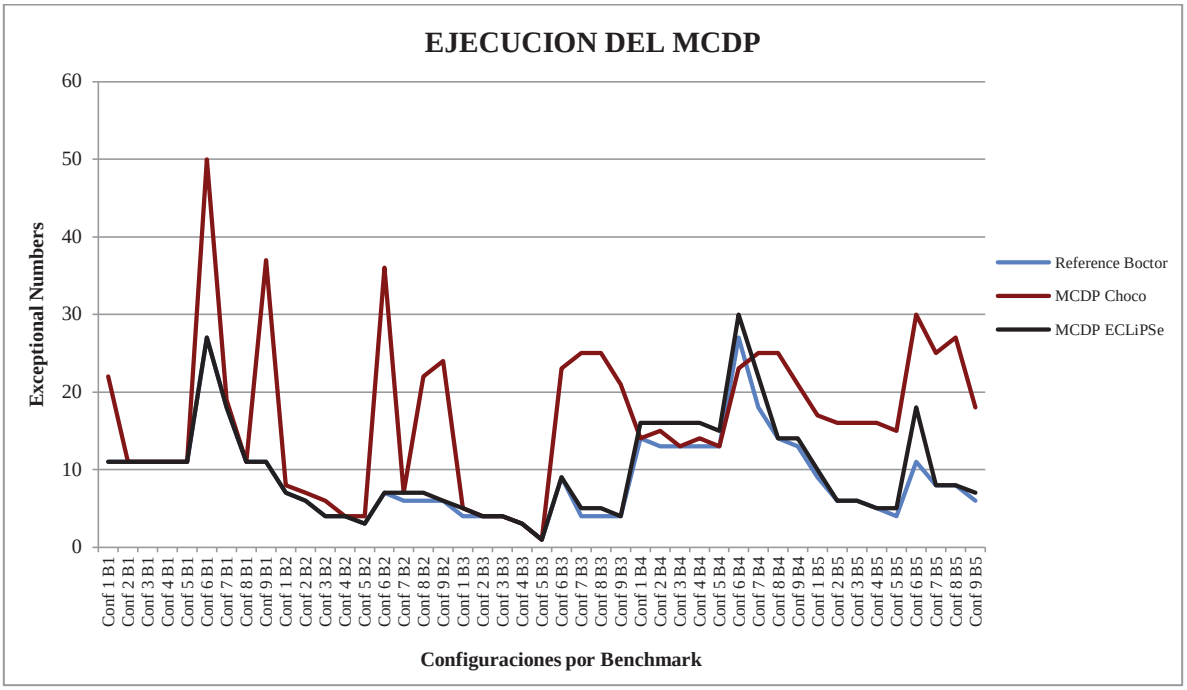


Figura 9.4 – Gráfico Comparativa Solvers Benchmarks 1 - 5

En los datos presentados recientemente se puede observar bastantes diferencias entre los *solvers*, diferencias claras y predecibles como por ejemplo, la cantidad de números excepcionales que presentan principalmente porque Choco y ECL^{i}PS^e trabajaron con un *timeout* de 3 horas aproximadamente, por lo que no siempre se llegó al óptimo.

A primera vista se puede apreciar que los viajes interceldarios o números excepcionales entre Choco y ECL^{i}PS^e son ampliamente distintos, considerando que estos diferencias están dadas por los modelos implementados y por la cantidad de iteraciones que cada solver debía realizar para llegar a ese óptimo.

En comparativa, véase el gráfico, ECL^{i}PS^e generalmente llega a valores mucho más cercanos al óptimo referenciado de Boctor o al mismo optimo en caso todos los casos quedando Choco con óptimos mucho más lejanos que los reales. Esto es claramente visible sobre todo en las configuraciones con 3 celdas de manufactura, donde básicamente el comportamiento de Choco, en base a ECL^{i}PS^e, es muy diferente a las soluciones optimas referenciadas por Boctor.

