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

LA PROBLEMÁTICA DEL MODELADO DE PROBLEMAS DE SATISFACCIÓN DE RESTRICCIONES

ILSE BERNARDITA ENGELS BALBI

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

Junio, 2010

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

LA PROBLEMÁTICA DEL MODELADO DE PROBLEMAS DE SATISFACCIÓN DE RESTRICCIONES

ILSE BERNARDITA ENGELS BALBI

Profesor Guía: Dr. Broderick Crawford Labrín
Profesor Co-referente: Silvana Roncagliolo De la Horra

Carrera: Ingeniería Civil en Informática

Junio, 2010

” Dedicado a mis padres por su profundo amor, fuerza y apoyo incondicional. Sin ustedes este sueño no hubiese sido posible.”

Agradecimientos

Agradezco a mis padres y hermanos por la paciencia, confianza y apoyo entregados siempre.
A Hernán por su constante apoyo, ánimo y compañía para que me mantuviese firme y saliera adelante.
A mis amigos por sus palabras de ánimo y apoyo.
A mis profesores, que me orientaron y apoyaron durante esta etapa.
A Dios, por entregarme lo necesario para poder desarrollarme como persona y profesionalmente.

Resumen

El mayor esfuerzo en la investigación de la Programación con Restricciones se ha centrado en encontrar formas eficaces para solucionar Problemas de Satisfacción de Restricciones. Sin embargo, el aspecto más fundamental de la resolución de un Problema de Satisfacción de Restricciones, el modelado del problema, ha recibido mucho menos atención. Esto es importante porque la selección de una formulación apropiada puede tener efectos en la eficacia de cualquier algoritmo que resuelve problemas de satisfacción de restricciones.

Considerando esta importancia se identificaron algunas de las características claves que participan en la definición de un modelo y se analizaron los efectos que puede tener la definición de un modelo en la búsqueda de soluciones de un problema, lo que servirá como una base para realizar estudios futuros.

Palabras claves: *Programación con Restricciones, Satisfacción de Restricciones, Modelado de Problemas.*

Abstract

The major research effort in Constraint Programming has focused on finding effective ways to solve Constraint Satisfaction Problems. However, the most fundamental aspect of solving a Constraint Satisfaction Problem, modeling of the problem, has received much less attention. This is important because the selection of an appropriate formulation can have dramatic effects on the effectiveness of any algorithm that solves Constraint Satisfaction Problems. Considering this importance, has identified some key features involved in the definition of a model and analyzed the effects it can have the definition of a model in the search for solutions to a problem, which will serve as a basis for future studies.

Keywords: *Constraint Programming, Constraint Satisfaction, Modeling Problems.*

Índice general

Lista de Figuras

Lista de Tablas

CAPÍTULO 1

Introducción

Muchos problemas en inteligencia artificial e informática, tales como: la asignación de recursos, planificación, establecimiento de horarios, configuración, y problemas de satisfacibilidad, pueden ser modelados como Problemas de Satisfacción de Restricciones, en inglés, Constraint Satisfaction Problem (CSP). Las restricciones se muestran en la mayoría de las áreas del ser humano. Se usan restricciones para dirigir el razonamiento como una clave del sentido común diario. "Puedo estar ahí a partir de las 5 ó 6", es una típica restricción que se utiliza para planificar el tiempo personal. En general no se soluciona una sola restricción sino un conjunto de restricciones. En los últimos años la Programación con Restricciones ha atraído la atención de gran cantidad de expertos de diversas áreas, debido a su potencial para resolver los problemas de la vida real.

La programación con restricciones, en inglés, Constraint Programming (CP), se basa en el modelado y resolución con restricciones [11]. Dado un problema descrito a través de variables, donde cada variable posee un dominio asociado, compuesto por un conjunto de potenciales valores y un conjunto de restricciones que representan las diferentes relaciones entre las variables que deben ser satisfechas para resolver el problema, la idea principal de la CP es usar el conocimiento de las restricciones para eliminar del dominio de las variables todos aquellos valores que no pueden formar parte de una solución al problema. De esta forma, es posible resolver un problema reduciendo los dominios de sus variables hasta conseguir aproximaciones muy cercanas al valor solución, o reducir el dominio de una o más variables hasta eliminar todos sus posibles valores.

La mayor parte de la investigación ha centrado su esfuerzo en diseñar algoritmos de búsqueda eficientes para resolver CSPs, y explotando información específica del dominio para resolver eficientemente aplicaciones individuales. Una línea de investigación reciente e importante en la comunidad científica es determinar cómo la formulación y reformulación de un problema afecta la eficiencia en la ejecución de algoritmos de resolución de restricciones, pasando a importar en este aspecto el tema del modelado de un CSP.

Seleccionar el modelo más apropiado es en general difícil. De hecho, no existen hasta el momento nociones objetivas y generales para la mejor formulación. En la primera parte del desarrollo del proyecto se obtuvo algunas características que forman parte del modelado. Estas características de modelado serán utilizadas para realizar la formulación de algunos problemas comunes existentes en la literatura, estableciendo de qué forma estos modelos influyen en la resolución de los problemas. Se utilizará el solver Eclipse para realizar las prácticas de los problemas y para implementar y evaluar la solución propuesta.

1.1 Objetivos Generales y Específicos

1.1.1 Objetivos Generales.

- Estudiar la problemática del modelado de CSP, establecer sus características principales y formular diferentes modelos de CSP a resolver con el solver Eclipse.

1.1.2 Objetivos Específicos.

- Estudiar y comprender los fundamentos de Programación con Restricciones.
- Comprender la teoría para modelar un problema combinatorial como un CSP.
- Estudiar y comprender el modelado de problemas clásicos existentes en la literatura.
- Estudiar y comprender las decisiones de modelado, definiendo las características que forman parte de un modelo.
- Estudiar y aplicar las características de modelado definidas.
- Comprender la estructura y notación utilizadas por Eclipse.
- Implementar el modelado de CSP utilizando Eclipse.
- Evaluar las características de modelado obtenidas, mediante la utilización de Benchmarks (de problemas clásicos) disponible en la literatura, entregando el análisis de los resultados obtenidos.

CAPÍTULO 2

Problemas de Satisfacción de Restricciones (CSP)

2.1 Definición y Conceptos Básicos

Dada la gran cantidad de terminologías existentes para denotar un CSP, es necesario formalizar su definición y la de sus componentes relacionados.

Un Problema de Satisfacción de Restricciones (CSP) de dominio finito, consiste en un conjunto finito de variables de decisión, cada una con un dominio finito asociado de posibles valores, y un conjunto finito de restricciones. Así, la definición formal sería:

- **CSP:** Un Problema de Satisfacción de Restricciones es una tupla $P = (X; D; C)$ donde:

  $i)$ $X = \{x_1, ..., x_n\}$ es un conjunto de n variables;
  $ii)$ $D = \{D_1, ..., D_n\}$ es el conjunto de dominios. Para cada variable x_i en X, D_{x_i} es el dominio de x_i. Se define el dominio de una variable como el conjunto finito de todos los valores posibles que pueden ser asignados a ella;
  $iii)$ $C = \{c_1, ..., c_m\}$ es un conjunto finito de restricciones. Una restricción c_i es una relación definida sobre un subconjunto de las variables.

  El conjunto de variables en la restricción c se denotará por V_c. Si el conjunto V_c tiene sólo uno o dos elementos, se hablará de restricciones unarias o binarias respectivamente. Las restantes restricciones son llamadas restricciones no binarias. así, se dice que un CSP es un ***CSP binario***, si todas sus restricciones son unarias o binarias.

  La estructura de un CSP puede ser representada por un ***grafo de restricciones*** (figura 2.1). El grafo de restricciones más simple utiliza CSP binarios, donde los vértices corresponden a las variables que están conectadas por un arista, sí y solo sí existe una restricción que incluya ambas variables.

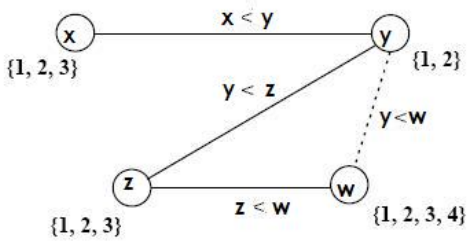

Figura 2.1: Representación de un CSP como grafo de restricciones

- **Notación de CSP:** A continuación se define una notación asociada a los CSP y que será utilizada en este proyecto.

  - El número de variables de un CSP se denota por n. La longitud del dominio de una variable x_i se denota por $d_i = |D_i|$. El número total de restricciones se denota por c y la *aridez* máxima por k.
  - Variables. Para representar las variables se utilizan las últimas letras del alfabeto en cursiva, por ejemplo, x, y, z. El conjunto de variables $x_i, ..., x_j$ se denota por $X_{i,...,j}$.
  - Dominio. El dominio de una variable x_i se denota por D_i. La asignación de un valor a a una variable x se denota mediante el par (x, a).
  - Restricción. Una restricción k-aria entre las variables $x_1, ...x_k$ se denota por $C_{(1,...,k)}$. Así, la restricción binaria entre x_i y x_j se representa por $C_{(i,j)}$.

  En el ejemplo de la figura 2.1 se observa que el número total de variables, n, es igual a 4. El dominio máximo es 4 y el dominio mínimo es 2,

- **Asignación de Variable:** Una asignación de variable (x, a) (también conocida como instanciación), es un par variable-valor que representa la asignación del valor a a la variable x. Una instanciación de un conjunto de variables es una tupla de pares ordenados, donde cada par ordenado (x, a) asigna el valor a a la variable x.

- **Solución:** Una solución a un CSP es una asignación de valores a todas las variables de forma que se satisfagan todas las restricciones.
  Una tupla $((x_1; a_1), ..., (x_n, a_n))$ es localmente consistente si satisface todas las restricciones formadas por las variables de la tupla.

  Es decir, una solución es una tupla consistente que contiene todas las variables del problema. Así, se dice que un problema es consistente, si existe al menos una solución, es decir una tupla consistente $(a_1, a_2, ..., a_n)$.

La tarea en un CSP es asignar un valor a cada variable tal que todas las restricciones sean satisfechas simultáneamente.

Si un CSP no tiene solución, el problema es llamado *inconsistente*.

2.2 Resolución de un CSP.

La resolución de un problema de satisfacción de restricciones requiere de tres etapas:

- **Modelar el Problema como un CSP**. El modelado expresa el problema mediante una sintaxis de CSP, es decir, mediante un conjunto de **variables**, **dominios** y **restricciones** del CSP.
- **Resolución del Modelo**. Esta etapa se entiende como la tarea de buscar una solución simple para el problema, aunque a veces se requiere encontrar todo el conjunto de soluciones, y en ciertos casos a causa del costo para encontrar una solución, el objetivo consiste en encontrar la mejor solución o una aproximación a ésta con respecto a unos recursos limitados (como un tiempo razonable).

  Actualmente la resolución de CSP suele ser realizada mediante el uso de diferentes técnicas, desde técnicas tradicionales a otras mucho más modernas. Existen diferentes métodos de resolución de CSPs y en general estos métodos para generar una solución se clasifican en cuatro categorías:

  (i) Las variantes de la búsqueda por backtracking,
  (ii) Los métodos que minimizan la redundancia en el proceso de búsqueda mediante la eliminación de aquellos valores que nunca pueden ser parte de una solución (estos métodos son denominados como algoritmos de filtrado, algoritmos de arco consistencia, algoritmos de propagación).
  (iii) Las técnicas de forward checking y las de look ahead,los cuales son algoritmos que combinan un método de propagación dentro de un algoritmo de backtracking, y,
  (iv) Los llamados algoritmos *guiados por la estructura* los cuales explotan la estructura del grafo de restricción del problema.
- **La Comprensión de la Solución**. La que corresponde fundamentalmente al área de análisis de programas.

2.3 Procesamiento de un CSP (Métodos de Búsqueda de Soluciones).

Las soluciones de un CSP pueden ser encontradas buscando (sistemáticamente) las posibles asignaciones de valores a las variables.

2.3.1 Búsqueda Sistemática

Desde el punto de vista teórico, la solución de un CSP es trivial a través del uso de exploración sistemática del espacio solución. No desde el punto de vista práctico, donde la eficiencia toma lugar. Aunque los métodos de búsqueda sistemática se ven muy simples y no eficientes son muy importantes porque ellos son la base de algoritmos más avanzados y eficientes.

El algoritmo básico de satisfacción de restricciones, que busca el espacio de asignación completo, es llamado *algoritmo de generación y prueba*, generate and test (GT). La idea de GT es simple: primero, se genera una asignación completa de variables en forma aleatoria, si esta asignación satisface todas las restricciones entonces se ha encontrado solución, sino, otra asignación es generada. GT es un algoritmo débil, que es usado si todo ha fallado. Su eficiencia es pobre debido a que descubre tardíamente las inconsistencias. Debido a lo anterior, hay dos formas para mejorar la eficiencia de GT:

- El generador de valoración es listo (informado). O sea, se genera la valoración de tal forma que el conflicto encontrado en la etapa de prueba se minimiza. Esta es la idea básica de los algoritmos estocásticos basados en búsqueda local.
- El generador es mezclado con el "*probador* ". Es decir, la validez de la restricción es probada tan pronto como sus respectivas variables son instanciadas. Este método es utilizado por el enfoque de *Backtracking*.

2.3.2 Backtracking.

Este algoritmo realiza una búsqueda en profundidad sobre el árbol del problema [7]. En cada nodo el algoritmo comprueba si las restricciones totalmente asignadas están satisfechas. Si es así, el algoritmo prosigue la búsqueda en profundidad. Sino, la rama actual no contiene soluciones y se realiza una vuelta atrás sobre la última variable asignada. Este algoritmo no especifica: $i)$ *el orden de las variables en una rama* y $ii)$ *el orden de los valores para una variable*. Estos aspectos se resuelven mediante heurísticas se selección de variable y selección de valor [1].

2.3.3 Técnicas de Consistencia.

Otro acercamiento a la solución de un CSP consiste en remover valores inconsistentes del dominio de la variables antes de obtener la solución. Estos métodos son llamados técnicas de

consistencia.

Existen varias técnicas de consistencia [9, 10] pero la mayoría de ellas no son completas. Es por esto que rara vez son utilizadas solas para resolver un CSP en forma completa. Los nombres de las técnicas de consistencia básicas derivan de la noción de Grafos. El CSP es usualmente representado como un *Grafo de Restricciones*, donde los nodos corresponden a las variables y los arcos son etiquetados por las restricciones. Esto requiere que el CSP esté en una forma especial, generalmente referida como *CSP Binario.*[1]

- Consistencia de Nodo (NC). Es la técnica de inconsistencia más simple. Remueve los valores desde el dominio de las variables que son inconsistentes con restricciones unitarias sobre las respectivas variables.
- Consistencia de Arco (AC). Esta es la técnica más utilizada. Remueve los valores desde el dominio de las variables que son inconsistentes con restricciones binarias.

2.3.4 Propagación de Restricciones.

La búsqueda sistemática y algunas técnicas de consistencia pueden ser utilizados en forma independiente para solucionar un CSP en forma completa. Una combinación de ambos enfoques es una forma más común para resolver CSP.

[1] Contiene restricciones binarias y unitarias solamente

CAPÍTULO 3

Modelado de un CSP

La importancia del modelado en la programación con restricciones ha sido considerada ampliamente, como lo da a conocer Puget en [15].

3.1 Representando un Problema.

Es difícil definir precisamente qué significa que un CSP represente un problema P. Una posible definición es que: un CSP $M = (X, D, C)$ representa un problema P, o M es un modelo de P, si cada solución de C corresponde a una solución de P y cada solución de P puede ser derivada de al menos una solución de C.

Esta definición no requiere que exista una relación uno a uno entre las soluciones de P y las de M. Esto se debe a que modelar un problema como un CSP a menudo introduce simetría, representando entidades que son indistinguibles en P a través de distintas variables o valores en M. Es así como múltiples soluciones de M pueden corresponder a la misma solución de P.

Hay muchas formas en las cuales un problema puede ser modelado como un CSP. Las decisiones tomadas en esta etapa son muy importantes en el proceso de solución del problema, y pueden llegar a tener un efecto importante en el costo de solución del problema. Hay muchos aspectos diferentes en la investigación de satisfacción de restricciones que afectan el proceso de selección del modelo de un CSP.

A continuación se presenta un ejemplo de un problema de satisfacción de restricciones, el problema criptográfico "send + more= money" y algunas de las formas de modelarlo, figura 3.1. La declaración del problema es: "Asignar a cada letra s, e, n, d, m, o, r, y un dígito diferente del conjunto $0, ..., 9$ de modo que se satisfaga la expresión $send + more = money$".

1. Una primera forma de realizar la formulación del problema es:

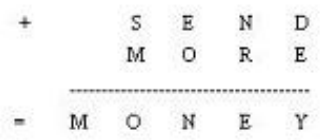

Figura 3.1: Problema SEND+MORE=MONEY

- Variables: Cada una de las letras,s, e, n, d, m, o, r, y.
- Dominio: El conjunto de enteros $0, ..., 9$.
- Restricciones: Todas las variables deben tomar distintos valores y se debe cumplir que $send + more = money$. Formalmente sería:
  - $10^3(s+m) + 10^2(e+o) + 10(n+r) + d + e = 10^4m + 10^3o + 10^2n + 10e + y$;
  - Restricción de todas diferentes s, e, n, d, m, o, r, y (*alldiferent* constraint)[18].

2. Otra forma de modelar el problema es utilizar bits de acarreo para descomponer la ecuación en una colección de pequeñas restricciones. Según el planteamiento del problema, M debería tomar el valor 1 y S solamente valores entre $1, ..., 9$. así, la definición formal sería:

   - Variables: Cada una de las letras, s, e, n, d, m, o, r, y. Además se incluyen tres variables adicionales c_1, c_2, c_3, que serán denominadas portadoras.
   - Dominio: Para las variables e, n, d, o, r, y es $0, ..,9$. Para la variables s es $1, ..., 9$, para m es 1, y el dominio de las variables portadoras es: $0, 1$.
   - Restricciones:
     - $e + d = y + 10c_1$
     - $c_1 + n + r = e + 10c_2$
     - $c_2 + e + o = n + 10c_3$
     - $c_3 + s + m = 10m + o$
     - Restricción de todas diferentes (*alldifferent*) s, e, n, d, m, o, r, y

Haciendo un breve análisis de estas formulaciones se puede concluir: De acuerdo a pruebas realizadas, en la primera alternativa al utilizar el algoritmo de backtracking (BT), este modelo no es muy eficiente porque con BT, todas las variables necesitan ser instanciadas antes de comprobar estas dos restricciones. De esta manera no se puede podar el espacio de búsqueda para agilizar la búsqueda de soluciones. Además, la primera restricción es una igualdad en la que forman parte todas las variables del problema[1], por lo que dificulta el proceso de consistencia.

[1]Restricción Global

En la segunda formulación, la ventaja del modelo es que las restricciones más pequeñas pueden comprobarse antes en la búsqueda de backtracking, y así podarse muchas inconsistencias.

En ambas formulaciones, la restricción *alldifferent*, se puede sustituir por un conjunto de "pequeñas" restricciones, por ejemplo, $s \neq e, ..., r \neq y$, y se obtiene un tercer y un cuarto modelo para el problema.

Otro ejemplo de Problema de Satisfacción de Restricciones es el caso del Problema de las "Series Mágicas". Este problema es un puzzle que consiste en la asignación de valores a una serie de números. O sea, para una serie de números de largo $n+1$, la tarea es asignar un número desde 0 a n, a cada posición de la serie, tal que el número de cada posición es igual al número de veces que aparece en la serie.

Así, dado un entero n, el problema consiste en encontrar una secuencia $S = (s_0, s_1, \ldots, s_n)$, tal que s_i representa el número de ocurrencias de i en S. Por ejemplo, si n=3, una solución al problema es $(2, 0, 2, 0)$. La anterior es una solución porque el número 0 aparece dos veces, en la posición 1 y 3, y el número 2 aparece también dos veces, en la posición 0 y 2. Otra solución es $(1, 2, 1, 0)$.

Algunas de las maneras de formular este problema son:
1. Primera formulación:

- Variables: s_0, s_1, ..., s_{n-1} para cada una de las n posiciones en la serie.
- Dominio: $\{0, ...,$n$\}$ para cada variable. Representa el valor en la posición de la serie.
- Restricciones:
  C_1 : Número de ocurrencias de i en $(s_0, s_1, \ldots, s_{n-1})$ es s_i.

2. Una segunda formulación es posible, si se agregan otras dos restricciones a la formulación anterior, conservando las definición de dominio y variables. Así, la nueva formulación sería:

- Variables: s_0, s_1, ..., s_{n-1} para cada una de las n posiciones en la serie.
- Dominio: $\{0, ...,$n$\}$ para cada variable. Representa el valor en la posición de la serie.
- Restricciones:
  C_1 : Número de ocurrencias de i en $(s_0, s_1, ..., s_{n-1})$ es s_i
  C_2 : Sumatoria de i = 0 hasta n de i*x_i es igual a n+1
  C_3 : Sumatoria de i = 0 hasta n de x_i es igual a n+1

Estas restricciones, C_2 y C_3, son conocidas por ser una característica del conjunto solución. Como resultado, ellas proporcionan restricciones adicionales, redundantes, basadas en la información del problema.

Como se puede observar, un problema se puede formular desde distintos enfoques y estas formulaciones dan como resultado diferentes ejecuciones de resolución de problemas.

