PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

"ESTRATEGIAS DE SELECCIÓN DE VARIABLES Y VALORES EN LA RESOLUCIÓN DE PROBLEMAS COMBINATORIALES UTILIZANDO PROGRAMACIÓN CON RESTRICCIONES"

MARY CLAUDIA ARANDA CABEZAS

INFORME FINAL DEL PROYECTO
PARA OPTAR AL TÍTULO PROFESIONAL DE
INGENIERO CIVIL EN INFORMÁTICA

Diciembre, 2006

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

"ESTRATEGIAS DE SELECCIÓN DE VARIABLES Y VALORES EN LA RESOLUCIÓN DE PROBLEMAS COMBINATORIALES UTILIZANDO PROGRAMACIÓN CON RESTRICCIONES"

MARY CLAUDIA ARANDA CABEZAS

Profesor Guía: Broderick Crawford Labrin

Profesor Co-referente: Nibaldo Rodríguez Agurto

Carrera: Ingeniería Civil en Informática

Diciembre, 2006

Dedicatoria

Dedicado a mis padres,
por su infinito amor, apoyo y comprensión.

Mary Claudia Aranda Cabezas

Agradecimientos

Quisiera agradecer a mi profesor guía, Broderick Crawford, por la constante ayuda, dirección y estímulo brindado, quien con sus consejos a contribuido a mi formación profesional y personal. Deseo agradecer a mi familia por el apoyo incondicional y desinteresado, por las palabras justas que llegan sin pedirlas y cuando más se necesitan…gracias por estar ahí, siempre.

Mary Claudia Aranda Cabezas

Resumen

La Programación con Restricciones es un paradigma para la representación y resolución de problemas combinatoriales. Muchos de estos problemas pueden modelarse como Problemas de Satisfacción de Restricciones y resolverse utilizando técnicas de Propagación de Restricciones y Enumeración. La propagación poda el árbol de búsqueda eliminando valores que no pueden participar en la solución y la enumeración consiste en dividir un CSP en dos CSPs más pequeños hasta que se alcance un CSP fallado o solucionado. En esta última, las Estrategias de Enumeración (Heurísticas de Selección de Variables y Valores) pueden ser usadas para reducir el costo de búsqueda en el proceso de resolución, por lo tanto en este trabajo se presenta una comparación entre el desempeño de diferentes heurísticas de selección de variables y valor, proporcionando un entendimiento profundo del efecto de este tipo de heurísticas en la resolución de problemas combinatoriales específicos (benchmarks).

Abstract

The Constraint Programming is a paradigm for the representation and solve of a wide variety of combinatorial problems. Many of these problems can be modeled like Constraint Satisfaction Problems and solving using Constraint Propagation and Enumeration Techniques. The propagation prunes the search tree by eliminating values that can’t participate in the solution and enumeration consists of dividing the original CSP in two smaller CSPs until a failed or solved CSP is reached. In this last one, the Enumeration Strategies (Variable and Value Selection Heuristics) can be used to reduce the search cost in the resolution process. In this work is presented a comparison between the performance of different Variable and Value Selection Heuristics, providing a deeper understanding of the effect of this type of heuristic in the resolution of specific combinatorial problems (benchmarks).

Índice

Índice de Figuras

Índice de Tablas

Lista de Abreviaturas

- ACM : Association for Computer Machinery
- BT : Backtracking
- COP : Constraint Optimization Problem
- CP : Constraint Programming
- CSP : Constraint Satisfaction Problem
- FF : First Fail
- FSS : Filtered Survivable Schedules
- GT : Generate and Test
- MC : Maximum Cardinality
- MD : Maximum Dregree
- MW : Minimum Width
- ORR : Operation Resource Reliance
- SDF : Smallest Domain First
- MeVal : Heurística de Selección de Valor, la cual selecciona el menor valor del dominio
- MaVal : Heurística de Selección de Valor, la cual selecciona el mayor valor del dominio
- MedVal : Heurística de Selección de Valor, la cual selecciona el valor medio del dominio
- MMedVal : Heurística de Selección de Valor, la cual selecciona el valor inmediatamente mayor al valor medio del dominio
- DMi : Heurística de Selección de Variable, la cual selecciona aquella variable con dominio mínimo

- DMa : Heurística de Selección de Variable, la cual selecciona aquella variable con dominio máximo
- BOr : Heurística de Selección de Variable estática, la cual selecciona una variable en base a un orden pre-establecido

CAPÍTULO 1

Introducción

1.1 Introducción

La Programación con Restricciones, en inglés Constraint Programming (CP), ha generado gran expectación entre expertos de muchas áreas debido a su potencial en la resolución de problemas reales. Esto no es de extrañar, ya que muchas de las decisiones que tomamos a la hora de resolver problemas cotidianos están sujetas a restricciones. Problemas tan cotidianos como cocinar o comprar un auto pueden depender tanto de aspectos interdependientes como conflictivos, cada uno de los cuales está sujeto a un conjunto de restricciones y encontrar una solución que dé satisfacción plena a alguno de estos aspectos, puede que no sea tan apropiada para otros.

La Programación con Restricciones ha sido definida como una tecnología de Software utilizada para describir y solucionar de manera efectiva grandes y complejos problemas, particularmente combinatoriales [1][2]. La idea de la Programación con Restricciones es resolver problemas mediante la declaración de restricciones propias del problema y encontrar soluciones que satisfagan todas las restricciones.

Para la resolución de muchos de estos problemas combinatoriales existentes en la vida real pueden utilizarse diversas técnicas, una de las más simples pero también la más costosa en términos del esfuerzo requerido, es encontrar una solución empleando la "fuerza bruta", técnica conocida en Programación con Restricciones como enumeración completa o exhaustiva, que consiste en generar asignaciones completas (una asignación

completa consiste en instanciar todas y cada una de las variables del problema con algún valor de su dominio) y comprobar si son solución, es decir, si satisface o no todas las restricciones del problema. Sin embargo, la Programación con Restricciones busca desarrollar formas de resolución que permitan reducir la cantidad de búsqueda a niveles aceptables. Es por esto que en el presente trabajo se hace un estudio acabado de las heurísticas de selección de variable y valor, y como ellas ayudan a reducir los niveles de búsqueda. Evaluando el desempeño de diferentes heurísticas con objeto de establecer características de su comportamiento en la resolución de diferentes problemas y de esta manera establecer una base teórica en la resolución "inteligente" o más eficiente de problemas combinatoriales. En el desarrollo del presente trabajo los problemas a resolver son denominados por la literatura como "puzzles", estos han sido utilizados en distintas investigaciones [6][12][21][22] como benchmarks y en el caso particular de este trabajo se utilizan para ilustrar el efecto de las heurísticas de selección de variable y valor en su resolución, objetivo principal de esta investigación.

Para la implementación y evaluación de las heurísticas se utiliza el sistema de programación Mozart, el cual está basado en el lenguaje de programación Oz [3][10][13].

El sistema Mozart fue desarrollado por el consorcio Mozart formado por grupos de investigación del Laboratorio de Sistemas de Programación de Saarbrucken, del Instituto Sueco para las ciencias de la Computación y de la Universidad Católica de Lovaina [4], dicho sistema cuenta con diversas características que lo hacen destacar por sobre sus pares a la hora de resolver tareas complejas. La utilización de Mozart en el presente trabajo ha sido motivada principalmente por estar basado en un lenguaje multiparadigma el cual permite utilizar distintos enfoques y de esta manera realizar una implementación más transparente. Además, posee una enumeración flexible donde el programador puede personalizar las estrategias de enumeración combinando distintas heurísticas de selección de variable y valor.

1.2 Objetivos

1.2.1 Objetivo General

Diseñar, implementar y evaluar el desempeño de diferentes heurísticas de selección de variable y valor en la resolución de problemas combinatoriales con Programación con Restricciones.

1.2.2 Objetivos Específicos

- Comprender el paradigma de Programación con Restricciones.

- Comprender el fundamento teórico y la relevancia de las heurísticas de selección de variable y valor en la resolución de problemas combinatoriales.

- Diseñar e implementar heurísticas de selección de variable y valor mediante la utilización del sistema de programación Mozart.

- Evaluar el rendimiento de las heurísticas de selección de variable y valor en la resolución de benchmarks de problemas combinatoriales clásicos, tipificados bajo la categoría de puzzles.

1.3 Organización del Documento

En este trabajo se aborda la resolución de problemas combinatoriales mediante la técnica completa de Programación con Restricciones, que involucra propagación de restricciones y enumeración. Concretamente se evalúan los efectos de las heurísticas de selección de variable y valor en la resolución de problemas de satisfacción de restricciones. La exposición se ha estructurado en seis capítulos con los contenidos que se detallan a continuación.

El capítulo 1 corresponde básicamente a una contextualización del tema tratado en este informe, en el cual se reflejan la motivación y objetivos del trabajo, y se describe de manera breve la metodología utilizada para el desarrollo del mismo.

Los capítulos 2 y 3 pretenden introducir los conceptos fundamentales en los que se basa el trabajo realizado. El capítulo 2 está dedicado al estudio de la Programación con Restricciones, esencialmente al planteamiento de los problemas como problemas de satisfacción de restricciones y a las técnicas de resolución, centrándose principalmente en describir la técnica de resolución completa que combina propagación de restricciones y enumeración. Por su parte, el capítulo 3 supone el inicio del tema central de este trabajo. En él se presenta la descripción de las heurísticas de selección de variable y valor, los conceptos fundamentales asociados a tales heurísticas y el funcionamiento general de éstas.

El capítulo 4 se dedica a exponer el funcionamiento de la plataforma de desarrollo a utilizar y cómo ésta permite la implementación de las técnicas utilizadas en el presente informe para la resolución de los problemas combinatoriales, describiendo además, cómo se implementan las heurísticas de selección de variable y valor que darán vida a las estrategias de enumeración.

Los capítulos 5 y 6 se dedican a la parte de implementación computacional. En el capítulo 5 se describen y modelan los problemas a resolver, se presenta una descripción de las heurísticas a implementar en su resolución y los indicadores de desempeños utilizados para evaluar dichas heurísticas. Además, se muestra parte de la representación de tales modelos en el lenguaje de programación utilizado en la implementación. Por su parte, en el capítulo 6 se lleva a cabo la evaluación de las distintas estrategias de enumeración formadas con las heurísticas de selección de variable y valor, las cuales fueron utilizadas en el desarrollo de los problemas planteados en el capítulo 5.

CAPÍTULO 2

Programación con Restricciones

2.1 Definición de Programación con Restricciones

La Programación con Restricciones es el estudio de sistemas computacionales basados en restricciones [1][2] y la idea central es resolver problemas mediante la declaración de restricciones propias del problema y encontrar soluciones, las cuales corresponden a una asignación de valor a cada variable de forma tal que se satisfagan todas las restricciones. Este paradigma de programación puede dividirse en dos ramas: la satisfacción de restricciones y la resolución de restricciones. Ambas ramas comparten la misma terminología pero el origen y técnicas de resolución son diferentes. La satisfacción de restricciones trata con problemas de dominio finito, en cambio, la resolución de restricciones está orientada principalmente a problemas de dominios infinitos [2] o más complejos. Por otra parte, la resolución de restricciones en lugar de utilizar métodos combinatoriales como lo hace la satisfacción de restricciones, utiliza algoritmos basados en técnicas matemáticas tales como Series de Taylor o métodos de Newton [29].

En el desarrollo del presente proyecto y derivado del párrafo anterior, se entenderá la Programación con Restricciones como un conjunto de técnicas para la resolución de problemas de satisfacción de restricciones.

2.2 Problemas de Satisfacción de Restricciones

Un problema de satisfacción de restricciones, en inglés Constraint Satisfaction Problem (CSP), consiste en una secuencia de variables $X = x_1, x_2,\ldots, x_n$, cada una con su respectivo dominio $D_{x_1} \ldots D_{x_n}$, y un conjunto finito C de restricciones que restringen los valores que las variables pueden tomar simultáneamente [14], donde la tarea es asignar un valor a cada variable satisfaciendo todas las restricciones. La notación más general [25] utilizada para un CSP consiste en lo siguiente:

$$\langle C; x_1 \in D_{x_1}, \ldots, x_n \in D_{x_n} \rangle \qquad (2.1)$$

A modo de ejemplo considere el siguiente CSP:
$\langle x_1 < x_2, x_2 \neq x_3; x_1 \in \{1,2,3,4\}, x_2 \in \{1,2,3\}, x_3 \in \{2,3\} \rangle$, donde la restricción $x_1 < x_2$ denota el subconjunto {(1, 2), (1,3), (2, 3)} de $D_{x_1} \times D_{x_2}$.

2.2.1 Notación

Antes de entrar en más detalle en los problemas de satisfacción de restricciones y el desarrollo del presente trabajo, a continuación se resume la notación básica a utilizar.

- Variable: Se denotarán con las últimas letras del alfabeto en cursiva con un subíndice, el cual puede ser un número entero o letras seleccionadas desde la mitad del alfabeto. Ejemplo: x_1, x_i, y_1, y_j

- Dominio: El dominio de la variable x_i se denotará por D_{x_i}. Por su parte, los valores individuales del dominio se denotarán con las primeras letras del alfabeto en cursiva, las cuales también pueden ir seguidas por un subíndice. Ejemplo: a, b, a_1, b_i

2.2.2 Conceptos

En el presente ítem se definirán conceptos básicos que son necesarios para la comprensión de un problema de satisfacción de restricciones, los cuales también se utilizarán a lo largo de este trabajo.

Asignación: Una asignación o instanciación de variable es un par (x_i, a) que representa la asignación del valor a a la variable x_i. Una instanciación de un conjunto de variables es una tupla de pares ordenados.
Una tupla $((x_1, a_1),\ldots, (x_i, a_i))$ es localmente consistente si satisface todas las restricciones formadas por las variables de la tupla [1].

Solución: Es una asignación de un valor de su dominio a cada variable de forma tal que se satisfaga cada restricción, es decir, una solución es una tupla consistente que contiene todas las variables del problema. Por otra parte, una solución parcial es una tupla consistente que contiene solo algunas variables del problema [1][14]. Se puede querer solucionar un CSP para:

- Determinar si existe una solución, sin necesariamente conocerla.
- Encontrar una o más soluciones

Aridad: La aridad de una restricción corresponde al número de variables que componen dicha restricción [1].

- restricción unaria, es aquella restricción que consta de una sola variable
  Ejemplo: $x_1 < 7$
- restricción binaria, es aquella restricción que consta de dos variables
  Ejemplo: $x_3 + x_4 \neq 3$
- restricción no binaria o n-aria, es una restricción que involucra un número arbitrario de 3 o más variables.

Ejemplo: $x_1 + 3x_2 - x_3 + x_4 \leq 10$

Un CSP que sólo tiene restricciones unarias o binarias es denominado CSP binario [30]. En general, la mayoría de los investigadores que trabaja con Problemas de Satisfacción de Restricciones centran su atención en CSP binarios por la simplicidad asociada a estos en comparación con los CSP no binarios, y además porque todo CSP no binario puede transformarse en un CSP binario equivalente [32][33]. Principalmente hay dos técnicas para trasladar las restricciones no binarias a binarias: la codificación dual y la codificación de variables ocultas [31].

Espacio de Búsqueda: Corresponde al producto cartesiano de los dominios de las variables, es decir, si se tiene un conjunto de n variables $\{x_1, x_2, \ldots, x_n\}$ cada una con dominio $D_{x_1}, D_{x_2}, ..., D_{x_n}$ respectivamente, entonces el espacio de búsqueda es igual a $D_{x_1} \times D_{x_2} \times ... \times D_{x_n}$. Por su parte, el tamaño del espacio de búsqueda corresponde a la cardinalidad del producto cartesiano:

$$Card(D_{x_1} \times D_{x_2} \times ... \times D_{x_n}) = Card(D_{x_1})Card(D_{x_2})...Card(D_{x_n})$$

Espacio de Solución: Corresponde a un subconjunto del espacio de búsqueda que satisface todas las restricciones.

Equivalencia: Se dice que dos problemas de satisfacción de restricciones P_1 y P_2 son equivalentes si tienen el mismo espacio solución.

CSOP: Constraint Satisfaction Optimization Problem, es un CSP estándar, junto con una función objetivo f a ser optimizada [14][25]. El valor de la función objetivo a menudo es representado por una variable z, junto con la "restricción" maximizar z o minimizar z para la maximización o minimización del problema, respectivamente [19]. El proceso para pasar de un CSP a un CSOP se describe a continuación:

Asumir que se quiere minimizar f. El valor de la función objetivo es representado por la variable z, entonces cuando se encuentra una solución para el CSP el correspondiente valor de z dice $z = b$, el cual sirve como una cota superior para el valor óptimo de f. Finalmente se agrega la restricción $z < b$ a todos los CSPs en el árbol de búsqueda y se continua.

2.2.3 Resolución de Problemas de Satisfacción de Restricciones

La resolución de un problema de satisfacción de restricciones consta de dos fases diferentes [1]:

1. Modelar el problema como un problema de satisfacción de restricciones. Dicho modelo expresa el problema mediante la sintaxis de un CSP, es decir, mediante un conjunto de variables, dominios y restricciones. (Ver sección 2.2)

2. Procesar el problema de satisfacción de restricciones resultante, para esto se utilizan [1][10][11]:

- ✓ Técnicas de consistencia (Propagación): se trata de técnicas basadas en la eliminación de valores inconsistentes de los dominios de las variables.

- ✓ Algoritmos de búsqueda: se basan en la exploración sistemática del espacio de solución.

Las técnicas de consistencia permiten deducir información del problema, aunque generalmente se utilizan en combinación con técnicas de búsqueda, ya que reducen el espacio de soluciones y los algoritmos de búsqueda exploran dicho espacio.

2.2.3.1 Técnicas de Consistencia

Una de las principales dificultades que suelen encontrarse en los algoritmos de búsqueda es la aparición de inconsistencias locales. Las inconsistencias locales son valores individuales (o combinación de valores) de las variables que no pueden participar en la solución porque no satisfacen alguna propiedad de consistencia, es decir, no satisfacen alguna restricción.

En la literatura pueden encontrarse diferentes niveles de consistencia local [1][2][11]:

- Consistencia de Nodo: Forzar este nivel de consistencia asegura que todos los valores en el dominio de una variable satisfacen todas las restricciones unarias sobre esa variable.
  - $\forall x_i \in X, \forall C_{x_i} \ \forall a \in D_{x_i}$: a satisface C_{x_i}

Ejemplo: $X = \{x_1, x_2\}$

$D_{x_1} = \{1, 2, 3, 4, 5\}$ y $D_{x_2} = \{1, 2, 3, 4, 5\}$

$C = \{x_1 \leq 3, x_2 \geq 1, x_1 \neq x_2\}$.

Es posible apreciar en la Figura 2.1 que el CSP planteado no es nodo consistente, esto debido a que el nodo x_1 no satisface la consistencia de nodo, es decir, x_1 posee valores en su dominio que no satisfacen la restricción unaria $x_1 \leq 3$.

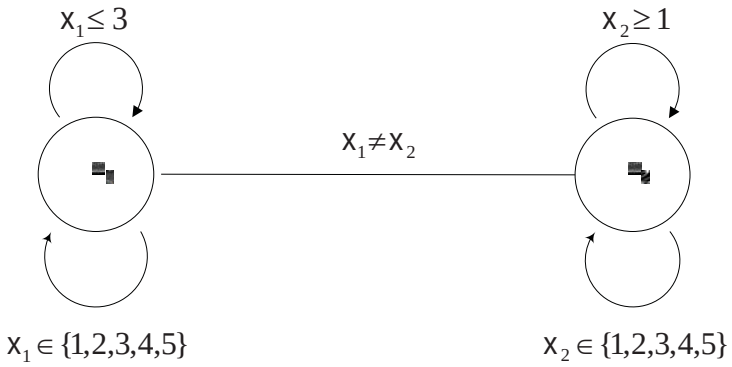


Figura 2. 1 Ejemplo de No Consistencia de Nodo

De esta manera, para que el CSP anterior sea nodo consistente, solo bastaría con eliminar los valores del dominio de la variable x_1 que no cumplen la restricción unaria (4 y 5), quedando el grafo de la figura anterior de la siguiente forma:

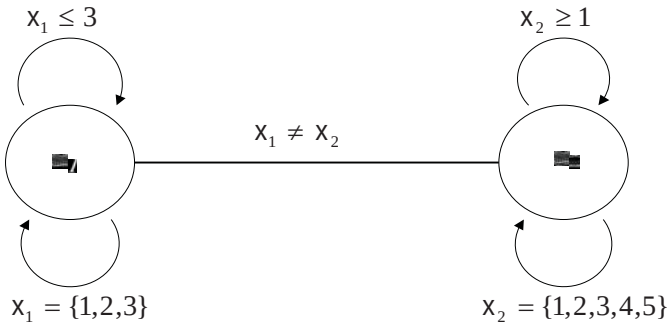


Figura 2. 2 Ejemplo de Consistencia de Nodo

- Consistencia de Arco: Se dice que un CSP es arco consistente si toda restricción binaria es arco consistente [11]. Una restricción binaria C sobre las variables x_1 y x_2, cuyos dominios son D_{x_1} y D_{x_2} respectivamente, es arco consistente si:
  - $\forall a \in D_{x_1}\ \exists b \in D_{x_2} : (a,b)$ satisface C
  - $\forall b \in D_{x_2}\ \exists a \in D_{x_1} : (a,b)$ satisface C

  De esta manera, una restricción binaria es arco consistente si cada valor en cada dominio tiene un soporte en el otro dominio, donde b es llamado un soporte para a si el par (a, b) satisface la restricción [11].