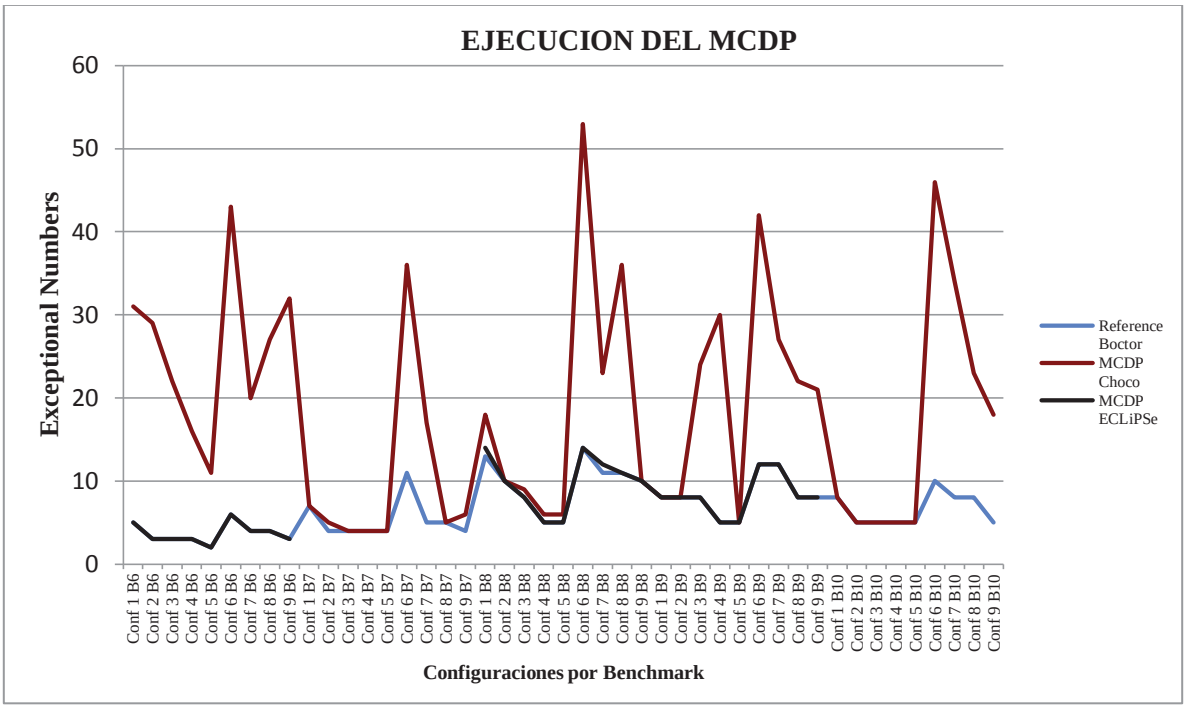


Figura 9.5 – Gráfico Comparativa Solvers Benchmarks 6 - 10

Es necesario destacar que el gráfico anterior refleja las situaciones de ejecución de los *benchmarks* 6 al 10, salvaguardando las excepciones de los *benchmark* número 8 y 10 en ECL^{i}PS^e , estos últimos sin consideración por encontrar soluciones no optimas en comparación con la referencia de Boctor. Por otra parte, al igual que el gráfico anterior del *benchmark* 1 al *benchmark* 5, se observan igualmente diferencias sustantivas entre ambos *solvers*, acentuando aún más las diferencias cuando se utilizan las configuraciones de 3 celdas de manufactura.

En balance, ver gráfico anterior, ECL^{i}PS^e al igual que en los *benchmarks* anteriores, logra un acercamiento más acentuado que Choco en términos de números excepcionales encontrados en base a la referencia de soluciones optimas, con excepción claro, del análisis de situaciones en los *benchmarks* 8 y 10 debido a las razones anteriormente mencionadas. Choco por su parte entrega, dependiendo del *benchmark*, soluciones acertadas o cercanas al optimo con configuraciones de 2 celdas; reflejándose también que para configuraciones con 3 celdas de manufactura, este *solver* entrega soluciones que son lejanas al óptimo.

La cantidad de *backtracks* es otros de los factores importantes a considerar al momento de realizar comparaciones. La cantidad de *backtracks* puede ayudar fácilmente a conocer cuáles son las estrategias que requieren de menor cantidad de tiempo en resolverse y cuales se tardarán más.

En las tablas que se presentaron se vislumbran las tendencias y que en la resolución de los *benchmarks* pequeños ya se pudieron visualizar. Por tanto, hay puntos de encuentro bastante importantes y peculiares. Por ejemplo (ver tablas resultado número 2,3,,4 y 6) se puede ver que el *benchmark* 2, 3, 4, 6 presenta un comportamiento similar en los dos *solvers* lo que hace pensar que las características de los problemas inciden mucho más que la resolución en uno o en el otro. En palabras simples, lo que es complicado de resolver para Choco también lo será para ECL^{i}PS^e, si bien, ECL^{i}PS^e presenta un mejor desempeño la complejidad se puede ver representada por el problema no por el *solver* que encontrará el óptimo.