3.2 Distintas Formas de Formular un Problema (*Viewpoints*).

Según lo estudiado en 3.1, diversos modelos de un problema P pueden ser obtenidos al mirar el problema desde diferentes puntos de vista, conocidos como ***Viewpoints***. Un Viewpoint es un par (X, D), donde X = X_1, ...X_n es un conjunto de variables y D es un conjunto de dominios. Para cada x_i que pertenece a X, el dominio asociado D_i es el conjunto de posibles valores para x. Debe ser posible asociar un significado a las variables y valores del CSP en términos del problema P. Las restricciones deben asegurar que cada solución del CSP es una solución válida para el problema P. Hay diversos puntos de vista que dan lugar a diferentes modelos de un problema.

En principio los valores del dominio pueden ser de cualquier tipo. En la práctica los tipos de datos comúnmente aceptados por los solvers de restricciones son enteros, booleanos y conjuntos de enteros. Otros tipos han sido propuestos, por ejemplo, multiconjuntos y tuplas. Los solvers pueden soportar estos tipos o proveer facilidades para que nuevos tipos de datos sean definidos.

Con excepción de muy pocos problemas, las variables de un CSP son usualmente implementadas usando estructuras de datos como listas o arreglos. En [1] se ha sugerido que los modelos matriciales, basados en matrices de variables, son un medio natural para modelar muchos problemas. Para algunas aplicaciones, otras estructuras son importantes, por ejemplo: para aplicaciones de redes se utilizan modelos basados en grafos.

Usualmente existen diferentes Viewpoints que pueden ser elegidos en el modelado de un problema. Así también pueden combinarse. Para explicar su definición e implicancia se tomará en cuenta la utilización de un Viewpoint individual. Así, al elegir un Viewpoint, el próximo paso es expresar las restricciones asegurando que las soluciones del CSP sean las correctas, es decir, sean las soluciones del problema P. Sin embargo, aunque esta correctitud es importante, no es suficiente si lo que interesa es cuán eficientemente puede ser resuelto un CSP.

Una buena regla para la elección de un Viewpoint es que las restricciones sean expresadas fácil y brevemente. Se deben preferir Viewpoint que permitan que el problema sea descrito con menor cantidad de restricciones como sea posible, las cuales otorguen eficiencia y baja comple-

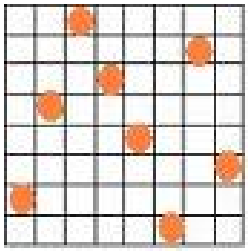

Figura 3.2: Puzzle N-reinas.

jidad en el algoritmo de propagación.

Un ejemplo de representación de un problema desde distintos puntos de vista lo da a conocer Nadel en [17], donde muestra nueve formas de formular el problema de las n-reinas (figura 3.2) como un CSP. El problema consiste en encontrar la forma de ubicar n reinas en un tablero de ajedrez de dimensiones nxn, de tal forma que ninguna reina pueda atacar a las otras. Como ejemplo se muestran dos viewpoints utilizados para la formulación del problema:

1. Sean r_1, ...r_n, las variables que representan las filas del tablero. El dominio de cada variable, que representa las columnas del tablero, es el conjunto de enteros 1, ..., n. La asignación (r_i, c) corresponde a la reina que está en la fila i y en la columna c.

   En este punto de vista, la regla de que exista sólo una reina en cada fila es satisfecha por el hecho de que a cada variable se le puede asignar un solo valor. Además, la condición que implica que exista una reina en cada columna puede ser expresada por la restricción $r_i \neq r_j$ para $1 \leq i < j \leq n$, o por una restricción *alldiferent* sobre r_1,. . . ,r_n.

2. Sean q_1, ...q_n, las variables que representan a las n reinas. El dominio de cada variable, que representan los cuadrados, es el conjunto de enteros $1, 2, ..., n^2$. La asignación (q_i, a) corresponde a la *iésima* reina en un cuadrado a.

En este punto de vista, las reglas son más difíciles de expresar. Se necesitan restricciones para asegurar que dos reinas no se encuentren en la misma fila. Si el elemento *fila* de un valor puede ser extraído, debe haber una restricción entre cada par de variables cuyo elemento fila no sean iguales. Las restricciones de columna podría ser tratadas de igual forma. Las restricciones de diagonal son más difíciles de escribir. Una posibilidad es declarar una restricción ampliada entre cada par de variables, registrando para cada uno de los n^2 valores, los valores que representan los cuadrados que no están en la misma fila, columna o diagonal, aunque la propagación del dominio sería costosa.

Por otra parte, las restricciones pueden expresar el hecho de que existe a lo más una reina en cada fila o columna, no que debe haber exactamente una. Aunque sólo soluciones correctas se encontrarán usando estas restricciones, el modelo permite soluciones parciales en que las reinas ya ubicadas atacan todos los cuadrados en una fila o columna, puesto que no hay nada explícito en las restricciones que prohíbe esto.

Por lo tanto, un modelo basado en el segundo viewpoint será menos eficiente de resolver que el modelo basado en el primer viewpoint. Se han sugerido directrices para ayudar a la resolución de problemas a encontrar una formulación de alta calidad. Existiendo algunas metodologías para mejorar la formulación de un CSP con respecto a una cierta clase de técnicas de resolución de problemas. Estas metodologías incluyen la adición o eliminación de restricciones redundantes, añadir o eliminar las variables redundantes, la explotación de simetrías, y la transformación de la formulación de un CSP en diferentes representaciones equivalentes.

Una apropiada formalización de modelado debe considerar:

- **Representar variables de decisión complejas**: Actualmente, la programación con restricciones proporciona variables de decisión cuyos dominios o valores de dominio son conjuntos finitos de elementos atómicos. Sin embargo, los problemas combinatorios a menudo requieren encontrar una estructura combinatorial no atómica. Por ejemplo, el *problema de los Golfistas* [2] que requiere asignar un conjunto G de golfistas en cada semana de juego. Por lo tanto, el objetivo es encontrar un conjunto de particiones del G, es decir, un conjunto de series de conjuntos de G. Modelar el problema requiere decidir cómo representar las variables de decisión, cuyo tipo es un conjunto de series de conjuntos de G, estructurado como una colección de variables de decisión atómicas. Representar las variables de decisión complejas es la tarea en el modelado de muchos problemas.

- **Identificar y romper simetrías**: Muchos modelos contienen simetrías, a menudo gran cantidad de ellas. Las simetrías en el modelo dan como resultado redundancias en el espacio de búsqueda. Modeladores expertos son capaces de identificar simetrías en un modelo y romperlas, ya sea introduciendo restricciones para "romper-simetrías" o usando un método de búsqueda de simetría. La simetría puede entrar en un modelo a través de dos fuentes: Una simetría es inherente al problema combinatorial o bien se ha introducido mediante el proceso de modelado.

- **Ejecutar transformaciones**: Un modelador humano puede mejorar un modelo a través de la realización de transformaciones tales como la introducción de restricciones implícitas o de variables auxiliares. Las transformaciones no cambian el nivel de abstracción en una especificación o modelo.

[2] Problema 10 en www.csplib.org

3.3 Identificación de Simetrías.

Un aspecto importante del modelado es tratar con la simetría. Ésta ocurre naturalmente en muchos problemas (por ejemplo, si se tienen dos máquinas idénticas para realizar una programación, o si se tienen dos tareas idénticas para procesar). La simetría también puede ser introducida al modelar un problema. Se debe tratar la simetría o sino se perdería mucho tiempo visitando soluciones simétricas.

Al aplicar una simetría sobre un estado s, se obtiene un nuevo estado s' que es equivalente a s. Una simetría induce una relación de equivalencia en el espacio de estados, agrupando en clases a todos los estados equivalentes por la simetría. Cada clase contiene soluciones (todos los estados de la clase son solución) o no soluciones (ningún estado de la clase es solución). Para resolver un CSP con simetrías, no es necesario visitar todos los estados, sino solamente un estado por clase de equivalencia. De esta forma disminuye drásticamente el tamaño del espacio y aumenta la eficiencia de la búsqueda.

Como se mencionó, el proceso de modelado puede introducir simetrías en el modelo. Para explicar lo anterior se pueden ver los siguientes ejemplos.

Ejemplo 1: Encontrar un conjunto de tres dígitos que sume 16.
Una forma de formular este problema es utilizando tres variables de decisión, x, y, z. Cuyo dominio se encuentra es el conjunto $\{0, \ldots, 9\}$. Además, se utiliza la restricción *alldifferent* que asegura que x, y y z toman diferentes valores, utilizando también la restricción que controla que la suma sea 16.

Como solución se puede obtener: $x = 1$, $y = 7$, $z = 8$. La solución anterior puede ser transformada a otra solución intercambiando los valores, es decir, $z = 1, x = 7, y = 8$. Esta simetría se introduce independientemente del conjunto de restricciones original.

Ejemplo 2: Otro ejemplo es el del tablero de ajedrez que se muestra en la figura 3.3. La solución al puzzle es única.

La pregunta que interesa responder es qué significa simetría en este ejemplo. Simetría se entiende como una operación que cambia la posición de las piezas, pero cuyo estado final cumple todas las restricciones si y solo si el estado inicial ocurrió.

Dada una solución cuya definición satisface todas las restricciones, se puede encontrar una nueva solución aplicando alguna simetría a la primera solución encontrada. Por ejemplo, según la figura 3.3, se pueden intercambiar los colores de cada pieza, esto es, donde aparecen piezas blancas que estén las negras y viceversa. Asimismo se puede rotar el tablero en un múltiplo de

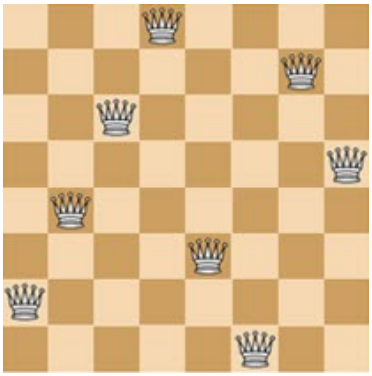

Figura 3.3: Solución al puzzle que implica ubicar nueve reinas y un rey en un tablero de ajedrez, con la condición que las piezas no estén en la misma línea (fila, columna o diagonal) que alguna reina del otro color

90 para producir nuevas soluciones. Finalmente, también se puede rotar el tablero sobre el eje horizontal, el eje vertical y ambos ejes diagonales. Puesto que estas simetrías pueden ser combinadas, hay 8 simetrías disponibles, incluyendo la operación identidad que deja cada pieza en el mismo lugar.

Introducir simetría de esta forma puede tener un importante efecto adverso en el esfuerzo de encontrar una solución a través de un árbol de búsqueda. Existe una variedad de formas para eliminar esta redundancia en el espacio de búsqueda, pero para ello es necesario tener conocimiento de la simetría en el modelo.

Surge una interrogante a responder. Por qué es importante la simetría. El principal motivo de la importancia es que explotando la simetría se puede reducir la cantidad de búsqueda necesaria para resolver el problema. Lo anterior posee una ventaja potencial enorme. Por ejemplo, si se supone que se busca una solución al tablero de la figura 3.3, y la primera asignación que se realiza es ubicar una reina blanca en la esquina superior izquierda del tablero. De hecho, la decisión de la búsqueda no era intentar con una reina blanca en la esquina superior izquierda del tablero, sino que por el contrario, la decisión estaba en intentar con todas las potenciales soluciones con una reina de cualquier color en cualquier parte del tablero. Como existen 8 simetrías, se tiene el potencial de reducir la búsqueda en un factor de 8.

Una segunda razón para la importancia de la simetría es que muchos problemas de restricciones poseen simetría. Por otra parte, el hecho de modelar puede incluir simetrías.

La más importante aplicación de la simetría en programación con restricciones es el *rompimiento de la simetría* como una forma de reducir la búsqueda. La meta del rompimiento de

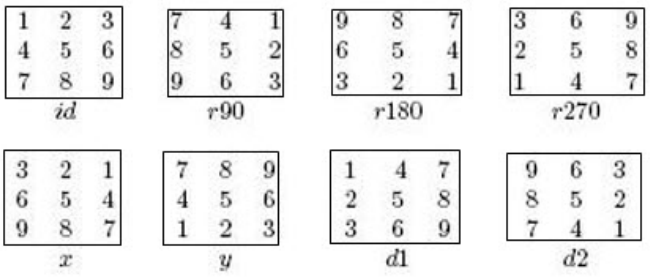

Figura 3.4: Simetrías en un tablero de ajedrez

simetría es nunca explorar dos estados que sean simétricos el uno con el otro, puesto que se sabe que el resultado de ambos debe ser el mismo[3].

Se identifican, comúnmente tres aproximaciones al rompimiento de simetría en la programación con restricciones. El primer acercamiento es **reformular el problema** de modo que tenga una cantidad reducida de simetría, o incluso no tenga. El segundo es **agregar rompimiento de simetrías a las restricciones antes de comenzar la búsqueda**. Y la tercera aproximación es **romper la simetría dinámicamente durante la búsqueda**, adaptando el procedimiento de búsqueda apropiadamente.

La característica principal que permite que una asociación biyectiva sobre un conjunto de objetos sea llamada una simetría es que deja algunas propiedades de estos objetos sin cambiar. De lo anterior se desprende que siempre la asociación de la identidad es una simetría.

- **Simetría en tablero de ajedrez**: Dado un tablero de ajedrez de 3×3, en que cada cuadro ha sido etiquetado con un número del 1 al 9, esos serán los números que serán movidos por las simetrías. Existen 8 simetrías naturales en un tablero de ajedrez. Se incluye también la simetría de la identidad, la que deja cada punto donde está. En la figura 3.4 se muestra la identidad en la esquina superior izquierda. El tablero puede ser rotado en $90, 180$ y 270 grados en el sentido de las manecillas del reloj. Las ubicaciones resultantes de los puntos del tablero después de estas rotaciones se muestran en la parte superior de la figura 3.4. Además, hay reflexión en el eje horizontal y en el eje vertical y en las dos diagonales principales, lo que se muestra en la parte inferior de la figura 3.4.

[3]La frase *rompimiento de simetría* puede llevar error a un error, debido a que actualmente no todos los métodos rompen simetrías en el sentido de crear un problema sin simetrías. Sin embargo, su uso se ha instaurado y sería aún más confuso intentar cambiarlo

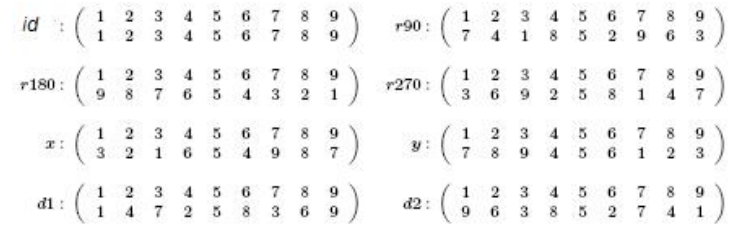

Figura 3.5: Permutaciones que representan las simetrías del tablero de ajedrez

Se puede ver la relación entre simetría y permutación de una forma fácil. Una permutación es una correspondencia uno a uno entre un conjunto y el mismo. Cada simetría define una permutación de un conjunto de puntos. En la figura 3.5 se muestra en la fila superior de cada simetría los números del tablero en forma ascendente sobre los cuales se realizan las permutaciones. La segunda fila muestra el número que ocupa la posición relacionada con el número de la fila superior. Por ejemplo, después de la rotación de $90°$ se ve que el punto 1 es reemplazado por el 7, el 7 es reemplazado por el 9, el 9 por el 3, y el 3 por el 1. Lo anterior entrega un *ciclo* $(1, 7, 9, 3)$.

En la programación con restricciones, siempre que un problema presenta simetría se puede construir un grupo para representar la simetría en el problema. Los elementos del grupo permutan puntos dependiendo de las simetrías en el problema particular.Los puntos sobre los que los elementos del grupo actúan suelen ser pares variable-valor.

Tomando como ejemplo el tablero de ajedrez de la figura 3.3 se tiene:

- Variables: El CSP tiene n^2 variables que corresponden a los cuadros del tablero de ajedrez. Para representar las variables se requiere de n^2 asignaciones.
- Valores: Para cada cuadro del tablero de ajedrez existen 5 posibles valores, una reina blanca, una reina negra, un rey blanco, un rey negro o vacío.
- Variable-Valor: Hay n^2 posibles asignaciones para las variables, y 5 posibles asignaciones para los valores. Para representar el par variable-valor se requiere de $5 \times n^2$ puntos.

3.3.1 Definiciones de Simetría.

Se ha presentado en la literatura variados trabajos sobre simetría en la satisfacción de restricciones y problemas relacionados, los cuales en general no entregan una definición clara de simetría. A continuación se entregan algunas definiciones de simetría que son utilizadas en la

programación con restricciones.

Existen dos tipos básicos de definición para la simetría en un CSP: Una que define simetría cono una propiedad de un conjunto de soluciones, y la que define simetría como una propiedad que puede ser identificada en la declaración del problema, sin resolverlo. Estas simetrías son conocidas como ***simetría de solución*** y ***simetría de problema o simetría de restricción***.

Una temprana definición de simetría de solución es aquella que la define como una permutación de las variables del problema que dejan el conjunto solución invariable. También se define simetría como una función biyectiva sobre el conjunto de soluciones de un CSP, esto permite que la simetría sea especificada por su efecto en la asignación de los valores a las variables.

Algunos autores han definido una simetría como un mapeo que deja las restricciones sin cambios, pero que a menudo restringen las asignaciones permitidas a esas que sólo afectan las variables o sólo los valores. Se debe apreciar que una restricción puede ser especificada extensionalmente por medio del listado de las tuplas permitidas, o intencionalmente por medio de una expresión tal como x<y desde donde las tuplas permitidas pueden ser determinadas. Intercambiar las variables en una restricción, en general la cambia. Por ejemplo, la restricción $x + y = z$, no es lo mismo que la restricción $x + z = y$.

En [14] se define el concepto de una restricción simétrica. Esto es, una restricción que no se ve afectada por el orden de las variables. Por ejemplo, la restricción de desigualdad binaria $\neq$, es simétrica. Define una simetría de un CSP como una permutación de las variables que mapean el conjunto de restricciones hacia un conjunto equivalente: cualquier restricción restricción simétrica y su mapeo hacia una restricción sobre el mismo conjunto de variables.

Puget en [14] restringe su definición de simetría a asignaciones que permutan las variables del problema solamente. En [12] se define la simetría que actúa tanto en las *variables* como en los *valores* de un CSP. Se define una simetría sobre un CSP con n variables como una colección de $n + 1$ asignaciones biyectivas $\Theta = \theta, \theta_1, ..., \theta_n$. La asignación Θ es una biyección sobre el conjunto de variables *{x*1, x2, . . . , *xn}*; cada θi es una biyección desde $D(xi)$ to $D(\theta(xi))$ (donde $D(xi)$ es el dominio D limitado a valores aceptables para xi a través de restricciones unarias). Estos mapeamientos también transforman cada restricción. El conjunto θ es llamado una simetría si no cambia el conjunto de restricciones C, como un todo.

En [12] la definición permite simetrías de variable (que permute sólo las variables) y de valor (permuta sólo los valores) como casos especiales, siendo por tanto, más general que muchas definiciones anteriores.

En [2]se extiende la idea de intercambiabilidad (permutación) ligeramente y distingue entre

simetría semántica y sintáctica en CSPs, que corresponde a los conceptos de ***simetría de solución*** *y* ***simetría de restricción***, respectivamente.

- **Simetría semántica**. dos valores ai y bi para una variable *vi* en un CSP son simétricas si se tiene la siguiente propiedad: existe una solución que asigna el valor *ai* a *vi* si y solo si hay una solución que asigna el valor *bi* a *vi*. Los valores son simétricos para todas las soluciones s_i, cada solución que contiene el valor a_i puede ser mapeada a una solución que contiene el valor b_i. Identificar simetrías semánticas requiere resolver el CSP para encontrar todas las soluciones, y examinarlas.

- **Simetría sintáctica**. Según [2] se define como:
  Sea $P = (X, D, C)$ un CSP binario, cuyas restricciones son todos miembros de algún conjunto R. Una permutación π de D es una simetría sintáctica si para todo $r_{i,j}$ en R se tiene (d_i,d_j) en $r_{i,j}$ =>($\pi(d_i)$,$\pi(d_j)$), en otras palabras, la permutación π no cambia ninguna relación de restricción de P, considerada como un conjunto de tuplas.

En virtud de todas las definiciones, las simetrías mapean soluciones a soluciones y no soluciones a las no soluciones; las definiciones difieren sobre si se trata de una definición de propiedad, de modo que cualquier mapeamiento biyectivo del tipo adecuado que conserve las soluciones es una simetría, o si es simplemente una consecuencia de dejar las restricciones sin cambios.

3.3.2 Simetría de Valor y Simetría de Variable.

Una simetría s es una simetría de variable si existe una biyección b sobre variables tal que $s(<x, v>) = (b(x), v)$ para cada par $<$ x, v $>$ donde x es una variable, y v el valor desde su dominio. Un tipo común de simetría de variable se presenta cuando se trata con matrices de decisión de variables.

Lo anterior se puede observar en el siguiente ejemplo que consiste en encontrar una relación sobre AxB, donde $A = 1, 2, 3$ y $B = 1, 2, 3$, tal que cada elemento de A está relacionado con dos elementos de B y cada elemento de B está relacionado con dos elementos de A.