  Un caso particular de la arco consistencia es la arco consistencia dirigida [11][1], donde dado un orden linear $\rightarrow$sobre las variables consideradas, se requiere la existencia de soportes solo en una dirección, esto es:
  Dadas las condiciones:
  - $\forall a \in D_{x_1}\ \exists b \in D_{x_2} : (a,b)$ satisface C, proporcionado el orden $x_1 \rightarrow x_2$
  - $\forall b \in D_{x_2}\ \exists a \in D_{x_1} : (a,b)$ satisface C, proporcionado el orden $x_2 \rightarrow x_1$

  Solo una de ellas necesita ser chequeada.

  Ejemplo: El CSP mostrado en la Figura 2.3, el cual esta constituido por $X = \{x_1, x_2\}$, $D_{x_1} = \{5\ldots10\}$, $D_{x_2} = \{3\ldots8\}$ y $C = \{x_1 < x_2\}$, no es arco consistente porque si se toma el valor 8 en el dominio de x_1, no existe un valor en el dominio de x_2 tal que se satisfaga la restricción $x_1 < x_2$. Por otra parte, el mismo CSP no es direccionalmente arco consistente, ya que si se considera el orden $x_1 \rightarrow x_2$ con $x_1 = 8$, no existe un valor en D_{x_2} que satisfaga la restricción binaria del CSP. Por su parte, si el orden considerado es $x_2 \rightarrow x_1$ con $x_2 = 4$, no hay un valor en D_{x_1} que satisfaga la restricción mencionada.

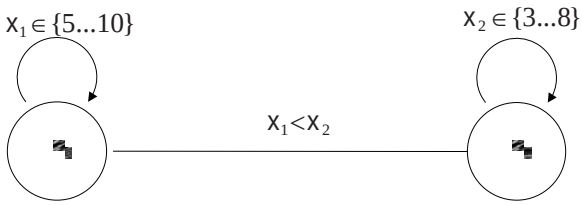


Figura 2. 3 CSP No Arco Consistente

Para que el CSP anterior sea direccionalmente arco consistente siguiendo el orden $x_2 \rightarrow x_1$, es preciso ajustar los dominios de manera tal que cada elemento en el dominio de x_2 tenga un soporte en D_{x_1}, este ajuste es mostrado en la Figura 2.4, donde además es posible apreciar que el CSP aún no es arco consistente. Para cambiar esto último, y alcanzar la arco consistencia los dominios de x_1 y x_2 deben ser reducidos tal como muestra la Figura 2.5.

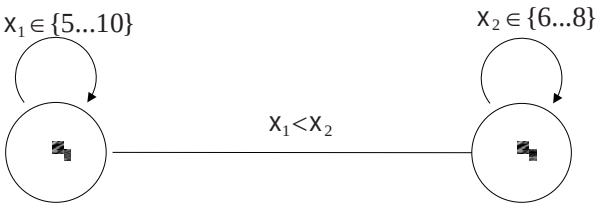


Figura 2. 4 CSP Direccionalmente Arco Consistente bajo el orden $x_2 \rightarrow x_1$

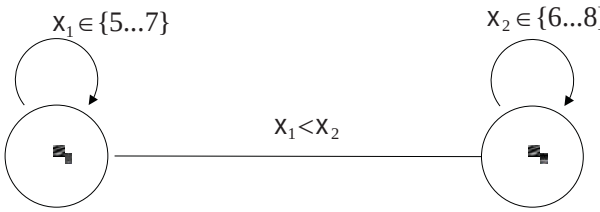


Figura 2. 5 CSP Arco Consistente

- Consistencia de Caminos: Formalmente, dos variables x_i y x_j son camino consistentes con x_k si para cada par de valores (a, b) que satisfacen las restricciones entre x_i y x_j, existe un valor c en el dominio de x_k tal que (a, c) y (b, c) satisfacen las restricciones entre x_i y x_k, y entre x_j y x_k respectivamente [14].

Ejemplo: El CSP formado por los elementos X = {x_1, x_2, x_3}, D_{x_1} = {0, 1, 2, 3, 4}, D_{x_2} = {1, 2, 3, 4, 5}, D_{x_3} = {5, 6, 7, 8, 9, 10}, C = {$x_1 < x_2$, $x_1 < x_3$, $x_2 < x_3$}, y mostrado en la Figura 2.6 no es camino consistente, ya que dado los valores $4 \in D_{x_1}$ y $5 \in D_{x_3}$, no existe un valor $b \in D_{x_2}$ tal que $4 < b$ y $b < 5$.

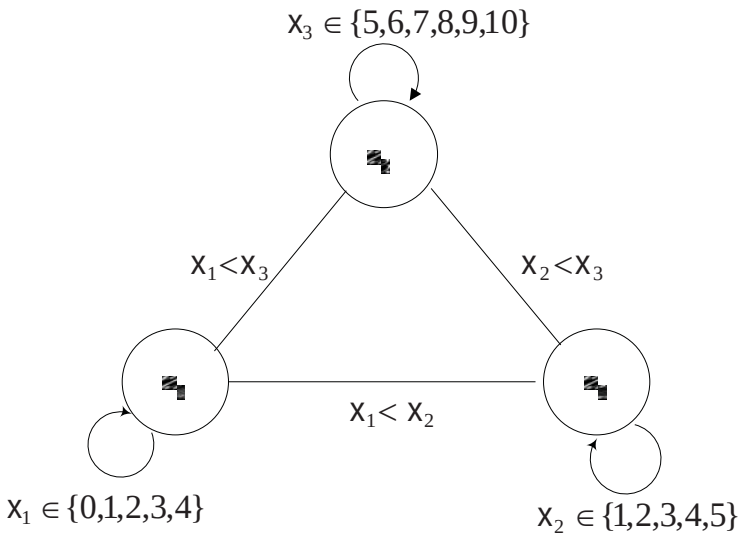


Figura 2. 6 CSP No Camino Consistente

De esta manera, si en el CSP anterior se reduce el dominio de la variable x_3 quedando D_{x_3} = {6, 7, 8, 9, 10} según se muestra en la Figura 2.7, se tendrá un CSP que cumple la consistencia de camino.

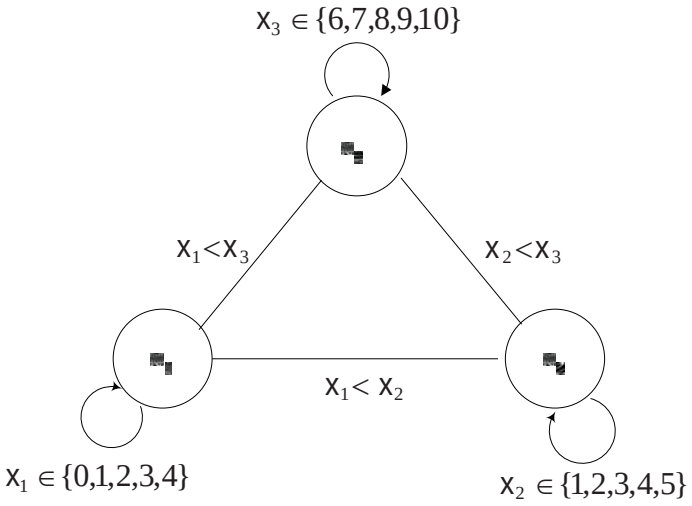


Figura 2. 7 CSP Camino Consistente

2.2.3.2 Algoritmos de Búsqueda

Un Problema de Satisfacción de Restricciones puede ser solucionado utilizando una técnica que consiste básicamente en que las variables son instanciadas, y dicha instanciación se comprueba para ver si satisface todas las restricciones. La versión no incremental de este tipo de técnica se conoce como Generación y Test (GT), el cual de forma sistemática genera todas las posibles asignaciones completas [30]. Cuando finaliza de generar una asignación completa, comprueba si esta asignación es una solución (es decir, comprueba si satisface todas las restricciones). En terminología de árboles, GT es un algoritmo que recorre el árbol primero en profundidad. Por su parte la versión incremental corresponde al conocido Backtracking Cronológico (BT), el cual recorre el árbol utilizando búsqueda primero en profundidad, al igual que GT, pero en cada nueva instanciación comprueba si las instanciaciones parciales ya realizadas son localmente consistentes. Si es así continua con la instanciación de una nueva variable. Por el contrario, si detecta inconsistencia, intenta asignar un nuevo valor a la última variable instanciada, si es posible, y en caso contrario retrocede a la variable asignada inmediatamente anterior [30]. La Figura 2.8 muestra el pseudo código de BT aplicado a la

resolución de un CSP, donde UNLABELLED corresponde a un conjunto de variables sin instanciar. COMPOUND_LABEL es un conjunto de variables ya instanciadas.

Si bien este tipo de métodos son completos, es decir, buscan a través del espacio de las posibles asignaciones de valores a las variables, garantizando que si existe solución ésta será encontrada o en caso contrario demostrando que el problema no tiene solución, su utilización es bastante ineficiente y en general impracticables en problemas grandes.

```
PROCEDIMIENTO BT (UNLABELLED, COMPOUND_LABEL, D, C)
BEGIN
  IF (UNLABELLED = { }) THEN
     return(COMPOUND_LABEL)
  ELSE
     Pick one variable x from UNLABELLED;
    REPEAT
       Pick one value v from Dx;
       Delete v from Dx;
       IF COMPOUND_LABEL + {<x,v>} violates no constraints THEN
          Result = BT (UNLABELLED -ã{x}, COMPOUND_LABEL + {<x,v>}, D, C);
          IF (Result <> NIL) THEN
              return(Result);
          END
      END
    UNTIL (Dx = { });
    return(NIL);
  END
END
```

Figura 2. 8 Algoritmo de búsqueda Backtracking [14]

2.2.3.3 Propagación y División

Las técnicas de búsqueda y algunas de las técnicas de consistencia pueden utilizarse independientemente para solucionar totalmente un problema de satisfacción de restricciones, pero esto rara vez sucede, por lo cual una combinación de ambos acercamientos es la manera más común para solucionar un CSP.

En la técnica de propagación y división, la división corta un CSP en sub-CSPs más pequeños de tal manera que la propagación puede actuar nuevamente. La propagación por su parte busca disminuir el espacio de búsqueda. Este método además de ser una técnica completa de resolución se caracteriza por ser uno de las técnicas de resolución de problemas de satisfacción de restricciones más eficientes.

La división de un CSP puede ser una división de dominio o una división de restricciones. La división de restricciones consiste básicamente en reemplazar una restricción por una restricción más pequeña. Por su parte, la división de dominio consiste en transformar una expresión de dominio en dos o más expresiones de dominio. En [11] se describen tres operaciones de división de dominio, cuya regla de transformación asociada, se representa gráficamente mediante una línea horizontal, sobre dicha línea se ubica una expresión de dominio a ser transformada en dos o más nuevas expresiones de dominio ubicadas bajo la línea horizontal y separadas mediante el símbolo "|".

- Enumeración $\dfrac{x \in D}{x \in \{a\} \mid x \in D - \{a\}}$

- Labeling $\dfrac{x \in \{a_1,\ldots, a_k\}}{x \in \{a_1\} \mid \ldots \mid x \in \{ a_k \}}$

- Bisección $\dfrac{x \in [a, b]}{x \in [a, \frac{a+b}{2}] \mid x \in [\frac{a+b}{2}, b]}$

Hasta este punto se han descrito técnicas de resolución de problemas de satisfacción de restricciones, con lo cual se puede afirmar a modo de resumen que la Programación con Restricciones tiene por meta encontrar una solución (o todas las soluciones) a un CSP dado, o una solución óptima (o todas las soluciones óptimas) a un CSOP dado, o bien demostrar la no existencia de dichas soluciones. Los problemas de satisfacción de restricciones actualmente son solucionados usando técnicas completas e incompletas. Específicamente, la comunidad de Programación con Restricciones utiliza una aproximación completa que alterna fases de Propagación Restricciones y Enumeración [9], donde la Propagación remueve valores inconsistentes del dominio, y la Enumeración [11] consiste en dividir el CSP original en dos CSPs más pequeños hasta que se alcance un CSP fallido o solucionado. Para dividir el dominio de la variable, es decir, iniciar la fase de Enumeración, primero se debe seleccionar una variable y luego decidir cómo se va a dividir su dominio. Este proceso es guiado por heurísticas de selección de variable y heurísticas de selección de valor. De esta manera, la estructura subyacente a la Programación con Restricciones se muestra en la Figura 2.9, donde el interés principal de este trabajo se ha enfocado en la componente de Búsqueda, cuyo marco general posee una estructura algorítmica similar a la mostrada en la Figura 2.10. En este marco general, la Propagación reduce el dominio de las variables, Enumeración consiste en tomar un CSP y dividirlo en dos CSPs más pequeños haciendo uso de la estrategia de enumeración, y proceder por casos es la función que administra la elección de los puntos creados por la enumeración, es decir, corresponde al orden en el cual se consideran los nuevos CSPs. El término fallido indica que no existe una asignación de valor para cada variable de manera que se satisfagan todas las restricciones, por el contrario solucionado indica que a cada variable se le ha asignado un valor de tal menara que se satisfacen todas las restricciones del CSP.

El orden en el cual se consideran los CSPs viene dado por la técnica de búsqueda, donde las más conocidas son Backtracking y Branch and bound (cuando se busca la solución óptima) [11].

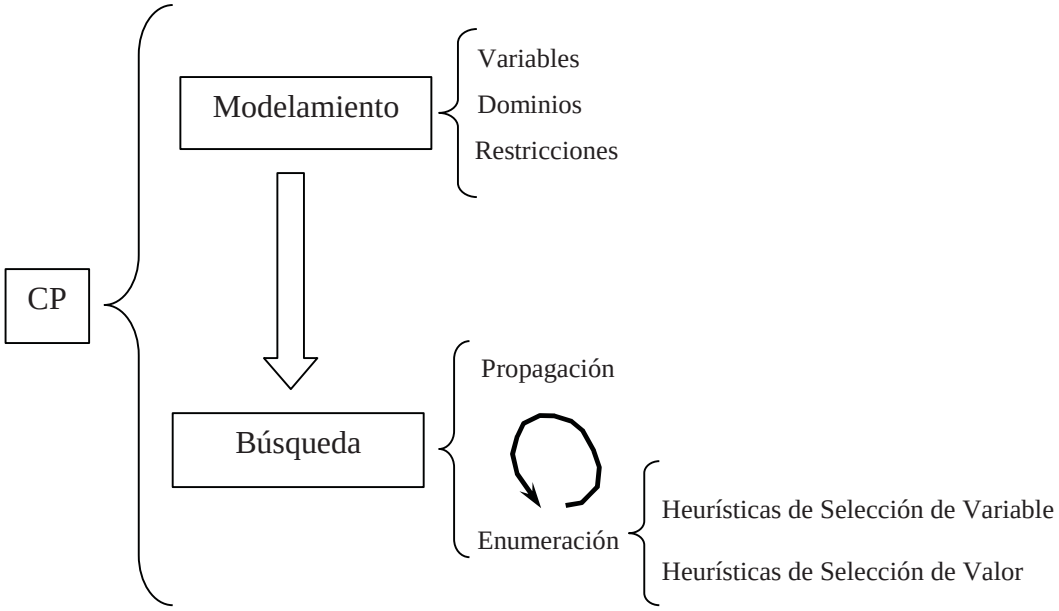


Figura 2. 9 Estructura de Programación con Restricciones.

```
WHILE no solucionado OR fallido
    Propagación
    IF no fallido AND no solucionado THEN
        Enumeración
        Proceder por casos
    END
END
```

Figura 2. 10 Búsqueda

CAPÍTULO 3

Heurísticas de Selección de Variable y Valor

3.1 Definición de Heurística

Una heurística, según se especifica en [11], establece cómo elegir entre muchas alternativas. Desde este punto de vista, una heurística de selección de variable y valor, comúnmente referida en la literatura como heurísticas de ordenación de variable y valor o en inglés variable and value ordering heuristics, consiste en imponer un orden de selección sobre las variables a instanciar y el valor a asignar en la fase de Enumeración.

El orden en el cual las variables y valores son seleccionados tiene un gran impacto sobre el proceso de búsqueda [5][7][8]. Por la importancia que tiene la selección adecuada de las variables y valores, y por el rol que juegan este tipo de heurísticas en dicha selección, el desarrollo del presente capítulo se centrará en conocer diferentes heurísticas y como funciona cada una de ellas.

3.2 Conceptos Relacionados

Para introducir algunas de las heurísticas de ordenación de variables y valores, y permitir una mejor comprensión de su aplicabilidad, en el presente ítem se definen y explican una serie de conceptos relacionados a CSP binarios, los cuales son de principal interés en este trabajo.

- Grafo de Restricciones: corresponde a la representación gráfica que se acostumbra a utilizar para representar un CSP binario, donde cada nodo corresponde a una variable del problema y los arcos muestran las restricciones entre las variables [30][14][11]. A continuación se presenta un ejemplo de CSP binario, cuya representación gráfica puede ser vista en la Figura 3.1.

Ejemplo:

$X = \{x_1, x_2, x_3, x_4\}$

$D = < D_{x_1}, D_{x_2}, D_{x_3}, D_{x_4} >$, donde $D_{x_1} = \{4, 5\}$, $D_{x_2} = \{3, 4, 5\}$, $D_{x_3} = \{1, 2\}$ y $D_{x_4} = \{4, 5\}$

$C = \{x_1 > x_2, x_2 > x_3, x_2 = x_4\}$

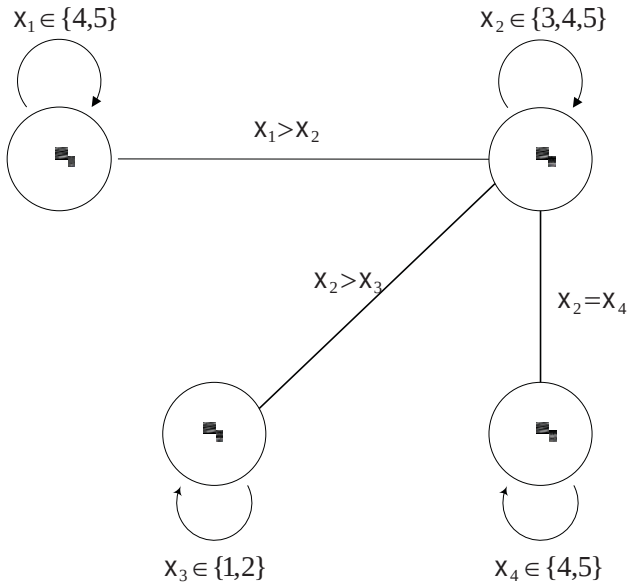


Figura 3. 1 Ejemplo de grafo de restricciones

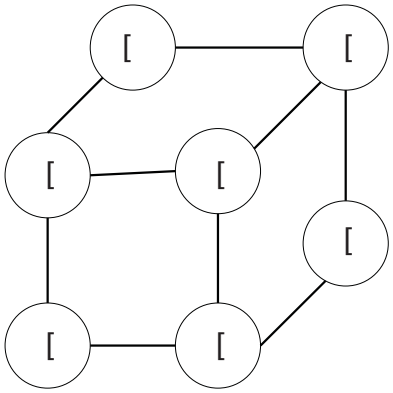
- Grado de un Nodo (x_i): es el número de nodos a los cuales el nodo x_i es adyacente [14].

- Anchura de Nodo (x_i): dado un grafo y un orden de sus nodos, la anchura de un nodo x_i es el número de nodos que están antes y son adyacentes a x_i [14].

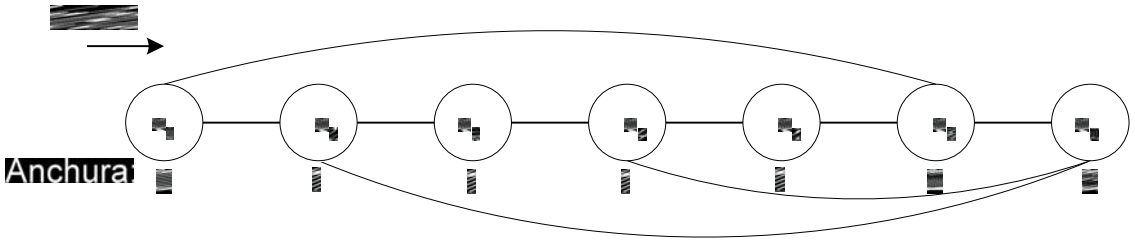
- Anchura de Ordenación: es la anchura máxima de todos los nodos en el grafo bajo esa ordenación establecida [14]. Máx{anchura x_i}

- Anchura de Grafo: corresponde a la anchura mínima del grafo bajo todos los posibles órdenes de sus nodos [14].

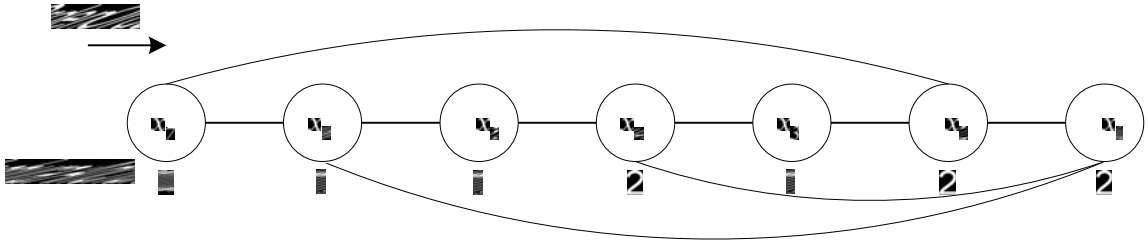
Por Ejemplo, la Figura 3.2 (a) muestra un grafo. Si el orden de los nodos es (x_1, x_2, x_3, x_4, x_5, x_6, x_7), entonces la anchura de cada nodo es 0, 1, 1, 1, 1, 2, 3, respectivamente (ver Figura 3.2 (b)). De esta manera la anchura de ordenación es 3, la cual corresponde a la máxima anchura entre todos los nodos. Por otra parte, si se consideran las ordenaciones (x_1, x_2, x_3, x_4, x_5, x_6, x_7) y (x_7, x_6, x_5, x_4, x_3, x_2, x_1) mostradas en las Figuras 3.2 (b) y 3.2 (c), se tiene que la anchura del grafo es 2 por corresponder a la mínima anchura de los dos órdenes mostrados (para este último calculo se debe obtener la anchura de ordenación de ambas ordenaciones). Considerando el mismo ejemplo de grafo mostrado en la figura 3.2(a), es posible obtener el grado de cada uno de los nodos que lo componen, de esta manera se tiene lo siguiente: Grado (x_1) = 2, Grado (x_2) = 3, Grado (x_3) = 3, Grado (x_4) = 3, Grado (x_5) = 2, Grado (x_6) = 3, Grado (x_7) = 3.