Finalmente como conclusión del análisis de la experimentación, el objetivo de todo el trabajo presentado y de las presentaciones de los experimentos realizados es conocer cuáles eran las mejores estrategias de resolución del problema del MCDP. Este objetivo está cumplido mediante las pruebas realizadas con el *benchmark* de 5x7 que en los dos casos resultó ser la heurística de selección de valor con el dominio más pequeño, *First Fail* para ECL^{i}PS^e y *Min Domain* para Choco.

Se pudo determinar que la heurística de selección de valor no tiene mucha incidencia en la resolución de los experimentos y que este problema en particular que está dado con un dominio entre 0 y 1 la heurística de valor *middle*, entrega los mismos valores en cuanto a *backtracks* y tiempo que la heurística *Min*, diferenciándose muy poco con la heurística ***Max***, por lo que queda de manifiesto que la elección de una heurística de selección de valor no presentaría aportes o desmedros en la búsqueda de resultados para el problema del MCDP.

Otro punto importante a destacar es que la complejidad de los problemas está dada por la cantidad de celdas que se permitan crear y por el *Mmax* que influyen también en la resolución del problema entregando más o menos posibilidades de agrupación de las máquinas, ya que el ser el *Mmax* cercano a la cantidad de máquinas del problema, la resolución e este es mucho más fácil para el o los *solvers* que deseen evaluarse con esa configuración.

Por último y a modo de información adicional se quiere mencionar que al igual que se presentaron heurísticas de selección de variables con buenos resultados como *Min Domain*, también existe una consistencia en los datos entregados por los dos *solvers*. La heurística con peor desempeño al momento de implementarla fue para los dos casos *Most Constrained*, heurística que elige el valor con un dominio más restringido. Esta heurística presentaba expectativas altas ya que en un comienzo por lo general convergía rápidamente hacia valores óptimos, pero su tiempo iterando al final era tan extenso que la transformó en ser la heurística de selección de variables con los resultados insatisfactorios.

10. Conclusiones

Al concluir este proyecto, básicamente se ha introducido y presentado un modelo hacia la resolución del problema del diseño de celdas de manufactura utilizando programación con restricciones. Con muchas y variadas técnicas se ha resuelto el MCDP, pero aún sin considerar la programación con restricciones; por tanto los algoritmos de búsqueda, técnicas de consistencia y heurísticas presentadas en el modelo entregan, en un marco teórico y práctico, el pilar estructural a la resolución del problema de diseño de celdas de manufactura.

No obstante de inconvenientes ha estado esta resolución, surgieron problemas y dificultades propias de cualquier investigación, aunque el buen desarrollo en términos de modularización de la problemática y el orden de objetivos ayudó en gran medida a la solución del MCDP. Consecutivamente y secundando lo anterior, el proyecto fue planificado en 3 fases; la primera de ellas y una de las más importantes constó del conocimiento de conceptos acordes a la manufacturación de productos en la industria metal-mecánica hasta ese entonces desconocidos, lo cual fue tremendamente determinante debido a que estos conceptos son pilares fundamentales en la solución de un problema de diseño de celdas de manufactura.

La segunda fase, también de vital importancia, consistió en obtener los conocimientos necesarios para comprender lo que es un problema de optimización de restricciones (COP) y como formalizar un problema cotidiano bajo este último. Esta etapa integró conceptos tales como fundamentación, algoritmos de búsqueda, técnicas de consistencia y heurísticas; conceptos que en su conjunto ayudaron a posibilitar el análisis, inicio y operación del modelo.

Seguidamente en la tercera fase se constituyeron las etapas determinantes y concluyentes del proyecto, donde básicamente se ha trabajado en la resolución del problema formalizado como un COP para la implementación en el *solver* ECL^{i}PS^e y en el *solver* Choco, con los cuales se obtuvo la resolución del problema. Bajo estos preceptos, en cada uno de estos *solvers* se han elaborado 2 metodologías de resolución; la primera de ellas constó de la solución al problema sin el empleo de heurísticas ni estrategias de resolución, por lo no se prestó mucha atención a variables como el tiempo de resolución en cada *solver*, sino a los datos obtenidos en función de la ejecución del modelo y que fueron significativamente importantes de entender. La segunda metodología empleada, fue el empleo de heurísticas y/o estrategias de resolución; bajo la ejecución de estas últimas se han obtenido una serie de datos muy valiosos, que junto a los datos anteriores ayudaron a generar valor al análisis de la experimentación y los resultados obtenidos.