Una forma común para modelar este problema es con una matriz bidimensional R indexada por A y B. El domino de todas las variables en R es $0, 1$, y $R[i, j] = 1$ si y solo si $< i, j >$ está en la relación. Para satisfacer las restricciones del problema, se hace que cada fila y columna sumen 2. Considerar la siguiente solución:

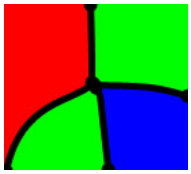

Figura 3.6: Coloreado de mapas

	1	2	3
1	1	1	0
2	0	1	1
3	1	0	1

Esta puede ser transformada hacia otra solución intercambiando los valores de los índices de las filas y/o columnas. Por ejemplo, al cambiar la primera y tercera columnas y la segunda y tercera filas, se obtiene:

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

Esta transformación conserva las soluciones. Este tipo de simetría de variable es conocida como ***simetría de fila*** y ***simetría de columna***. El término general es ***simetría de índice***.

Una simetría s es una ***Simetría de Valor*** si existe una biyección b tal que $s(< x, v >) =< x, b(v) >$ para cada par $< x, v >$. Un ejemplo de simetría de valor es el problema de coloración del mapa, figura 3.6, donde hay un conjunto de colores y se quiere colorear cada región del mapa de manera que las regiones adyacentes tengan distintos colores [1]. En una solución cualquiera, al intercambiar la región coloreada verde y la región roja, se genera otra solución.

3.3.3 Simetría de Restricción y Simetría de Solución.

A continuación se entregan dos definiciones de simetrías para problemas de satisfacción de restricciones, las cuales son lo suficientemente generales como para incluir los tipos de simetrías mencionados antes y otros que no se han nombrado.

Definición: Para una instancia $P = (X, D, C)$ una ***simetría de solución*** de P es una permutación del conjunto X x D que conserva el conjunto de soluciones de P.

Así, una simetría de solución es una asignación biyectiva definida sobre un conjunto de posibles pares variable-valor de un CSP, que mapea solución a solución. Esta definición permite *simetría de variable* y *de valor* como casos especiales.

Para exponer la definición de *simetría de restricción* primero se debe describir una estructura matemática asociada con cualquier instancia de CSP. Para un caso de CSP binario, el detalle de las restricciones puede ser capturado en un grafo, [8], denominado la **microestructura** de la instancia.

Para el caso del CSP binario, $P = (X.D, C)$, la microestructura de P es un grafo con un conjunto de vértices XxD, donde cada arco corresponde a una asignación permitida por una restricción específica, o corresponde a una asignación permitida debido a que no existen restricciones entre las variables asociadas.

Por lo tanto, los vértices de la microestructura corresponden a los pares variable-valor del CSP. Para tratar de mejor forma la definición de simetría se trabaja en la definición con el complemento de la microestructura, que posee el mismo conjunto de vértices, pero con arcos que unen todos los pares de vértices no permitidos por alguna restricción, o bien son asignaciones incompatibles para la misma variable.

Así, se dice que dos vértices (v_1, a_1) y (v_2, a_2) en el complemento de la microestructura están conectados por un arco sí y sólo si:

1. Los vértices v_1 y v_2 están en el alcance (ámbito) de alguna restricción, pero la asignación de a_1 a v_1 y a_2 a v_2 no es permitida por la restricción; o
2. v_1=v_2 y a_1 es distinto a a_2.

Recordando que cualquier conjunto de vértices de un grafo que no contiene un arco se llama *conjunto independiente*, se tiene como consecuencia de la definición anterior que una solución a una instancia de un CSP P es un conjunto independiente de tamaño |X| en el complemento de su microestructura.

Para el caso no binario, la microestructura en un *hipergrafo* cuyo conjunto de vértices es el conjunto de pares variable-valor. En este caso, el conjunto de vértices E es un hiperarco del complemento de la microestructura que representa una asignación no permitida por una restricción, o bien consiste de un par de asignaciones incompatibles para la misma variable. Así, el conjunto de vértices $\{(v_1, a_1), (v_2, a_2), \ldots, (v_k, a_k)\}$ es un hiperarco sí y sólo si:

1. $\{v_1, v_2, \ldots, v_k\}$ es el conjunto de variables en el ámbito de alguna restricción, pero la restricción no permite la asignación $\{ (v_1, a_1), (v_2, a_2), \ldots, (v_k, a_k)\}$;o

2. $K = 2$, $v_1 = v_2$ y $a_1 \neq a_2$.

Como ejemplo, considerar el siguiente sistema de ecuaciones lineales sobre enteros en módulo 2, esto es $1 + 1 = 0$, puede modelarse como una instancia de CSP P=(X,D,C), con X={w,x,y,z}, D={0,1} y C=$\{c_1, c_2, c_3\}$, donde c_1, c_2, c_3 corresponden a cada una de las ecuaciones.

$$x + y + z = 0$$

$$w + y = 1$$

$$w + z = 0$$

El complemento de la microestructura de P se muestra en la figura 3.7. Se caracteriza por tener 8 vértices: <w,0>,<w,1>,<x,0>,<x,1><y,0>,<y,1>,<z,0>,<z,1> y doce hiperarcos.

La ecuación $x+y+z = 0$ no permite la asignación $\{(x, 0), (y, 0), (z, 1)\}$ y otras tres asignaciones. Así, el complemento de la microestructura tiene cuatro hiperarcos ternarios que surgen desde la restricción $x + y + z = 0$. Cada restricción binaria tiene dos hiperarcos binarios. En total el complemento de la microestructura tiene cuatro hiperarcos ternarios y cuatro hiperarcos binarios que surgen desde las tuplas no permitidas. También, hay cuatro hiperarcos binarios, uno por cada variable, que corresponde a los pares de valores distintos para la misma variable, por ejemplo, el hiperarco $\{(y, 0), (y, 1)\}$.

Recordando que un automorfismo de un grafo o hipergrafo es un mapeo biyectivo de los vértices que conserva los arcos. Así, con estas definiciones se puede definir ***simetría de restricción*** como sigue.

Definición: Para una instancia de CSP $P = (X, D, C)$, una *simetría de restricción* es un automorfismo del complemento de la microestructura de P, figura 3.7.

Ejemplo. Considerando las simetrías de restricción del CSP definido en el ejemplo anterior, cuyo complemento de microestructura se muestra en la figura 3.7. Los automorfismos de este grafo son:

- $((w, 0)(w, 1))((y, 0)(y, 1))((z, 0)(z, 1))$;
- $((w, 0)(w, 1))((y, 0)(z, 1))((y, 1)(z, 0))$;
- $((w, 0)(w, 1))((y, 0)(z, 0))((y, 1)(z, 1))$.

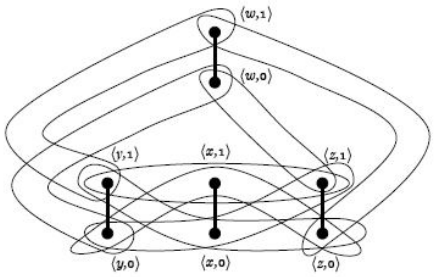

Figura 3.7: Complemento de la microestructura definida en el ejemplo.

Además del mapeo de la identidad (las permutaciones de los vértices son escritas en forma cíclica, esto es, cambia primero los vértices $(w, 0)$ y $(w, 1)$, cambiando simultáneamente $(y, 0)$ y $(y, 1)$, y cambiando $(z, 0)$ y $(z, 1)$, pero dejando $(x, 0)$ y $(x, 1)$ sin cambiar).

Así, los cuatro mapeos descritos, son el *grupo de simetría de restricción del CSP*. También se ve en este ejemplo que puede haber más simetrías de solución que simetrías de restricción. El CSP tiene sólo dos soluciones , $\{(w, 0), (x, 1), (y, 1)(z, 0)\}$ y $\{(w, 1), (x, 1), (y, 0), (z, 1)\}$. La permutación $\{(w, 0), (z, 0), (y, 1)\}$. La que realiza un mapeo de $(w, 0)$ a $(z, 0)$, $(z, 0)$ a $(y, 1)$, $(y, 1)$ a $(w, 0)$ y deja todos los otros pares variable-valor sin cambiar, es una *simetría de solución.*

En [4] se aclara la relación entre simetría de solución y simetría de restricción. Lo anterior se debe a que se considera importante distinguir entre los dos tipos de simetrías, pues en general puede haber mayor cantidad de simetrías de solución que de simetrías de restricción para un CSP dado

Como conclusión de este apartado se puede decir que la principal razón para identificar simetría en un CSP es para reducir el esfuerzo de búsqueda en la exploración de asignaciones consideradas simétricamente equivalentes. Es evidente que, si el grupo de simetrías de solución es más grande que el grupo de simetría de restricción, se dará una mayor reducción de la búsqueda a través de la utilización de simetrías de solución.

Cuando se encuentran todas las soluciones, el objetivo de romper las simetrías es encontrar una solución de cada clase de equivalencia de simetría.

3.3.4 Simetría en n-reinas.

Basándose en el problema de las n-reinas (figura 3.2), se ilustra la relación existente entre *simetría de restricción* y *simetría de solución*. Este problema es útil para analizar la simetría

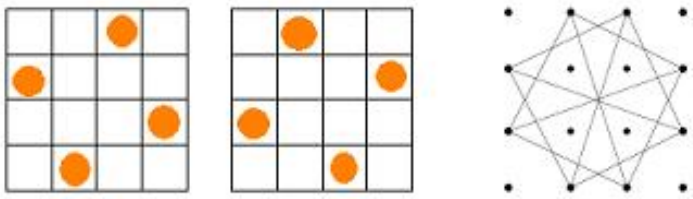

Figura 3.8: Soluciones a n-reinas, n igual a 4, y el complemento del grafo binario.

debido a que la formulación de CSP común tiene diferentes tipos de simetría, algunas de las cuales van más allá del alcance de las primeras definiciones([4]).

La formulación común del problema de n-reinas como un CSP tiene n variables que corresponden a las filas del tablero, llamadas $r_1, r_2, ..., r_n$. El dominio de los valores corresponde a las columnas del tablero,llamados $D = 1, 2, ..., n$. Las restricciones pueden ser expresadas como sigue, para un CSP binario dado:

- para todo $i, j, 1 \leq i \leq n, r_i \neq r_j$;
- para todo $i, j, 1 \leq i \leq n, |r_i - r_j| \neq |i - j|$.

Considerado como un objeto geométrico, un tablero de ajedrez tiene ocho simetrías: reflexión en el eje vertical y horizontal y las dos diagonales, rotación en 90^0, 180^0 y 270^0, y la identidad.

Claramente, el conjunto de soluciones para una instancia de un problema de n-reinas es invariante bajo cada una de las ocho simetrías geométricas del tablero de ajedrez. Por lo tanto, cada una de esas simetrías geométricas es una simetría de solución del problema de n- reinas para algún n dado, de acuerdo a la definición de simetría de solución entregado en 3.3.3. Sin embargo, puede haber muchas otras simetrías de solución para los casos de este problema. A continuación se muestran algunos de ellos.

El problema de 3-reinas no tiene soluciones; como en cualquier otro CSP sin solución.

El problema de las 4-reinas tiene dos soluciones, como se muestra en la figura 3.8. En este ejemplo se muestra el grafo, en el cual cada arista representa la asignación al par variable-valor que es permitida por las soluciones, dibujando así la posición de cada vértice, que representa un par variable-valor, y un cuadro del tablero de ajedrez, que corresponde a su posición sobre el tablero.

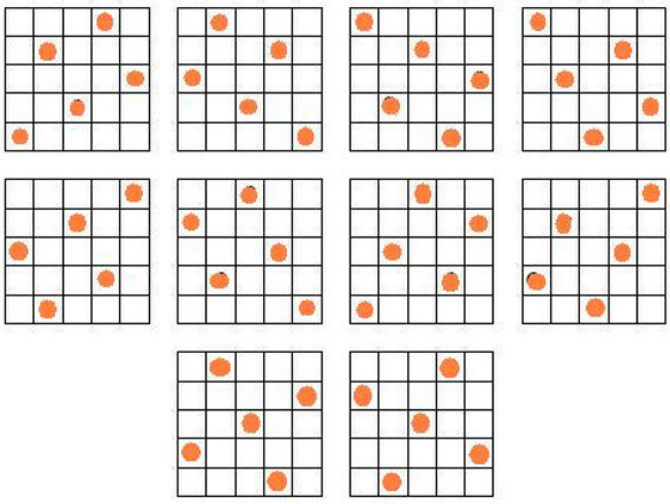

Figura 3.9: Soluciones a n-reinas, n igual a 5.

El problema de las 5-reinas tiene diez soluciones, las que se muestran en la figura 3.9.

3.3.5 Estrategias para Explotar Simetrías.

Uno de los motivos principales para identificar simetrías en un CSP es el de reducir el esfuerzo de búsqueda mediante la no exploración de las asignaciones que son simétricamente equivalentes a las asignaciones consideradas en el resto de la búsqueda. Se puede ver que si el grupo de simetrías de solución es más grande que el grupo de simetrías de restricción, habrá potencialmente una mayor reducción de la búsqueda al usar simetrías de solución, si ellas pueden ser identificadas anticipadamente.

La simetría en un CSP es usualmente identificada, en la práctica, aplicando el conocimiento humano, es decir, el programador ve que alguna transformación puede transformar una solución hipotética en otra hipotética solución. La definición de simetría de restricción entregada en 3.3.3 puede ser usada para confirmar que las transformaciones candidatas son verdaderas simetrías.

Se han desarrollado diversas estrategias para explotar simetrías en la resolución de un CSP. Algunas de ellas son:

1.) **Incorporar nuevas restricciones para romper simetrías**. En [14] se muestra que algunas simetrías en Problemas de Satisfacción de Restricciones pueden ser removidas a través de la incorporación de ciertas restricciones a la formulación del problema, lo que elimina aquellas

restricciones simétricas. El nuevo problema mantiene todas las restricciones no simétricas pero elimina las simétricas. Las nuevas restricciones se pueden agregar antes de comenzar la búsqueda o durante ella.

2.) **Modificación de la estrategia de búsqueda.** Se modifica la estrategia de búsqueda para evitar visitar estados simétricos a los ya visitados. Para ellos se definen las soluciones canónicas y se podan los subárboles que no las contienen. Esto puede ser visto en [7].

3.) **Ruptura de simetrías durante la búsqueda.** En [6] se introducen los árboles de búsqueda que excluyen simetría, y se describe con mayor detalle la implementación de esta técnica, denominada SBDS por sus definición en inglés. La idea básica de esta técnica es la de introducir restricciones a un problema de modo que, después de backtracking la restricción de SBDS garantice que no hay equivalente simétrico de la decisión permitida. Esta es una técnica dinámica, ya que no se puede añadir restricciones hasta no saber qué decisión de búsqueda se está realizando. En términos generales SBDS puede trabajar sobre todo tipo de decisiones de búsqueda. En otras palabras, cuando se visita un subárbol se añaden restricciones que evitan visitar un subárbol simétrico en el futuro.

4.) **Heurística de ruptura de simetrías.** Al asignar una variable se rompen simetrías. En [12] se estudia cómo la simetría puede ser usada para guiar la búsqueda. Concretamente se lleva la búsqueda hacia subespacios con un alto grado de asignaciones no simétricas, rompiendo tantas simetrías como sea posible con cada asignación de variable. La heurística de ruptura de simetría propone romper tantas simetrías como sea posible, orientando la búsqueda hacia subespacios con menor número de estados simétricos.

3.4 Expresando Restricciones.

Una vez que se ha llegado a un punto de vista que permite tener restricciones que sean expresadas fácil y brevemente, existen opciones para elegir la exactitud de las restricciones. La forma en que son escritas las restricciones afecta la eficiencia del modelo resultante, ya que afecta la forma en que las restricciones propagarán durante la búsqueda.

En [11] se menciona que la forma de las restricciones puede cambiar la cantidad de información que la propagación entrega. Para ilustrar lo anterior se utilizan las siguientes restricciones: Sean x, y, z variables enteras.

- $c_1 : x = y$;
- $c_2 : x + y = z$;

- $c_3 : 2y = Z$.

Si $C = c_1, c_2$ y $C' = c_1, c_3$, C y C' son equivalentes en el sentido que tienen ambas las mismas soluciones. Sin embargo, si C y C' son hechas localmente consistentes, entonces en C' el dominio de z será entero, no siendo necesariamente verdadero para C.

Para llegar a tener un buen modelo del problema P, es necesario conocer el rango de restricciones soportado por el solver, el nivel de consistencia exigido y tener alguna idea del grado de complejidad del algoritmo de propagación. Lo anterior está muy lejos del ideal declarativo existente. A continuación se describen algunas de las opciones disponibles cuando se describen restricciones:

3.4.1 Combinación de Restricciones.

La combinación de restricciones con el mismo ámbito puede ser una forma de expresarlas más brevemente. La unión de dos restricciones con el mismo ámbito permite solo las tuplas que ambas permiten en forma separada.

Hacer cumplir el mismo nivel de consistencia local sobre una conjunción $c_1 \wedge c_2$ como sobre c_1 y c_2 en forma separada eliminará al menos el mismo número de valores del dominio. Sin embargo, puede o no reducir el tiempo de ejecución, dependiendo de cuánto tiempo se consume en hacer cumplir la consistencia en la conjunción y en las restricciones en forma separada.

Un ejemplo se puede observar en el problema de las n-reinas [13]. Analizando el primer punto de vista definido en 3.1, donde las variables $x_1, x_2, ..., x_n$ representan las filas de cada tablero, y los valores son $1, 2, ..., n$, representando las columnas, la regla para que dos reinas no estén en la misma columna o diagonal puede ser expresada a través de más de una restricción entre cada par de variables, x_i y x_j, con $i<j$. Por ejemplo:

- $x_i \neq x_j$;
- $x_i - x_j \neq j - i$;
- $x_j - x_i \neq j - i$.

La figura 3.10 muestra un estado al que se puede llegar durante la búsqueda, cuando $n = 6$. dos variables, x_1 y x_2 ya han sido asignadas y los cuadros marcados con "X" ya no están disponibles.

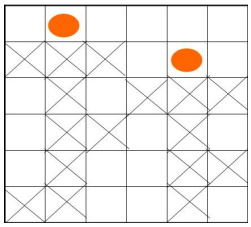

Figura 3.10: Un estado de búsqueda en 6-reinas.

3.4.2 Eliminación de Variables.

Ciertamente la reducción del número de variables se contrapone con la idea de utilizar diversos puntos de vista para modelar un problema. Es probable que un modelo que requiere pocas asignaciones de variables para describir las soluciones del problema sea un mejor modelo.

3.4.3 Restricciones Globales.

Desde el punto de vista de los algoritmos, es habitual ver las restricciones como relaciones entre variables, sin asumir ninguna semántica. Esta idea no es adecuada cuando se trabaja sobre problemas reales. Existe un conjunto de restricciones con un significado bien definido que aparecen a menudo en las aplicaciones. Explotar su semántica permite obtener procedimientos de propagación más eficientes, con lo que disminuye la complejidad de los algoritmos.

Estas restricciones suelen ser no binarias, y se denominan genéricamente *restricciones globales*. El ejemplo clásico de restricción global es la restricción descomponible *alldifferent* $(x_1, x_2, ..., x_i)$. Esta restricción es equivalente a un conjunto de $(i \cdot (i-1))/2$ restricciones binarias distintas entre las variables. Comúnmente, una restricción global es semánticamente redundante, en el sentido de que la misma relación puede ser expresada como una conjunción de varias restricciones más simples.

Los solvers poseen un rango de restricciones globales, desarrollado para reemplazar conjuntos particulares de restricciones que ocurren frecuentemente. Las restricciones globales permiten que una sola restricción sobre un número de variables reemplace un conjunto de restricciones.

Algunos ejemplos de restricciones globales útiles existentes se presentan a continuación:

- **Restricción *Alldifferent:*** Es la restricción más conocida, más influyente y más estudiada en la programación con restricciones, aparte de su simplicidad y aplicabilidad práctica.

Definición[16]**:** Sean las variables $x_1, x_2, ..., x_n$, luego:

$$alldifferent\ (x_1, x_2, ..., x_n) = (\ (d_1, d_2, ..., d_n) \mid \forall\ d_i \epsilon\ D(x_i), \forall\ i \neq j, d_i \neq d_j\ )$$

Un problema que puede ser modelado con restricciones *alldifferent* es el conocido problema de las n-reinas, 3.2, Una forma de modelar este problema es la de introducir una variable x_i para cada fila $i = 1, ..., n$, que va desde la columna 1 a la n. Esto significa que en una fila i, una reina es ubicada en la x_i columna. El dominio de cada x_i es $D(x_i) = 1, 2, ..., n$, y se expresa el "no ataque de las reinas"por las siguientes restricciones:

1. $x_i \neq x_j\ para\ 1 \leq i < j \leq n$,
2. $x_i - x_j \neq i - j\ para\ 1 \leq i < j \leq n$,
3. $x_i - x_j \neq j - i\ para\ 1 \leq i < j \leq n$.

La restricción (1) dice que dos reinas no pueden estar en la misma columna y las restricciones (2) y (3) establecen los casos de las diagonales. A continuación se muestra un modelo más preciso, después de haber reordenado los términos de las restricciones (2) y (3), transformándose el modelo en:

1. $alldifferent(x_1, x_2, ..., x_n)$,
2. $alldifferent(x_1 - 1, x_2 - 2, ..., x_n - n)$,
3. $alldifferent(x_1 + 1, x_2 + 2...., x_n + n)$,
4. $x_i\ \epsilon\ (1, 2, ..., n)\ para\ 1 \leq i \leq n$.