(a) Un grafo de restricciones



(b) Anchura de los nodos dado el orden $x_1, x_2, x_3, x_4, x_5, x_6, x_7$



(c) Anchura de los nodos dado el orden $x_7, x_6, x_5, x_4, x_3, x_2, x_1$

Figura 3. 2 Grafo de restricciones y la anchura de nodo de acuerdo a un orden dado

3.3 Heurísticas de Selección de Variable

Al hacer un recorrido por la literatura es posible encontrar diferentes heurísticas de selección de variable, las cuales se pueden clasificar según el momento en que establecen el orden de selección, es así que las heurísticas de selección de variable se clasifican en:

- Heurísticas de Selección Estáticas: Genera un orden fijo de las variables antes de iniciar la búsqueda, y son entonces seleccionadas siempre en ese orden.

- Heurísticas de Selección Dinámica: Puede cambiar el orden de las variables dinámicamente. Para generar dicho orden se basa en información generada durante la búsqueda.

En la literatura existen una serie de heurísticas de ordenación de variables estáticas, las cuales se basan en la información inicial proporcionada por el grafo de restricciones con la que se acostumbra a representar un CSP binario. Sin embargo, dichas heurísticas son incapaces de percibir o tomar en cuenta los cambios que puedan generarse en los dominios de las variables producto de la ejecución de una etapa de propagación de restricciones durante el proceso de resolución, es por esto que las heurísticas de ordenación de variables estáticas han quedado relegadas principalmente al uso en algoritmos de backtracking [1][11], donde no se lleva a cabo propagación de restricciones. Por otra parte, las heurísticas de selección de variable dinámicas no son factibles para este tipo de algoritmos de búsqueda ya que no hay información adicional disponible durante el proceso de búsqueda con la cual poder modificar el orden inicial de las variables.

La idea que hay detrás de las heurísticas de selección de variable dinámica se basa en el principio de "Fail-First", el cual sugiere que "para tener éxito se debe intentar primero donde es más posible fallar", de esta manera las situaciones sin salida pueden identificarse antes ahorrando espacio de búsqueda [35].

El la literatura [6][11][14] se han propuesto una serie de heurísticas tanto estáticas como dinámicas, las cuales se proceden a revisar a continuación:

3.3.1 Heurísticas de Selección Estáticas

- Minimum Width (MW): Esta heurística ordena las variables según la ordenación de anchura mínima, para lo cual en primer lugar se selecciona la variable con el mínimo número de variables conectadas y se coloca al final del orden, dicha variable y todos sus arcos adyacentes se eliminan del grafo original. Posteriormente se selecciona la siguiente variable y el proceso continúa de la misma forma [1][14]. En la Figura 3.3 es posible apreciar el proceso de aplicación de la heurística descrita, donde inicialmente se selecciona x_7 como la variable menos conectada y se elimina dicha variable y sus relaciones. Posteriormente se seleccionan $\{x_6, x_5\}$, luego $\{x_4, x_3, x_2\}$ y finalmente x_1, dando origen al orden de selección: $x_1, x_2, x_3, x_4, x_5, x_6$.

- Max–Static–Degree o Maximum Degree (MD): Esta heurística selecciona las variables decrecientemente de acuerdo a su grado en el grafo de restricciones original [1]. Considerando el grafo de la Figura 3.4(a), la heurística MD obtiene en primer lugar los grados de cada variable en el grafo de restricciones y luego las ordena decrecientemente obteniendo de esta forma el orden de selección de las variables, el cual para este caso particular corresponde al siguiente: $\{x_1, x_2\}$, $\{x_3, x_4, x_5\}$, x_6, x_7.

- Minimun Domain Variable (MDV): Esta heurística selecciona las variables de acuerdo a la menor cardinalidad de su dominio, es decir, las variables con dominio más pequeño son elegidas antes [11]. En relación a la aplicación de la heurística MDV, es preferible su versión dinámica denominada Minimum Remaining Values descrita a en la sección siguiente.

3.3.2 Heurísticas de Selección Dinámicas

- Minimum Remaining Values (MRV): Esta heurística en cada paso se selecciona la variable con el dominio de instanciación más pequeño [1].

- Maximum Cardinality (MC): Esta heurística selecciona la primera variable arbitrariamente y luego en cada paso, selecciona la variable que es adyacente al conjunto más grande de las variables ya instanciadas [14][1]. En la Figura 3.4 se muestra un ejemplo del proceso llevado a cabo por la heurística MC, donde finalmente se establece que el orden de selección de variables para el grafo de restricciones de la Figura 3.4(a) corresponde al siguiente: x_7, x_6, x_5, x_4, x_3, x_2, x_1.

- Domdeg [6]: Esta heurística seleccionada la variable que minimiza la proporción entre el tamaño de dominio y el número de variables adyacentes no instanciadas.

$$\frac{|D_{x_i}|}{\text{Degree}(x_i)}$$

De manera general, en lo que respecta a definición de heurísticas de selección de variable, es posible utilizar criterios directamente relacionados con el problema de satisfacción de restricciones que se pretende resolver, es así como en la literatura se pueden encontrar la heurística Brélaz, desarrollada para el Graph Coloring Problem [18] o la heurística Orr (Operation Resource Reliance), desarrollada para la resolución de problemas de scheduling [20], entre otras.

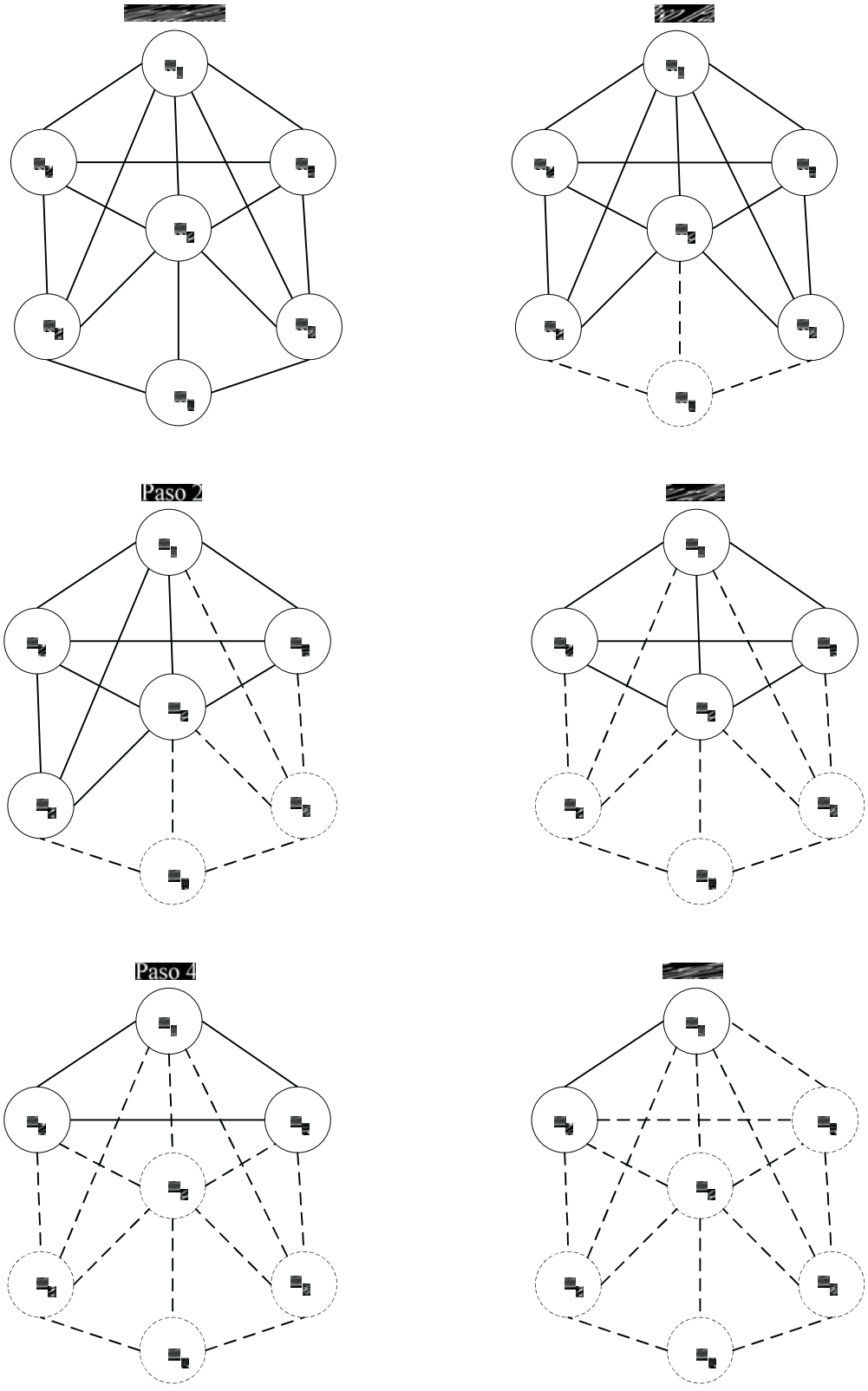
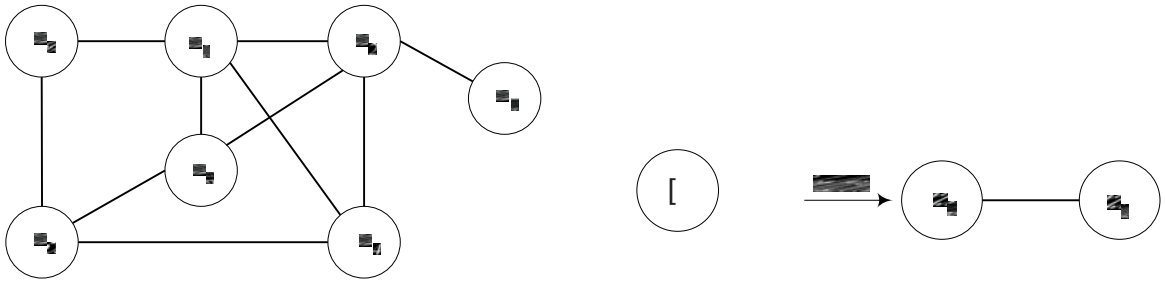


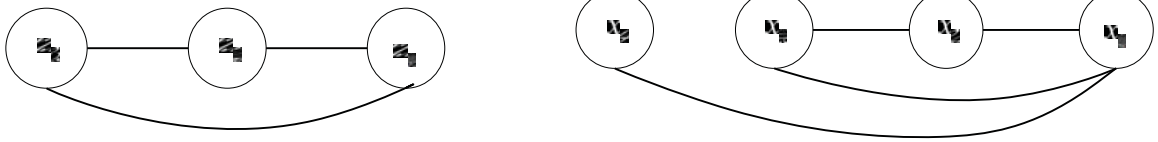
Figura 3. 3 Aplicación de la Heurística Minimum Width



(a) Dado un grafo

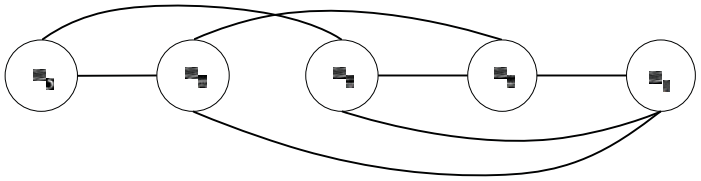
(b) Elegir un nodo Arbitrariamente

(c)

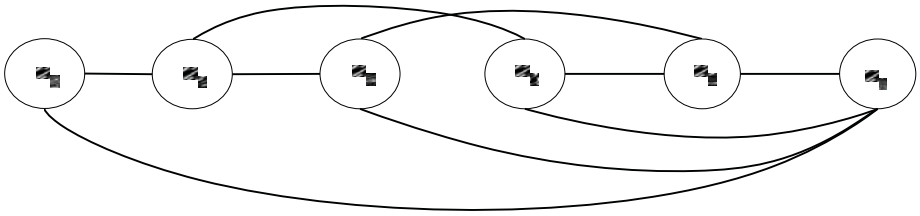


(d)

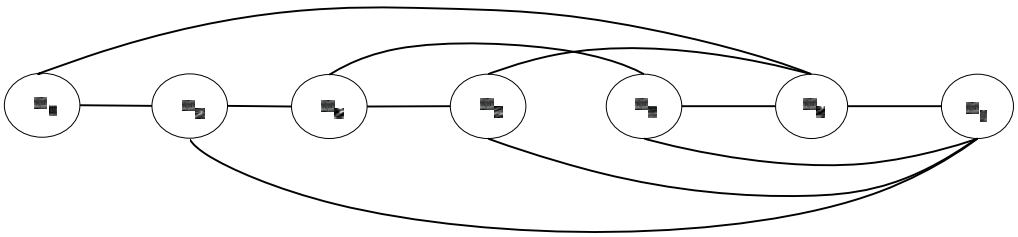
(e)



(f)



(g)



(h)

Figura 3. 4 Ejemplo de Heurística Maximum Cardinality

3.4 Heurísticas de Selección de Valor

A diferencia de las heurísticas de selección de variable, se han realizado pocos trabajos en relación a heurísticas de selección de valor. La idea básica que persigue una heurística de selección de valor es encontrar el valor de la variable actual "más prometedor", es decir, seleccionar el valor que tenga más probabilidad de llevarnos a una solución y así reducir el riesgo de tener que retroceder. En la práctica, lo que mejor se puede hacer es seleccionar el valor que tenga menos probabilidad de llevarnos a una falla, esto último conocido como el principio "Succeed-First", es decir, el valor con un mayor número de soportes es preferible [35].

Si bien se ha realizado poco trabajo, en la literatura [1][11] es posible encontrar heurísticas ampliamente utilizadas a la hora de resolver un problema de satisfacción de restricciones, las cuales se presentan a continuación.

- Point Domain Size: Esta heurística asigna un peso a cada valor de la variable actual dependiendo del número de variables futuras que se quedaran con ciertos tamaños de dominio. Este peso es conocido como punto de valor [1] [35][36]. Por ejemplo, se toma un valor, y por cada dominio que queda de tamaño 1, se le asignan 8 puntos de valor. Por cada dominio que genere de tamaño 2, se le asignan 4 puntos de valor a dicha variable. Por cada dominio que genere de tamaño 3, se le asignan 2 puntos de valor a dicha variable, y por cada dominio que genere de tamaño 4, se le asigna 1 punto de valor a dicha variable. Se prueba con todas los valores para la variable actual. Finalmente se elige el valor con la menor cantidad de puntos de valor, o peso.

- Máx Domain Size: Esta heurística selecciona aquel valor de la variable actual que crea los máximos tamaños de dominio en las variables futuras [1].

- Min Conflicts: Esta heurística ordena los valores de acuerdo a los conflictos en los que éstos están involucrados con las variables futuras. El proceso consisten en asociar a cada valor a de la variable actual, el número total de valores en los dominios de las variables futuras adyacentes que son incompatibles con a, de esta forma se selecciona el valor asociado a la suma más baja [1][14].

- Promise: Esta heurística, para cada valor a de la variable actual, cuenta el número de valores que soportan a a (que son compatibles con a) en cada variable adyacente futura y se toma el producto de las cantidades contadas, tal producto se llama la promesa de un valor. Procediendo según este criterio, se selecciona el valor que posee máxima promesa [1] [15].

Al igual que en las heurísticas de selección de variable, existen heurísticas de selección de valor definidas de acuerdo a las características particulares del problema a resolver, así es el caso de la heurística Fss (Filtered Survivable Schedules) desarrollada para la resolución de problemas de scheduling [20].

A continuación, con la intención de ejemplificar el efecto de las heurísticas de selección de variable y valor en el proceso de resolución de un problema de satisfacción de restricciones, se considera el grafo de restricciones de la Figura 3.5 correspondiente al siguiente CSP:

Variables: x_1, x_2, x_3, x_4, x_5

Dominios: $x_1 \in \{1,2,3,4\}, x_2 \in \{1,2,3\}, x_3 \in \{1,2,3\}, x_4 \in \{1,2,3,4\}, x_5 \in \{2,3,4,5\}$

Restricciones:

(1) $x_1 \neq x_2$ (5) $x_3 \neq x_4$ (9) $x_1 - x_3 \neq 2$

(2) $x_1 \neq x_3$ (6) $x_3 \neq x_5$ (10) $x_1 - x_3 \neq -2$

(3) $x_1 \neq x_4$ (7) $x_1 - x_2 \neq 1$ (11) $x_1 - x_4 \neq 3$

(4) $x_2 \neq x_3$ (8) $x_1 - x_2 \neq -1$ (12) $x_1 - x_4 \neq -3$

(13) $x_2 - x_3 \neq 1$ (15) $x_3 - x_4 \neq -1$ (17) $x_3 - x_5 \neq -2$

(14) $x_3 - x_4 \neq 1$ (16) $x_3 - x_5 \neq 2$

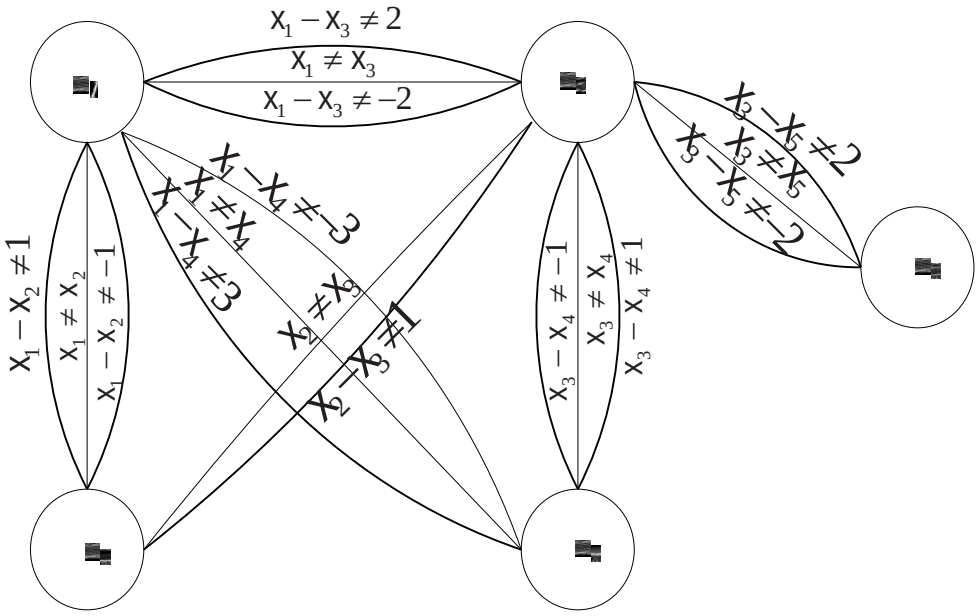


Figura 3. 5 Grafo de Restricciones de un CSP

Definido el CSP, se procede a su resolución utilizando la siguiente combinación de heurísticas: MC + Promise y MW + Min Conflicts, constituyendo de esta forma dos estrategias de enumeración. La utilización de cada una de las estrategias dará origen a dos procesos de resolución distintos, donde el seguimiento de cada proceso será separado en iteraciones. Los cálculos y resultados obtenidos en cada iteración son registrados en una tabla cuya estructura se presenta a continuación:

x_s	a_1	…	a_n
x_i	v_1	…	v_n
…	…		
…	…		
x_m	v_m		
	t_1	…	t_n

Donde:

x_s: variable actualmente selecciona en la iteración

$a_1 \ldots a_n$: valores en el dominio de x_s

$x_i \ldots x_m$: variables sin instanciar relacionadas con x_s

$v_1 \ldots v_n$: para el caso de la heurística Min Conflicts, corresponderán al número total de valores en el dominio de x_i que son incompatibles con $a_1 \ldots a_n$, respectivamente. En el caso de la heurística Promise, dichos valores corresponden al número de soportes que tienen los valores $a_1 \ldots a_n$, en el dominio de la variable x_i.

Lo anterior se repite para cada una de las variables $x_i \ldots x_m$.

$t_1 \ldots t_n$: corresponden a la suma (en el caso de Min Conflicts) o producto (en el caso de Promise) de los valores $v_1 \ldots v_m$.

Utilizando la primera estrategia de enumeración (MC + Promise), donde el orden de selección de variables se establece según lo descrito en la sección 3.3. En la primera iteración, donde se selecciona el valor 4 para la variable x_4, las restricciones (3), (11), (12), (14) y (15) ejecutan propagación sobre los dominios de las variables, reduciendo el dominio de x_1 y x_3.. Una fase similar es llevada a cabo en la segunda iteración, donde la propagación permite reducir los dominios de ciertas variables a tamaño 1. El proceso de propagación es realizado en todos los nodos y en cada uno de los ejemplos donde las restricciones pueden inferir nueva información.