En definitiva, actualmente se ha resuelto, analizado y se han inferido los resultados en términos de la satisfacción de restricciones matriciales, parámetros de configuración y variables de resultado con experimentación minuciosa de optimización del problema para determinar el óptimo en función de minimizar la cantidad de viajes de partes entre máquinas. Luego de esto, se han analizado y comparado los resultados obtenidos tanto en el *solver* ECL^{i}PS^e como en Choco determinando la mejor estrategia de resolución. Convergiendo producto de aquello, a la publicación científica del trabajo realizado bajo el título de *paper* "*Solving Manufacturing Cell Design Problems Using Constraint Programming*" [32].

Sin embargo, también el análisis de resultados permite visualizar que el modelo de resolución propuesto es susceptible a mejoras tanto en su diseño como quizás en su implementación final. Por tanto y consecutivamente, como posibilidades de trabajo futuro, es clara la necesidad de probar el desempeño de la implementación del modelo en ambos *solvers* de programación con restricciones y evaluar la efectividad y eficiencia en la resolución del MCDP y su inclusión en el modelo propuesto.

Finalmente aunque la investigación del MCDP utilizando programación con restricciones se muestra enfocada a un problema específico, puede ser aplicada en cualquier ámbito organizacional en el que se requiera agrupar procesos, funciones o procedimientos con cierto grado de similitud entre sí.

11. Referencias Bibliográficas y Bibliografía

[1] Boctor, F. (1991). A linear formulation of the machine-part cell formation problem. *International Journal Production Research*, 29(2), 343-356.

[2] Chen, W. H., & Srivastava, B. (1994). Simulated annealing procedures of forming machine cells in group technology. *European Journal of Operational Research*, 75, 100-111.

[3] Wu, T-h., Chang, C.-C., Chung, S.-H (2008). A simulated annealing algorithm for manufacturing cell formation problems. *Expert Systems with Applications*, 34(3), 1609-1617.

[4] Venugopal, V., & Narendran, T. T. (1992). A genetic algorithm approach to the machine component grouping problem with multiple objectives. *Computers and Industrial Engineering*, 22.4, 469-480.

[5] Gupta, Y., Gupta, M., Kumar, A., & Sundaram, C. (1996). A genetic algorithm-based approach to cell composition and layout design problems. *International Journal of Production Research*, 34(2), 447-482.

[6] Aljaber, N., Baek, W., &Chen, C, L, (1997). A Tabu Search approach to the cell formation problem. *Computers and Industrial Engineering*, 32(1), 169-185.

[7] Lozano, S., Adenso, B., Salinas, I., & Gimenez, L. (1999). A one-step Tabu search algorithm for manufacturing cell design. *Journal of the Operational Research Society*, 50(5), 509-516.

[8] Andres, C., & Lozano, S. (2006). A particle swarm optimization algorithm for part machine grouping. *Robotics and Computer-Integrated Manufacturing*, 22(5-6), 468-474.

[9] Ming, L.C. & Ponnambalam, S.G. (2008). A hybrid GA/PSO for the concurrent design of cellular manufacturing system. *IEEE International Conference on Systems, Man and Cybernetics*, 1855-1860.

[10] Mehdizadeh, E. & Tavakkoli-Moghaddam, R. (2009). A fuzzy particle swarm optimization algorithm for a cell formation problem. *Proceedings of IFSA-EUSFLAT*, 1768-1772.

[11] Durán, O., Rodriguez, N. & Consalter, L.A. (2009). Collaborative particle swarm optimization with a data mining technique for manufacturing cell design. *Expert Systems with Applications*, 37,1563–1567.

[12] R. Soto, (2009). Languages and Model Transformation in Constraint Programming. PhD Thesis, CNRS, LINA, Université de Nantes, France, pages 9-10.

[13] Barták et al, 1999. Roman Barták, Constraint programming: in the pursuit of the holy grail, In proceedings of WDS99 (invited lecture), Prague, pp. 7, June.1998.