- **Restricción *Element*:** Sea y una variable entera, z una variable con dominio finito y c un arreglo de variables, esto es $c = [x_1, x_2, ..., x_n]$. La restricción *element* establece que z es igual a la y-ésima variable en c, o $z = x_y$. Una definición más formal es:

$$element(y, z, x_1, ..., x_n) = \{(e, f, d_1, d_n) \mid e\ \epsilon\ D(y),\ f\ \epsilon\ D(z), \forall\ i\ d_i\ \epsilon\ D(x_i),\ f = d_e\ \}.$$

Este modelo puede ser aplicado en muchos problemas prácticos, especialmente cuando se desea modelar subíndices variables ([16]).

- **Restricciones de *Suma* y *Knapsack* (mochila):**La restricción *Sum* es una de las que ocurre más frecuentemente. Sean, $x_1, x_2, ..., x_n$ las variables. Para cada variable x_i se asocia un escalar $c_i\ \epsilon\ Q$. Además, sea z una variable con dominio $D(z) \subseteq Q$. La restricción

Sum se define como :

$Sum(x_1, ..., x_n, z, c) = \{ (d_1, ..., d_n, d) \mid \forall\ i\ d_i \in D(x_i), d \in D(z), d = \sum_{i=1}^{n} c_i d_i \}$. También se escribe $z = \sum_{i=1}^{n} c_i x_i$.

La restricción ***Knapsack*** es una variante de la restricción ***Suma***. En lugar de restringir la suma a un valor específico, la restricción ***Knapsack*** establece que la sumatoria posee un límite superior u y un límite inferior l. tradicionalmente se escribe $l \leq \sum_{i=1}^{n} c_i x_i \leq u$. En [16] se establece l y u como una variable z, tal que $D(z) = [l, u]$. Luego, se define la restricción ***Knapsack*** como:

$Knapsack(x_1, ..., x_n, z, c) =$

$$\{ (d_1, ..., d_n, d) \mid \forall\ i\ d_i \in D(x_i), d \in D(z), d \leq \sum_{i=1}^{n} c_i d_i \} \bigcap$$
$$\{ (d_1, ..., d_n, d) \mid \forall\ i\ d_i \in D(x_i), d \in D(z), \sum_{i=1}^{n} c_i d_i \leq d \},$$

lo que corresponde a $min\ D(z) \leq \sum_{i=1}^{n} c_i x_i \leq max\ D(z)$.

3.4.4 Variables Auxiliares.

En la sección 3.4 se comentaron algunas formas de representar las restricciones cuando se elige una manera de representar un problema. Sin embargo, si se incluyen nuevas variables se pueden tener mayores opciones de representación y un mayor potencial de obtener un modelo más eficiente.

Las variables auxiliares son variables que se introducen en un modelo, debido a la dificultad de expresar las restricciones en términos de las variables existentes o a expresar las restricciones de una forma que se propague mejor. En [5, 17] se ejemplifica la utilización de variables auxiliares en un problema. En [5] un número de autos debe ser hecho en una línea de producción. Cada uno de ellos requiere de una o más opciones que se instalan en diferentes estaciones de la línea. La estaciones de opción tienen capacidad menor que las del resto de la línea de producción. Por ejemplo, la estación podría ser capaz de hacer frente a lo más a un auto de cada dos. Los autos se han organizado en una secuencia de producción a fin de que estas capacidades no sean superadas.

Según [5], el punto de vista inicial tiene variables s_i, $1 \leq i \leq n$, donde n es el número de autos a producirse, o sea la longitud de la secuencia de producción. El valor s_i representa al auto a ser producido en la posición i en la secuencia, o más precisamente la *clase* de auto. Puesto que los autos que requieren el mismo conjunto de opciones pueden ser considerados como idénticos. Algunas de las restricciones serían por ejemplo: El número de variables asignada a un número específico es igual al número de autos en la clase correspondiente. Sin embargo, la opción de las

capacidades es difícil de expresar con estas variables por sí solas.

Es así como se introducen variables auxiliares booleanas o_{ij}, $1 \leq i \leq n$, $1 \leq j \leq m$ tal que $o_{ij} = 1$, si y sólo si el auto en la i-ésima ubicación en la secuencia requiere la opción j. Las restricciones que expresan la capacidad de opción son expresadas en términos de estas variables, suponiendo que la capacidad de la opción 1 es un auto de cada dos. Luego, la capacidad de la opción puede ser impuesta usando las restricciones:

$$o_{i,1} + o_{i+1,1} \leq 1 \; para \; 1 \leq i < n$$

Las restricciones también deben expresar la relación entre las variables auxiliares y las variables originales, en este caso se hace por las restricciones:
$o_{i,j} = \lambda_{s_i,j}$, $1 \leq i \leq n$, $1 \leq j \leq m$, donde $\lambda_{k,j} = 1$ si y sólo si la clase de auto k requiere la opción j.

Usualmente las variables auxiliares no son suficientes para formular un problema. Sin embargo, las variables auxiliares en el problema de secuencias de autos puede constituir un modelo, puesto que cada secuencia de producción válida puede ser especificada como una completa asignación de estas variables.

Las variables auxiliares también son útiles en algunas ocasiones para ser utilizadas como variables de búsqueda, lo que se da a conocer en [17].

3.4.5 Restricciones Implícitas.

También conocidas como restricciones redundantes, son aquellas restricciones que están contenidas en las restricciones que definen el problema. No cambian el conjunto solución y por lo tanto son lógicamente redundantes.

Aunque las restricciones implícitas pueden reducir significativamente el tiempo de ejecución, la elección de restricciones implícitas útiles es considerado un arte. Se establecen dos criterios básicos. Primero, se requieren restricciones implícitas para las cuales especializados y eficientes algoritmos de propagación de restricciones están disponibles, o restricciones de aridez pequeña. En segundo lugar,las circunstancias en las cuales una restricción implícita lleva a la poda de valores adicionales deben ser obvias. Sin embargo, si bien estos criterios son deseables, no son suficientes. La experimentación es necesaria.

Una condición necesaria para que una restricción implícita sea útil en la reducción de la búsqueda es que "prohíba" una o más asignaciones compuestas que permiten las restricciones existentes. Una restricción compuesta que es prohibida por una restricción implícita no conduce a una solución, es así que al realizar esta operación no cambia el espacio solución. Sin la restricción implícita la asignación podría ocurrir durante la búsqueda y probar que no puede ser

determinada puede tomar una gran cantidad de tiempo. En [5] se da a conocer un ejemplo de este tipo de restricciones.

3.4.5.1 Restricciones Implícitas y Espacio de Búsqueda.

Asegurar que una restricción implícita no permite una asignación que de otra forma sería realizada no es suficiente para garantizar que las restricciones implícitas reduzcan el espacio de búsqueda. Puede ser que una asignación prohibida no ocurriera durante la búsqueda de todas formas, dado cierto orden de búsqueda. Por lo anterior, en backtracking, el orden según el cual las variables son asignadas puede afectar el hecho de si será beneficioso o no incorporar una restricción implícita o no.

Por ejemplo en [3] se comenta sobre agregar una restricción implícita entre dos restricciones, q y r, cuando existen restricciones binarias en un CSP entre p y q y entre p y r. La restricción c_{qr} puede ser obtenida de la unión de c_{pq} y c_{qr}. Los autores de [3] muestran a través de un algoritmo de backtracking, que si tres variables p, q y r son asignadas en este orden, la restricción implícita c_{qr} no tendrá efecto sobre el número de nodos visitados. Por otro lado si el CSP ya contiene las restricciones c_{pr} y c_{qr}, al agregar la restricción c_{pq} puede reducir el número de nodos visitados, con el mismo orden de búsqueda.

3.5 En Busca de Soluciones

El problema de las n-reinas mostrado en (ver figura) consistente en ubicar n reinas en un tablero de ajedrez con la condición de que ninguna sea alcanzada por otra, es un buen ejemplo para poder visualizar la forma en que un algoritmo de búsqueda de soluciones trabaja.

Como ya se mencionó, se puede representar el problema de las n-reinas como un CSP usando n variables, $q_1, ..., q_n$, las que representan las filas del tablero de ajedrez. El dominio de cada variable sería $1, ..., n$, que representa las columnas del tablero.

La idea es encontrar una asignación de valor a cada variable tal que todas las restricciones sean satisfechas. Es necesario un tipo de algoritmo para resolver éste y otros CSPs. Algunas de las propiedades que se desea posean estos algoritmos son:

- Solidez: La solución que el algoritmo encuentra es una solución genuina.
- Completitud: El algoritmo garantiza encontrar una solución en un tiempo finito. si es que existe alguna. Por lo tanto si finaliza sin encontrar una solución, se debe a que el problema no tiene solución.

Solidez es la más importante. Completitud también es una necesidad, así todos los algoritmos que se utilicen serán completos. Pero existen algoritmos incompletos para resolver CSPs (por ejemplo algoritmos de búsqueda local).

Si los problemas son muy difíciles de resolver, un algoritmo completo podría tardar demasiado en demostrar que una instancia no tiene solución. Así, un algoritmo que en teoría es completo puede llegar a ser incompleto en la práctica. En estas circunstancias es mejor utilizar un algoritmo que se sabe es incompleto, si tiene una mejor oportunidad de encontrar una solución cuando existe una. Por otro lado, si un problema no tiene solución, un algoritmo incompleto no tendrá forma de detectarlo, no importando el tiempo que utilice en ejecutarse, mientras que un algoritmo completo puede ser capaz de probar que existe solución.

3.5.1 Generar y Probar.

Una forma simple para encontrar una solución a un CSP es generar todas las posibles asignaciones a las variables, de alguna forma sistemática, y probar cada una para ver si satisfacen las restricciones. Si ocurre esto, se tiene una solución y se puede parar (so solo se quiere una solución), de lo contrario se continúa. Si todas las asignaciones ha sido probadas y no se ha encontrado una solución, entonces no existe una solución para el problema. Así, este algoritmo es tanto *sólido* como *completo*.

si cada variable tiene m valores posibles y hay n variables, entonces existen m^n posibles asignaciones. Se tendrán que examinar todas éstas, por si no existe una solución o la última asignación es la única solución o si se quiere una única solución o se quieren todas la soluciones o se quiere la mejor solución (en cuyo caso se encuentran todas las soluciones y se escoge la mejor). Si existe solución y se desea escoger sólo una, se podría ser muy afortunado y encontrar que la primera asignación que se examina satisface las restricciones. En general, sin embargo, se trata de un algoritmo realmente torpe.

Para explicar lo anterior, suponiendo que se tiene que las restricciones no permiten $v_1 = 1$ y $v_2 = 1$, como en el caso de las n-reinas por ejemplo. Hay m^{n-2} asignaciones en que esta combinación de valores aparece. Se deben generar todas las asignaciones y saber cuáles de ellas no son válidas, pues el algoritmo no tiene forma de saber cuáles de ellas son un error.La mayoría de los algoritmos para encontrar soluciones a un CSP se basa en búsqueda sistemática, y pueden evitar en mayor o menor medida, considerar asignaciones parciales que no llevan a una solución.

3.5.2 Búsqueda Espacio-Estado.

Esta es una técnica fundamental en Inteligencia Artificial para resolver problemas. Dado un *estado inicial*, uno o más *estados finales*, y un conjunto de *operadores* los que producirán los posibles sucesores de un estado, se quiere saber cómo llegar de un estado inicial a un estado final.

Se puede representar la búsqueda por un estado final como un árbol, con los nodos representando los estados (el estado inicial será el nodo raíz) y los arcos los posibles operadores. Buscando la solución, se construye gradualmente el árbol, eligiendo en forma repetida un nodo y uno de los operadores principales, creando un nuevo nodo que representa el estado resultante. Dos técnicas de búsqueda importantes son *Búsqueda en profundidad* y *Búsqueda en amplitud*.

En la *Búsqueda en Profundidad*, siempre se elige explorar un nodo que es hijo del nodo más recientemente explorado. si este nodo no tiene hijos se retrocede hasta el camino que condujo a este nodo hasta que se encuentra un nodo con un hermano inexplorado. Se debe guardar en memoria todos los hermanos no explorados de un nodo en el camino desde el nodo actual hacia la raíz del árbol (en caso de que se deba volver a uno de ellos).

En la *Búsqueda en Amplitud*, cada nodo en el nivel d es explorado antes que un nodo en el nivel $d + 1$. Los requerimientos de memoria en este caso son más grandes (y aumentan tanto como se baje en el árbol).

Algunas de las ventajas de la búsqueda en amplitud son:

- Se encuentra el camino más corto hacia el estado final (lo cual es importante en algunos problemas.)
- En la búsqueda en amplitud se puede en algunos tipos de problemas quedar atrapado en la exploración de un camino infinito y así nunca llegar a alcanzar un estado final.

3.5.2.1 Encontrar una solución a un CSP como una búsqueda Espacio-Estado.

Una posible forma de encontrar una solución a un CSP en términos de búsqueda espacio-estado se describe a continuación:

- En el **estado inicial**, no se han asignado valores a las variables.
- Un **estado final** es un estado en que a cada variable se ha asignado un valor y todas las restricciones son satisfechas (o sea, corresponde a una solución).
- Un **estado intermedio** corresponde a soluciones en que a algunas variables les han sido asignados valores y las restricciones relevantes son satisfechas.

- Los **operadores** son: - seleccionar una variable a la cual aún o se le ha asignado un valor; - un nodo donde una variable ya ha sido seleccionada, seleccionar uno de los valores en su dominio.

El árbol de búsqueda que se forma de este modo cuando se resuelve un CSP tiene características especiales:

- el árbol de búsqueda es finito; no puede haber caminos de largo infinito en él.
- todas las soluciones ocurren en el mismo nivel en el árbol (por ejemplo, profundidad $2n$, si n es el número de variables, y se contabiliza el nodo raíz del árbol como profundidad 0. Se tienen n niveles que corresponden a la elección de cada variable en un turno, alternando con n niveles en los que se elige un valor para la variable actual.)

Debido a estas características, se elige el algoritmo de búsqueda en profundidad más que el algoritmo de búsqueda en anchura.

Se asumirá, para efectos de explicación, que el orden en el que las variables son seleccionadas es predeterminado, así que la selección de la próxima variable no se muestra como parte del proceso de búsqueda. Sin embargo, la estrategia de selección de variable en efecto juega un importante rol en la resolución exitosa de un CSP.

El algoritmo más simple basado en búsqueda en profundidad es llamado *Backtracking Simple*.

1. Elegir una variable a la cual no se le ha asignado un valor. Si no existe ninguna, entonces se detiene: una solución ha sido encontrada.
2. Elegir un valor para esta variable.
3. Probar si la nueva solución parcial satisface todas las restricciones relevantes (por ejemplo, todas las restricciones que afectan a la variable actual y una o más de las variables anteriores, a las que ya se le ha asignado un valor):
    - si es así, volver al paso 1.
    - si no es así, elegir otro valor para esta variable; si no hay más valores, retroceder a una variable que aún posee valores que no han sido probados; si no hay tal variable, parar: no hay solución.

La figura 3.11 muestra el árbol de búsqueda para el problema de 4-reinas usando backtracking simple. (Como ya se mencionó, se ignora la selección de la próxima variable en este árbol de búsqueda; se asume que las variables son consideradas en el orden q_1,q_2,q_3,q_4).

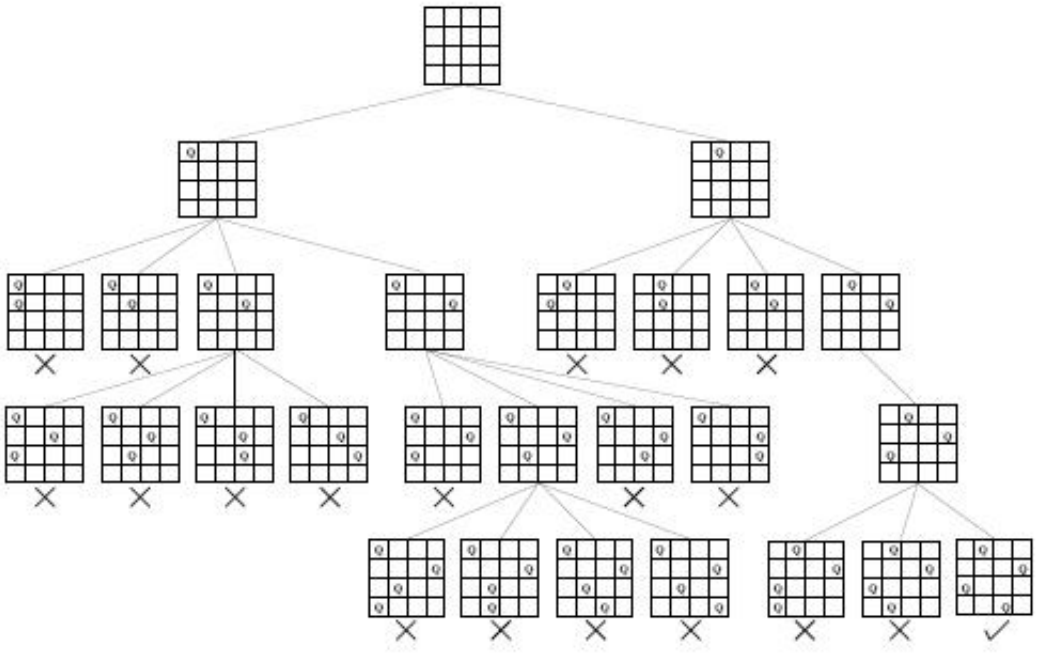

Figura 3.11: Árbol de búsqueda para 4-reinas usando backtracking simple.

3.5.3 Forward Checking.

Debido a que el algoritmo de backtracking sólo revisa las restricciones entre la variable actual y la variable anterior, lo que intenta muchas asignaciones las cuales pueden fallar. Por ejemplo, si la reina en la primera fila, está en la primera columna, no tiene sentido tratar de ubicar otra reina en esta columna. El algoritmo de forward checking intenta evitar asignaciones inútiles buscando el efecto futuro de una asignación sobre las futuras variables (estas variables son las que aún no han sido asignadas).

Cuando un valor es asignado a la variable actual, cualquier valor en el dominio de una variable futura que entra en conflicto con la nueva solución parcial es removida (temporalmente) del dominio. La ventaja de esto es que si el dominio de una futura variable, llamada v_f, llega a estar vacío, se sabe inmediatamente que la solución parcial actual es inconsistente, y como antes, se intenta para otro valor de la variable actual o se intenta el algoritmo de backtrack a la variable anterior; el estado de los dominios de las variables futuras, que estaba antes que las asignaciones fracasaran, es restaurado. Con backtracking simple, este fracaso será detectado mucho más tarde. Cuando v_f llega a ser la variable actual y se intenta hacer una asignación, se descubrirá que ninguno de sus valores son consistentes con la solución parcial actual. Forward checking por lo tanto permite ramas del árbol de búsqueda que llevarán al fracaso al ser podadas

más temprano que con backtracking simple.

Esto puede ser mostrado usando el problema de las 4-reinas. Si se comienza ubicando una reina en la primera fila, entonces ninguna de las otras reinas puede ser ubicada en la misma columna o en la misma diagonal. Los valores correspondientes al cuadrado atacado por esta reina pueden ser removidos desde el dominio de las variables representadas por las reinas en las filas 2 ó 4, hasta que esta rama conduzca a un callejón sin salida, y la reina en la primera fila sea movida. El árbol de búsqueda completo construido a través de forward checking para este problema se muestra en la figura 3.12. Los cuadrados con cruces denotan valores removidos desde el dominio de variables futuras en asignaciones pasadas y actuales.

Cada vez que una nueva variable es considerada, se garantiza que todos sus valores restantes sean consistentes con las variables anteriores, así que chequear una asignación con una asignación pasada ya no es necesario.

Las únicas restricciones que se necesitan considerar cuando se realiza forward checking son esas que involucran tanto la variable actual y exactamente una variable futura. Estas restricciones pueden ser restricciones binarias o restricciones de una aridez mayor en las cuales a todas excepto a estas dos variables se les han asignado valores. En este último caso las restricciones se convierten en binarias (aunque esto puede ser sólo temporal: se puede encontrar después que se tiene que retroceder y deshacer algunas de estas asignaciones). Este es un ejemplo de por qué las restricciones binarias o restricciones que involucran sólo un pequeño número de variables son tan importantes en un CSP; se puede utilizar restricciones binarias para propagar los efectos de una asignación a una variable no asignada, aún si las restricciones no son binarias en un principio, ellas llegarán a ser binarias durante la búsqueda, cuando a las variables se les asignen valores.

En cada nodo, forward checking hace más trabajo que en backtracking simple, de modo que, en general, puede tomar más tiempo encontrar una solución. Por ejemplo, es muy fácil satisfacer las restricciones,puede ser que el backtracking simple encuentre una solución sin tener que retroceder a la variable anterior; en este caso, chequear el efecto de asignaciones sobre el dominio de variables futuras carece de esfuerzo. Esto rara vez sucede en la práctica, sin embargo, casi siempre forward checking es mucho más rápido que backtracking simple.