Primera iteración ($x_4 = 4$)

x_4	1	2	3	4
x_1	2	3	3	2
x_3	1	0	1	2
	2	0	3	4

Segunda iteración ($x_1 = 3$)

x_1	2	3		
x_2	0	1		
x_3	1	1		
	0	1		

De esta manera se selecciona el valor asociado a la máxima promesa, que el caso de la primera iteración corresponde al valor 4 y en la segunda iteración se selecciona 3. En la Figura 3.6 es posible ver paso a paso el árbol de búsqueda generado con el proceso descrito.

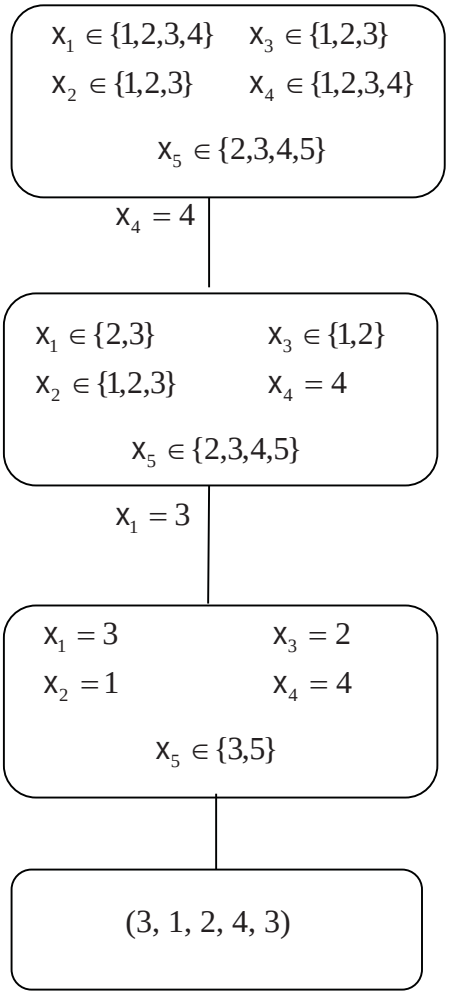


Figura 3. 6 Resolución con MC + Promise

Una segunda instancia de resolución del CSP descrito, considera la utilización de la estrategia de enumeración que combina MW + Min Conflicts. Para utilizar la heurística de selección de variable se utiliza el proceso descrito en la Figura 3.3, donde en caso de tener más una posibilidad de selección de variables, el empate se rompe haciendo una selección lexicográfica, donde finalmente el orden de selección de variables corresponde a x_5, x_2, x_1, x_3, x_4.

Primera iteración ($x_5 = 2$)

x_5	2	3	4	5
x_3	1	2	1	1
	1	2	1	1

Segunda iteración ($x_2 = 1$)

x_2	1	2	3	
x_1	2	3	3	
x_3	1	1	1	
	3	4	4	

Tercera iteración ($x_2 = 2$)

x_2	2	3	
x_1	3	3	
x_3	1	1	
	4	4	

Cuarta iteración ($x_5 = 4$)

x_5	3	4	5
x_3	2	1	1
	2	1	1

Quinta iteración ($x_2 = 1$)

x_2	1	2	3	
x_1	2	3	3	
x_3	1	1	1	
	3	4	4	

Sexta iteración ($x_5 = 5$)

x_5	3	5
x_3	2	1
	2	1

Séptima iteración ($x_2 = 1$)

x_2	1	2	3	
x_1	2	3	3	
x_3	1	2	1	
	3	5	4	

Sexta iteración ($x_2 = 3$)

x_2	2	3	
x_1	3	3	
x_3	2	1	
	5	4	

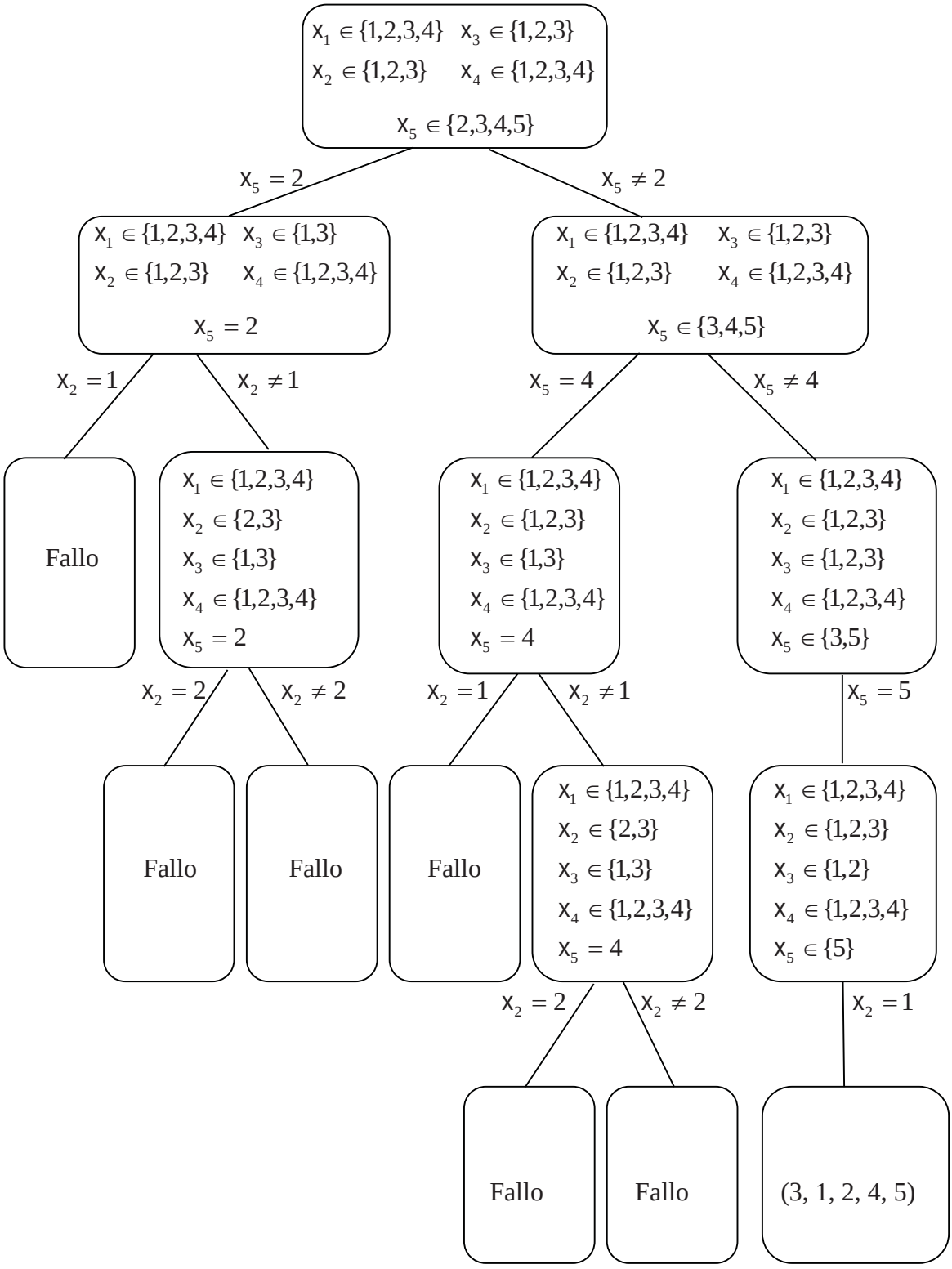


Figura 3. 7 Resolución con MW + Min Conflicts

Observando los dos procesos es posible ver que la utilización de heurísticas de selección de variable y valor influye en la eficiencia de resolución, la cual es reflejada a través de la cantidad de nodos generados. De esta manera, en una misma instancia del problema la solución puede ser encontrada en un par de pasos versus una decena de ellos incluyendo varios intentos fallidos, lo que claramente impactara aún más si se consideran adecuadamente dichas heurísticas en problemas más complejos.

De esta manera, bajo el supuesto que la utilización adecuada de heurísticas de selección de variable y valor mejora la eficiencia de resolución de un CSP, a continuación se estudia la resolución de puzzles que permitan evaluar el desempeño de diferentes heurísticas de selección de variable y valor.

CAPÍTULO 4

Programación con Restricciones en Mozart

4.1 Espacio de Cómputo

Como se mencionó en capítulos anteriores, el proceso de resolución de problemas combinatoriales mediante la utilización de programación con restricciones consiste en alternar fases de Propagación y Enumeración, esta última conocida en Mozart como Distribución.

El guión principal de un modelo con restricciones en Mozart se ejecuta en un espacio de cómputo. El espacio de cómputo está compuesto de un almacén de restricciones, propagadores y un distribuidor. En dicho espacio se hace tanta propagación como sea posible hasta llegar a un estado estable. Si aún no se tiene una solución, o si se quieren otras, se puede hacer distribución creando subespacios de cómputo.

La solución del espacio consiste básicamente en mapear las variables a números enteros de tal manera de satisfacer las restricciones en el almacén de restricciones y todas las restricciones impuestas por los propagadores.

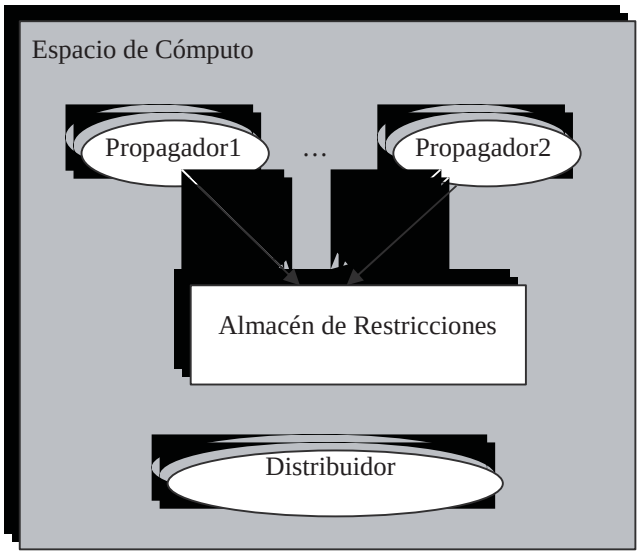


Figura 4. 1 Espacio de Cómputo

4.2 Dominio Finito y Restricciones

La atención de este trabajo se enfocará en la resolución de problemas combinatoriales de dominio finito y la implementación de heurísticas de selección de variable y valor en la plataforma Mozart. A continuación se especifican los conceptos asociados a la plataforma.

Dominio Finito: corresponde a un conjunto de números enteros no negativos [3]. La notación utilizada para el dominio finito {m,…, n} corresponde a m#n.

Problema de Dominio Finito: corresponde a un conjunto finito P de restricciones cuantificadoras libres tales que P contiene una restricción de dominio para cada variable que ocurre en una restricción de P [3].

Ejemplo: Restricciones cuantificadoras libres: $x_1 \neq x_2$, $x_1 \neq x_3$, $x_2 \neq x_3$

Restricciones de dominio: $D_{x_1} = \{1, 2, 3\}$, $D_{x_2} = \{1, 2, 3\}$ y $D_{x_3} = \{1, 2, 3\}$

Restricción: es una fórmula de lógica de predicado.
Ejemplo: $x_1 + x_2 = 5x_3$

Restricción de Dominio: una restricción de este tipo toma la forma $x_i \in D_{x_i}$, donde D_{x_i} es un dominio finito. Las restricciones de dominio pueden expresar restricciones de la forma $x_1 = n$, lo cual también es equivalente a: $x_1 \in m\#n$.

Restricción Básica: una restricción de este tipo toma alguna de las siguientes formas: $x_i = n$ o $x \in D_{x_i}$, donde D_{x_i} es un dominio finito.

4.3 Propagación

La propagación de restricciones es una regla de inferencia para problemas de dominio finito que reduce los dominios de las variables. La arquitectura computacional para la propagación de restricciones es llamada espacio y consiste en un número de propagadores conectados a un almacén de restricciones [3].

Propagador: es un agente concurrente que trata de reducir los dominios de las variables. El propagador impone restricciones no-básicas, las cuales pueden ser de la forma "$x_1 < x_2$" o "todos los valores de $x_1,\ldots,x_n$ son diferentes". Un propagador tiene la capacidad de decir restricciones a un almacén de restricciones, de hecho la semántica operacional de un propagador se determina si el propagador puede decir al almacén una restricción básica o no. La semántica operacional de un propagador debe respetar la semántica declarativa del propagador, la cual esta dada por la restricción que el propagador impone.

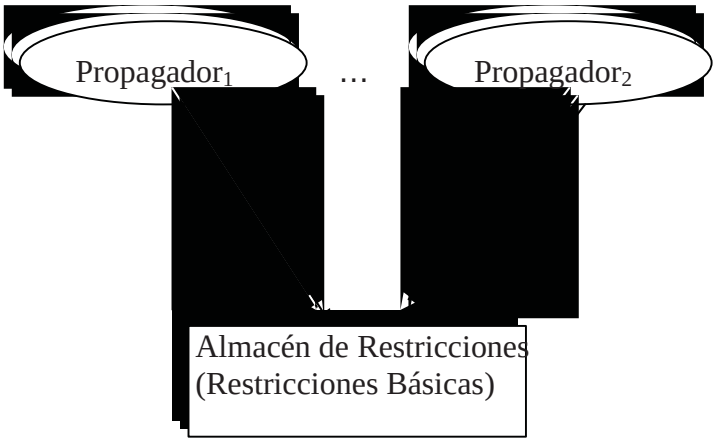


Figura 4. 2 Propagación en Mozart (Espacio)

Durante la etapa de propagación, los propagadores pueden tomar distintos estados los cuales se describen a continuación:

- Propagador Fallido: un propagador es inconsistente si no hay asignación variable que satisface el almacén de restricciones y la restricción impuesta por el propagador. Por lo tanto, se dice que el propagador es fallido si la semántica operacional que realiza este es inconsistente.

  Ejemplo: $x_1 + x_2 = 6$ sobre $x_1 \in 3\#9$ y $x_2 \in 4\#9$

- Propagador Implicado: un propagador es implicado si cada asignación variable que satisface el almacén de restricciones también satisface la restricción impuesta por el propagador.

- Propagador Estable: un propagador es estable si es fallido o su semántica operacional no puede decir nueva información al almacén de restricciones.

- Espacio Fallido: un espacio es fallido si falla uno de sus propagadores.

- Espacio Estable: el espacio es estable si todos sus propagadores son estables.

- Espacio Resuelto: un espacio esta resuelto si no se falla y no hay más propagadores.

4.3.1 Esquema Operacional

En Mozart hay dos esquemas establecidos para la operación de un propagador, la propagación de dominio y la propagación de intervalo. La propagación de dominio reduce los dominios de las variables tanto como sea posible, por su parte, la propagación de intervalo reduce solamente los límites de un dominio. En la práctica, la propagación de intervalo es generalmente preferible por sobre la propagación de dominio debido a sus costos computacionales más bajos.

Para ejemplificar lo descrito anteriormente considerar el propagador $2 * x_1 = x_2$ sobre el almacén de restricciones $x_1 \in 1\#10$ $x_2 \in 1\#7$. Bajo propagación de dominio, el propagador puede reducir los dominios a $x_1 \in 1\#3$, $x_2 \in \{2, 4, 6\}$ y bajo propagación de intervalo, el propagador puede reducir los límites del dominio a $x_1 \in 1\#3$ y $x_2 \in 2\#6$.

4.3.2 Estado Incompleto

La propagación de restricciones no es un método de solución completo, esto debido a que puede suceder que un espacio tiene una única solución y la propagación de restricciones no la encuentre o que el espacio no tenga solución y la propagación de restricciones no conduzca a un propagador fallido.

Considerar que se tienen los siguientes propagadores, $x_1 + x_2 = 20$, $x_1 > 3x_2$ sobre el almacén de restricciones $x_1 \in \{0,\ldots, 20\}$, $x_2 \in \{0,\ldots, 20\}$, el segundo propagador reduce

los dominios a $x_1 \in \{1,\ldots, 20\}$, $x_2 \in \{0,\ldots, 6\}$. Luego, el primer propagador deja los dominios $x_1 \in \{14,\ldots, 20\}$, $x_2 \in \{0,\ldots, 6\}$. Los propagadores lograron reducir los dominios de las variables x_1 y x_2 pero no lograron encontrar una solución al problema.

El ejemplo anterior es muy elemental pero sirve para ilustrar el concepto de estado incompleto de la propagación de restricciones.

4.4 Distribución

Como la propagación de restricciones no es un método de solución completo, se combina propagación de restricciones y distribución, proporcionando un método completo de solución para problemas de dominio finito. Para solucionar un problema de dominio finito P, se puede elegir una restricción C y solucionar $P \cup \{C\}$ y $P \cup \{\sim C\}$. Agregar esta nueva restricción no cambia la solución.

A continuación se describe las forma de proceder, para lo cual se considera que dado un problema, se instala un espacio cuyo almacén de restricciones contenga las restricciones básicas y cuyos propagadores impongan las restricciones no básicas del problema. Entonces se ejecutan los propagadores hasta que el espacio llegue a ser estable. Si se falla o se soluciona el espacio, no se hace nada. Por el contrario, se elige una variable aún no determinada x_i y un entero n, y se hacen dos copias del espacio de cómputo original, agregando un propagador que impone $x_i = n$ a la primera copia y un propagador que impone $x_i \neq n$ a la segunda. De esta forma, los propagadores pueden volver a actuar en ambos espacios.

La elección de la variable y la elección del valor con el cual se crean la copia del espacio se realizan mediante una estrategia de distribución. Dicha estrategia de distribución esta compuesta de heurísticas de selección de variable y heurísticas de selección de valor, las que en conjunto guían la etapa de distribución, de tal forma que la

elección sea la más adecuada, ya que de esta elección depende la eficiencia con que se solucione el problema de satisfacción de restricciones, de aquí que se considera de vital importancia entender estas estrategias y las posibilidades que brinda la plataforma Mozart para su implementación.

4.4.1 Estrategias de Distribución

Un distribuidor es un agente computacional que implementa estrategias de distribución [3]. Generalmente, una estrategia de distribución se define sobre una secuencia $x_1,..., x_n$ de variables. Cuando es necesario un paso de distribución, la estrategia selecciona una variable aún no determinada en la secuencia y distribuye sobre esa variable.

En [3] se definen algunas posibilidades estándares para distribuir sobre una variable x_i:

- Distribuir con $x_i = l$, donde l es el menor valor posible para x_i.
- Distribuir con $x_i = u$, donde u es el, mayor valor posible para x_i.
- Distribuir con $x_i = m$, donde m es un valor posible para x_i que está en el medio del menor y el mayor valor posible para x_i.
- Distribuir con $x_i \leq m$, donde m es un valor posible para x_i que está en el medio del menor y el mayor valor posible para x_i.

En general, las estrategias de distribución pueden ser clasificadas en estrategias de distribución genéricas y estrategias de distribución específicas del problema. Las estrategias de distribución genéricas son estrategias generales que no dependen del problema, por su parte las estrategias de distribución específicas del problema, como su nombre lo refleja, dependen del problema que se quiera resolver, es decir, el criterio

usado en la estrategia depende de las características del problema para acelerar el proceso de distribución.

Estrategias de distribución existentes en Mozart son:

- Estrategia de Distribución Ingenua (Naive): Esta estrategia seleccionará la variable indeterminada de más a la izquierda en la secuencia y en cuanto al valor seleccionará el límite más pequeño del dominio. Sentencia del lenguaje: {FD.distribute naive xv}, xv debe ser un vector de enteros en un dominio finito, entonces la estrategia seleccionará la variable indeterminada de más a la izquierda en xv.

- Estrategia de Distribución Primer Fallo (First–Fail): Esta estrategia selecciona la variable indeterminada de más a la izquierda en la secuencia cuyo dominio en el almacén de restricciones tiene tamaño mínimo, en cuanto al valor selecciona el límite más pequeño del dominio. En otras palabras selecciona la variable indeterminada de más a la izquierda para la cual el número de valores diferentes posibles es mínimo.
  Sentencia del lenguaje: {FD.distribute ff xv}, xv debe ser un vector de enteros en un dominio finito, entonces la estrategia seleccionará la variable indeterminada de más a la izquierda en xv cuyo dominio es mínimo.

A modo de ejemplo de la estrategia de distribución first-fail, considere el problema bosquejado por las siguientes restricciones: $x_1 \neq 7$; $x_3 \neq 2$; $x_1 - x_3 = 3 * x_2$; $x_1 \in \{1\#8\}$; $x_2 \in \{1\#10\}$; $x_3 \in \{1\#10\}$. El proceso computacional se describe a continuación:

Nodo 1: se ejecuta una etapa de propagación y los dominios de las variables quedan reducidos a lo siguiente: $x_1 \in \{4\#6\ 8\}$; $x_2 \in \{1\#2\}$; $x_3 \in \{1\ 3\#5\}$. Luego, por distribución se instancia $x_2 = 1$.

Nodo 2: Se realiza distribución, seleccionando x_1 con valor 4, esto genera la solución $x_1 = 4$, $x_2 = 1$, $x_3 = 1$ en el nodo 3, y en el nodo 4 los dominios de las variables quedan reducidos a: $x_1 \in \{6\ 8\}$, $x_2 = 1$, $x_3 \in \{3\#5\}$ (notar que el dominio de x_1 se redujo por propagación)

Nodo 4: se hace distribución, seleccionando la variable x_1 con el valor 6 lo que genera la solución del nodo 5 $x_1 = 6$, $x_2 = 1$, $x_3 = 3$. El nodo 6 (que corresponde a $x_1 \neq 6$) proporciona la solución $x_1 = 8$, $x_2 = 1$, $x_3 = 5$.