[14] Marinescu, R y Dechter, R. "AND/OR Tree Search for Constraint Optimization". In the 6th International Workshop on Preferences and Soft Constraints, of the Tenth International Conference on Principles and Practice of Constraint Programming. CP´2004.

[15] Sabin, D. y Freuder, E.Contradicting conventional wisdom in constraint satisfaction. En Proceedings of European Conference on AI (ECAI´94), páginas 125-129, 1994.

[16] Haralick, R. y Elliot, G. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, 14:263-313, 1980.

[17] Dechter, G.MIT Encyclopedia of Congnitive Science: Constraint Satisfaction (MITECS), article code 26. Department of Computer and Information Science, University of California, USA, páginas 10, 11. 1998.

[18] Hunter Frost, Daniel, (1997). Algorithms and Heuristics for Constraint Satisfaction Problems, *University of California*, 36-39.

[19] Freuder, E. C. (1982). A sufficient condition for backtrack-free search. JACM, 21(11):958-965, 1982.

[20] Dechter, Rina and Itay Meiri (1994). Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. Artificial Intelligence, 68:211-241.

[21] N.Keng y D.Yun. A planning/scheduling methodology for the constrained resources problem. In Proceeding of International Joint Conference on Artificial Intelligence, IJCAI-89, pages 999-1003, 1989.

[22] P.A. Geelen. Dual viewpoint heuristic for binary constraint satisfaction problems. *In proceeding of European Conference of Artificial Intelligence*. (ECAI´92), pages 31-35, 1992.

[23] F. Koenigsberger, The Use of Group Technology in the Industries of Various Countries. H. Hodges, Technology in the Ancient World, Penguin Books, 1971. The Bible, Ecclesiastes 1.9.

[24] V. Cordova. (2007). Estudio para la implantación de una celda mecánica. *Instituto Politécnico Nacional , México*. Páginas 10-19.

[25] P. Medina, E. Cruz, M. Pinzón (2010). *Generación de Celdas de Manufactura Usando un Algoritmo de Ordenamiento Binario*. Scientia et Technica Año XVI, No 44, Abril de 2010. Universidad Tecnológica de Pereira. ISSN 0122-1701. Páginas 106-108.

[26] Sitio web oficial del proyecto de ECL^{i}PS^e CLP, *(Visited, 10/11/2010)*. Available at http://eclipseclp.org/

[27] Krzysztof R. APT and Mark Wallace, *Constraint Logic Programming using ECL^{i}PS^e* . Cambridge University Press, pages 13-209

[28] Mark Wallace. Constraint programming. Chapter 17 of, *The Handbook of Applied Expert Systems*, Section four, CRC Press, 1997.

[29] Official website of ECL^{i}PS^e Solver, Manual Reference *(Visited, 07/08/2011)*. Available at http://www.eclipseclp.org/doc/bips/lib/ic/search-6.html

[30] Official website of Choco Solver, *(Visited, 02/05/2011)*. Available at http://www.choco-constraints.net

[31] Xiangyong Li, M.F.Baki e Y.P.Aneja (2010). *An Ant Colony Optimization Metaheuristics for Machine-Part Cell Formations Problems.* International Journal Computers & Operations Research, 37(2010), 2071-2081.

[32] R. Soto, H. Kjellerstrand, J. Gutiérrez, A. López, B. Crawford, and E. Monfroy. Solving Manufacturing Cell Design Problems Using Constraint Programming. In proceedings of the 25th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE), pages 400-406, LNCS 7345, Springer, 2012.