A través del siguiente ejemplo se muestra que forward checking puede ahorrar una cantidad arbitraria de trabajo comparado con backtracking simple: Suponer que se tiene las variables $x_1, x_2, x_3, ..., x_n$ donde x_1, x_2, x_n tienen dominio $1, 2$ y las restricciones sobre estas tres variables son que todas ellas deben tener diferentes valores. Claramente este subproblema, y por lo tanto el problema entero, no es factible. (No le interesa cuál es el dominio de las variables restantes $x_3, ..., x_{n-1}$, o qué restricciones existen sobre estas variables: se asumirá que esta parte del pro-

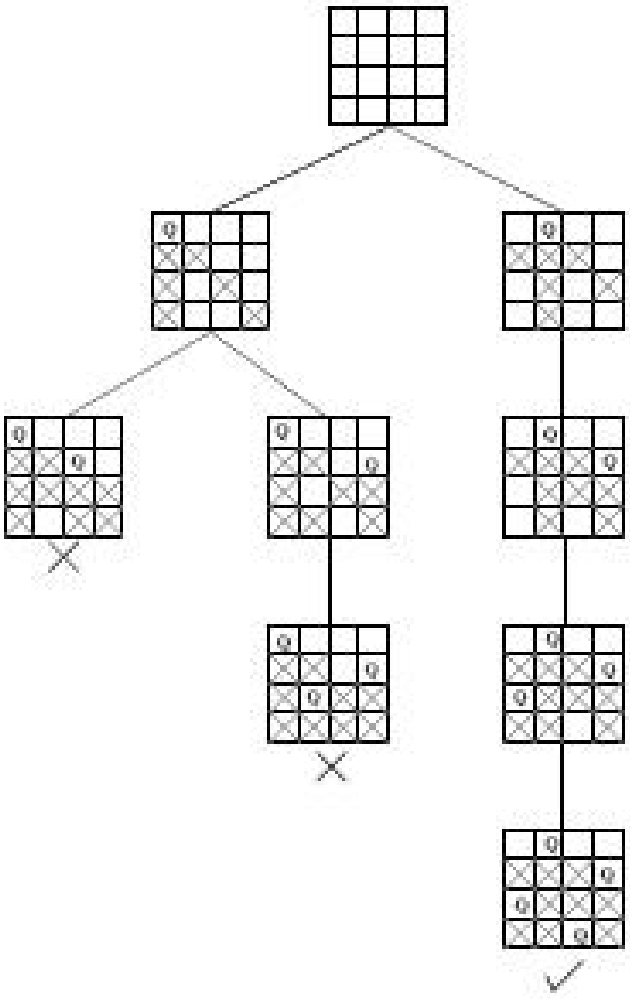

Figura 3.12: Árbol de búsqueda para 4-reinas usando forward checking.

blema puede ser resuelto sin dificultad.) El algoritmo de backtracking instanciará las variables x_1 y x_2 a 1 y 2 respectivamente, y luego asignará valores a $x_3, ..., x_{n-1}$ a su vez, antes de descubrir que no hay valor para x_n que sea consistente con las primeras dos asignaciones. Entonces volverá a x_{n-2} y así sucesivamente, aunque estas variables no son parte del subproblema que causa la dificultad. Tomará una gran cantidad de tiempo descubrir que el problema no tiene solución. Forward checking, por otro lado, descubrirá que no hay valores disponibles en el dominio de x_n tan pronto como los valores se asignen a x_1 y x_2, nunca considerará asignar valores a las variables restantes.

3.5.4 Ordenamiento de Variable y de Valor.

En el ejemplo de las n-reinas explicado anteriormente, se han considerado las variables en el orden $q_1, q_2, ..., q_n$, y los valores en el orden $1, 2, ..., n$. No existe una razón en particular para creer que es la mejor orden en que se consideran las variables y los valores, y la selección de la próxima variable a considerar y el valor que se asigna a ésta es claramente parte del proceso de búsqueda.

El orden en que las variables son asignadas tiene un efecto dramático en el efecto utilizado para resolver un CSP, como el orden en que cada valor de variable es considerado. Por ejemplo, suponer que el problema tiene una solución en la cual $v_1 = l_1, v_2 = l_2, ...,$ y sucederá que se elige el valor para cada variable de tal forma que el primer valor considerado para cada variable es su valor en esta solución. Entonces se puede encontrar esta solución incluso utilizando backtracking simple, muy rápidamente.

Un buen orden de variable reducirá el tamaño del árbol explorado por el algoritmo de búsqueda, en comparación con un pobre ordenamiento.

El orden de variable puede ser un *orden estático*, en el cual el orden de las variables se especifica antes que la búsqueda comience, y no cambia posteriormente, o u *orden dinámico* en el cual la elección de la próxima variable a ser considerada en cualquier punto depende del estado actual de la búsqueda. El ordenamiento dinámico no es factible para todos los algoritmos de árbol de búsqueda: por ejemplo, con backtracking simple no hay información extra disponible durante la búsqueda que pueda ser usada para realizar una elección diferente de ordenamiento desde el orden inicial: todas las variables futuras ven lo mismo como lo hicieron al principio, así, no hay motivo para cambiar el orden.

CAPÍTULO 4

Formalización de las Características de Modelado a Utilizar

4.1 Características de Modelado Seleccionadas

De la información descrita en el capítulo anterior se han extraído algunas características de modelado que serán consideradas para analizar de qué forma influyen en la resolución de un CSP. Con ellas se realizarán algunas formulaciones de problemas clásicos existentes en la literatura.
Las características de modelado a utilizar son las siguientes:

1. Utilización de diversos *Viewpoints* para representar un problema.

2. Identificar simetrías: Al modelar un problema se puede introducir simetría. El principal motivo de la importancia es que explotando la simetría se puede reducir la cantidad de búsqueda necesaria para resolver el problema.

3. Utilizar restricciones implícitas: Definidas como aquellas restricciones que están contenidas en las restricciones que definen el problema.

4. Utilizar restricciones globales: Estas restricciones permiten que una sola restricción sobre un número de variables reemplace un conjunto de restricciones.

5. Utilizar variables auxiliares: Son variables que se introducen en un modelo, debido a la dificultad de expresar las restricciones en términos de las variables existentes o a expresar las restricciones de una forma que se propague mejor.

Para aplicar el estudio realizado sobre las características que intervienen en un modelado de problemas como un CSP, se utilizaron los siguientes problemas:

1. N-Reinas

2. Cuadrados Mágicos

3. SEND + MORE = MONEY

Se han modelado los problemas utilizando una combinación de las características identificadas. En el siguiente capítulo se describirán los modelos definidos.

4.2 Resultados Esperados

De acuerdo al estudio de las características de modelado que se ha realizado, se identifican algunos resultados que se espera obtener producto de la implementación de los modelos definidos. Estos son:

- Al introducir restricciones para explotar simetrías, la cantidad de soluciones disminuye en una proporción igual a la cantidad de simetrías existentes. Es decir, una proporción igual a 8.
- Al introducir restricciones para romper simetrías, el tiempo de ejecución disminuye en comparación con el problema sin restricciones para romper simetrías.
- Al reemplazar las restricciones individuales por restricciones globales, la propagación es mayor, por lo que el tiempo de búsqueda disminuye.
- Al utilizar variables auxiliares, el tiempo de búsqueda disminuye.
- Al utilizar restricciones implícitas, el tiempo de ejecución disminuye.

CAPÍTULO 5

Formulación de Modelos

5.1 Cuadrados Mágicos

5.1.1 Descripción

Un **Cuadrado Mágico** es la disposición de una serie de números enteros en un cuadrado o matriz de tal forma que la suma de los números por cada columna, fila y diagonal sea la misma, valor conocido como la *constante mágica*. Usualmente los números empleados para rellenar las casillas son consecutivos, de 1 a n^2, siendo n el número de columnas y filas del cuadrado mágico (por ejemplo, el valor para n en la figura 5.1 es 3).

Se debe considerar que la constante mágica es igual a:

$$K = \frac{n(n^2+1)}{2} \tag{5.1}$$

4	9	2
3	5	7
8	1	6

Figura 5.1: Cuadrado Mágico, n= 3.

5.1.2 Formulación Modelo 1

Se consideran para este modelo las siguientes características:

- Se utilizan "restricciones globales".
- No se utilizan restricciones para remover simetrías.

Para representar este problema como un CSP se utilizan las variables que representen las posiciones de cada número en el cuadrado. Los posibles valores para cada variable son los números entre$\{1, \ldots, n^2\}$.
El valor para la constante mágica k es: $\frac{n(n^2+1)}{2} = k$.

1. Variables: x_{ij} =Posición del número en la fila i y columna j. Con i=$\{1, \ldots, n\}$ y j=$\{1, \ldots, n\}$.
2. Dominio: $\{1, \ldots, n^2\}$.
3. Restricciones:

$\sum_{j=1}^{n} x_{ij} = k$	$\forall i = 1, \ldots, n$	Restricción en las Filas
$\sum_{i=1}^{n} x_{ij} = k$	$\forall j = 1, \ldots, n$	Restricción en las Columnas
$\sum_{i=1}^{n} x_{ii} = k$		Restricción en la Diagonal positiva
$\sum_{i=1,j=i-1}^{n} x_{i(n-j)} = k$		Restricción en la Diagonal negativa
$alldifferent(x_{ij})$	$\forall i, j = 1, \ldots, n$	Restricción para todos los valores diferentes

5.1.3 Formulación Modelo 2

Se consideran para este modelo las siguientes características:

- Se utilizan "restricciones globales".
- Se utilizan restricciones para remover simetrías.

Se utiliza la misma representación que en el modelo 1 (ver 5.1.2),considerando que cada variable representa la posición del número ubicado en la fila i y columna j. Además, los posibles valores para cada variable son los números entre $\{1, \ldots, n^2\}$.

El valor para la constante mágica k es: $\frac{n(n^2+1)}{2} = k$.

1. Variables: x_{ij} =Posición del número en la fila i y columna j. Con $i = 1, \ldots, n$ y $j = 1, \ldots, n$.
2. Dominio: $\{1, \ldots, n^2\}$.

3. Restricciones:

$\sum_{j=1}^{n} x_{ij} = k$	$\forall i = 1, \ldots, n$	Restricción en las Filas
$\sum_{i=1}^{n} x_{ij} = k$	$\forall j = 1, \ldots, n$	Restricción en las Columnas
$\sum_{i=1}^{n} x_{ii} = k$		Restricción en la Diagonal positiva
$\sum_{i=1,j=i-1}^{n} x_{i(n-j)} = k$		Restricción en la Diagonal negativa
$alldifferent(x_{ij})$	$\forall i, j = 1, \ldots, n$	Restricción para todos los valores diferentes
$x_{n1} < x_{1n}$		Restricción para romper simetrías
$x_{11} < x_{1n}$		Restricción para romper simetrías
$x_{11} < x_{nn}$		Restricción para romper simetrías
$x_{11} < x_{n1}$		Restricción para romper simetrías

5.1.4 Formulación Modelo 3

Se consideran para este modelo las siguientes características:

- No se utilizan "restricciones globales".
- Se utilizan restricciones para remover simetrías.

Se utiliza la misma representación que en el modelo anterior (ver 5.1.2), considerando que cada variable representa la posición del número ubicado en la fila i y columna j. Además, los posibles valores para cada variable son los números entre $\{1, \ldots, n^2\}$.

El valor para la constante mágica k es: $\frac{n(n^2+1)}{2} = k$.

1. Variables: x_{ij} = Posición del número en la fila i y columna j. Con $i = 1, \ldots, n$ y $j = 1, \ldots, n$.
2. Dominio: $\{1, \ldots, n^2\}$.
3. Restricciones:

$\sum_{j=1}^{n} x_{ij} = k$	$\forall i = 1, \ldots, n$	Restricción en las Filas
$\sum_{i=1}^{n} x_{ij} = k$	$\forall j = 1, \ldots, n$	Restricción en las Columnas
$\sum_{i=1}^{n} x_{ii} = k$		Restricción en la Diagonal positiva
$\sum_{i=1,j=i-1}^{n} x_{i(n-j)} = k$		Restricción en la Diagonal negativa
$x_{n1} \neq x_{1n}$		Restricción para romper simetrías
$x_{11} \neq x_{1n}$		Restricción para romper simetrías
$x_{11} \neq x_{nn}$		Restricción para romper simetrías
$x_{11} \neq x_{n1}$		Restricción para romper simetrías
$x_{ij} \neq x_{ji}$	$\forall i, j = 1, \ldots, n$	Restricción todos los valores diferentes

5.1.5 Formulación Modelo 4

Se consideran para este modelo las siguientes características:

- No se utilizan "restricciones globales".
- No se utilizan restricciones para remover simetrías.

Se utiliza la misma representación que en el modelo anterior, considerando que cada variable representa la posición del número ubicado en la fila i y columna j. Además, los posibles valores para cada variable son los números entre $\{1, \ldots, n^2\}$.

El valor para la constante mágica k es: $\frac{n(n^2+1)}{2} = k$.

1. Variables: x_{ij} = Posición del número en la fila i y columna j. Con $i = 1, \ldots, n$ y $j = 1, \ldots, n$.
2. Dominio: $\{1, \ldots, n^2\}$.
3. Restricciones:

$\sum_{j=1}^{n} x_{ij} = k$	$\forall i = 1, \ldots, n$	Restricción en las Filas
$\sum_{i=1}^{n} x_{ij} = k$	$\forall j = 1, \ldots, n$	Restricción en las Columnas
$\sum_{i=1}^{n} x_{ii} = k$		Restricción en la Diagonal positiva
$\sum_{i=1,j=i-1}^{n} x_{i(n-j)} = k$		Restricción en la Diagonal negativa
$x_{ij} \neq x_{ji}$	$\forall i, j = 1, \ldots, n$	Restricciones para todos los valores diferentes

5.2 SEND + MORE = MONEY

5.2.1 Descripción

El problema **SEND + MORE = MONEY** consiste en asignar a cada letra $\{s, e, n, d, m, o, r, y\}$ un dígito diferente del conjunto $\{0, ..., 9\}$ de modo que se satisfaga la expresión $send + more = money$.

5.2.2 Formulación Modelo 1

Se consideran para este modelo las siguientes características:

- Se utilizan "restricciones globales".
- No se utilizan "variables auxiliares".

1. Variables: $\{s, e, n, d, m, o, r, y\}$.

2. Dominio:$\{0, ..., 9\}$.

3. Restricciones:

   Restricción en las Filas $10^3(s+m) + 10^2(e+o) + 10(n+r) + d + e = 10^4 m + 10^3 o + 10^2 n + 10e + y$

   Restricción para todas las variables diferentes alldifferent(s,e,n,d,m,o,r,y)

5.2.3 Formulación Modelo 2

Otra forma de modelar el problema es utilizando variables auxiliares para descomponer la ecuación en una colección de pequeñas restricciones.
Se consideran para este modelo las siguientes características:

- Se utilizan "variables auxiliares".
- Sí se utilizan "restricciones globales".

1. Variables: $\{s, e, n, d, m, o, r, y, c_1, c_2, c_3\}$.

2. Dominio: Para

$$\{e, n, d, o, r, y\} = \{0, ..., 9\}$$
$$s = \{1, ..., 9\}$$
$$m = \{1\}$$
$$\{c_1, c_2, c_3\} = \{0, 1\}$$

3. Restricciones:

$$e + d = y + 10c_1$$
$$c_1 + n + r = e + 10c_2$$
$$c_2 + e + o = n + 10c_3$$
$$c_3 + s + m = 10m + o$$
$alldifferent(s, e, n, d, m, o, r, y)$ Restricción para todas las variables diferentes

5.2.4 Formulación Modelo 3

Otra forma de modelar el problema es utilizando variables auxiliares para descomponer la ecuación en una colección de pequeñas restricciones.
Se consideran para este modelo las siguientes características:

- Se utilizan "variables auxiliares".
- Se identifica "restricción implícita".

- Se utiliza "restricción global".

1. Variables: $\{s, e, n, d, m, o, r, y, c_1, c_2, c_3\}$.

2. Dominio: Para

$$\{e, n, d, o, r, y\} = \{0, ..., 9\}$$
$$s = \{1, ..., 9\}$$
$$m = \{1\}$$
$$\{c_1, c_2, c_3\} = \{0, 1\}$$

3. Restricciones:

$e + d = y + 10c_1$	
$c_1 + n + r = e + 10c_2$	
$c_2 + e + o = n + 10c_3$	
$c_3 + s + m = 10m + o$	
$c_1 - 9c_2 - 10c_3 + o = 0$	Restricción obtenida de Restricción 2 y 3
$alldifferent(s, e, n, d, m, o, r, y)$	Restricción para todas las variables diferentes

5.2.5 Formulación Modelo 4

Otra forma de modelar el problema es, además de introducir variables auxiliares para descomponer la ecuación en una colección de pequeñas restricciones, se genera una colección de restricciones individuales para establecer que cada una de las letras son distintas entre si.
Se consideran para este modelo las siguientes características:

- Se utilizan "variables auxiliares".
- No se utilizan "restricciones globales".

1. Variables: $\{s, e, n, d, m, o, r, y, c_1, c_2, c_3\}$.

2. Dominio: Para

$$\{e, n, d, o, r, y\} = \{0, ..., 9\}$$
$$s = \{1, ..., 9\}$$
$$m = \{1\}$$
$$\{c_1, c_2, c_3\} = \{0, 1\}$$

3. Restricciones:

$e + d = y + 10c_1$
$c_1 + n + r = e + 10c_2$
$c_2 + e + o = n + 10c_3$
$c_3 + s + m = 10m + o$
$c_1 - 9c_2 - 10c_3 + o = 0$ Restricción obtenida de Restricción 2 y 3
$s \neq e$ Restricción para todas las variables diferentes
$s \neq n$
$s \neq d$
$s \neq o$
$s \neq r$
$s \neq y$
$s \neq m$
$e \neq n$
$e \neq d$
$e \neq o$
$e \neq r$
$e \neq y$
$e \neq m$
$n \neq d$
$n \neq o$
$n \neq r$
$n \neq y$
$n \neq m$
$d \neq o$
$d \neq r$
$d \neq y$
$d \neq m$
$o \neq r$
$o \neq y$
$o \neq m$
$r \neq y$
$r \neq m$
$y \neq m$

5.3 N-reinas

5.3.1 Descripción

El problema de las ***n*-reinas** consiste en encontrar una distribución de *n* reinas en un tablero de ajedrez de *n* x *n* de modo tal, que éstas no se ataquen. Así, no pueden encontrarse dos reinas en la misma fila, columna o diagonal.
El Problema de las n-reinas es conocido usualmente como uno relacionado a un juego y también como un problema apropiado para probar algoritmos nuevos. Sin embargo, tiene otras aplicaciones ya que se le considera como un modelo de máxima cobertura. Esto es, una solución al problema de las n-reinas garantiza que cada objeto puede ser accesado desde cualquiera de sus ocho direcciones vecinas (dos verticales, dos horizontales y cuatro diagonales) sin que tenga conflictos con otros objetos.
El problema de las *n* reinas cuenta con lo siguiente:

- Dos reinas no pueden estar en la misma Fila.
- Dos reinas no pueden estar en la misma Columna.
- Dos reinas no pueden estar en la misma Diagonal.

5.3.2 Formulación Modelo 1

Se consideran para este modelo, las siguientes características:

- No se utilizan "restricciones globales".
- Se utiliza un "viewpoint" para modelar el problema.

1. Variables: Este modelo utiliza una variable binaria x_{ij} para representar la existencia de una reina en el casillero (i, j). Con $i = 1, \ldots, n$ y $j = 1, \ldots, n$.
2. Dominio: Si existe una reina en el casillero ubicado en la posición ij, $x_{ij} = 1$ y si no, vale 0.
3. Restricciones: Las restricciones abarcan las filas, las columnas,diagonales positivas y diagonales negativas.

    - Restricción para filas: Se impide que existan dos reinas en la misma fila. Esto se verifica sumando todos los valores de las casillas x_{ij} para una fila determinada. Así, dicha suma debería ser igual a 1 para todas las filas ya que debería haber un solo 1 en cada fila. Esta restricción se puede resumir de la siguiente forma:

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i \in N \tag{5.2}$$

- Restricción para columnas: Se impide que existan dos reinas en la misma columna. Esto se verifica sumando todos los valores de las casillas x_{ij} para una columna determinada. Así, dicha suma debería ser igual a 1 para todas las columnas ya que debería haber un sólo 1 en cada columna. Esta restricción se puede resumir de la siguiente forma:

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j \in N \tag{5.3}$$

- Restricción en diagonales: En las diagonales se debe revisar la definición de los valores de recorrido de la variable. En las diagonales, tanto positivas como negativas, debe haber a lo más una reina. Esto se verifica sumando los valores de las casillas que forman las distintas diagonales.