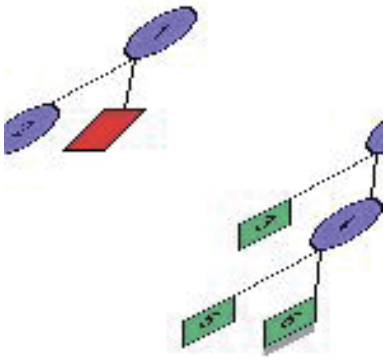


Figura 4. 3 Árbol de búsqueda obtenido con estrategia First-Fail

- Estrategia de Distribución Split: Esta estrategia selecciona la variable indeterminada de más a la izquierda en la secuencia cuyo dominio tiene tamaño mínimo, y crea dos puntos de elección con el valor del centro del dominio.

- Estrategia de Distribución Generic: Esta estrategia permite personalizar ciertos parámetros de forma tal de crear una estrategia de distribución que se ajuste a las necesidades de resolución del problema. Esta estrategia recibe los parámetros para order (que variable se seleccionará), filter (grupo de variables a considerar), select (variable a enumerar), value (que valor se seleccionará) y procedure (se aplica solo cuando se alcanza la estabilidad). Un ejemplo de la estructura de la estrategia es la siguiente:

- ✓ generic(order: +Order <= Size
  filter: +Filter <= undet
  select: +Select <= id
  value: +Value <= min
  procedure: +Proc <= proc{$} skip end)

  Considera los elementos de la secuencia para los cuales Filtre es verdadero. Selecciona el elemento de más a la izquierda, el cual es mínimo con respecto a Order y selecciona la variable correspondiente según Select y crea dos puntos de elección para el valor seleccionado por Value.

Hasta este punto se han visto cuatro estrategias de distribución disponibles en el sistema Mozart, dichas estrategias son implementadas por distribuidores incorporados en la plataforma, es decir, el programador solo tiene que hacer uso de ellas por medio de alguna sentencia del lenguaje, o mediante la incorporación de parámetros que permitan la personalización de la estrategia, esto quiere decir, una adecuación a las necesidades del problema, por medio de la estrategia generic.

4.5 Árbol de Búsqueda

Alternando fases de propagación y distribución se obtienen un árbol de búsqueda, donde cada nodo del árbol corresponde a un espacio, y cada hoja del árbol corresponde a un espacio que ha sido solucionado o fallido. El árbol de búsqueda es siempre finito ya que solo hay un número finito de muchas variables todas ellas restringidas a priori a un dominio finito.

La idea es que a cada nivel se aplica propagación al espacio actual, si se falla o se soluciona no se hace nada. Por el contrario, se aplica una fase de distribución, y así se prosigue hasta construir la solución. En la Figura 4.4 es posible ver el árbol de búsqueda generado llevando a cabo este proceso.

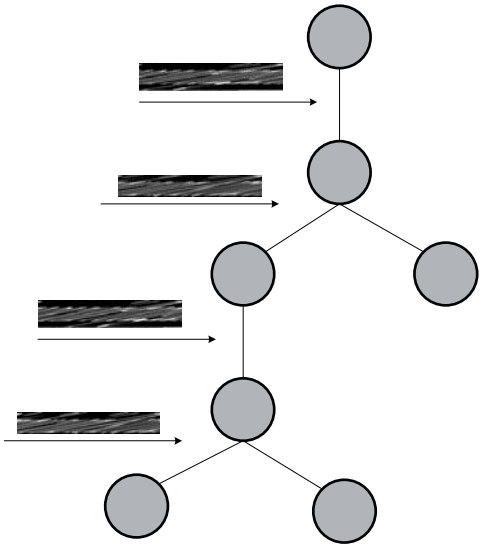


Figura 4. 4 Árbol de búsqueda

El explorador de Mozart genera un árbol de búsqueda muy similar al de la Figura 4.4, la diferencia radica en la notación utilizada, ya que los espacios solucionados se representan mediante un diamante verde y los espacios fallidos mediante cuadrados rojos (ver Figura 4.5).

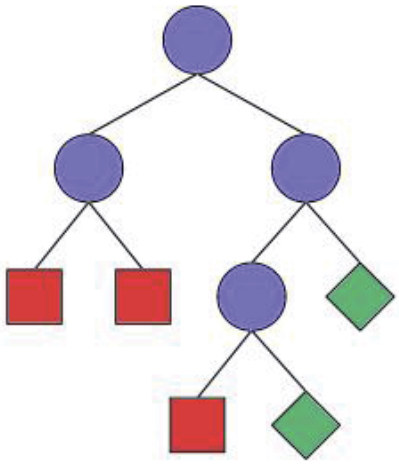


Figura 4. 5 Ejemplo de árbol de búsqueda generado en Mozart

4.5.1 Orden de Búsqueda

El orden en el cual se explora el árbol de búsqueda si bien no tiene un impacto en la forma ni en el tamaño del árbol, este orden sí genera un impacto en los recursos de tiempo y de memoria necesitados para encontrar una o todas las soluciones [3]. Por lo cual es preciso tener en cuenta aspectos tales como:

- Si se esta interesado en encontrar solo una solución, no es necesario explorar el árbol de búsqueda completo.

- Si se esta interesado en encontrar todas las soluciones, es necesario entonces explorar el árbol de búsqueda entero. No obstante, si se explora el árbol con una exploración primero en profundidad (Depth-First) o primero en anchura (Breadth-First) habrá una gran diferencia en la memoria necesitada, ya que los requerimientos de memoria de la exploración Breadth-First son exponencialmente más grandes que los de la exploración Depth-First.

En el caso particular del presente trabajo, el árbol de búsqueda se explora siempre primero en profundidad (Depth-First). Además, cuando se distribuye con una restricción C, se explora primero el espacio obtenido con C y luego el espacio obtenido con ~C.

CAPÍTULO 5

Implementación Computacional

5.1 Problemas

En la presente sección se presenta una descripción detalla tanto de lo problemas resueltos como de su implementación en el sistema de desarrollo Mozart, dichos problemas han sido utilizados para ilustrar el efecto de las heurísticas de selección de variable y valor en su resolución.

5.1.1 N- Reinas

El problema consiste básicamente en ubicar N reinas sobre un tablero de ajedrez de dimensión NxN, donde N > 3, de modo que no se ataquen. La Figura 5.1 muestra una solución al problema para N = 8.

5.1.1.1 Modelo

Las reinas son numeradas de 1 a N, tal que la k-ésima reina siempre es ubicada en la k-ésima columna. Para cada reina i se tiene una variable x_i que indica la fila en la cual se ubica la reina. El modelo descrito hasta el momento garantiza que dos reinas nunca serán ubicadas en la misma columna.

Para asegurar que dos reinas nunca serán ubicadas en la misma fila, se debe imponer la restricción que las variables $x_1 \ldots x_n$ sean todas distintas:

$$x_i \neq x_j \quad \forall i, j \text{ talque } 1 \leq i < j \leq N \qquad (5.1)$$

Por otra parte, se debe asegurar que dos reinas nunca serán ubicadas en la misma diagonal, con lo cual se deben imponer las siguientes restricciones:

$$x_i - x_j \neq i - j \quad \forall i, j \text{ talque } 1 \leq i < j \leq N \quad (5.2)$$
$$x_i - x_j \neq j - i \quad \forall i, j \text{ talque } 1 \leq i < j \leq N \quad (5.3)$$

5.1.1.2 Script

Basándose en el modelo anterior es posible obtener una representación en el lenguaje Oz, la cual permitirá obtener el Script que resolverá el problema planteado, dicha representación se explica en los párrafos siguientes.

La tupla Row permite crear las variables $x_1 \ldots x_n$ definidas en el modelo, y se representa en el lenguaje a través de la siguiente sentencia:

```
{FD.tuple queens N 1#N Row}
```

N corresponde al número de reinas, esta variable es recibida como parámetro de entrada en el script. Por su parte 1#N establece el dominio en el cual se mueven las variables $x_1 \ldots x_n$ del problema. Una vez establecida la definición de las variables se procede al establecimiento de las restricciones, de esta manera representar la restricción (5.1) del modelo a través del lenguaje significa que se debe incluir la sentencia siguiente:

```
{FD.distinct Row}
```

La sentencia anterior asegura que las variables $x_1 \ldots x_n$ definidas en el modelo serán todas distintas. Por su parte el fragmento de código siguiente representa y asegura que se de cumplimiento a las restricciones (5.2) y (5.3) del modelo.

```
{For 1 N-1 1
   proc {$ I}
     {For I+1 N 1
        proc{$ J}
          Row.I - Row.J \=: I - J
          Row.I - Row.J \=: J - I
        end}
   end}
```

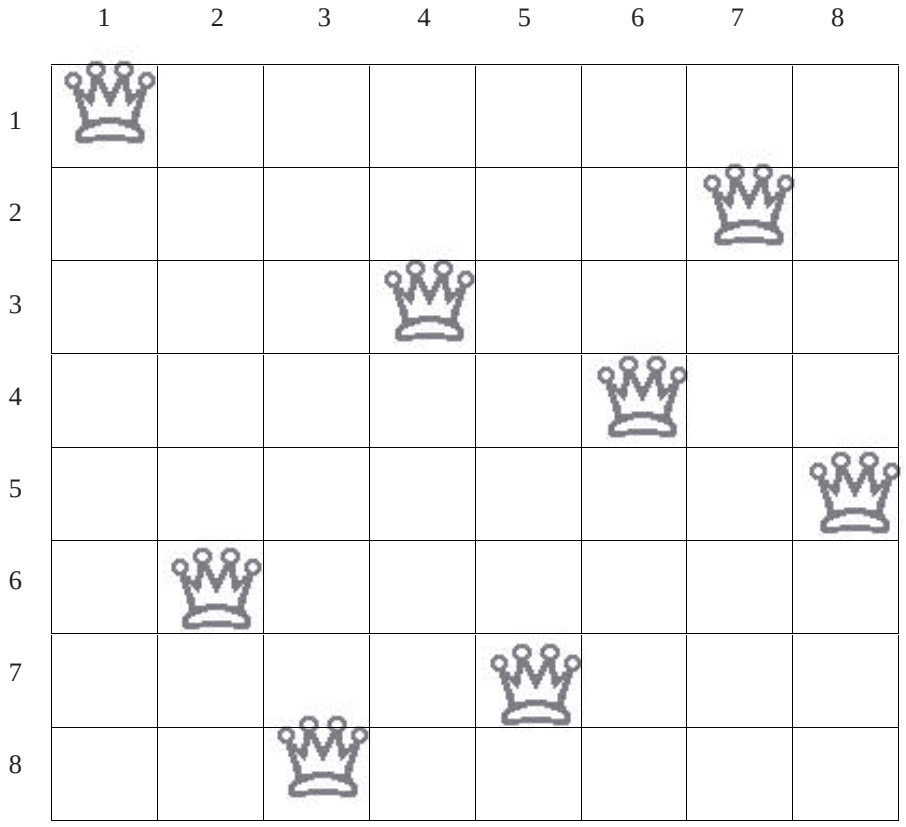


Figura 5. 1 Una de las 92 soluciones del problema de 8 – Reinas

5.1.1.3 Caracterización

El problema de N- Reinas es un problema de satisfacción de restricciones binario, discreto y finito, cuyo modelo establecido en 5.1.1.1 permite determinar para un N dado correspondiente al número de reinas, los siguientes atributos característicos:

- Tamaño de los dominios: el tamaño de los dominios de las variables es igual a N.
- Número de variables: el modelo establecido establece el uso de N variables
- Tamaño del espacio de búsqueda: N^N

Con las características anteriormente expuestas es posible obtener la Tabla 5.1 en la cual se muestran los tamaños de dominios para distintas instancias del problema:

Tabla 5. 1 Espacio de Búsqueda para instancias de N-Reinas

Instancia N – Reinas	Tamaño del Espacio Búsqueda (N^N)
4	256
10	$1x10^{10}$
20	$1.0485x10^{26}$
50	$8.8817x10^{84}$
100	$1x10^{200}$

5.1.2 Cuadrados Mágicos

Este puzzle consiste en encontrar para un N dado una matriz de NxN tal que cada celda de la matriz es un número entero entre 1 y N^2, todos los campos de la matriz son distintos entre si, y la suma de las filas, columnas, y las dos diagonales son todas iguales. En la Figura 5.2 es posible ver una solución para la instancia N = 3 del cuadrado mágico.

2	7	6
9	5	1
4	3	8

Figura 5. 2 Una Solución para N = 3

5.1.2.1 Modelo

El modelo tiene una variable x_{ij} para cada celda (i, j) de la matriz y una variable S que requiere que la suma de cada fila, columna y diagonal sean igual a S. Por otra parte, el modelo debe restringir que todos los campos de la matriz sean distintos entre si, y que la suma de las filas, las columnas, y las dos diagonales son todas iguales. Es así como el modelo establece las siguientes restricciones:

$$\sum_{j=1}^{N} x_{ij} = S \quad \forall i \in \{1,\ldots, N\} \quad (5.4) \qquad \sum_{i=1}^{N} x_{ij} = S \quad \forall j \in \{1,\ldots, N\} \quad (5.5)$$

$$\sum_{i=1}^{N} x_{ii} = S \quad (5.6) \qquad \sum_{i=1}^{N} x_{iN-i} = S \quad (5.7)$$

La restricción (5.4) asegura que la suma de cada una de las filas será igual a S, la restricción (5.5) asegura que la suma de cada una de las columnas será igual a S y finalmente las restricciones (5.6) y (5.7) aseguran que la suma de cada una de las diagonales será igual a S.

El puzzle de los cuadrados mágicos es extremadamente duro de resolver para N grande [3][17], por lo cual en algunos casos se establecen algunas restricciones adicionales que ayudan a mejorar su resolución, este tipo de restricciones adicionales son denominadas como restricciones redundantes y restricciones que eliminan simetrías.

Una restricción redundante es una restricción implicada o derivada de las restricciones que definen el modelo, y la idea fundamental para utilizar este tipo de restricciones en la resolución de un CSP es reducir el esfuerzo de búsqueda requerido para solucionar un CSP. Por su parte, una restricción que elimina simetrías es una restricción que elimina soluciones, de tal manera de hacer más pequeño el espacio de búsqueda.

En el presente trabajo y para el caso particular del problema de los cuadrados mágicos se ha considerado la utilización de las siguientes restricciones adicionales:

- Restricciones que eliminan simetrías: en este caso se han considerado restricciones de orden en las esquinas de la matriz, ejemplos de estas restricciones son del tipo: $x_{11} < x_{NN}$

- Restricciones redundantes: en este caso se considera que la suma de las sumas de las filas debe ser igual a la suma de todas las celdas, lo cual equivale a restringir lo siguiente: $N^2 \frac{(N^2 + 1)}{2} = N*S$

5.1.2.2 Script

El script con el cual se representa el modelo anterior en el lenguaje de programación Oz recibe como parámetro de entrada una de las dimensiones de la matriz, correspondiente a la variable N. El mapeo de las variables x_{ij} del modelo al lenguaje de programación se realiza a través de la tupla Square la cual se implementa a través de la siguiente sentencia de código:

```
{FD.tuple square NN 1#NN Saqure}
```

En la sentencia anterior, NN es igual a N^2. Luego, para asegurar que todas las celdas de la matriz sean distintas entre sí se impone la siguiente restricción en el lenguaje de programación:

```
{FD.distinct Square}
```

Como la tupla Square define la matriz como una especie de arreglo unidimensional (tupla), se debe tener una función que permita tener acceso a cada elemento que represente una celda (i, j), esto se realiza a través de la función Fiel a la cual se accede con la sentencia siguiente:

```
{Field I J}
```

En lo que respecta a las restricciones del modelo, específicamente las restricciones (5.4), (5.5) son mapeadas a las siguientes sentencias en el script de programación:

```
{Assert fun {$ I} {Field I I} end}

{Assert fun {$ I} {Field I N+1-I} end}
```

Por su parte, las restricciones (5.6) y (5.7) son representadas en el lenguaje de programación a través de las siguientes sentencias del lenguaje:

```
{For 1 N 1
   proc {$ J} {Assert fun {$ I} {Field I J} end}
end}

{For 1 N 1
   proc {$ I} {Assert fun {$ J} {Field I J} end}
end}
```

El procedimiento Assert realiza las sumas de cada una de las filas, columnas y diagonales, y obliga a que todas estas sean igual a Sum, dando así cumplimiento por completo a las restricciones (5.4), (5.5), (5.6) y (5.7) del problema.

```
proc {Assert F}
```

```
        {FD.sum {Map ListNum F} '=:' Sum}
    end
```

5.1.2.3 Caracterización

Para el caso de los cuadrados mágicos y específicamente para el modelo establecido en este trabajo se puede determinar lo siguiente:

- Tamaño de los dominios: es igual a N^2.
- Número de variables: el modelo utiliza N^2 variables.
- Tamaño del espacio de búsqueda: $(N^2)^{N^2}$

Considerando que N es una de las dimensiones de la matriz en la Tabla 5.2 se observa el tamaño del espacio de búsqueda para distintas instancias del problema.

Tabla 5. 2 Espacio de Búsqueda para instancias de Cuadrados Mágicos

N	Tamaño del Espacio Búsqueda $(N^2)^{N^2}$
3	$3.8742x10^{8}$
4	$1.8446x10^{19}$
5	$8.8817x10^{34}$
7	$6.6009x10^{82}$

Al observar la tabla anterior es posible ver que el tamaño del espacio de búsqueda para cada una de las instancias crece muy rápido, lo que hace pensar que el problema de los cuadrados mágicos puede volverse más duro de resolver que el problema de las N–Reinas, sin embargo como se establece en 5.1.2.1, esto se aborda utilizando restricciones redundantes y restricciones que eliminan simetrías. El uso de tales restricciones no se limita al problema de los cuadrados mágicos, sin embargo en el presente trabajo solo se

utiliza en dicho problema como una forma de percibir las diferencias producidas al resolver el problema utilizando este tipo de restricciones y sin utilizar dichas restricciones.

5.1.3 Cuadrados Latinos

Un Cuadrado Latino de orden N se define como una matriz de NxN donde sus elementos son números enteros entre 1 y N con la propiedad que cada uno de los N enteros ocurre exactamente una vez en cada fila y exactamente una vez en cada columna de la matriz.

1	2	3	4	5
2	3	4	5	1
3	4	5	1	2
4	5	1	2	3
5	1	2	3	4

Figura 5. 3 Cuadrado Latino de orden 5

5.1.3.1 Modelo

El modelo tiene una variable x_{ij} que representa la celda (i, j) de la matriz, donde cada una de estas celdas puede tomar valores entre 1 y N. Por otra parte se tiene la restricción que permiten asegurar que cada uno de los N enteros ocurre una vez en cada fila, mostrada en la restricción (5.8):

$$\forall i \in \{1..N\}\ \text{AllDifferent}\ (x_{i1}, x_{i2}, x_{i3}, \ldots, x_{iN}) \quad (5.8)$$

Y además se restringen las variables para que los N enteros ocurran una vez en cada columna, según se muestra a continuación:

$$\forall j \in \{1..N\}\ \text{AllDifferent}\ (x_{1j}, x_{2j}, x_{3j},\ldots, x_{Nj}) \quad (5.9)$$

5.1.3.2 Script

El script con el cual se representa el modelo anterior en el lenguaje de programación recibe como parámetro de entrada una de las dimensiones de la matriz, correspondiente a la variable N. El mapeo de las variables x_{ij} del modelo al lenguaje de programación se realiza a través de la tupla Square la cual se implementa a través de la siguiente sentencia de código:

```
{FD.tuple square NN 1#N Square}
```

En la sentencia anterior NN es una constante que toma el valor de N^2 y representa el número de variables definidas en el modelo descrito en 5.1.3.1, por su parte 1#N representa el dominio de las variables definidas en el modelo, lo que significa que cada variable puede tomar un valor entre 1 y N.

La restricción (5.8), es representada en el lenguaje de programación a través de la siguiente sentencia de código:

```
{ForAll ListNum
  proc {$ Fila }
    F = {FoldL ListNum
         fun {$ Lista Columna}
          {Field Fila Columna}|Lista
         end
```

```
      nil}
 in
   {FD.distinct F}
   {FD.sum F '=:' ((N+1)*N) div 2}
 end}
```

La idea perseguida con el fragmento de código anterior es verificar que los elementos de una fila sean todos distintos entre si, dicha verificación se realiza para todas y cada una de las fila de la matriz. En dicho fragmento de código la variable F es una lista que contiene los elementos que constituyen una fila, para la cual se establece que todos los elementos en F son distintos, lo cual se logra con la sentencia:

```
{FD.distinct F}
```

El mapeo al lenguaje de programación para el caso de la ecuación (5.9) que establece que los elementos de una columna son todos distintos entre si, es equivalente a lo descrito anteriormente y se expresa a través de las siguientes sentencias:

```
{ForAll ListNum
 proc {$ Columna}
   C = {FoldL ListNum
        fun {$ Lista Fila}
          {Field Fila Columna}|Lista
        end
        nil}
 in
   {FD.distinct C}
   {FD.sum C '=:' ((N+1)*N) div 2}
 end}
```

Como la tupla Square define la matriz como una especie de arreglo unidimensional se utiliza la función Fiel, la cual permite tener acceso a cada elemento (i, j) como si fuera una matriz. Dicha función es idéntica a la descrita en 5.1.2.2.