Anexos

A.- Código fuente del MCDP en ECL^{i}PS^e

```
:-lib(ic).

:-lib(ic_global).

:-lib(ic_search).

:-lib(branch_and_bound).

:-lib(lists).

:-lib(ic_global).

:-lib(ic_search).

:-lib(listut).


mcdp(Matrix_Machine_Part, Matrix_Machine_Cell, Matrix_Part_Cell) :-

% Implementación del MCDP bajo sintaxis ECLiPSe

      Machines = 5,

      Cells = 2,

      Parts = 7,

      Mmax is 4,

      Sum = 1,

      Matrix_Machine_Part = [](     [](1, 0, 0, 0, 1, 1, 1),

                                    [](0, 1, 1, 1, 1, 0, 0),

                                    [](0, 0, 1, 1, 1, 1, 0),

                                    [](1, 1, 1, 1, 0, 0, 0),

                                    [](0, 1, 0, 1, 1, 1, 0)),
```

```
Matrix_Machine_Part[1..5, 1..7] :: 0..1,

dim(Matrix_Machine_Cell, [Machines,Cells]), % make the Machine and Cell
variables

Matrix_Machine_Cell[1..Machines, 1..Cells] :: 0..1,  % set the domains

( for(J,1,Cells),param(Machines,Cells,Matrix_Machine_Cell,Sum,Mmax) do

(   for(I,1,Machines),param(Machines,Cells,Matrix_Machine_Cell,Mmax,Sum,J)
do

      sum(Matrix_Machine_Cell[I,1..Cells]) #=  Sum,

      sum(Matrix_Machine_Cell[1..Machines,J]) #=<  Mmax

)

),

dim(Matrix_Part_Cell, [Parts,Cells]),                % make the Parts and
Cell variables

Matrix_Part_Cell[1..Parts, 1..Cells] :: 0..1,  % set the domains

( for(I,1,Parts),param(Cells,Matrix_Part_Cell, Sum) do

      sum(Matrix_Part_Cell[I,1..Cells]) #=  Sum

),

(for(K,1,Cells) * for(I,1,Machines)* for(J,1,Parts),

fromto(0,In,Out,TotalCost),

      param(Matrix_Machine_Part,Matrix_Machine_Cell,Matrix_Part_Cell) do

      A is Matrix_Machine_Part[I,J],

      Z is Matrix_Part_Cell[J,K],

      Y is Matrix_Machine_Cell[I,K],

      Out #= In + A*Z*(1-Y)

),

term_variables([Matrix_Machine_Part,Matrix_Machine_Cell,Matrix_Part_Cell]
,Vars),

%flatten_array(Matrix_Machine_Part, Vars),

%flatten_array(Matrix_Machine_Cell, Vars2),

%flatten_array(Matrix_Part_Cell, Vars3),
```

```
%minimize(search(kaka,0,first_fail,indomain_min,complete,[]),TotalCost),

minimize(search(Vars,        0,        input_order,        indomain_max,
complete,[]),TotalCost),

%minimize((labeling(Matrix_Machine_Part),labeling(Matrix_Machine_Cell),la
beling(Matrix_Part_Cell)), TotalCost),

writeln(viajes:TotalCost).
```