  Si se analiza la formación de las diagonales positivas, se ve que éstas estarán definidas por la suma $k = i + j$. Así, todos los pares (i, j) que tengan el mismo valor para k, estarán en la misma diagonal positiva.
  De esta forma, se puede definir la restricción para las diagonales positivas de la siguiente forma:

$$\underbrace{\sum_{i=1}^{n}\sum_{j=1}^{n}}_{i+j=k} x_{ij} \leq 1 \qquad \forall k = 2, \ldots, 2n \quad o \quad k = 3, \ldots, 2n-1 \tag{5.4}$$

  De igual forma, se analizan las diagonales negativas.Se puede verificar que éstas están definidas por la resta $k = i - j$ varía en el rango $\{2, \ldots, 2n\}$ o $\{3, \ldots, 2n-1\}$ dependiendo si se consideran las casillas de las esquinas.
  De esta forma, se puede definir la restricción para las diagonales negativas de la siguiente forma:

$$\underbrace{\sum_{i=1}^{n}\sum_{j=1}^{n}}_{i-j=k} x_{ij} \leq 1 \qquad \forall k = -n, \ldots, n \quad o \quad k = -n+1, \ldots, n-1 \tag{5.5}$$

Formalizando el modelo, se tiene lo siguiente, tomando en cuenta la cantidad de reinas igual a n:

1. Variables:

$$x_{ij} = \begin{cases} 1 & \text{si existe una reina en la posición (i,j)} \\ 0 & \text{si no existe} \end{cases}$$

2. Dominio: $\{0, 1\}$.

3. Restricciones:

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i = 1, \ldots, n \qquad \text{Restricción en las Filas}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j = 1, \ldots, n \qquad \text{Restricción en las columnas}$$

$$\underbrace{\sum_{i=1}^{n}\sum_{j=1}^{n}}_{i+j=k} x_{ij} \leq 1 \quad \forall k = 3, \ldots, 2n-1 \qquad \text{Restricción en las diagonales positivas}$$

$$\underbrace{\sum_{i=1}^{n}\sum_{j=1}^{n}}_{i-j=k} x_{ij} \leq 1 \quad \forall k = -n+1, \ldots, n-1 \qquad \text{Restricción en las diagonales negativas}$$

5.3.3 Formulación Modelo 2

Se consideran para este modelo, las siguientes características:

- No se utilizan "restricciones globales".
- No se utilizan restricciones para remover simetrías.
- Viewpoint 2.

1. Variables: Otra forma de modelar el mismo problema es utilizando la variable x_i para representar la posición de la reina en la fila i.

2. Dominio: En este caso, el dominio de x_i sería $\{1, \ldots, n\}$. A diferencia del modelo anterior (ver 5.3.2).

3. Restricciones: Estas abarcan las columnas y todas las diagonales, sin distinguir entre positivas y negativas. Las filas se verifican por defecto, al tener que asignar un valor a la variable. Es decir, por la forma del modelo, no puede haber dos reinas en la misma fila, por lo que esa restricción se omite.
   La restricción de las columnas impide que existan dos reinas en la misma columna. Esto se verifica observando los valores de las variables. Si estos son distintos, entonces se cumple con la restricción, si son iguales, no. Esta restricción se puede resumir de la siguiente forma:

$$x_i \neq x_j \qquad \forall i \neq j \quad coniyj \in N \tag{5.6}$$

   Igualmente se tiene:

$$x_i - x_j \neq 0 \qquad \forall i \neq j \quad coniyj \in N \tag{5.7}$$

   Al igual que en el modelo anterior (ver 5.3.2), en las diagonales, tanto positivas como negativas, debe haber a lo más una reina. Dada la definición del modelo, el valor de la

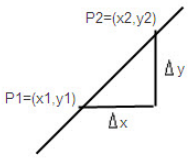

Figura 5.2: Cálculo de la pendiente de una recta.

variable indica la columna y el índice de la variable, la fila. Utilizando esos valores, se puede calcular la pendiente de una recta que pase por los dos casilleros, x_i y x_j que se está verificando. Sabiendo que todas las diagonales positivas forman un ángulo de 45^0 respecto a la base del tablero y que todas las diagonales negativas forman un ángulo de -45^0 respecto a la misma base, si se calcula la pendiente y el ángulo resultante es aproximadamente 45^0, entonces los dos casilleros están en la misma diagonal.

En general, para poder encontrar la pendiente de una recta, basta con conocer dos puntos de esa recta y calcular la diferencia entre las coordenadas de ambos puntos. Por ejemplo, si se quiere calcular la pendiente de la recta de la figura 5.2, se calcula la variación de x, $\Delta x = x_2 - x_1$ y la variación de y, $\Delta y = y_2 - y_1$. Después, se calcula $\frac{\Delta x}{\Delta y}$ y con este resultado se calcula la tangente. Como la tangente de $45 = 1$, eso significa que el Δx y el Δy deben ser iguales. Para este caso, como no se quiere que las reinas estén en la misma diagonal, esas diferencias deben ser distintas. De esta forma, se puede definir la restricción para las diagonales de la siguiente forma:

$$|x_i - x_j| \neq |i - j| \qquad \forall i \neq j \quad \text{con i y j } \in N \tag{5.8}$$

Formalizando el modelo, se tiene lo siguiente, tomando en cuenta la cantidad de reinas igual a n:

1. Variables: x_i = posición de la reina en la fila i, con i=$\{1, \ldots, n\}$.
2. Dominio: $\{1, \ldots, n\}$
3. Restricciones:

$x_i \neq x_j$	$\forall i \neq j$	con i y j $\in \{1,\ldots,n\}$	Restricción en las Columnas
$\lvert x_i - x_j\rvert \neq \lvert i - j\rvert$	$\forall i \neq j$	con i y j $\in \{1,\ldots,n\}$	Restricción en las Diagonales

5.3.4 Formulación Modelo 3

Se considera para este modelo el modelo entregado anteriormente (ver 5.3.3), para el cual se toman en cuenta las siguientes características:

- Se utilizan "restricciones globales".
- No se utilizan restricciones para remover simetrías.

La formulación del modelo es la siguiente:

1. Variables: $x_i =$ posición de la reina en la fila i, con i=$\{1, \ldots, n\}$.
2. Dominio: $\{1, \ldots, n\}$
3. Restricciones:

$alldifferent(x_1, \ldots, x_n)$	Restricción en las Columnas
$alldifferent(x_1 - 1; x_2 - 2; \ldots; x_n - n)$	Restricción en Diagonal
$alldifferent(x_1 + 1; x_2 + 2; \ldots; x_n + n)$	Restricción en Diagonal

5.3.5 Formulación Modelo 4

Se considera para este modelo el modelo entregado anteriormente (ver 5.3.3), para el cual se toman en cuenta las siguientes características:

- Se utilizan restricciones para remover simetrías.
- No se utilizan "restricciones globales".

La formulación del modelo es la siguiente:

1. Variables: $x_i =$ posición de la reina en la fila i, con i=$\{1, \ldots, n\}$.
2. Dominio: $\{1, \ldots, n\}$
3. Restricciones:

$x_i \neq x_j$	$\forall i \neq j$ con i y j $\in \{1,\ldots,n\}$	Restricción en las Columnas
$\lvert x_i - x_j \rvert \neq \lvert i - j \rvert$	$\forall i \neq j$ con i y j $\in \{1,\ldots,n\}$	Restricción en las Diagonales
$r_{90}x_i = r_{180}x_i =$ $r_{270}x_i = rd_1x_i = rd_2x_i$	con i $\in \{1,\ldots,n\}$	Restricción de simetrías

CAPÍTULO 6

Resultados Obtenidos y Análisis

Los modelos definidos han sido implementados en el solver Eclipse [1], con el fin de validar los resultados esperados que se dieron a conocer en el capítulo anterior.

6.1 Cuadrados Mágicos

El resultado de la ejecución de cada uno de los problemas implementados para *Cuadrados Mágicos* se muestra en la Tabla 6.1.

Tabla 6.1: Resultados obtenidos para *Cuadrados Mágicos*

Modelo	Características	Número de Soluciones	Tiempo Total (seg.)	Back. Totales	Nodos Visitados
1	Se utilizan restricciones globales No se utilizan restricciones para remover simetrías	7.040	45,69	71720	226032
2	No se utilizan restricciones globales Se utilizan restricciones para remover simetrías	880	5,7	6610	22907
3	Se utilizan restricciones globales Se utilizan restricciones para remover simetrías	880	4,19	6141	22197
4	No se utilizan restricciones globales No se utilizan restricciones para remover simetrías	7040	62,8750	78384	241978

[1] http://www.eclipseclp.org/

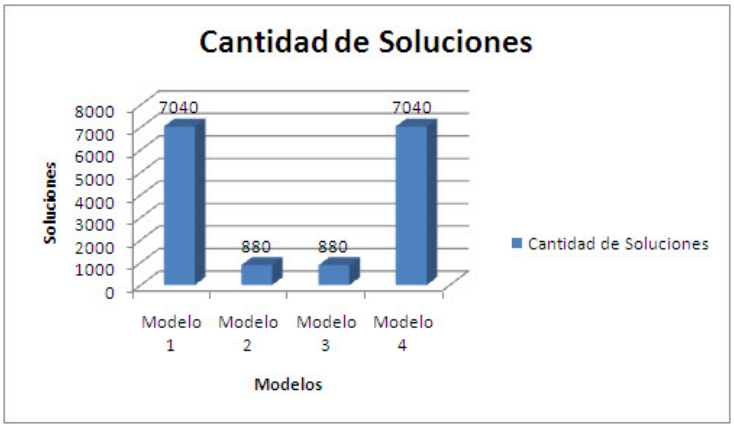

Figura 6.1: Cantidad de soluciones de cada modelo definido para Cuadrados Mágicos.

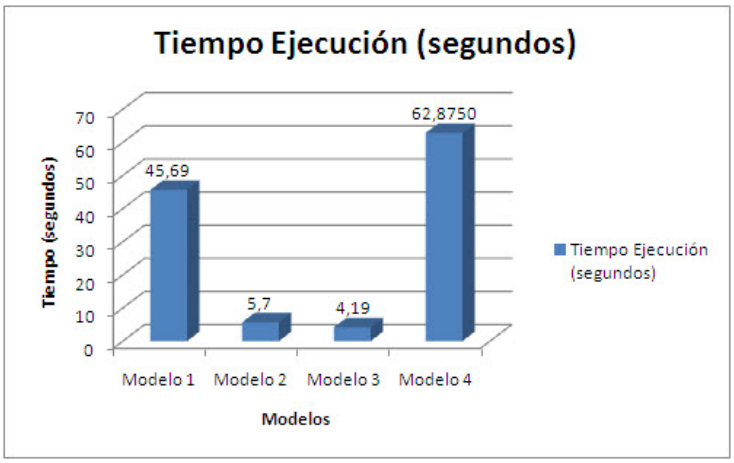

Figura 6.2: Tiempo de ejecución de cada modelo definido para Cuadrados Mágicos.

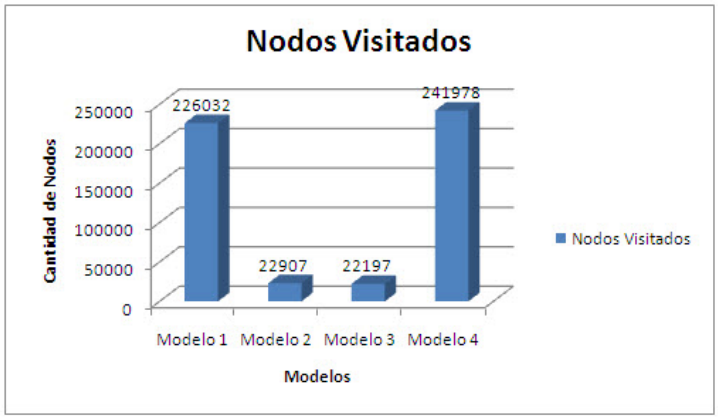

Figura 6.3: Nodos visitados en cada modelo definido para Cuadrados Mágicos.

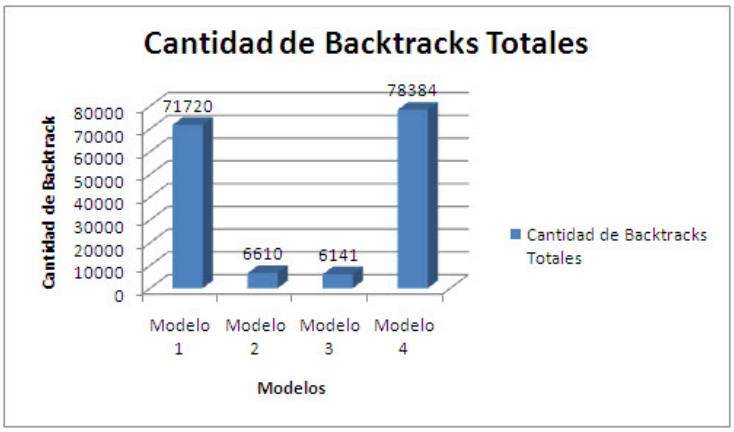

Figura 6.4: Backtrack totales en cada modelo definido para Cuadrados Mágicos.

6.1.1 Análisis de Resultados

Se observa que se cumple la regla asociada a cuando se introducen restricciones para romper simetrías durante la búsqueda. Esto es:

- Sea NSCRS = Número de soluciones con restricciones para romper simetrías
- Sea NSSRS= Número de soluciones sin restricciones para romper simetrías
- NSCRS = NSSRS / 8

Es decir, al introducir restricciones para romper simetrías se disminuye en 8 veces la cantidad de soluciones, lo que se relaciona con el número de simetrías que se identifican en una matriz.

El tiempo total de ejecución varía de acuerdo a si se utilizan restricciones para remover simetrías o no, variando de menor forma si se utilizan restricciones globales. Esto es, se puede observar que el tiempo utilizado por el modelo 3 es aproximadamente 15 veces menor que el tiempo utilizado por el modelo 4. Considerando que en el modelo 3 se utilizan restricciones para romper simetrías y se utilizan restricciones globales y en el caso del modelo 4 no se utilizan ni restricciones globales ni restricciones para remover simetrías.

El número de nodos visitados es menor cuando se utilizan restricciones para remover simetrías (modelos 2 y 3). Si a esto se agregan restricciones globales, el tiempo de ejecución y el número de nodos visitados será menor.

6.2 SEND + MORE = MONEY

El resultado de la ejecución de cada uno de los problemas implementados para *SEND + MORE = MONEY* se muestra en la Tabla 6.2.

Tabla 6.2: Resultados obtenidos para $SEND + MORE = MONEY$

Modelo	Características	Soluciones	Tiempo Total (seg.)	Back. Totales	Nodos Visitados
1	Se utilizan restricciones globales Se utilizan variables auxiliares	1	0,001	0	8
2	Se utilizan restricciones globales No se utilizan variables auxiliares	1	0,01	0	8
3	No se utilizan restricciones globales Se utilizan variables auxiliares	1	0,02	0	8
4	No se utilizan restricciones globales Se utilizan variables auxiliares se identifica restricción implícita	1	0,03	0	8

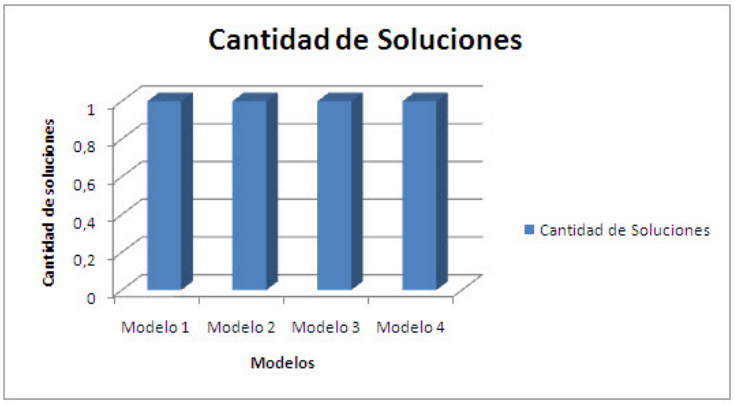

Figura 6.5: Cantidad de soluciones de cada modelo definido para SEND + MORE + MONEY.

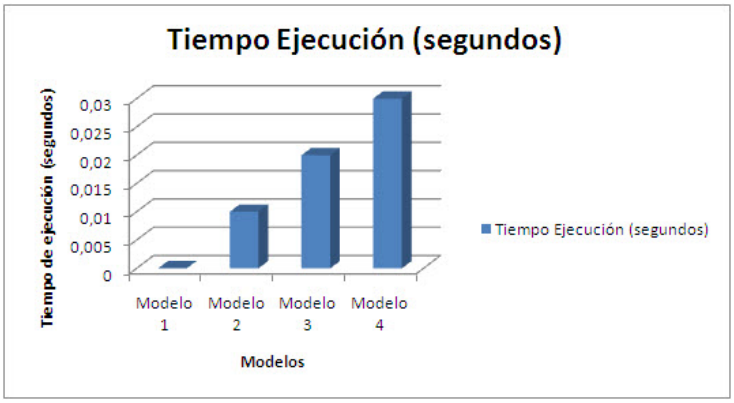

Figura 6.6: Tiempo de ejecución de cada modelo definido para SEND + MORE + MONEY.

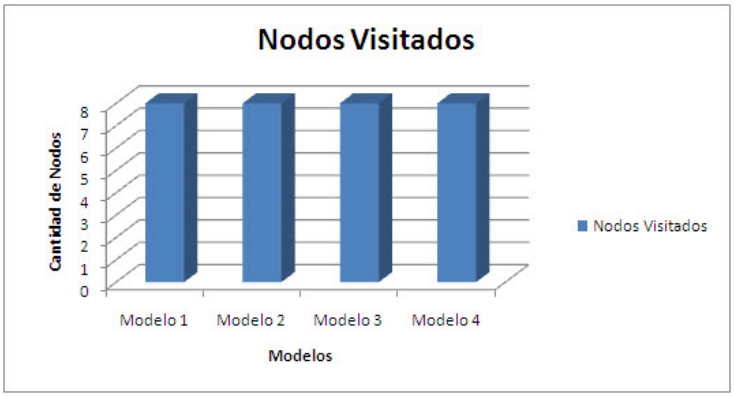

Figura 6.7: Nodos visitados en cada modelo definido para SEND + MORE + MONEY.

6.2.1 Análisis de Resultados

Se observa que el tiempo de ejecución, si bien es pequeño para este problema, se reduce al utilizar variables auxiliares en comparación al no utilizar variables auxiliares.
El menor tiempo de ejecución se obtiene al utilizar restricciones globales y variables auxiliares. La cantidad de soluciones para este tipo de problemas no se reduce, pues es única. Siendo útiles las características de modelado para reducir los tiempos de ejecución.

6.3 N-Reinas

El resultado de la ejecución de cada uno de los problemas implementados para *N-Reinas* se muestra en la Tabla 6.3. Se utiliza el valor de n igual a 8 para realizar la comparación de la

implementación de todos los modelos definidos.

Tabla 6.3: Resultados obtenidos para *N-Reinas*. N = 8.

Modelo	Soluciones	Características	Tiempo Total	Back. Totales	Nodos Visitados
1	Se utiliza un Viewpoint 1 No se utilizan restricciones para remover simetrías No se utilizan restricciones globales	92	0,125	51	206
2	Se utiliza Viewpoint 2 No se utilizan restricciones globales No se utilizan restricciones para remover simetrías	92	0,11	178	577
3	Se utilizan restricciones globales No se utilizan restricciones para remover simetrías	92	0,09	167	561
4	No se utilizan restricciones globales Se utilizan restricciones para remover simetrías	12	0,03125	34	103

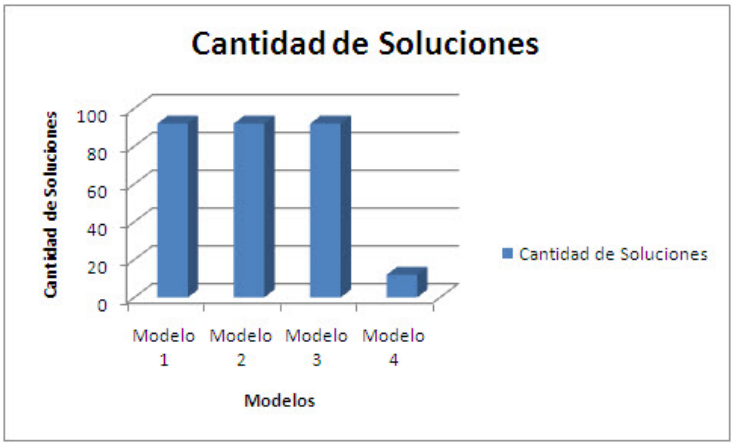

Figura 6.8: Cantidad de soluciones de cada modelo definido para n-reinas. N = 8.

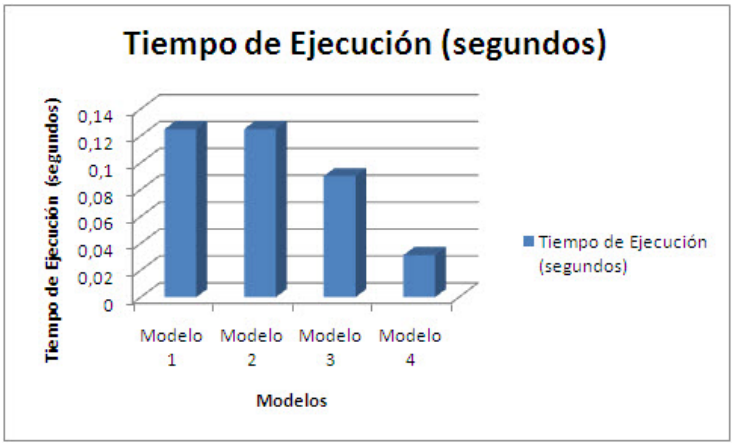

Figura 6.9: Tiempo de ejecución de cada modelo definido para n-reinas. $N = 8$.