5.1.3.3 Caracterización

El modelo presentado en 5.1.3.1 permite establecer atributos que caracterizan al problema de los cuadrados latinos y visualizar someramente que tan problemática se puede convertir la resolución del mismo. Considerando N como una de las dimensiones de la matriz, dichos atributos se mencionan a continuación:

- Tamaño de los dominios: N .
- Número de variables: el modelo utiliza N^2 variables.
- Tamaño del espacio de búsqueda: N^{N^2}

En la Tabla 5.3 se detalla el tamaño del espacio de búsqueda par cada una de las instancias del problema de los cuadrados latinos.

Tabla 5. 3 Espacio de Búsqueda para instancias de Cuadrados Latinos

N	Tamaño del Espacio Búsqueda $(N)^{N^2}$
3	$1.9683x10^{4}$
4	$4.2949x10^{9}$
5	$2.9802x10^{17}$
10	$1x10^{100}$
15	$4.1738x10^{264}$

5.1.4 Sudoku

Sudoku es un puzzle jugado en una matriz de 9x9 (estándar) parcialmente llena (ver Figura 5.4), la cual esta compuesta de submatrices o "regiones" 3x3. La tarea es completar las celdas vacías de modo que cada columna, fila y región contengan números de 1 a 9 exactamente una vez.

	2	6				8	1	
3			7		8			6
4				5				7
	5		1		7		9	
		3	9		5	1		
	4		3		2		5	
1				3				2
5			2		4			9
	3	8				4	6	

Figura 5. 4 Ejemplo de un Problema Sudoku

5.1.4.1 Modelo

El modelo tiene una variable x_{ij} para cada celda (i, j) de la matriz, la cual representa el valor que cada celda puede tomar (1 a 9). Para restringir que cada fila y cada columna tengan valores de 1 a 9 exactamente una vez se deben imponer las siguientes restricciones:

$$\forall i \in \{1...9\} \text{ AllDifferent } \{x_{i1}, x_{i2},..., x_{i9}\} \quad (5.10)$$

$$\forall j \in \{1...9\} \text{ AllDifferent } \{x_{1j}, x_{2j},..., x_{9j}\} \quad (5.11)$$

Por otra parte, cada celda en la región S_{kl} con $0 \leq k, l \leq 2$ deben ser distintas:

$$\forall ij \ \text{AllDifferent} \{x_{ij}, x_{i(j+1),} x_{i(j+2)}, x_{(i+1)j}, x_{(i+1)(j+1)}, x_{(i+1)(j+2)}, x_{(i+2)j}, x_{(i+2)(j+1)}, x_{(i+2)(j+2)}\} \quad (5.12)$$

con $i = k * 3 + 1$ y $j = l * 3 + 1$

5.1.4.2 Script

El script con el cual se representa el modelo anterior en el lenguaje de programación recibe como parámetro de entrada un problema Sudoku similar al mostrado en la Figura 5.4. El mapeo de las variables x_{ij} del modelo al lenguaje de programación se realiza a través de la siguiente sentencia de código:

```
Root = {Map {List.make 9}
fun {$ Row}
    Row = {List.make 9}
    Row  ::: 1#9
    {FD.distinct Row}
    Row
end
 }
```

En la sentencia anterior, Row ::: 1#9 establece el dominio de las variables y {FD.distinct Row} asegura que todos los elementos de una fila sean distintos entre si, dando cumplimiento de esta forma la restricción (5.10). Por otra parte, la restricción (5.11), la cual establece que todos los elementos de una columna deben ser distintos entre si, es representada en el lenguaje de programación a través del siguiente fragmento de código:

```
    for X in 1; X =< 9; X+1 do
      {FD.distinct {FoldR Root fun {$ Row Prev} {Nth Row X} | Prev end nil}}
    end
```

Para asegurar el cumplimiento de la restricción (5.12) es necesario establecer las siguientes líneas de código:

```
for X in 0; X =< 2; X+1 do
  for Y in 0; Y =< 2; Y+1 do
    {FD.distinct
      {FoldR
        {List.filterInd Root fun {$ I Row}
                                  I >= 1+Y*3 andthen
                                  I =< 3+Y*3
                                end}
         fun {$ Row Prev}
            {Append
               {List.filterInd Row fun {$ I X}
                                        I >= 1+Y*3 andthen
                                        I =< 3+Y*3
                                      End}
                Prev
             }
          end
          nil
        }
      }
    end
  end
```

5.1.4.3 Caracterización

El modelo presentado en 5.1.4.1 permite establecer atributos que caracterizan a Sudoku y visualizar someramente que tan costosa puede convertirse su resolución. El problema Sudoku abordado en este trabajo y que corresponde al estándar según la literatura, esta formado por una matriz de dimensiones 9x9, de esta manera si consideramos N = 9, se tienen los siguientes atributos característicos:

- Tamaño de los dominios: N .
- Número de variables: de acuerdo al modelo presentado, se tienen N^2 variables.
- Tamaño del espacio de búsqueda: N^{N^2}

5.2 Heurísticas

En el capítulo 4 se explica en detalle el proceso de resolución de problemas combinatoriales utilizando las técnicas de propagación y distribución en la plataforma Mozart, donde la etapa de distribución de tal proceso es llevada a cabo por distribuidores quienes son los encargados de implementar las conocidas estrategias de distribución, dichas estrategias están constituidas por heurísticas de selección de variable y heurísticas de selección de valor, por lo cual en la presente sección se proporciona una descripción de las heurísticas a utilizar en la resolución de los problemas anteriormente expuestos.

5.2.1 Selección de Variables

De las tres heurísticas que se mencionan a continuación, las dos primeras corresponde a heurísticas de selección dinámicas y la última corresponde a una heurística de selección estática (ver sección 3.3).

- Seleccionar la variable con tamaño de dominio mínimo (DMi)

Una vez que se ha llegado a un estado estable o espacio estable según lo definido en la sección 4.3, y no se ha encontrado solución, se analiza el dominio de cada una de las variables aún no instanciadas, es decir, aquellas variables con tamaño de dominio mayor a 1, y se selecciona aquella con tamaño de dominio más pequeño.

Ejemplo: en un momento dado se tiene la variable $x_1 \in 1\#10$ y la variable $x_2 \in 1\#3$, utilizando esta heurística se seleccionará como variable a distribuir la variable x_2 por tener solo 3 elementos en su dominio, cantidad menor a la cantidad de elementos del dominio de x_1.

- Seleccionar la variable con tamaño de dominio máximo (DMa)

La idea de esta heurística es similar a la anterior, sin embargo en este caso se selecciona aquella variable aún no instanciada cuyo tamaño de dominio corresponde al más grande.

- Seleccionar la variable en base a un orden previamente establecido (BOr)

Corresponde a una heurística estática, ya que para seleccionar la variable se establece un orden para selección de variables antes de iniciar la búsqueda, y luego se van seleccionando las variables siempre en este mismo orden, sin considerar los cambios en los dominios de las variables producto de la ejecución de fases de propagación. A modo de ejemplo considere que en un momento dado se tienen las siguientes variables con sus respectivos dominios: $x_1 \in 1\#4$, $x_2 \in 3\#7$ y $x_3 \in 4\#5$. Si antes de iniciar cualquier cómputo se establece el orden de selección de variables x_2, x_3, x_1 este orden se preserva hasta que se encuentre la solución buscada.

En el presente informe, la heurística BOr se ha definido específicamente para la resolución del problema de los cuadrados mágicos, y consiste en seleccionar los elementos de la diagonal principal primero, luego los elementos del triángulo superior a la

diagonal principal y finalmente los elementos del triángulo inferior a la diagonal principal.

Ejemplo: considere N = 3

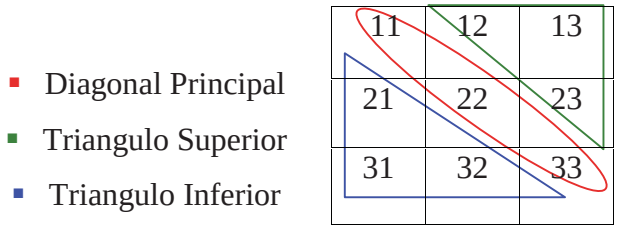


Figura 5. 5 División de la Matriz para N = 3

Lo anterior esta motivado principalmente por el hecho que los elementos de la diagonal principal están relacionados con más elementos de la matriz (ver Figura 5.5) que aquellos que no se encuentran en la diagonal (ver Figura 5.6), lo cual sugiere que al instanciar un elemento de la diagonal se están restringiendo más elementos, conduciendo de esta manera a una propagación de restricciones más eficaz.

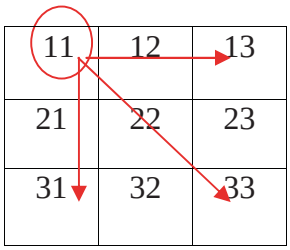


Figura 5. 6 Elemento de la diagonal

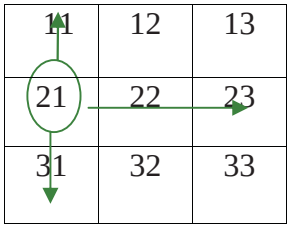


Figura 5. 7 Elemento no perteneciente a la diagonal

5.2.2 Selección de Valor

Ya se ha mencionado que una vez que se está en un estado estable, se debe seleccionar una variable aún no instanciada sobre la cual distribuir, una vez hecho esto último corresponde determinar con qué valor se distribuirá sobre esa variable, para lo cual se utilizan las heurísticas de selección de valor. A continuación se presentan las heurísticas utilizadas en la resolución de los problemas descritos en el presente trabajo.

- Seleccionar el menor valor del dominio

Esta heurística establece que se elija siempre el valor más pequeño del dominio. De ahora en adelante como objeto del presente trabajo se hará referencia a la heurística como MeVal.

Ejemplo: en un momento dado se tienen las variables $x_1 \in 1\#10$ y $x_2 \in 1\#3$, de las cuales se selecciona la variable x_1. Ahora, utilizando esta heurística de selección de valor se analiza el dominio de x_1 y se determina elegir el valor 1 por corresponder al valor más pequeño.

- Seleccionar el mayor valor del dominio

Heurística que en el presente informe se denomina como MaVal, y que básicamente es similar a la heurística anterior, pero en lugar de haber elegido el valor 1 del dominio de la variable x_1, se selecciona el valor 10 por corresponder al valor más grande del dominio.

- Seleccionar el valor medio del dominio

La heurística selecciona aquel valor del dominio que se encuentra más cerca de la mitad del dominio, para lo cual calcula la media aritmética entre los límites (superior e inferior) del dominio de la variable seleccionada y en caso de haber empate se selecciona el valor más pequeño. De ahora en adelante como objeto del presente trabajo se hará referencia a la heurística como MedVal.

Ejemplo: si ya se ha seleccionado la variable x_1 cuyo dominio es 2#4, para seleccionar el valor a instanciar se calcula el promedio entre los límites del dominio, esto es: (2+4) / 2 = 3, por lo tanto el valor que se selecciona en esta etapa corresponde al 3. Si el dominio de x_1 fuese 1#4 el valor a seleccionar sería 2, ya que el promedio es (1+4) / 2 = 2,5 y en caso de empate siempre se selecciona el valor menor.

- Seleccionar el valor inmediatamente mayor al valor medio del dominio

  En esta heurística se establece el valor medio del dominio, pero se selecciona aquel valor menor del dominio que a su vez es mayor al valor medio del dominio.

Considere el ejemplo anterior, cuando el dominio de la variable es 2#4 en la heurística anterior se selecciona el valor 3, en cambio con la actual heurística se selecciona el valor 4. Así, cuando el dominio de la variable es 1#4 la actual heurística seleccionará el valor 3. Para efectos del presente informe la heurística será referenciada como MMedVal.

5.2.3 Representación de las Estrategias de Distribución

Como ya se ha mencionado, las estrategias de distribución están constituidas por heurísticas de selección de variable y heurísticas de selección de valor, las cuales han sido descritas en la sección anterior. A continuación se proporciona una explicación de la representación en el lenguaje de programación de las estrategias formadas con dichas heurísticas.

Oz soporta la distribución a través de la sentencia {FD.distribute Dist Sol}, donde el vector Sol es distribuido de acuerdo a la especificación Dist, cual puede ser una etiqueta naive, ff, split o generic. En el caso particular de las estrategias implementadas en el presente informe se utiliza la etiqueta generic, a la cual se le ajustan los parámetros para order y value según lo especificado en la sección 4.4.

Para las estrategias que utilizan heurísticas de selección de variable del tipo DMi el parámetro para order se fija en size, lo cual permite seleccionar la variable cuyo tamaño de dominio es mínimo. Por su parte el parámetro para value puede ser fijado en min, max o mid para las heurísticas de valor del tipo MeVal, MaVal y MedVal respectivamente. Un ejemplo de la estrategia constituida por las heurísticas DMi y MeVal se representa a través de la siguiente sentencia:

```
{FD.distribute generic(order: size value: min) Sol}
```

De manera particular, en la estrategia constituida por la heurística de selección de valor del tipo MMedVal el parámetro de value debe ser fijado a una función que retorne aquel valor del dominio que es inmediatamente mayor al valor medio, la implementación se logra a través de la siguiente sentencia:

```
{FD.distribute generic(order: size
                       value: fun{$ S}
                                 {FD.reflect.nextLarger S {FD.reflect.mid S}}
                              end) Sol}
```

Al igual que para las estrategias que utilizan heurísticas de selección de variable del tipo DMi, para las estrategias que utilizan heurísticas del tipo DMa se deben fijar los parámetros para order y value, pero en lugar de fijar order a size, se fija a una función que nos permita discriminar entre el tamaño de los dominios de las variables y elegir aquella que tenga el tamaño de dominio mayor, dicha función se expresa como sigue:

```
fun {$ X Y}
   if {FD.reflect.size X} > {FD.reflect.size Y} then
      true
   else
     false
```

```
    end
end
```

Para el caso de value se conserva lo explicado para las estrategias que utilizan heurísticas de selección de variable del tipo DMi. Un ejemplo de la estrategia constituida por las heurísticas DMa y MeVal se representa a través de la siguiente sentencia:

```
{FD.distribute generic(order: fun {$ X Y}
                                  if {FD.reflect.size X} > {FD.reflect.size Y} then
                                    true
                                  else
                                    false
                                  end
                                end
                       value: min) Sol}
```

Todo lo expuesto para las estrategias que utilizan heurísticas del tipo DMi y DMa referente a value se conserva para las estrategias que utilizan heurísticas de selección de variable estáticas (BOr), sin embargo order es fijado a naive, lo cual establece que se selecciona la variable de más a la izquierda en una lista que establece el orden de selección, en este caso en particular denominada Lista2. Un ejemplo de una estrategia que utiliza la heurística BOr y MeVal se establece con la siguiente sentencia:

```
{FD.distribute generic (order: naive value: min) {Map Lista2 fun {$ I} Sol.I end}}
```

Para efectos prácticos de este informe las distintas estrategias de distribución serán numeradas de 1 a 12 y se identifican de la siguiente manera: S_1, S_2, S_3,…, S_{12}, quedando constituidas como se muestra en la Tabla 5.4.

Tabla 5. 4 Constitución de las Estrategias de Distribución

	Heurística de Selección de Variable		Heurística de Selección de Valor
S_1	DMi	+	MeVal
S_2	DMi	+	MaVal
S_3	DMi	+	MedVal
S_4	DMi	+	MMedVal
S_5	DMa	+	MeVal
S_6	DMa	+	MaVal
S_7	DMa	+	MedVal
S_8	DMa	+	MMedVal
S_9	BOr	+	MeVal
S_{10}	BOr	+	MaVal
S_{11}	BOr	+	MedVal
S_{12}	BOr	+	MMedVal

5.2.4 Indicadores de Desempeño

Una estrategia de distribución, como se ha mencionado, consiste en seleccionar una variable y un valor de su dominio con los cuales distribuir el espacio actual. Dichas estrategia de distribución, tal como se menciona a lo largo del capítulo 3, son guiadas por heurísticas de selección de variable y heurísticas de selección de valor. Dependiendo del orden en que se seleccionan tales variables y valores se puede generar diferencias significativas en la resolución del problema. Con el objetivo de medir y establecer tales diferencias es necesario establecer algún indicador de desempeño que permita de alguna manera reflejar con claridad el efecto de las estrategias y formar un juicio respecto del comportamiento de estas, para lograr esto último se llevo a cabo un recorrido teórico que

permitió identificar una serie de indicadores utilizados en la literatura [23][24], con lo cual una vez realizado el estudio y evaluada las características de la técnica de resolución aquí empleada se establecieron como indicadores de desempeño (utilizados como criterios de eficiencia) los siguiente:

- Número de Backtracking (B): muestra la cantidad de malas decisiones hechas durante la búsqueda de la solución, es decir, cálculos o decisiones ejecutadas sin conducir a una solución.

- Número de Enumeraciones (E): esta métrica cuenta la cantidad de nodos o espacios generados para encontrar la solución del problema, incluyendo las buenas enumeraciones que conducen a una solución y las malas enumeraciones que obligan a retroceder.

- Tiempo: esta métrica mide el tiempo requerido para resolver el problema.

Entonces, para establecer la eficiencia de las estrategias de distribución se deben considerar cada una de las métricas seleccionadas, por lo cual una estrategia tendrá un mejor desempeño con respecto a otra, si logra reducir el valor que toman estas métricas.

5.3 Descripción de la Implementación

La implementación que permite, tanto la resolución de los problemas expuestos con anterioridad y la implementación de las estrategias de distribución antes descritas, es llevada a cabo mediante una combinación de paradigmas, los cuales incluyen Programación con Restricciones y Programación Orientada a Objetos, tal combinación ha sido posible gracias a la característica de multiparadigma del lenguaje Oz.

Mediante la Programación con Restricciones se crean los script de resolución de los problemas, es decir, se realiza el mapeo del modelo del problema al lenguaje de programación, con esto básicamente se establecen las restricciones del modelo. Por su parte, con la Programación Orientada a Objetos se modulariza el sistema en componentes con funciones claramente establecidas facilitando la medición de indicadores y aspectos de interoperabilidad del sistema.

La plataforma de desarrollo Mozart proporciona ciertas abstracciones que facilitan la resolución de problemas mediante Programación con Restricciones según lo establecido en [10], sin embargo estas abstracciones del lenguaje impiden visualizar con claridad el proceso de propagación de restricciones y distribución llevado a cabo, ocultando información que en el caso de esta investigación resulta relevante, especialmente para la medición de indicadores de desempeño, es así como la Programación Orientada a Objetos a permitido transparentar el proceso y tener acceso a cierta información, tal como el número de espacios generados, los estados de un espacio, entre otros. En cuanto a la interoperabilidad del sistema, la programación Orientada a Objetos ha facilitado la lectura de datos de entrada necesarios para la resolución del problema y lo que es aún más relevante, ha permitido la generación de archivos con datos obtenidos durante el proceso de resolución, en este caso particular se generan archivos con las mediciones hechas de cada indicador.

Los elementos esenciales del componente de software desarrollado se describen a continuación:

- Etapa de Control o Iniciación: es la etapa encargada de iniciar el proceso de resolución, acá se establece el script a utilizar basándose en el problema seleccionado para la resolución.

- Clases:
  - ✓ Clase Distribute: considerando lo descrito en [4] esta parte de la implementación es la encargada de realizar la etapa de distribución, para lo cual se evalúa la condición del espacio actual (fallido, estable, resuelto) y en base a esto se determinan las acciones a seguir. Es aquí donde se puede realizar el registro de la cantidad de enumeraciones y backtracking, además de controlar que el proceso de resolución no exceda el tiempo limite establecido.

  - ✓ Clase TexFile: esta clase tiene relación con la interoperabilidad del sistema. Permite generar los archivos de salida conteniendo las mediciones hechas de cada indicador de desempeño y para cada instancia del problema resuelto.

- Mapeo del Modelo: corresponde al script que permite la representación del problema en el lenguaje de programación, acá se establecen las variables, dominios y restricciones del problema a resolver.

- Parámetros: en este punto se le indica al algoritmo el problema y la instancia de dicho problema a resolver a través del paso de parámetros. En este punto, y considerando el número de instancias resueltas para cada problema, se automatizó el paso de parámetros mediante un script Python.

CAPÍTULO 6

Análisis de Resultados

6.1 Consideraciones Previas a la Simulación Computacional

Para la implementación de los script presentados en secciones anteriores se utiliza la plataforma de desarrollo Mozart, su utilización en el presente trabajo persigue el objetivo implícito de obtener el máximo de sus potencialidades poco exploradas hasta el momento en la realidad más próxima. La versión utilizada de la plataforma desarrollo corresponde a la versión 1.3.1, y las pruebas realizadas se ejecutaron en Athlon XP 1800+ con 256 MB de memoria RAM.

Cada uno de los problemas expuestos en el capítulo 5 se resuelve utilizando las ocho primeras estrategias listadas en la Tabla 5.4 del mismo capítulo, y adicionalmente se utilizan las estrategias $S_9 \ldots S_{12}$ en la resolución del problema de los cuadrados mágicos, esto último debido principalmente a que dichas estrategias están constituidas por una heurística de selección de variables diseñada específicamente para tal problema.

Llevar a cabo lo expuesto en el párrafo anterior permite evaluar el desempeño de las estrategias de distribución basándose principalmente en las métricas definidas en la sección 5.2.4 del presente informe.

Cada ejecución tiene un tiempo limite de 10 minutos, tiempo durante el cual se debe encontrar una solución, de no encontrarla se detiene la ejecución del script y los resultados se muestran con el símbolo “-”.

6.2 Búsqueda de la Primera Solución

En la presente sección se muestran las tablas con los resultados obtenidos en resolución de los problemas buscando la primera solución.

En general es posible apreciar que para instancias pequeñas las estrategias no reflejan diferencias significativas, sin embargo al observar los resultados obtenidos para cada uno de los problemas se puede visualizar que el desempeño de las estrategias varía a medida que N aumenta, esto debido principalmente a que con el incremento de N el tamaño el espacio de búsqueda crece drásticamente [34].