B.- Código fuente del MCDP en Choco

```
import static choco.Choco.*;
import choco.cp.model.CPModel;
import choco.cp.solver.CPSolver;
import choco.cp.solver.constraints.*;
import choco.cp.solver.*;
import choco.kernel.model.variables.integer.*;
import choco.kernel.*;
import choco.kernel.solver.*;
import choco.kernel.model.*;
import choco.kernel.model.variables.*;
import choco.kernel.model.constraints.*;
import choco.kernel.model.variables.set.*;
import choco.cp.solver.search.integer.varselector.*;
import choco.cp.solver.search.integer.valiterator.*;
import choco.cp.solver.search.integer.valselector.*;
import choco.cp.solver.search.integer.branching.*;

import choco.kernel.model.variables.scheduling.TaskVariable;

import java.io.*;
import java.util.*;
import java.text.*;

public class MCDP14_Benchmark_5x7 {



      public static void main(String[] args) {

            // constants of model

      int maquinas=7; // invariable for the new model
      int partes=11; // invariable for the new model
      int suma=1; // invariable for the new model
      int celdas=7;// invariable for the new model
      int mmax=3;//variable
```

```
      // matrix benchmark variable matrix_machine_part

      int [][] matrix_machine_part = {{1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1},
                                      {1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0},
                                      {0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0},
                                      {0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0},
                                      {0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0},
                                      {0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1},
                                      {0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0}};



// start CP model

      Model m = new CPModel();

// model CP variables

IntegerVariable [][] matrix_machine_cell = new
IntegerVariable[maquinas][celdas];
IntegerVariable [][] matrix_part_cell = new
IntegerVariable[partes][celdas];


// define domain matrix variable matrix_machine_cell

            for(int i = 0; i < maquinas; i++) {
                  for(int j = 0; j < celdas; j++) {
                      matrix_machine_cell[i][j]= makeIntVar("x_" + i +
"_" + j, 0, 1 );
                  }
          }

// first constraint: matrix_machine_cell sum rows = 1

      for(int i=0;i<maquinas;i++){

            m.addConstraint(eq(sum(matrix_machine_cell[i]), 1));

      }

// second constraint: max machines number per cell at matrix_machine_cell

for(int i = 0; i < celdas; i++) {
                  ArrayList<IntegerVariable> col = new
ArrayList<IntegerVariable>();
                  int j=0;
                  //System.out.println("j: " +  j);
                  for(j = 0; j < maquinas; j++) {
                        //System.out.println("j: " +  j);

                      col.add(matrix_machine_cell[j][i]);
                  }
                  m.addConstraint(leq(sum(col.toArray(new
IntegerVariable[1])), mmax));
}
```

```
//define domain matrix_part_cell

for(int i = 0; i < partes; i++) {
        for(int j = 0; j < celdas; j++) {
              matrix_part_cell[i][j]= makeIntVar("x_" + i + "_" + j, 0, 1
);
            }
}

// third constraint: sum rows matrix_part_cell = 1

for(int i=0;i<partes;i++){

             m.addConstraint(eq(sum(matrix_part_cell[i]), suma));
}


/* objective function */


//array for save the solutions

IntegerVariable z = makeIntVar("z", 0, 10000);
            IntegerVariable[] z_array_1 = new
IntegerVariable[maquinas*maquinas*partes];

// define domain for auxiliary array in order to apply the sum

for(int i = 0; i < maquinas*maquinas*partes; i++) {

      z_array_1[i] = makeIntVar("z_array_1" + i, 0, 1);

}

// start objective function

            int contador=0;

            for(int k=0;k<celdas;k++){

                  for(int i=0;i<maquinas;i++){

                        for(int j=0;j<partes;j++){

                              //z_array_1[contador] = makeIntVar("z_" + i
+"_" + k, 0, 1);

      m.addConstraint(eq(mult(mult(matrix_machine_part[i][j],matrix_part_
cell[j][k]),minus(1,matrix_machine_cell[i][k])),z_array_1[contador]));

                              contador= contador+1;

                  }

            }
      }
```

```
      m.addConstraint(eq(sum(z_array_1), z));


/*Our solver */

      Solver s = new CPSolver();


/* Read the Model */

      s.read(m);

/* Statistics of Model */

s.monitorTimeLimit(true);
s.monitorBackTrackLimit(true);
s.monitorNodeLimit(true);
s.monitorFailLimit(true);

/* We do Statistics for the Solution */

CPSolver.setVerbosity(CPSolver.SOLUTION);

/* Search Strategy 1 - Common Strategy with ECLiPSe */

//s.addGoal(new AssignVar(new MostConstrained(s), new MinVal()));
//s.addGoal(new
choco.cp.solver.search.integer.branching.AssignOrForbidIntVarVal(new