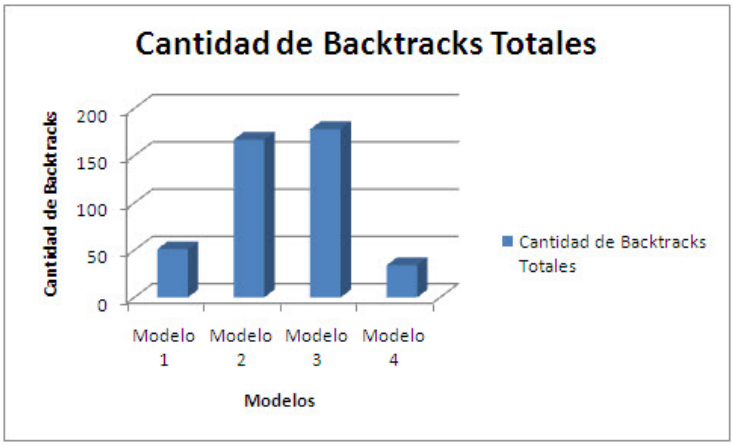

Figura 6.10: Nodos visitados en cada modelo definido para n-reinas. $N = 8$.

6.3.1 Análisis de Resultados

Para el problema de las n-reinas, se realizaron ejecuciones para distintos valores de n, obteniendo para valores de n mayores o iguales a 8 que, el número de soluciones sin restricciones para romper simetrías se encuentra directamente relacionado con el número de soluciones sin restricciones para romper simetrías en una proporción aproximada de 8, de acuerdo a lo que se observa en la tabla 6.4.

Tabla 6.4: Relación entre soluciones para modelos que utilizan restricciones para remover simetrías y modelos que no utilizan restricciones para remover simetrías.

N	NSSRS	NSCRS	Proporción
8	92	12	7,666
9	352	46	7,652
10	724	92	7,869
11	2680	341	7,859
12	14200	1787	7,946
13	73712	9233	7,983
14	365596	45752	7,990
15	2279184	285053	7,996

Se observa, además, que al introducir restricciones para remover simetrías, el número de nodos visitados y la cantidad de backtrack totales son menores que los resultados obtenidos para los otros modelos que fueron implementados.

Al utilizar restricciones globales, se observa que el tiempo de ejecución disminuye. Se realizó la prueba con otros valores de n mayores a 8 obteniéndose diferencias pequeñas entre ambos tiempos (tiempo de ejecución para problema sin restricciones globales y tiempo de ejecución para problema con restricciones globales).

Con relación a la cantidad de backtrack y número de nodos visitados, la cantidad disminuye al ejecutar el problema con restricciones globales en comparación con el resultado al ejecutar el problema sin restricciones globales.

CAPÍTULO 7

Conclusiones

Se han descrito los conceptos que se consideran importantes y decisivos al momento de decidir por un modelo, designando a algunos como las características de modelado a analizar. Para ésto se han formulado algunos problemas clásicos existentes en la literatura, utilizando las características de modelado identificadas: Identificación de simetrías, uso de restricciones globales, utilización de diversos Viewpoints, presencia de restricciones implícitas.

Respecto a las formulaciones de modelado realizadas se puede concluir que de acuerdo a lo estudiado se comprueba que la **simetría** juega un papel importante al momento de modelar un problema. Se puede observar esto en los modelos del problema de los cuadrados mágicos. Si se comparan las soluciones de los modelos 1 y 3 respectivamente, se puede ver que el modelo que no explota simetrías, el modelo 1, posee 880 soluciones simétricas, que corresponde a 8 veces las soluciones del modelo 3. Así, al explotar las simetrías, se reduce en 8 veces la cantidad de soluciones, debido a que se reducen las 8 diagonales presentes en una matriz. También se puede ver este análisis para el caso del modelo 4 de n-reinas, donde para valores de n mayores a 8 se obtiene que la cantidad de soluciones para el modelos sin restricciones para remover simetrías es aproximadamente 8 veces la cantidad de soluciones para el modelo con restricciones para remover simetrías.

El principal objetivo de la detección de las simetrías en la programación con restricciones sería usarla para mejorar el rendimiento de la búsqueda de soluciones a un problema. La ruptura simetrías se considera fundamental para la resolución práctica de algunos problemas, donde existen simetrías que causan una búsqueda redundante. Sin embargo, se puede concluir que son difíciles de utilizar, ya sea debido a un rendimiento impredecible como por las habilidades requeridas para realizar su implementación.

No se puede decir que exista un método fácil para trabajar con las simetrías, pero al lograr su implementación se pueden obtener resultados beneficiosos para mejorar el rendimiento de cada problema.

Estudiar los resultados que se obtienen al ejecutar cada uno de los problemas que involucran restricciones para remover simetrías, resulta interesante para concluir una mayor cantidad de lecciones aprendidas para un futuro trabajo.

En relación con la utilización de **restricciones globales**, se observa que los tiempos de ejecución disminuyen. Se considera que al trabajar en problemas más complejos con restricciones globales se mejorará en forma sustancial la eficiencia y solución de los problemas.

Al utilizar distintos **viewpoint** es necesario considerar que se deben utilizar otras características de modelado para poder obtener resultados más eficientes en comparación con otros modelos. Por lo tanto, se concluye que el uso de distintos viewpoint no es por sí sólo una forma de identificar reducciones en la búsqueda de soluciones a los Problemas de Satisfacción de Restricciones.

El uso de **restricciones implícitas** en los problemas utilizados no entregaron reducción en la búsqueda de forma significativa, en comparación con otros modelos. Debiese utilizarse en otros problemas asociados a ecuaciones lineales, para poder comparar y ver si existen beneficios importantes asociados a su utilización.

Para estudios futuros se deja abierta la idea para trabajar con técnicas de búsqueda distintas a backtracking y utilizar problemas con complejidades mayores para así observar desde otros puntos de vista el efecto positivo que tiene el utilizar las características de modelado que se estudiaron en este trabajo de título.

Bibliografía

[1] Federico Barber and Miguel A. Salido. Introducción a la programación de restricciones. *Inteligencia Artificial. Revista Iberoamericana de IA*, pages 13–33, 2003.

[2] Belaid Benhamou. Study of symmetry in constraint satisfaction problems. 1994.

[3] James E. Borrett and Edward P. K. Tsang. A context for constraint satisfaction problem formulation selection. *Constraints*, pages 299–327, 2001.

[4] David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, pages 115–137, 2006.

[5] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. pages 290–295, 1988.

[6] Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. pages 599–603, 2000.

[7] Solomon W. Golomb and L. Baumert. Backtrack programming. *Journal of the ACM*, pages 516–524, 1965.

[8] Philippe Jégou. On the consistency of general constraint-satisfaction problems. pages 114–119, 1993.

[9] Vipin Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, page 32.

[10] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, pages 99–118, 1977.

[11] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

[12] Pedro Meseguer and Carme Torras. Exploiting symmetries within constraint satisfaction search. *Artif. Intell.*, pages 133–163, 2001.

[13] B.A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, pages 16–23, 1990.

[14] J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. pages 350–360, 1993.

[15] Jean-Francois Puget. Constraint programming next challenge: Simplicity of use. pages 5–8, 2004.

[16] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. 2006.

[17] Barbara M. Smith. Symmetry and search in a network design problem. pages 336–350, 2005.

[18] W. van Hoeve. The alldifferent constraint: A survey, 2001.

ANEXOS

A. CÓDIGO FUENTE CUADRADOS MÁGICOS

```
%Archivo: Modelos.pl

:-lib(ic).
:- lib(lists).
magic(N) :-
init_backtrackst,
init_nodesv,
statistics(times, [T0|_]),
% modelo1(N,Vars,Square), % no remueve simetrias, posee restricciones globales
% modelo2(N,Vars,Square), % remueve simetrias, no posee restricciones globales
% modelo3(N,Vars,Square), % con simetrias, con variables globales
% modelo4(N,Vars,Square), % no remueve simetrias, no posee restricciones globales
labs(Vars,B,ND),
get_nodesv(ND),
statistics(times, [T1|_]),
TimeFinal is T1-T0,  % Calculo el tiempo total
write(' & Tiempo Total:& '),write(TimeFinal),
write(' & Bactrack totales:& '),write(B),
printf(" & Nodos visitados: & %d & %n ", [ND]).

%MODELO 1
modelo1(N,Vars,Square) :-
       NN is N*N,                          % tamaño del cuadrado
       Sum is N*(NN+1)//2,             % constante mágica
       printf("Sum = %d%n", [Sum]),
       dim(Square, [N,N]),             % variables
       Square[1..N,1..N] :: 1..NN,           % dominio
       Rows is Square[1..N,1..N],
       flatten(Rows, Vars),
       alldifferent(Vars),             % restricción de todas diferentes
       for(I,1,N),
       foreach(U,UpDiag),
       foreach(D,DownDiag),
       param(N,Square,Sum)
       do
       Sum #= sum(Square[I,1..N]),           % suma of row I
       Sum #= sum(Square[1..N,I]),           % suma of column I
       U is Square[I,I],
       D is Square[I,N+1-I]
       ),
```

```
       Sum #= sum(UpDiag),              % suma de diagonales
       Sum #= sum(DownDiag).

%MODELO 2
modelo2(N,Vars,Square) :-
       NN is N*N,                              % cantidad de números
       Sum is N*(NN+1)//2,              % suma mágica
       printf("Número  mágico = %d%n", [Sum]),
       dim(Square, [N,N]),              % Variables
       Square[1..N,1..N] :: 1..NN,             % Dominio
       Rows is Square[1..N,1..N],
       flatten(Rows, Vars),

Square[1,1] #\= Square[1,2],                        Square[1,3] #\= Square[1,4],
       Square[1,1] #\= Square[1,3],                 Square[1,3] #\= Square[2,1],
       Square[1,1] #\= Square[1,4],                 Square[1,3] #\= Square[2,2],
       Square[1,1] #\= Square[2,1],                 Square[1,3] #\= Square[2,3],
       Square[1,1] #\= Square[2,2],                 Square[1,3] #\= Square[2,4],
       Square[1,1] #\= Square[2,3],                 Square[1,3] #\= Square[3,1],
       Square[1,1] #\= Square[2,4],                 Square[1,3] #\= Square[3,2],
       Square[1,1] #\= Square[3,1],                 Square[1,3] #\= Square[3,3],
       Square[1,1] #\= Square[3,2],                 Square[1,3] #\= Square[3,4],
       Square[1,1] #\= Square[3,3],                 Square[1,3] #\= Square[4,1],
       Square[1,1] #\= Square[3,4],                 Square[1,3] #\= Square[4,2],
       Square[1,1] #\= Square[4,1],                 Square[1,3] #\= Square[4,3],
       Square[1,1] #\= Square[4,2],                 Square[1,3] #\= Square[4,4],
       Square[1,1] #\= Square[4,3],                 Square[1,4] #\= Square[2,1],
       Square[1,1] #\= Square[4,4],                 Square[1,4] #\= Square[2,2],
       Square[1,2] #\= Square[1,3],                 Square[1,4] #\= Square[2,3],
       Square[1,2] #\= Square[1,4],                 Square[1,4] #\= Square[2,4],
       Square[1,2] #\= Square[2,1],                 Square[1,4] #\= Square[2,2],
       Square[1,2] #\= Square[2,2],                 Square[1,4] #\= Square[2,3],
       Square[1,2] #\= Square[2,3],                 Square[1,4] #\= Square[2,4],
       Square[1,2] #\= Square[2,4],                 Square[1,4] #\= Square[3,1],
       Square[1,2] #\= Square[3,1],                 Square[1,4] #\= Square[3,2],
       Square[1,2] #\= Square[3,2],                 Square[1,4] #\= Square[3,3],
       Square[1,2] #\= Square[3,3],                 Square[1,4]  #\= Square[3,4],
       Square[1,2] #\= Square[3,4],                 Square[1,4]  #\= Square[4,1],
       Square[1,2] #\= Square[4,1],                 Square[1,4]  #\= Square[4,2],
       Square[1,2] #\= Square[4,2],                 Square[1,4]  #\= Square[4,3],
       Square[1,2] #\= Square[4,3],                 Square[1,4]  #\= Square[4,4],
       Square[1,2] #\= Square[4,4],                 Square[2,1] #\= Square[2,2],
```

```
Square[2,1] #\= Square[2,3],
Square[2,1] #\= Square[2,4],
Square[2,1] #\= Square[3,1],
Square[2,1] #\= Square[3,2],
Square[2,1] #\= Square[3,3],
Square[2,1] #\= Square[3,4],
Square[2,1] #\= Square[4,1],
Square[2,1] #\= Square[4,2],
Square[2,1] #\= Square[4,3],
Square[2,1] #\= Square[4,4],
Square[2,2] #\= Square[2,3],
Square[2,2] #\= Square[2,4],
Square[2,2] #\= Square[3,1],
Square[2,2] #\= Square[3,2],
Square[2,2] #\= Square[3,3],
Square[2,2] #\= Square[3,4],
Square[2,2] #\= Square[4,1],
Square[2,2] #\= Square[4,2],
Square[2,2] #\= Square[4,3],
Square[2,2] #\= Square[4,4],
Square[2,3] #\= Square[2,4],
Square[2,3] #\= Square[3,1],
Square[2,3] #\= Square[3,2],
Square[2,3] #\= Square[3,3],
Square[2,3] #\= Square[3,4],
Square[2,3] #\= Square[4,1],
Square[2,3] #\= Square[4,2],
Square[2,3] #\= Square[4,3],
Square[2,3] #\= Square[4,4],
Square[2,4] #\= Square[3,1],
Square[2,4] #\= Square[3,2],
Square[2,4] #\= Square[3,3],
Square[2,4] #\= Square[3,4],
Square[2,4] #\= Square[4,1],
Square[2,4] #\= Square[4,2],
Square[2,4] #\= Square[4,3],
Square[2,4] #\= Square[4,4],
Square[3,1] #\= Square[3,2],
Square[3,2] #\= Square[3,3],
Square[3,2] #\= Square[3,4],
Square[3,2] #\= Square[4,1],
Square[3,2] #\= Square[4,2],
Square[3,2] #\= Square[4,3],
Square[3,2] #\= Square[4,4],
Square[3,3] #\= Square[3,4],
Square[3,3] #\= Square[4,1],
Square[3,3] #\= Square[4,2],
Square[3,3] #\= Square[4,3],
Square[3,3] #\= Square[4,4],
Square[3,4] #\= Square[4,1],
Square[3,4] #\= Square[4,2],
Square[3,4] #\= Square[4,3],
Square[3,4] #\= Square[4,4],
Square[4,1] #\= Square[4,2],
Square[4,1] #\= Square[4,3],
Square[4,1] #\= Square[4,4],
Square[4,2] #\= Square[4,3],
Square[4,2] #\= Square[4,4],
Square[4,3] #\= Square[4,4],
(
for(I,1,N),
foreach(U,UpDiag),
foreach(D,DownDiag),
param(N,Square,Sum)
do
      Sum #= sum(Square[I,1..N]),      % suma de filas
      Sum #= sum(Square[1..N,I]),      % suma de columnas
      U is Square[I,I],
      D is Square[I,N+1-I]
),
Sum #= sum(UpDiag),              % suma de diagonales
```

```
        Sum #= sum(DownDiag),
        Square[1,1] #< Square[1,N],            % removiendo simetrías
        Square[1,1] #< Square[N,N],
        Square[1,1] #< Square[N,1],
        Square[1,N] #< Square[N,1].

%MODELO 3
modelo3(N,Vars,Square) :-
        NN is N*N,                             % tamaño del cuadrado
        Sum is N*(NN+1)//2,             % constante mágica
        printf("Sum = %d%n", [Sum]),
        dim(Square, [N,N]),             % variables
        Square[1..N,1..N] :: 1..NN,            % dominio
        Rows is Square[1..N,1..N],
        flatten(Rows, Vars),
        alldifferent(Vars),             % restricción de todas diferentes
        nl,write('Las variables son: '),write(Vars),
        (
        for(I,1,N),
        foreach(U,UpDiag),
        foreach(D,DownDiag),
        param(N,Square,Sum)
        do
            Sum #= sum(Square[I,1..N]),  % suma de filas
            Sum #= sum(Square[1..N,I]),  % suma de columnas
            U is Square[I,I],
            D is Square[I,N+1-I]
        ),
        Sum #= sum(UpDiag),             % suma de diagonales
        Sum #= sum(DownDiag),
        Square[1,1] #< Square[1,N],            % removiendo simetrías
        Square[1,1] #< Square[N,N],
        Square[1,1] #< Square[N,1],
        Square[1,N] #< Square[N,1].


%MODELO 4
modelo4(N,Vars,Square) :-
        NN is N*N,                             % tamaño del cuadrado
        Sum is N*(NN+1)//2,             % constante mágica
        printf("Sum = %d%n", [Sum]),
        dim(Square, [N,N]),             % definición de variable
```

```
        Square[1..N,1..N] :: 1..NN,            % dominio de variable (rango)
        Rows is Square[1..N,1..N],
        flatten(Rows, Vars),
        Square[1,1] #\= Square[1,2],           % restricciones para todas las
        posiciones diferentes. Sin Restricción global

Square[1,1] #\= Square[1,3],                        Square[1,3] #\= Square[3,4],
        Square[1,1] #\= Square[1,4],                Square[1,3] #\= Square[4,1],
        Square[1,1] #\= Square[2,1],                Square[1,3] #\= Square[4,2],
        Square[1,1] #\= Square[2,2],                Square[1,3] #\= Square[4,3],
        Square[1,1] #\= Square[2,3],                Square[1,3] #\= Square[4,4],
        Square[1,1] #\= Square[2,4],                Square[1,4] #\= Square[2,1],
        Square[1,1] #\= Square[3,1],                Square[1,4] #\= Square[2,2],
        Square[1,1] #\= Square[3,2],                Square[1,4] #\= Square[2,3],
        Square[1,1] #\= Square[3,3],                Square[1,4] #\= Square[2,4],
        Square[1,1] #\= Square[3,4],                Square[1,4] #\= Square[2,2],
        Square[1,1] #\= Square[4,1],                Square[1,4] #\= Square[2,3],
        Square[1,1] #\= Square[4,2],                Square[1,4] #\= Square[2,4],
        Square[1,1] #\= Square[4,3],                Square[1,4] #\= Square[3,1],
        Square[1,1] #\= Square[4,4],                Square[1,4] #\= Square[3,2],
        Square[1,2] #\= Square[1,3],                Square[1,4] #\= Square[3,3],
        Square[1,2] #\= Square[1,4],                Square[1,4]  #\= Square[3,4],
        Square[1,2] #\= Square[2,1],                Square[1,4]  #\= Square[4,1],
        Square[1,2] #\= Square[2,2],                Square[1,4]  #\= Square[4,2],
        Square[1,2] #\= Square[2,3],                Square[1,4]  #\= Square[4,3],
        Square[1,2] #\= Square[2,4],                Square[1,4]  #\= Square[4,4],
        Square[1,2] #\= Square[3,1],                Square[2,1] #\= Square[2,2],
        Square[1,2] #\= Square[3,2],                Square[2,1] #\= Square[2,3],
        Square[1,2] #\= Square[3,3],                Square[2,1] #\= Square[2,4],
        Square[1,2] #\= Square[3,4],                Square[2,1] #\= Square[3,1],
        Square[1,2] #\= Square[4,1],                Square[2,1] #\= Square[3,2],
        Square[1,2] #\= Square[4,2],                Square[2,1] #\= Square[3,3],
        Square[1,2] #\= Square[4,3],                Square[2,1] #\= Square[3,4],
        Square[1,2] #\= Square[4,4],                Square[2,1] #\= Square[4,1],
        Square[1,3] #\= Square[1,4],                Square[2,1] #\= Square[4,2],
        Square[1,3] #\= Square[2,1],                Square[2,1] #\= Square[4,3],
        Square[1,3] #\= Square[2,2],                Square[2,1] #\= Square[4,4],
        Square[1,3] #\= Square[2,3],                Square[2,2] #\= Square[2,3],
        Square[1,3] #\= Square[2,4],                Square[2,2] #\= Square[2,4],
        Square[1,3] #\= Square[3,1],                Square[2,2] #\= Square[3,1],
        Square[1,3] #\= Square[3,2],                Square[2,2] #\= Square[3,2],
        Square[1,3] #\= Square[3,3],                Square[2,2] #\= Square[3,3],
```

```
        Square[2,2] #\= Square[3,4],                Square[3,1] #\= Square[3,2],
        Square[2,2] #\= Square[4,1],                Square[3,2] #\= Square[3,3],
        Square[2,2] #\= Square[4,2],                Square[3,2] #\= Square[3,4],
        Square[2,2] #\= Square[4,3],                Square[3,2] #\= Square[4,1],
        Square[2,2] #\= Square[4,4],                Square[3,2] #\= Square[4,2],
        Square[2,3] #\= Square[2,4],                Square[3,2] #\= Square[4,3],
        Square[2,3] #\= Square[3,1],                Square[3,2] #\= Square[4,4],
        Square[2,3] #\= Square[3,2],                Square[3,3] #\= Square[3,4],
        Square[2,3] #\= Square[3,3],                Square[3,3] #\= Square[4,1],
        Square[2,3] #\= Square[3,4],                Square[3,3] #\= Square[4,2],
        Square[2,3] #\= Square[4,1],                Square[3,3] #\= Square[4,3],
        Square[2,3] #\= Square[4,2],                Square[3,3] #\= Square[4,4],
        Square[2,3] #\= Square[4,3],                Square[3,4] #\= Square[4,1],
        Square[2,3] #\= Square[4,4],                Square[3,4] #\= Square[4,2],
        Square[2,4] #\= Square[3,1],                Square[3,4] #\= Square[4,3],
        Square[2,4] #\= Square[3,2],                Square[3,4] #\= Square[4,4],
        Square[2,4] #\= Square[3,3],                Square[4,1] #\= Square[4,2],
        Square[2,4] #\= Square[3,4],                Square[4,1] #\= Square[4,3],
        Square[2,4] #\= Square[4,1],                Square[4,1] #\= Square[4,4],
        Square[2,4] #\= Square[4,2],                Square[4,2] #\= Square[4,3],
        Square[2,4] #\= Square[4,3],                Square[4,2] #\= Square[4,4],
        Square[2,4] #\= Square[4,4],                Square[4,3] #\= Square[4,4],
        (
            for(I,1,N),
            foreach(U,UpDiag),
            foreach(D,DownDiag),
            param(N,Square,Sum)
        do
            Sum #= sum(Square[I,1..N]),         % suma de filas
            Sum #= sum(Square[1..N,I]),         % suma de columnas
            U is Square[I,I],
            D is Square[I,N+1-I]
        ),
        Sum #= sum(UpDiag),                     % suma de diagonales
        Sum #= sum(DownDiag).


%UTILIDADES
lab(AllVars,BT) :-
        init_backtracks,
        init_nodesv,
       (fromto(AllVars, Vars, Rest, [])
```

```
      do
             choose_var_first_list(Vars, Var, Rest),
             indomain(Var,max),
             count_backtracks,
             count_nodesv
       ),
      get_backtracks(BT).

labs(AllVars,BT,ND) :-
       init_backtracks,
        init_nodesv,
       setval(cond,[]),
       getval(cond,COND),
       (fromto(AllVars, Vars, Rest, COND)
        do
             choose_var_MRV(Vars, Var, Rest),
             choose_val(Var,Val),
             Var = Val,
             count_backtracks,
             count_nodesv
       ),
       get_backtracks(BT),
        get_nodesv(ND).

%Selecciona variables
%El primero de la lista
choose_var_first_list(List,Var,Rest):-
       List = [Var|Rest],
       get_domain_size(Var,L).

%Selecciona la variable con
%dominio más pequeño
choose_var_MRV(List,Var,Rest):-
       delete(Var,List,Rest, 0, first_fail),
       get_domain_size(Var,L).

choose_val(Var,Val):-
       get_domain_as_list(Var,Domain),
       member(Val,Domain).
```