Por otra parte, al observar los resultados obtenidos se percibe que las estrategias constituidas por la heurística DMi (S_1,…, S_4) tienen mejor comportamiento en aquellas instancias en que el espacio de búsqueda crece, esto en comparación con estrategias que son guiadas por la heurística DMa (S_5,…, S_8). Tales diferencias ocurren principalmente porque la heurística DMi conduce a un espacio fallido lo antes posible, permitiendo podar el árbol de búsqueda. Conducir a un espacio fallido rápidamente consiste en elegir variables con pocos elementos en su dominio aumentando la probabilidad de fallar antes de generar un árbol de búsqueda demasiado grande.

Tabla 6. 1 N-Reinas: Enumeraciones, Backtracking, tiempo de CPU en ms.

	4-reinas			10-reinas			20-reinas			50-reinas			100-reinas		
	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)
S_1	5	3	10	24	17	10	76	60	15	1065	1021	118	137	41	403
S_2	5	3	9	24	17	11	76	60	15	1065	1021	118	137	41	560
S_3	1	0	9	6	0	10	41	27	13	56	11	72	110	13	554
S_4	1	0	11	7	1	10	89	74	18	47	1	73	100	3	606
S_5	6	3	10	820	807	45	650752	650717	51138	-	-	-	-	-	-
S_6	6	3	10	820	807	45	650752	650717	51297	-	-	-	-	-	-
S_7	1	0	10	6	1	10	252979	252952	22235	-	-	-	-	-	-
S_8	1	0	9	38	30	11	5018	5001	309	-	-	-	-	-	-

Tabla 6. 2 Cuadrados Mágicos: Enumeraciones, Backtracking, tiempo de CPU en ms

(Con Restricciones Redundantes + Restricciones que eliminan simetrías)

N^2	9			16			25			49		
	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)
S_1	4	1	1	741	717	30	208861	208827	12669	-	-	-
S_2	13	7	1	1681	1655	63	9113251	9113227	505231	-	-	-
S_3	8	5	1	2465	2449	81	2187	2158	76	101510	101424	4415
S_4	13	7	1	406	384	27	7061	7037	452	7139	7081	487
S_5	16	7	1	45097	45043	3680	-	-	-	-	-	-
S_6	14	8	1	681	646	46	-	-	-	-	-	-
S_7	22	10	1	847094	847055	48519	-	-	-	-	-	-
S_8	23	19	1	358775	358772	27115	557090	557608	44189	-	-	-
S_9	5	2	0	49	34	3	26475	26428	1396	-	-	-
S_{10}	13	7	1	508	491	24	80761	80728	5078	-	-	-
S_{11}	8	5	1	1323	1296	56	393	375	21	-	-	-
S_{12}	13	7	1	791	763	55	12	1	2	82107	82047	6649

Observando los resultados obtenidos para el problema de los cuadrado mágicos (ver Tabla 6.2), uno de los aspectos a destacar son los buenos resultados obtenidos por las estrategias constituidas por la heurística BOr, y en particular por aquellas que utilizan dicha heurística en combinación con heurísticas de selección de valor MedVal y MMedVal, este buen comportamiento reflejado en la tabla de resultados se debe principalmente a lo expresado en la sección 5.2.1, donde se explica básicamente que la definición de la heurística BOr conduce a una propagación de restricciones más eficaz producida por seleccionar en primer lugar elementos de la diagonal principal, los cuales están relacionados con un mayor número de elementos de la matriz, provocando de esta forma restringir más elementos de la matriz.

Para el problema de los cuadrados mágicos es posible apreciar (ver Tabla 6.2 y la Tabla 6.3) que las restricciones redundantes y restricciones que eliminan simetrías ayudan a obtener mejores resultados en virtud de los indicadores definidos. Esto último se debe a que una restricción redundante reduce el esfuerzo de búsqueda requerido para solucionar un CSP mediante la realización de propagación. Por su parte, una restricción que elimina

simetrías es una restricción que elimina soluciones, de tal manera de hacer más pequeño el espacio de búsqueda.

Por otra parte, es posible mencionar que el tamaño del espacio de búsqueda tienen una gran incidencia en el proceso de resolución, donde su crecimiento exponencial vuelve el proceso considerablemente más costoso, esto último es posible apreciarlo tanto al resolver distintas instancias de un mismo problema como al resolver los diferentes problemas expuestos.

Tabla 6. 3 Cuadrados Mágicos sin Restricciones Redundantes

N^2	9		16		25	
	(E)	(B)	(E)	(B)	(E)	(B)
S_1	99	86	1718	1685	-	-
S_2	99	86	1718	1685	-	-
S_3	175	165	51190	51166	8853	8811
S_4	12	5	50063	50032	-	-
S_5	306	286	5765	5701	-	-
S_6	306	286	5765	5701	-	-
S_7	37910	37884	-	-	-	-
S_8	27	19	-	-	-	-
S_9	86	73	10533	10495	-	-
S_{10}	86	73	10533	10495	-	-
S_{11}	116	109	8946	8917	366	333
S_{12}	54	42	2840	2817	-	-

Tabla 6. 4 Cuadrados Latinos: Enumeraciones, Backtracking, tiempo de CPU en ms.

N	3			4			5			10			15		
	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)
S_1	3	0	9	6	0	10	10	0	10	67	1	18	165	7	76
S_2	3	0	10	6	0	10	10	0	10	67	1	18	165	7	78
S_3	3	0	10	6	0	10	12	0	12	70	2	18	163	5	67
S_4	3	0	10	6	0	10	10	0	11	70	4	31	309	138	229
S_5	3	0	12	8	0	10	94	77	16	-	-	-	-	-	-
S_6	3	0	9	8	0	10	94	77	15	-	-	-	-	-	-
S_7	4	0	10	8	0	10	1644	1625	106	-	-	-	-	-	-
S_8	3	0	10	8	0	10	36	21	12	-	-	-	-	-	-

En lo que respecta a la resolución de Sudoku, se han utilizado diferentes instancias del puzzle publicadas en el periódico “The Times” [27], donde usualmente se proporciona la dificultad del puzzle. Por otra parte también se ha utilizado una colección [28] de puzzles que poseen solo 17 números dados en la matriz inicial, esta cantidad corresponde al número conocido más pequeño de números dados en la matriz inicial [26], estos últimos no son clasificados por dificultad.

En la Tabla 6.5 y Tabla 6.6 se resumen los resultados de las pruebas realizadas, en donde se presentan la fuente desde donde se obtuvo el puzzle, el grado (nivel de dificultad), la heurística utilizada para resolver cada instancia y los distintos indicadores de desempeño medidos durante la ejecución de las pruebas.

Si bien algunos autores establecen que la cantidad de números dados inicialmente no tiene incidencia en el grado de dificultad de Sudoku, los resultados aquí obtenidos muestran que bajo la técnica de resolución empleada, la eficiencia de resolución deja bastante que desear cuando se tienen solo 17 números inicialmente en comparación con los otros casos (fácil, medio y difícil), donde la cantidad de números dados esta por sobre los 25.

Tabla 6. 5 Sudoku resuelto con heurísticas DMi (tamaño de dominio mínimo)

		S_1			S_2			S_3			S_4		
Fuente	Grado	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)
[28]	Ninguno	84	52	14	220	195	21	1308	1283	88	183	159	26
[28]	Ninguno	2836	2815	153	271	249	23	11074	11048	603	124	102	22
[27]	Fácil	7	3	11	17	13	11	7	3	10	17	13	12
[27]	Medio	16	6	11	174	164	19	16	6	11	174	164	26
[27]	Difícil	27	16	11	24	18	11	27	16	11	24	18	12

Tabla 6. 6 Sudoku resuelto con heurísticas DMa (tamaño de dominio máximo)

		S_5			S_6			S_7			S_8		
Fuente	Grado	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)	(E)	(B)	(t)
[28]	Ninguno	-	-	-	-	-	-	-	-	-	-	-	-
[28]	Ninguno	-	-	-	-	-	-	-	-	-	-	-	-
[27]	Fácil	18554	18537	1799	274476	274472	28149	24195	24169	2582	721773	72155	7484
[27]	Medio	-	-	-	121135	121113	12868	-	-	-	88720	88706	9763
[27]	Difícil	-	-	-	-	-	-	93138	93105	9158	-	-	-

Para finalizar, es necesario destacar que una vez analizados los resultados obtenidos en cada uno de los problemas resueltos se percibe que diferentes estrategias de enumeración tienen rendimientos significativamente diferentes en distintas instancias de un mismo problema. Por otra parte, se aprecia también el hecho que no existe una estrategia de enumeración que sea la mejor para todos los problemas expuestos, lo cual hace pensar que tomar una decisión a priori respecto de una buena estrategia de enumeración es un trabajo bastante difícil.

6.3 Búsqueda de Todas las Soluciones

Encontrar todas las soluciones de un problema involucra necesariamente generar el árbol de búsqueda completo, se esta forma se pensó inicialmente que utilizar heurísticas de selección de variable y valor poco o nada pueden hacer por mejorar este proceso, sin embargo analizando los resultados obtenidos en la resolución de los problemas en busca de todas soluciones es posible ver que sí se generan diferencias, y básicamente utilizar heurísticas de selección de variable y valor adecuadas en el proceso de buscar todas las soluciones tiene un efecto similar al mostrado por dichas heurísticas en la resolución

buscando una solución. Estas diferencias son posible percibirlas debido a que utilizando la técnica de resolución descrita, los valores de las variables son asignados a través de una secuencia de puntos de elección binarios, es decir, cuando una variable x_i es seleccionada para insatnciación, los valores son asignados en el orden $\{a_1, a_2,.., a_n\}$, donde la primera elección crea dos alternativas $x_i = a_1$ y $x_i \neq a_1$. De esta manera la rama izquierda es explorada, si esta falla o si se requieren todas las soluciones, se vuelve al punto de elección inicial y la rama derecha es explorada. El proceso descrito y los resultados obtenidos serian muy diferente si se utilizaran algoritmos de búsqueda del tipo "Backtracking Cronológico" y estructura de árboles N-arios ($N > 2$) [12].

En la Tabla 6.5 es posible observar las mediciones realizadas para el problema de los cuadrados mágicos, donde solo para las instancias 3 y 4 fue posible encontrar todas las soluciones, esto por que al tener que generar el árbol de búsqueda completo el proceso de búsqueda se vuelve mucho más complejo y costoso, por lo cual para instancias grandes del problema, donde el tamaño del espacio de búsqueda es considerablemente mayor, esto se vuelve casi impracticable impidiendo encontrar las soluciones dentro del tiempo límite impuesto. También es posible observar que las estrategias guiadas por la heurística estática BOr (S_9, S_{10}, S_{11}, S_{12}) claramente presentan un mejor comportamiento que las estrategias que usan otro tipo de heurísticas.

Tabla 6. 7 Cuadrados Mágicos (Buscando todas las soluciones):
Enumeraciones, Backtracking, Tiempo de CPU en ms

N^2	9			16		
	(E)	(B)	(t)	(E)	(B)	(t)
S_1	12	6	1	23168	11413	894
S_2	14	7	1	22346	11002	874
S_3	14	7	1	35914	17786	1212
S_4	14	7	1	27602	13630	1875
S_5	46	23	3	834792	417225	68566
S_6	48	24	3	600878	300268	50617
S_7	208	104	11	-	-	-
S_8	46	23	3	3245426	1622542	258269
S_9	10	5	0	14210	6934	745
S_{10}	14	7	1	14558	7108	753
S_{11}	16	8	1	19500	9579	895
S_{12}	14	7	1	16392	8025	1207

Conclusión

En este trabajo se ha presentado un estudio acabado de heurísticas de selección de variable y valor, y como su utilización afecta la eficiencia de resolución de problemas combinatoriales, dicha eficiencia de resolución es medida en base a ciertas métricas, permitiendo reflejar los cambios producidos al utilizar distintas heurísticas. El trabajo realizado incluye el modelamiento y resolución, tanto de problemas clásicos a resolver como de las estrategias de distribución guiadas por las heurísticas propuestas. La implementación se realizó en el sistema de programación Mozart, el cual se basa en el Lenguaje de Programación Oz.

Mediante las pruebas realizadas se logró mostrar claramente que es posible obtener mejores resultados en el proceso de búsqueda utilizando criterios adecuados de selección de variables y valores. En esencia, seleccionar una variable en un momento dado del proceso de búsqueda implica determinar si los nodos o espacios descendientes del espacio actual tienen o no solución, el que tengan solución no genera mayor conflicto ya que cualquier variable seleccionada es adecuada, sin embargo si en los nodos descendientes no hay solución lo más adecuado es seleccionar aquella variable que antes descubre tal situación, y así evitar empezar a realizar cálculos innecesarios que tarde o temprano no van a servir. Es por esto, que las estrategias evaluadas en este trabajo y que son guiadas por la heurística que selecciona la variable con tamaño de dominio mínimo (DMi) presentan un mejor comportamiento en comparación con las otras estrategias, ya que DMi lo que busca es asignar aquella variable que aparentemente conduce antes a un espacio fallido evitando estos cálculos innecesarios.

Atendiendo a las pruebas realizadas se pudo observar que las posibilidades de resolución de un determinado problema dependen del tamaño del espacio de búsqueda, esto último se puede apreciar al observar las diferencias generadas en los resultados obtenidos en la resolución de distintas instancias de un mismo problema.

Por otra parte, si se observan los resultados obtenidos en la resolución del problema de los cuadrados mágicos con restricciones redundantes y restricciones que eliminan simetrías, y sin este tipo de restricciones, se aprecia más claramente la diferencia producida por el tamaño del espacio de búsqueda, este último resulta ser un factor importante en la eficiencia de resolución, por una parte tiene incidencia el tamaño inicial del espacio de búsqueda y por otra que tan efectiva es la reducción del tamaño de dicho espacio durante el proceso de búsqueda. En el primer caso resulta de vital importancia el modelo utilizado, específicamente la definición de las variables y los dominios de ellas, en cuanto al segundo caso es relevante las restricciones utilizadas, de manera que estas reduzcan considerablemente el tamaño del espacio, es en esto último donde las restricciones redundantes y las restricciones que eliminan simetrías juegan un papel importante, por una lado las restricciones redundantes si bien son lógicamente derivadas de las restricciones básicas del modelo, su implementación no es lógica sino computacional, con lo cual es posible que su inclusión permita podar parte del árbol de búsqueda no podado por las restricciones básicas del problema. En cuanto a las restricciones que eliminan simetrías estas reducen el espacio de búsqueda mediante la eliminación de las soluciones simétricas, y en caso de querer otras o todas las soluciones del problema se generan por simetría a partir de la que ya se tiene, y no por un proceso de búsqueda mucho más costoso. También se puede concluir, de manera mucho más especifica y centrada en los resultados obtenidos de la resolución del problema de los cuadrados mágicos, es que una heurística de selección de variable estática no necesariamente presentará un mal comportamiento (según lo establecido por algunos autores) frente a una heurística dinámica, esto queda condicionado básicamente al modelo que se utiliza para representar el problema y a la definición de la heurística considerando dicho modelo.

Para finalizar entonces, es posible afirmar que las heurísticas de selección de variable y valor afectan la eficiencia del proceso de búsqueda, haciéndolo menos problemático o costoso, esto ha quedado demostrado experimentalmente a través de los resultados obtenidos. Además, producto de todo el estudio realizado se establece que es necesario considerar el modelo utilizado para la representación del problema al momento de diseñar dichas heurísticas, es decir, es preciso inicialmente centrar los esfuerzos en modelar adecuadamente el problema para luego en base a dicho modelo definir las heurísticas de selección de variable y valor.

Resulta interesante mencionar que producto de este trabajo se han logrado publicaciones en congresos y conferencias a nivel nacional e internacional. A continuación se listan los documentos publicados:

- Enumeration Strategies in Constraint Programming for Solving Puzzles [37]
- Variable and Value Selection Heuristics:Application to Solve Puzzles [39]
- Estrategias de Enumeración en Programación con Restricciones para la resolución de Puzzles [38]

Finalmente, atendiendo al estudio realizado y los resultados obtenidos, se plantea la posibilidad probar la utilidad de las heurísticas de selección de variable y valor en la resolución de problemas de optimización e incursionar en la definición de este tipo de heurísticas utilizando otras técnicas tales como Búsqueda Local o Ant Colony Optimization Metaheuristic (ACO). Por otra parte, y dado que se ha podido observar que no existe una estrategia de enumeración que sea la mejor para todos los problemas expuestos, se esta interesado en extender este trabajo a futuras investigaciones que permitan proponer algún mecanismo de resolución eficiente para diferentes problemas, independizando en mayor nivel la relación problema-estrategia.

Referencias

[1] Federico Barber y Miguel A. Salido. Introducción a la Programación de Restricciones. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. Nº 20, pp. 13-30, 2003

[2] R. Barták. On-Line Guide to Constraint Programming. First Edition, 1998 http://kti.ms.mff.cuni.cz/~bartak/constraints/index.html

[3] C. Schulte and G. Smolka. Finite Domain Constraint Programming in Oz. A Tutorial. 1998

[4] Julio Hernando Vargas. Programación Concurrente con Restricciones en Mozart. Scientia Et Technica, Año X, Nº 24, Mayo 2004.

[5] A. Kwan and E. P K. Tsang. Comparing CSP Algorithms Without Considering Variable Ordering Heuristics can be Misleading. Tech. Report CSM-262, Colchester, UK, 1995.

[6] Patrick Prosser, J. Christopher Beck, and Richard Wallace. Toward Understanding Variable Ordering Heuristic for Constraint Satisfaction Problems. Fourteenth Irish Artificial Intelligence and Cognitive Science Conference – AICS 2003, pp 11 – 16.

[7] Paula Sturdy. Learning Good Variable Orderings. CP (Francesca Rossi, ed.), Lecture Notes in Computer Science, Vol. 2833, pp. 997, Springer, 2003

[8] S. Bain, J. Thornton and A. Sattar. Envolving Variable Ordering Heuristics for Constrained Optimisation. Principles and Practice of Constraint Programming - CP 2005, Barcelona, España, Octubre 2005.

[9] Carlos. Castro, Eric Monfroy, Christian, Figueroa and Rafael Meneses. An approach for dynamic split strategies in constraint solving, MICAI, Lecture Notes in Computer Science, Vol. 3789, pp. 162-174, Springer, 2005.

[10] Peter Van Roy and Seif Haridi. Concepts, Techniques, and Models of Computer Programming. MIT press, Cambridge, March 2004

[11] K. Apt. Principles of Constraint Programming. Cambridge University Press, 2003

[12] Barbara M. Smith and Paula Sturdy. Value Ordering for Finding All Solutions. IJCAI. (Leslie Pack Kaelbling and Alessandro Saffiotti, eds.), Professional Book Center, pp. 311- 316, 2005.

[13] Tobias Müller and Martin Müller. Finite Set Constraint in Oz. Workshop Logische Programmierung (Technische Universität München), pp. 104-115, 1997.

[14] E. Tsang. Foundations of Constraint Satisfaction. Academic Press, London, 1993

[15] P. A. Geelen. Dual Viewpoint heuristics for binary constraint satisfaction problems. Proc. Of the 10th ECAI, pp. 31-35, Vienna, Austria, 1992.

[16] J. Christopher Beck, Patrick Prosser and Richard Wallace. Variable Ordering Heuristics Show Promise. CP (Mark Wallace, ed.), Lecture Notes in Computer Science, Vol. 3258, pp. 711-715, Springer, 2004

[17] C. Schulte, G. Smolka, T. Müller and D. Brill. Constraint Programming in Alice. A Tutorial. Julio - Noviembre 2005

[18] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. Principles and Practice of Constraint Programming, pp. 179-193, 1996

[19] Willem Jan van Hoeve. Operations Research Techniques in Constraint Programming. PhD thesis, University of Amsterdam, 2005.

[20] Norman M. Sadeh, Mark S. Fox. Variable and Value Ordering Heuristics for the Job Shop Scheduling Constraint Satisfaction Problem. Artificial Intelligence 86, Nº 1, pp. 1-41, 1996

[21] M. Cadoli, T. Mancini, F. Patrizi. SAT as an Effective Solving Technology for Constraint Problems. The 16th International Symposium on Methodologies for Intelligent Systems (ISMIS 96), Bari, Italia, 2006.

[22] Eric Monfroy, Carlos Castro, and Broderick Crawford. Using Local Search for Guiding Enumeration in Constraint Solving. AIMSA (Jérôme Euzenat and John Domingue, eds.), Lecture Notes in Computer Science,Vol. 4183, pp. 56-65, Springer, 2006.

[23] James E. Borrett, Edward P. K. Tsang, and N. R. Walsh, Adaptive constraint satisfaction: The quickest first principle, ECAI (Wolfgang Wahlster, ed.), John Wiley and Sons, Chichester, 1996, pp. 160–164.

[24] Eric Monfroy, Carlos Castro, and Broderick Crawford. Adaptive enumeration strategies and metabacktracks for constraint solving, ADVIS (Tatyana M. Yakhno and Erich J. Neuhold, eds.), Lecture Notes in Computer Science, vol. 4243, pp. 354–363, Springer, 2006.

[25] P. Zoeteweij. Composing Constraint Solvers. Impreso por PrintPartners Ipskamp, Enschede, 2005.

[26] Helmut Simonis, Sudoku as a constraint problem, Proc. 4th Int. Works. Modelling and Reformulating Constraint Satisfaction Problems (Brahim Hnich, Patrick Prosser, and Barbara Smith, eds.), pp. 13–27, 2005.

[27] The Times. Spring Su Doku. April 7, 2007.