MostConstrained(s), new MinVal()));
//s.addGoal(new AssignVar(new MostConstrained(s), new MidVal()));
//s.addGoal(new
choco.cp.solver.search.integer.branching.AssignOrForbidIntVarVal(new
MostConstrained(s), new MidVal()));
//s.addGoal(new AssignVar(new MostConstrained(s), new MaxVal()));
//s.addGoal(new
choco.cp.solver.search.integer.branching.AssignOrForbidIntVarVal(new
MostConstrained(s), new MaxVal()));


/* Search Strategy 2 - Own Choco*/

//s.addGoal(new AssignVar(new MinDomain(s), new MinVal()));
//s.addGoal(new AssignVar(new MinDomain(s), new IncreasingDomain()));
//s.addGoal(new AssignVar(new MinDomain(s), new DecreasingDomain()));
//s.addGoal(new AssignVar(new MinDomain(s), new MidVal()));
//s.addGoal(new AssignVar(new MinDomain(s), new MaxVal()));
//s.addGoal(new AssignVar(new MinDomain(s), new RandomIntValSelector()));


/* Search Strategy 3 - Own Choco*/

//s.addGoal(new AssignVar(new DomOverDeg(s), new MinVal()));
//s.addGoal(new AssignVar(new DomOverDeg(s), new MidVal()));
//s.addGoal(new AssignVar(new DomOverDeg(s), new MaxVal()));
```

```
/* Search Strategy 4 - Own Choco*/
//s.addGoal(new AssignVar(new RandomIntVarSelector(s), new MinVal()));
//s.addGoal(new AssignVar(new RandomIntVarSelector(s), new MidVal()));
//s.addGoal(new AssignVar(new RandomIntVarSelector(s), new MaxVal()));

/* Search Strategy 5 - Own Choco*/

//s.addGoal(new AssignVar(new StaticVarOrder(s), new MinVal()));
//s.addGoal(new AssignVar(new StaticVarOrder(s), new MidVal()));
//s.addGoal(new AssignVar(new StaticVarOrder(s), new MaxVal()));

/* Search Strategy 6 - Own Choco*/

//s.addGoal(new AssignVar(new MaxDomain(s), new MinVal()));
//s.addGoal(new AssignVar(new MaxDomain(s), new MidVal()));
//s.addGoal(new AssignVar(new MaxDomain(s), new MaxVal()));

/* Search Strategy 7 - Own Choco*/

//s.addGoal(new AssignVar(new MaxRegret(s), new MinVal()));
//s.addGoal(new AssignVar(new MaxRegret(s), new MidVal()));
//s.addGoal(new AssignVar(new MaxRegret(s), new MaxVal()));




/* Other Heuristics for to try the behavior*/


//s.attachGoal(new DomOverWDegBranching(s, new DecreasingDomain()));
//BranchingFactory.minDomMinVal(s);
//s.addGoal(new AssignVar(new MinDomain(s), new DecreasingDomain()));
//s.attachGoal(new AssignVar(new MinDomain(s), new IncreasingDomain()));

/* Heuristics Variable Selector */

//s.setVarIntSelector(new MinDomain(s)); // the best?
//s.setVarIntSelector(new DomOverDeg(s)); // not so bad
//s.setVarIntSelector(new DomOverDynDeg(s)); // por evaluar: hasta 1 hora
            //s.setVarIntSelector(new MostConstrained(s)); // worse than
DomOverDeg
            //s.setVarIntSelector(new RandomIntVarSelector(s)); // not
good after 20 minutes
 //s.setVarIntSelector(new MinDomain(s)); // good
            //s.setVarIntSelector(new StaticVarOrder(s)); // the best?
            //s.setVarIntSelector(new MaxDomain(s)); // to check , but
after 10 minutes , bad
//s.setVarIntSelector(new MaxRegret(s)); // so good
//s.setVarIntSelector(new MaxValueDomain(s)); // evaluar
//s.setVarIntSelector(new MinValueDomain(s)); // evaluar

/* Heuristics Value Iterator Own Choco */

        //s.setValIntIterator(new DecreasingDomain()); // no good
        //s.setValIntIterator(new IncreasingDomain()); // worse than
DecreasingDomain
```

```
  /* Heuristics Value Selector */

//s.setValIntSelector(new MinVal());
//s.setValIntSelector(new MaxVal());
//s.setValIntSelector(new MidVal());
//s.setValIntSelector(new RandomIntValSelector()); // not, bad...


/* solve the problem */
s.minimize(s.getVar(z), true); // minimize

/* solve at least one solution*/

//s.solve();

s.printRuntimeStatistics();
System.out.println("Soluciones  Encontradas: " + s.getNbSolutions());
//System.out.println(s.pretty());


if (s.isFeasible()) {
            //System.out.println(s.pretty());
            System.out.println("Cost Exceptional Numbers: " +
s.getVar(z).getVal());


        }  else {
            System.out.println("No solution."); }



            /* Print the values of variables */

            for(int i = 0; i < maquinas; i++){
                  for(int j = 0; j < celdas; j++){


      //System.out.print(s.getVar(matrix_machine_cell[i][j]).getVal()+"")
;

      System.out.print(s.getVar(matrix_machine_cell[i][j]).pretty());


                  }

            System.out.println();
            }

            System.out.println("\n");

            for(int i = 0; i < partes; i++){
                  for(int j = 0; j < celdas; j++){
```

```
      System.out.print(s.getVar(matrix_part_cell[i][j]).getVal()+"");

      //System.out.print(s.getVar(matrix_part_cell[i][j]).pretty());


                  }

            System.out.println();
}


            System.out.println("Cost Exceptional Numbers: " +
s.getVar(z).getVal());

            System.out.println();
            System.out.println();
            System.out.println();
            System.out.println();

      }

}
```