Para medir BT

```
:- local variable(backtracks), variable(deep_fail).
init_backtracks :-
       setval(backtracks,0).                  % Se inicializa la variable        %
                                              bactracks en cero
init_backtrackst :-
       setval(backtrackst,0).

get_backtrackst(B) :-
       getval(backtrackst,B).

get_backtracks(B) :-
       getval(backtracks,B).

count_backtracks :-
       setval(deep_fail,false).

count_backtracks :-
       getval(deep_fail,false),
       setval(deep_fail,true),
       incval(backtracks),
       incval(backtrackst),
       fail.


Para contar el número de nodos visitados
init_nodesv:-
       setval(nodesv,0).

get_nodesv(ND):-
       getval(nodesv, ND).

count_nodesv:-
       incval(nodesv).
```

B. CÓDIGO FUENTE SEND+MORE=MONEY

```
% Archivo: MODELOS_SMM.pl
% llamado de librerías
:-lib(ic).
:-lib(lists).
:-lib(viewable).
:-lib(branch_and_bound).
:-lib(ic_sbds).

%MODELO 1
modelo1([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
      init_nodesv,
      statistics(times, [T0|_]),
      Digits = [S,E,N,D,M,O,R,Y], % definicion de variables
      Digits :: [0..9], % definicion de dominios
      alldifferent(Digits), % definicion de restricciones
      S #\= 0,
      M #\= 0,
      1000*S + 100*E + 10*N + D
      + 1000*M + 100*O + 10*R + E
      #= 10000*M + 1000*O + 100*N + 10*E + Y,
      shallow_backtrack(Digits,B),
      statistics(times, [T1|_]),
      TimeFinal is T1-T0, % Calculo el tiempo total
      printlist(Digits),
      write('Valores finales para S,E,N,D,M,O,R,Y:& '), printlist(Digits),
      write(' & Tiempo Total:& '),write(TimeFinal),
      get_backtracks(BT),
      get_nodesv(ND),
      write(' & Bactrack totales:& '),write(BT),
      printf(" & Nodos visitados: & %d %n ", [ND]).

% MODELO 2
modelo2(Digits) :-
      init_backtrackst,
      init_nodesv,
      statistics(times, [T0|_]),
      Digits = [S,E,N,D,M,O,R,Y], % definición de variables
```

```
     Digits :: [0..9], % definición de dominio
     Carries = [C1,C2,C3,C4], % variables auxiliares
     Carries :: [0..1], % dominio de variables auxiliares
     alldifferent(Digits), % restricciones
     S #\= 0,
     M #\= 0,
     C1#= M,
     C2 + S + M #= O + 10*C1,
     C3 + E + O #= N + 10*C2,
     C4 + N + R #= E + 10*C3,
     D + E #= Y + 10*C4,
     shallow_backtrack(Carries,B),
     shallow_backtrack(Digits,B),
     term_variables(Digits, Domain),
     statistics(times, [T1|_]),
     TimeFinal is T1-T0, % Calculo el tiempo total
     nl,write('Resultado Final:'),nl, printlist(Digits),
     nl,write('Tiempo Total:'),nl,write(TimeFinal),
     get_backtracks(B),
     get_nodesv(ND),
     nl,write('Bactrack totales:'),nl,write(B),
     nl,printf("Nodos visitados %d nodos%n", [ND]).

% Modelo 3
modelo3(Digits) :-
     init_backtrackst,
     init_nodesv,
     statistics(times, [T0|_]),
     Digits = [S,E,N,D,M,O,R,Y], % definicion de variables
     Digits :: [0..9], % definicion de dominio
     Carries = [C1,C2,C3,C4], % variables auxiliares
     Carries :: [0..1], % dominio de variables auxiliares
     S #\= E,                                  S #\= Y,
     S #\= N,                                  E #\= N,
     S #\= D,                                  E #\= D,
     S #\= M,                                  E #\= M,
     S #\= O,                                  E #\= O,
     S #\= R,                                  E #\= R,
```

```
     E #\= Y,                                  M #\= Y,
     D #\= M,                                  O #\= R,
     D #\= O,                                  O #\= Y,
     D #\= R,                                  R #\= Y,
     D #\= Y,                                  S #\= 0,
     N #\= D,                                  M #\= 0,
     N #\= M,                                  C1#= M,
     N #\= O,                                  C2 + S + M #= O + 10*C1,
     N #\= R,                                  C3 + E + O #= N + 10*C2,
     N #\= Y,                                  C4 + N + R #= E + 10*C3,
     M #\= O,                                  D + E #= Y + 10*C4,
     M #\= R,
     term_variables(Digits, Domain),
     lab(Carries,B), % etiquetado
     lab(Digits,B), % etiquetado
     statistics(times, [T1|_]),
     TimeFinal is T1-T0, % Calculo el tiempo total
     nl,write('Resultado Final:'),nl, printlist(Digits),
     nl,write('Tiempo Total:'),nl,write(TimeFinal),
     get_backtracks(B),
     get_nodesv(ND),
     nl,write('Bactrack totales:'),nl,write(B),
     nl,printf("Nodos visitados %d nodos%n", [ND]).

% MODELO 4
modelo4(Digits) :-
     init_backtrackst,
     init_nodesv,
     statistics(times, [T0|_]),
     Digits = [S,E,N,D,M,O,R,Y], % definición de variables
     Digits :: [0..9], % definición de dominio
     Carries = [C1,C2,C3,C4], % variables auxiliares
     Carries :: [0..1], % dominio de variables auxiliares
     alldifferent(Digits), % restricciones
     S #\= 0,
     M #\= 0,
     C1#= M,
     M #= 1,
```

```
     C2 + S + M #= O + 10*C1,
     C3 + E + O #= N + 10*C2,
     C4 + N + R #= E + 10*C3,
     D + E #= Y + 10*C4,
     10*C2 + 9*C3 - C4 - O - R #=0,
     term_variables(Digits, Domain),
     lab(Carries,B),
     lab(Digits,B),
     statistics(times, [T1|_]),
     TimeFinal is T1-T0, % Calculo el tiempo total
     nl,write('Resultado Final:'),nl, printlist(Digits),
     nl,write('Tiempo Total:'),nl,write(TimeFinal),
     get_backtracks(BT),
     get_nodesv(ND),
     nl,write('Bactrack totales:'),nl,write(BT),
     nl,printf("Nodos visitados %d nodos%n", [ND]).

% MODELO 5
Modelo5(Digits) :-
     init_backtrackst,
     init_nodesv,
     statistics(times, [T0|_]),
Digits = [S,E,N,D,M,O,R,Y], % definición de variables
Digits :: [0..9], % definición de dominios
S #\= E,                                  D #\= O,
S #\= N,                                  D #\= R,
S #\= D,                                  D #\= Y,
S #\= M,                                  N #\= D,
S #\= O,                                  N #\= M,
S #\= R,                                  N #\= O,
S #\= Y,                                  N #\= R,
E #\= N,                                  N #\= Y,
E #\= D,                                  M #\= O,
E #\= M,                                  M #\= R,
E #\= O,                                  M #\= Y,
E #\= R,                                  O #\= R,
E #\= Y,                                  O #\= Y,
D #\= M,                                  R #\= Y,
```

```
S #\= 0,                                    + 1000*M + 100*O + 10*R + E
M #\= 0,                                    #= 10000*M + 1000*O + 100*N + 10*E
1000*S + 100*E + 10*N + D                   + Y,
term_variables(Digits, Domain),
shallow_backtrack(Digits,B),
statistics(times, [T1|_]),
TimeFinal is T1-T0, % Calculo el tiempo total
nl,write('Tablero Final:'),nl, printlist(Digits),
nl,write('Tiempo Total:'),nl,write(TimeFinal),
get_backtracks(B),
get_nodesv(ND),
nl,write('Bactrack totales:'),nl,write(B),
nl,printf("Nodos visitados %d nodos%n", [ND]).

% UTILIDADES
lab(AllVars,B) :-
      init_backtracks,
      init_nodesv,
      ( fromto(AllVars, Vars, Rest, [])
      do
            choose_var(Vars, Var, Rest),
            count_backtracks, % agregado
            count_nodesv,
            choose_val(Var, Val),
            Var = Val
      ),
      get_backtracks(B),
      get_nodesv(ND).

shallow_backtrack(AllVars,B) :-
      init_backtracks,
      init_nodesv,
      (foreach(Var,AllVars) do
            once (indomain(Var)),
            count_backtracks, % agregado
            count_nodesv
      ),
      get_backtracks(B),
```

```
      get_nodesv(ND).

labs(AllVars,BT) :-
      init_backtracks,
      setval(cond,[]),
getval(cond,COND),
(fromto(AllVars, Vars, Rest, COND)
do
      choose_var_MRV(Vars, Var, Rest),
      choose_val(Var, Val),
      Var = Val,
      count_backtracks
),
get_backtracks(BT).

% Selección de Variables
% El con dominio más grande
choose_var(List,Var,Rest):-
      List = [Var|Rest],
      get_domain_size(Var,L).

% Selecciona la variable con dominio más pequeño
choose_var_MRV(List,Var,Rest):-
      delete(Var,List,Rest, 0, first_fail),
      get_domain_size(Var,L).

% Selección de Valor
choose_val(Var,Val):-
      get_domain_as_list(Var,Domain),
      member(Val,Domain).

% Para medir BT
% Va contando los Bactrack es decir cuando una variable que tenía un valor
% es desinstanciada y se elige otra variable.
:- local variable(backtracks), variable(deep_fail).
init_backtracks :-
      setval(backtracks,0). % Se inicializa la variable
```

```
% bactracks en cero
init_backtrackst :-
      setval(backtrackst,0).

get_backtrackst(B) :-
      getval(backtrackst,B).

get_backtracks(B) :-
      getval(backtracks,B).

count_backtracks :-
      setval(deep_fail,false).

count_backtracks :-
      getval(deep_fail,false),
      setval(deep_fail,true),
      incval(backtracks),
      fail.

% Contar el número de nodos visitados
init_nodesv:-
      setval(nodesv,0).

get_nodesv(ND):-
      getval(nodesv, ND).

count_nodesv:-
      incval(nodesv).
```

C. CÓDIGO FUENTE N-REINAS

```
% Archivo: Modelo_1_2_3.pl
:-lib(ic).
:-lib(lists).
:-lib(branch_and_bound).
:-lib(ic_sbds).

queen(N,Board):-
       statistics(times, [T0|_]),
       % modelo1(N, Board),                          % Se define el Modelo 1
       % modelo2(N,Board),                    % Se define el Modelo 2
       % modelo3(N, Board),                          % Se define el Modelo 3
       statistics(times, [T1|_]),
       TimeFinal is T1-T0,
       write('Tablero Final:& '),write(Board),
       write(' & Tiempo Total:& '),write(TimeFinal),
       get_backtracks(BT),
       get_nodesv(ND),
       write(' & Bactrack totales:& '),write(BT),
       printf(" & Nodos visitados: & %d %n ", [ND]).

% MODELO 1
modelo1(N, Board) :-
       length(Board, N),
      Board :: 1..N,
       alldifferent(Board),      % Las filas son distintas
      ( fromto(Board, [Q1|Cols], Cols, []) do
       ( foreach(Q2, Cols), param(Q1), count(Dist,1,_) do
              noattack(Q1, Q2, Dist)
              )
        ).
noattack(Q1,Q2,Dist) :-
       Q2 #\= Q1,
       Q2 - Q1 #\= Dist,
       Q1 - Q2 #\= Dist.

% MODELO 3
Modelo3 (N,Board) :-
       length(Board,N),                % Hay N reinas
       Board :: 1..N,                  % Las filas están en el intervalo 1..N
```

```
	segura(Board),			% No hay en más de una reina en las
					diagonales
	alldifferent(Board),		% Las filas son distintas
	lab(Board) .		% Buscar los valores de L

%segura(L) se verifica si las reinas colocadas en las filas L no se atacan
%diagonalmente
segura([]).
	segura([Y|Board]) :-
	no_ataca(Y,Board,1),
	segura(Board).

%no ataca(Y,L,D) se verifica si: Y es un número, L es una lista de números %[n1, . .
. , nm]
%y D es un número tales que, la reina colocada en la posición (x,Y) no %ataca a las
colocadas en las posiciones (x + d, n1), . . , (x + d +m, nm).

no_ataca(_Y,[],_).

no_ataca(Y1,[Y2|L],D) :-
	Y1-Y2 #\= D,
	Y2-Y1 #\= D,
	D1 is D+1,
	no_ataca(Y1,L,D1).

% MODELO 2
Modelo2(N, Board) :-
	dim(Board, [N]),
	Board[1..N] :: 1..N,
	( for(I,1,N), param(Board,N) do
		( for(J,I+1,N), param(Board,I) do
			Board[I] #\= Board[J],
			Board[I] #\= Board[J]+J-I,
			Board[I] #\= Board[J]+I-J
		)
	),
	Board =.. [_|Vars],
	lab(Vars).

% UTILIDADES
lab(AllVars) :-
	init_backtracks,
```

```
      init_nodesv,
      ( fromto(AllVars, Vars, Rest,
       do
       choose_var(Vars, Var, Rest),
             count_backtracks,            % Ejemplo de Indicador
              count_nodesv,
              % se llama a la heurística que se define previamente
              choose_val(Var, Val),
              % se asigna el valor elegido a la variable. Se ejecuta %propagación, si
              falla vuelve para elegir otro variable.
              Var = Val
       ).

% Selección de Variables
% Selecciono el primero de la lista
choose_var(List,Var,Rest):-
% se separa al vars tomando la cabecera, el resto de la lista es devuelto  % en
Rest, por ello al final el rest sera igual a []
       List = [Var|Rest],
       get_domain_size(Var,L).

% Selección de Valor
choose_val(Var,Val):-
       get_domain_as_list(Var,Domain),
       member(Val,Domain).

% Para medir BT
% Este va contando los Bactrack es decir cuando una variable que tenía un  % valor
es desinstanciada y se elige otra variable.

:- local variable(backtracks), variable(deep_fail).
init_backtracks :-
       setval(backtracks,0).

get_backtracks(B) :-
       getval(backtracks,B).

count_backtracks:-
      setval(deep_fail,false).

count_backtracks :-
      getval(deep_fail,false),
```

```
      setval(deep_fail,true),
      incval(backtracks),
      fail.

% Para contar el numero de nodos visitados
init_nodesv:-
      setval(nodesv,0).

get_nodesv(ND):-
      getval(nodesv, ND).

count_nodesv:-
      incval(nodesv).

% MODELO 4
% Archivo: Modelo_4.pl
:- set_flag(gc_interval, 100000000).
:- lib(fd).
:- lib(fd_sbds).

% queens/2 encuentra todas las soluciones
% N es el tamaño del tablero
% Se encuentran todas las soluciones descartando las simétricas
queens(N) :-
      setval(solutions, 0),
      statistics(times, [T0|_]),
      (
          queens(N, Board),
          init_backtracks,
          sbds_labeling(Board), % llamado a etiquetado para simetría
          writeln(Board),       % se imprime la solución encontrada
          incval(solutions),           % incrementa el valor de soluciones
          fail
      ;
          true
      ),
      get_backtracks(B),
      get_nodesv(ND),
      statistics(times, [T1|_]),
      T is T1-T0,
      getval(solutions, S),
```

```
	printf("\n %d soluciones encontradas para %d reinas en %w segundos con %d
backtracks y %d nodos visitados\n",[S,N,T,B,ND]).

queens(N, Board) :-
	length(Board, N),
	Board :: 1..N,
	alldifferent(Board),
	( fromto(Board, [Q1|Cols], Cols, []) do
		( foreach(Q2, Cols), param(Q1), count(Dist,1,_) do
			noattack(Q1, Q2, Dist)
		)
	),
	Matrix =.. [[] | Board],
	%listado de funciones para simetría
	Syms = [
		r90(Matrix, N),
		r180(Matrix, N),
		r270(Matrix, N),
		rx(Matrix, N),
		ry(Matrix, N),
		rd1(Matrix, N),
		rd2(Matrix, N)
	],
	%se inicializa sbds
	fd_sbds:(sbds_initialise(Matrix, Syms, #=, [])).

noattack(Q1,Q2,Dist) :-
	fd: (Q2 #\= Q1),
	fd:(Q2 - Q1 #\= Dist),
	fd:(Q1 - Q2 #\= Dist).

% Predicados para simetría.
%Input: Matriz de variables de búsqueda
%Input: Dimensión del tablero, N
%Input: [X,Y] Índice en la matriz de la variable para la cual se aplica simetría
%Input: Valor para la variable
%Output:  Variable simétrica SymVar
%Output:  Valor simétrico SymValue

r90(Matrix, N, Index, Value, SymVar, SymValue) :-
	SymVar is Matrix[Value],
	SymValue is N + 1 - Index.
```

```
r180(Matrix, N, Index, Value, SymVar, SymValue) :-
       SymVar is Matrix[N + 1 - Index],
       SymValue is N + 1 - Value.

r270(Matrix, N, Index, Value, SymVar, SymValue) :-
       SymVar is Matrix[N + 1 - Value],
       SymValue is Index.

rx(Matrix, N, Index, Value, SymVar, SymValue) :-
       SymVar is Matrix[N + 1 - Index],
       SymValue is Value.

ry(Matrix, N, Index, Value, SymVar, SymValue) :-
       SymVar is Matrix[Index],
       SymValue is N + 1 - Value.

rd1(Matrix, _N, Index, Value, SymVar, SymValue) :-
       SymVar is Matrix[Value],
       SymValue is Index.

rd2(Matrix, N, Index, Value, SymVar, SymValue) :-
       SymVar is Matrix[N + 1 - Value],
       SymValue is N + 1 - Index.

% Estrategias de Búsqueda

sbds_labeling(AllVars) :-
init_nodesv,
       ( foreach(Var, AllVars) do
              count_backtracks,
              count_nodesv,
             sbds_indomain(Var)         % Selección de Valor
         ).

% Reemplazo para indomain/1 que toma en cuenta SBDS.
sbds_indomain(X) :-
       nonvar(X).

sbds_indomain(X) :-
       var(X),
       mindomain(X, LWB),
```

```
       fd_sbds:(sbds_try(X, LWB)),
       sbds_indomain(X).

% Utilidades

search_space(Vars, Size) :-
       ( foreach(V,Vars), fromto(1,S0,S1,Size) do
           dvar_domain(V,D),
           S1 is S0*dom_size(D)
       ).

% Para contar cantidad de backtracks
:- local variable(backtracks), variable(deep_fail).

init_backtracks :-
        setval(backtracks,0).

get_backtracks(B) :-
        getval(backtracks,B).

count_backtracks :-
        setval(deep_fail,false).
count_backtracks :-
        getval(deep_fail,false),
        setval(deep_fail,true),
        incval(backtracks),
       fail.

% Para contar el número de nodos visitados

init_nodesv:-
       setval(nodesv,0).

get_nodesv(ND):-
       getval(nodesv, ND).

count_nodesv:-
       incval(nodesv).
```