[28] G. Royle. Minimum Sudoku. http://www.csse.uwa.edu.au/~gordon/sudoku17

[29] R. Barták. Constraint Programming: In Pursuit of the holy grail. In Proceedings of the Week of Doctoral Students (WDS), Prague, Czech Republic, June 1999.

[30] Felip Manyá y Carla Gomes. Técnicas de Resolución de Problemas de Satisfacción de Restricciones. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. N° 19, pp. 169 – 180, 2003.

[31] Miguel A. Salido. Técnicas para el manejo de CSPs no binarios. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. N° 20, pp. 95 – 110, 2003.

[32] Fahiem Bacchus and Meter Van Beek. On the conversion between non-binary and binary Constraint Satisfaction Problem. AAAI/IAAI, pp. 310 – 318, 1998.

[33] Francesca Rossi, Charles J. Petrie and Vasant Dhar. On the equivalence of constraint satisfaction poblems. ECAI, pp. 550 – 556, 1990.

[34] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. IPSJ SIG Notes, 2002(103):9–16.

[35] B. Smith. Succeed-First or Fail-First: A case study in variable and value ordering. In: Proceeding of the Parallel Computing Technologies 4th International Conference, pp. 321-330. Yaroslavl, 1997.

[36] D. Frost, R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In: International Joint Conference on Artificial Intelligence, IJCAI, 1995, pp. 572-578

[37] Broderick Crawford, Mary Aranda, Carlos Castro, Eric Monfroy. Enumeration Strategies in Constraint Programming for solving Puzzles. Electronics, Robotics and Automotive Mechanics Conference, CERMA 2007, Cuernavaca Morelos, México, 25-28 of September, 2007. (Paper indexed by IEEE ISI Proceedings)

[38] Broderick Crawford, Mary Aranda, Carlos Castro y Eric Monfroy. Estrategias de Enumeración en Programación con Restricciones para la Resolución de Puzzles. VII Congreso Chileno de Investigación Operativa, OPTIMA 2007, Puerto Montt, Chile, 21-23 de Noviembre, 2007.

[39] Broderick Crawford and Mary Aranda. Variable and Value Selection Heuristics: Application to Solve Puzzles. In Proc. of 6th Mexican International Conference on Artificial Intelligence, MICAI 2007, Aguascalientes, Mexico, 4-10 of November, 2007. (Paper accepted but not yet published - Research in Computing Science)

Apéndices

Apéndice A: Código Fuente

A.1 Búsqueda de una Solución

El código que a continuación se muestra permite resolver cada uno de los problemas planteados en este trabajo. La etiqueta <ESTRATEGIA> indica la sección en donde se debe incluir la estrategia de distribución a utilizar, las cuales se muestran en el Apéndice C.

```
functor
import
   T at 'TiempoMilisegundos.so{native}'
   FD
   Application
   Open
   OS
   Space
define
   Lista2
    A = {NewDictionary}
    A.1 := S1
    A.2 := S2
    A.3 := S5
    A.4 := Se
    A.5 := Sm
    A.6 := Sd
```

S1 =[
[_ _ _ _ _ _ _ 1 _]
[4 _ _ _ _ _ _ _ _]
[_ 2 _ _ _ _ _ _ _]
[_ _ _ _ 5 _ 4 _ 7]
[_ _ 8 _ _ _ 3 _ _]
[_ _ 1 _ 9 _ _ _ _]
[3 _ _ 4 _ _ 2 _ _]
[_ 5 _ 1 _ _ _ _ _]
[_ _ _ 8 _ 6 _ _ _]
]

S2 =[
[_ _ _ _ _ _ _ 1 _]
[4 _ _ _ _ _ _ _ _]
[_ 2 _ _ _ _ _ _ _]
[_ _ _ _ 5 _ 6 _ 4]
[_ _ 8 _ _ _ 3 _ _]
[_ _ 1 _ 9 _ _ _ _]
[3 _ _ 4 _ _ 2 _ _]
[_ 5 _ 1 _ _ _ _ _]
[_ _ _ 8 _ 7 _ _ _]
]

S5 =[
[_ _ _ _ _ _ _ 1 2]
[_ _ 8 _ 3 _ _ _ _]
[_ _ _ _ _ _ _ 4 _]
[1 2 _ 5 _ _ _ _ _]
[_ _ _ _ _ 4 7 _ _]
[_ 6 _ _ _ _ _ _ _]
[5 _ 7 _ _ _ 3 _ _]
[_ _ _ 6 2 _ _ _ _]
[_ _ _ 1 _ _ _ _ _]
]

```
Se =[
        [1 _ 3 _ _ 4 _ _ 7]
        [_ _ 2 _ _ 7 _ 3 _]
        [9 8 _ _ 5 _ _ _ _]
        [_ _ _ 6 _ 2 _ 7 3]
        [_ _ 8 _ 1 _ 6 _ _]
        [3 2 _ 5 _ 9 _ _ _]
        [_ _ _ _ 2 _ _ 9 6]
        [_ 3 _ 9 _ _ 7 _ _]
        [7 _ _ 1 _ _ 4 _ 2]
      ]

Sm =[
        [1 7 _ _ _ _ 2 _ 3]
        [5 8 _ _ 9 _ _ _ _]
        [_ _ _ 3 7 _ 8 _ 5]
        [_ _ 2 4 _ _ _ _ _]
        [_ 5 4 _ _ _ 1 2 _]
        [_ _ _ _ _ 7 6 _ _]
        [6 _ 5 _ 4 1 _ _ _]
        [_ _ _ _ 5 _ _ 3 6]
        [9 _ 7 _ _ _ _ 4 1]
     ]

Sd =[
    [9 _ _ _ 2 _ _ 6 8]
    [8 _ _ 3 _ _ 7 _ _]
    [_ 7 _ _ 8 _ _ _ _]
    [_ _ _ 8 _ 9 _ 3 _]
    [3 _ 1 _ _ _ 8 _ 5]
    [_ 8 _ 7 _ 5 _ _ _]
    [_ _ _ _ 6 _ _ 5 _]
    [_ _ 6 _ _ 2 _ _ 7]
    [5 1 _ _ 7 _ _ _ 2]
   ]


class TextFile
   from Open.file Open.text
end

Pantalla={fun {$}
            A={New TextFile init(name:stdout)}
          in
            proc {$ M}
```

```
            {A write(vs:M)}
          end
        end}

  Argumentos={
          Application.getCmdArgs
          record(
            ene(single  char: &n type:int )
            timeout(single char: &t type:int default:1000*60*10)
            problema(single char:&e type:int default:1)
            backtrack(single type:int default:0)
            )
          }
  ProblemaSeleccionado=case Argumentos.problema of 1 then
                        reinas
                      [] 2 then
                        magicos
                      [] 3 then
                        latin
                      else
                        sudoku
                      end



  fun {Queens N}
   proc {$ Row}
     L1N ={MakeTuple c N}
   in
     {FD.tuple queens N 1#N Row}
     {FD.distinct Row}
     {For 1 N-1 1
      proc {$ I}
        {For I+1 N 1
         proc{$ J}
           Row.I - Row.J \=: I - J
           Row.I - Row.J \=: J - I
         end}
      end}
    end
   end

fun {MagicSquare N}
  NN  = N*N
  ListNum = {List.number 1 N 1}
```

```
   Lista = {List.number 1 NN 1}
   Post = {FD.tuple post NN 1#NN}
   fun{Campo I J}
      {List.nth Lista ((I-1)*N + J)}
   end
in
   {For 1 N 1
    proc{$ I}
       {For 1 N 1
        proc{$ J}
           if I \= J then
              if I < J then
                 Post.(((J*(J-1))div 2)+N-J+I+1) = {Campo I J}
              else
                 Post.(((I*(I-1))div 2)+N+((N*(N-1))div 2)-I+J+1)={Campo I J}
              end
           else
              Post.I = {Campo I I}
           end
        end}
    end}
   Lista2 = {Record.toList Post}
   proc {$ Square}
      fun {Field I J}
         Square.((I-1)*N + J)
      end
      proc {Assert F}
         {FD.sum {Map ListNum F} '=:' Sum}
      end
      Sum = {FD.decl}
   in
      {FD.tuple square NN 1#NN Square}
      {FD.distinct Square}

      {Assert fun {$ I} {Field I I} end}
      {Assert fun {$ I} {Field I N+1-I} end}

      {For 1 N 1
       proc {$ I} {Assert fun {$ J} {Field I J} end} end}

      {For 1 N 1
       proc {$ J} {Assert fun {$ I} {Field I J} end} end}
      {Field 1 1} <: {Field N N}
      {Field N 1} <: {Field 1 N}
      {Field 1 1} <: {Field N 1}
```

```
      {Field 1 N} <: {Field N N}
      NN*(NN+1) div 2 =: N*Sum
   end
  end

  fun {LatinSquare N}
    NN  = N*N
    ListNum = {List.number 1 N 1}
  in
    proc {$ Square}
      fun {Field I J}
         Square.((I-1)*N + J)
      end
      proc {Assert F}
        {FD.sum F '=:' ((N+1)*N) div 2}
      end
    in
      {FD.tuple square NN 1#N Square}
      {ForAll ListNum
       proc{$ I}
          F={FoldL ListNum fun{$ Lista Col} {Field I Col} | Lista end nil}
       in
        {FD.distinct F}
        {Assert F}
       end}

      {ForAll ListNum
       proc{$ J}
          C={FoldL ListNum fun{$ Lista Fil} {Field Fil J} | Lista end nil}
       in
        {FD.distinct C}
        {Assert C}
       end}
     end
    end

    fun {Sudoku Spec}
    proc {$ Root}
      Root={Map {List.make 9}
            fun {$ Row}
               Row={List.make 9}
               Row ::: 1#9
               {FD.distinct Row}
               Row
            end
```

```
         }
      for X in 1;X=<9;X+1 do
         {FD.distinct {FoldR Root fun {$ Row Prev} {Nth Row X}|Prev end nil}}
      end
      for X in 0;X=<2;X+1 do
         for Y in 0;Y=<2;Y+1 do
            {FD.distinct
             {FoldR
              {List.filterInd Root fun {$ I Row} I>=1+Y*3 andthen I=<3+Y*3 end}
              fun {$ Row Prev}
                 {Append
                  {List.filterInd Row fun {$ I X} I>=1+Y*3 andthen I=<3+Y*3 end}
                  Prev
                 }
              end
              nil
             }
            }
         end
      end
      {List.forAllInd Spec
       proc {$ Y R}
          Row={Nth Root Y}
       in
          {List.forAllInd R
           proc {$ X I}
              if {Not {IsFree I}} then
                 {Nth Row X} :: I
              end
           end
          }
       end
      }
   end
end

   class Distribute
      prop locking
      attr
         original
         backtracking
         cantEspacios
         ultimo_fallo
         enumeraciones
         backtrackingAcumulado
```

```
   enumeracionesAcumulado
   cantEspaciosAcumulado

 meth init(EspacioOriginal)
   original<-EspacioOriginal
   backtracking<-0
   cantEspacios<-0
   ultimo_fallo<-0
   enumeraciones<-0
   backtrackingAcumulado<-0
   enumeracionesAcumulado<-0
   cantEspaciosAcumulado<-0
 end

meth busca()
   fun {Backtrack Espacio P CantEsp}

      case {Space.ask Espacio}
      of failed then
         backtracking <- @backtracking +1
         ultimo_fallo<-P
         nil
      [] succeeded then  {Space.merge Espacio}
      [] alternatives(N) then
        Espacio2 = {Space.clone Espacio}
         if {T.getElapsedTime $} > Argumentos.timeout then
          {Pantalla "Termino por Timeout.\n"}
           {Application.exit 0}
         end
      in
         cantEspacios<-CantEsp+1

         {Space.commit Espacio 1}
         enumeraciones<-@enumeraciones+1

         case {Backtrack Espacio P+1 @cantEspacios}
         of nil then
            if (@ultimo_fallo-1)==P then
               skip
            else
               backtracking<-@backtracking + (@ultimo_fallo -1 -P)
            end
            {Space.commit Espacio2 2}
            enumeraciones<-@enumeraciones+1
            {Backtrack Espacio2 P+1 @cantEspacios}
```

```
            elseof Solucion then
               Solucion
            end
          end

        end

        Solucion
        Temp={Space.clone @original}

      in
        thread
           {Space.inject Temp
            proc {$ Sol}
               <ESTRATEGIA>
            end}
           Solucion={Backtrack Temp 0 0}
           {Pantalla "Milisegundos:"#{T.getElapsedTime $}#"\n"}
           {Pantalla "Backtracking:"#@backtrackingAcumulado+@backtracking#"
Enumeraciones:"#@enumeraciones+@enumeracionesAcumulado#"
Espacios:"#@cantEspacios+@cantEspaciosAcumulado#"\n"}
           {Pantalla "\n"}
           {Application.exit 0}
        end
      end
   end


   GeneradorProblema=case ProblemaSeleccionado of reinas then
                        {Queens Argumentos.ene}
                     [] magicos then
                        {MagicSquare Argumentos.ene}
                     [] latin then
                        {LatinSquare Argumentos.ene}
                     else
                        {Sudoku (A.(Argumentos.ene))}
                     end

      Algoritmo ={New Distribute init({Space.new GeneradorProblema})}

      {T.setInitTime _}

      {Algoritmo busca()}
end
```

A.2 Búsqueda de Todas las Soluciones

El código que a continuación se muestra permite resolver cada uno de los problemas planteados en este trabajo en busca de todas las soluciones. La etiqueta <ESTRATEGIA> indica la sección en donde se debe incluir la estrategia de distribución a utilizar, las cuales se muestran en el Apéndice C.

```
functor
import
   T at 'TiempoMilisegundos.so{native}'
   FD
   Application
   Open
   OS
   Space

define
   Lista2
   class TextFile
      from Open.file Open.text
   end

   Pantalla={fun {$}
               A={New TextFile init(name:stdout)}
            in
               proc {$ M}
                  {A write(vs:M)}
               end
            end}

   Argumentos={
              Application.getCmdArgs
              record(
                 ene(single  char: &n type:int)
                 timeout(single char: &t type:int default:1000*60*10)
                 problema(single char:&e type:int default:1)
                 backtrack(single type:int default:0)
                 )
             }
```

```
   ProblemaSeleccionado=case Argumentos.problema of 1 then
                           reinas
                        [] 2 then
                           magicos
                        else
                           latin
                        end
   fun {Queens N}
    proc {$ Row}
       L1N ={MakeTuple c N}
    in
       {FD.tuple queens N 1#N Row}
       {FD.distinct Row}
       {For 1 N-1 1
        proc {$ I}
           {For I+1 N 1
            proc{$ J}
               Row.I - Row.J \=: I - J
               Row.I - Row.J \=: J - I
            end}
        end}
     end
    end
fun {MagicSquare N}
   NN  = N*N
   ListNum = {List.number 1 N 1}
   Lista = {List.number 1 NN 1}
   Post = {FD.tuple post NN 1#NN}
   fun{Campo I J}
     {List.nth Lista ((I-1)*N + J)}
   end
in
   {For 1 N 1
    proc{$ I}
       {For 1 N 1
        proc{$ J}
           if I \= J then
              if I < J then
                 Post.(((J*(J-1))div 2)+N-J+I+1) = {Campo I J}
              else
                 Post.(((I*(I-1))div 2)+N+((N*(N-1))div 2)-I+J+1)={Campo I J}
              end
           else
              Post.I = {Campo I I}
           end
```

```
        end}
     end}
    Lista2 = {Record.toList Post}
    proc {$ Square}
       fun {Field I J}
          Square.((I-1)*N + J)
       end
       proc {Assert F}
          {FD.sum {Map ListNum F} '=:' Sum}
       end
       Sum = {FD.decl}
    in
       {FD.tuple square NN 1#NN Square}
       {FD.distinct Square}

       {Assert fun {$ I} {Field I I} end}
       {Assert fun {$ I} {Field I N+1-I} end}

       {For 1 N 1
        proc {$ I} {Assert fun {$ J} {Field I J} end} end}

       {For 1 N 1
        proc {$ J} {Assert fun {$ I} {Field I J} end} end}
       {Field 1 1} <: {Field N N}
       {Field N 1} <: {Field 1 N}
       {Field 1 1} <: {Field N 1}
       {Field 1 N} <: {Field N N}
       NN*(NN+1) div 2 =: N*Sum
  end
 end

 fun {LatinSquare N}
    NN  = N*N
    ListNum = {List.number 1 N 1}
in
    proc {$ Square}
       fun {Field I J}
          Square.((I-1)*N + J)
       end
       proc {Assert F}
          {FD.sum F '=:' ((N+1)*N) div 2}
       end
    in
       {FD.tuple square NN 1#N Square}
       {ForAll ListNum
```

```
    proc{$ I}
       F={FoldL ListNum fun{$ Lista Col} {Field I Col} | Lista end nil}
    in
     {FD.distinct F}
     {Assert F}
    end}

   {ForAll ListNum
    proc{$ J}
       C={FoldL ListNum fun{$ Lista Fil} {Field Fil J} | Lista end nil}
    in
     {FD.distinct C}
     {Assert C}
    end}
  end
 end

class Distribute
   prop locking
   attr
      original
      backtracking
      ultimo_fallo
      enumeraciones
      backtrackingAcumulado
      enumeracionesAcumulado
      Solucion
   meth init(EspacioOriginal)
      original<-EspacioOriginal
      backtracking<-0
      ultimo_fallo<-0
      enumeraciones<-0
      backtrackingAcumulado<-0
      enumeracionesAcumulado<-0
   end

   meth busca()
      fun {Backtrack Espacio P}
         case {Space.ask Espacio}
         of failed then
            backtracking <- @backtracking +1
            ultimo_fallo<-P
            Solucion
         [] succeeded then  {Space.merge Espacio}|Solucion
         [] alternatives(N) then {SolveLoop Espacio 1 N Solucion P}
```

```
          end
        end

        fun {SolveLoop Esp I N SolTail Pp}
          if {T.getElapsedTime $} > Argumentos.timeout then
              {Pantalla "Termino por Timeout.\n"}
             {Application.exit 0}
         end
          if I > N then
             SolTail
          elseif I == N then
             {Space.commit Esp I}
             enumeraciones<-@enumeraciones+1
             {Backtrack Esp Pp+1}
          else
             Espacio2 = {Space.clone Esp}
             NewTail = {SolveLoop Esp I+1 N SolTail Pp}
          in
             {Space.commit Espacio2 I}
             enumeraciones<-@enumeraciones+1
             {Backtrack Espacio2 Pp+1}
          end
         end

        Solucion
        Temp={Space.clone @original}

      in
        thread
           {Space.inject Temp
            proc {$ Sol}
               <ESTRATEGIA>
            end}
           Solucion = {Backtrack Temp 0}
           {Pantalla "Milisegundos:"#{T.getElapsedTime $}#"\n"}
           {Pantalla "Backtracking:"#@backtrackingAcumulado+@backtracking#"
Enumeraciones:"#@enumeraciones+@enumeracionesAcumulado#"\n"}
           {Pantalla "\n"}
           {Application.exit 0}
        end
      end
    end
```

```
 GeneradorProblema=case ProblemaSeleccionado of reinas then
                      {Queens Argumentos.ene}
                   [] magicos then
                      {MagicSquare Argumentos.ene}
                   else
                      {LatinSquare Argumentos.ene}
                   end

     Algoritmo ={New Distribute init({Space.new GeneradorProblema})}

     {T.setInitTime _}

     {Algoritmo busca()}
end
```

Apéndice B: Estrategias

- DMi + MeVal

```
{FD.distribute generic(order: size value:min) Sol}
```

- DMi + MaVal

```
{FD.distribute generic(order: size value:max) Sol}
```

- DMi + MedVal

```
{FD.distribute generic(order: size value:mid) Sol}
```

- DMi + MMedVal

```
{FD.distribute generic(order: size
                       value: fun{$ S}
                                 {FD.reflect.nextLarger S {FD.reflect.mid S}}
                              end) Sol}
```

- DMa + MeVal

```
{FD.distribute generic(order: fun {$ X Y}
                                 if {FD.reflect.size X} > {FD.reflect.size Y} then
                                    true
                                 else
                                    false
                                 end
                              end
                       value: min) Sol}
```

- DMa + MaVal

```
{FD.distribute generic(order: fun {$ X Y}
                          if {FD.reflect.size X} > {FD.reflect.size Y} then
                           true
                          else
                           false
                          end
                         end
                value: max) Sol}
```

- DMa + MedVal

```
{FD.distribute generic(order: fun {$ X Y}
                          if {FD.reflect.size X} > {FD.reflect.size Y} then
                           true
                          else
                           false
                          end
                         end
                value: mid) Sol}
```

- DMa + MMedVal

```
{FD.distribute generic(order: fun {$ X Y}
                          if {FD.reflect.size X} > {FD.reflect.size Y} then
                           true
                          else
                           false
                          end
                         end
                value: fun{$ S}
                        {FD.reflect.nextLarger S {FD.reflect.mid S}}
                       end) Sol}
```

- BOr + MeVal

```
{FD.distribute generic (order: naive value: min) {Map Lista2 fun {$ I} Sol.I
end}}
```

- BOr + MaVal

{FD.distribute generic (order: naive value: max) {Map Lista2 fun {$ I} Sol.I end}}

- BOr + MedVal

{FD.distribute generic (order: naive value: mid) {Map Lista2 fun {$ I} Sol.I end}}

- BOr + MMedVal

```
{FD.distribute generic(order: naive
                       value: fun{$ Sol}
                                 {FD.reflect.nextLarger Sol {FD.reflect.mid Sol}}
                              end) {Map Lista2 fun {$ I} Sol.I end}}
```