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

RESOLUCIÓN DEL CONSTRAINED FINANCIAL PORTFOLIO SELECTION PROBLEM UTILIZANDO PROGRAMACIÓN CON RESTRICCIONES

CAMILA ANDREA ALLENDES FARÍAS

HANS NICOLÁS BERENDSEN MELLA

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

ENERO 2012

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

RESOLUCIÓN DEL CONSTRAINED FINANCIAL PORTFOLIO SELECTION PROBLEM UTILIZANDO PROGRAMACIÓN CON RESTRICCIONES

CAMILA ANDREA ALLENDES FARÍAS

HANS NICOLÁS BERENDSEN MELLA

Profesor Guía: **Ricardo Soto De Giorgis**

Profesor Co-referente: **Broderick Crawford Labrín**

Carrera: **Ingeniería de Ejecución en Informática**

Enero 2012

Dedicatoria

A Dios, a nuestros padres y a todas las
personas que nos alentaron a cumplir nuestros anhelos,
sin importar las dificultades que encontráramos en el camino.

Agradecimientos

Agradecemos a nuestros padres, quienes nos brindaron su apoyo incondicional y nos impulsaron a seguir cuando veíamos que el camino se ponía cuesta arriba. Fueron ellos quienes nos recordaron día a día que nada en la vida es fácil, y que los sueños son para hacerlos realidad. Sin ellos, este anhelo no se habría podido concretar.

A nuestros familiares y compañeros, quienes siempre tuvieron la palabra de aliento necesaria para no desfallecer cuando las fuerzas nos abandonaban. En especial a nuestros amigos de toda la vida, los que sin importar cuán difícil se viera el panorama, siempre confiaron en que seríamos capaces de salir adelante y podríamos superar cualquier obstáculo que se interpusiera en nuestro camino.

A nuestros profesores que nos entregaron las herramientas necesarias para afrontar esta tarea y salir airosos de ella.

En pocas palabras, a todas las personas que se cruzaron por nuestros caminos estos años de estudio y esfuerzo.

Por todo esto, infinitas gracias a todos.

Resumen

El objetivo de este proyecto es modelar y resolver el problema de la Selección de Carteras con Restricciones Financieras, en inglés, Constrained Financial Portfolio Selection Problem (CFPSP), utilizando programación con restricciones. La idea de este problema es determinar el porcentaje a invertir de cada acción perteneciente a un determinado portafolio de manera tal que el riesgo sea minimizado. En el presente proyecto, el problema se resuelve utilizando el modelo propuesto por el economista Harry Markowitz y los solvers Eclipse y Minizinc.

Palabras Claves: Portfolio Selection Problem, Solver, Constraint Programming, Constraint Satisfaction Problem.

Abstract

The goal of this project is to model and solve the Constrained Financial Portfolio Selection Problem (CFPSP) by using constraint programming. The idea of this problem is to determine the percentage of each stock to invest from a given portfolio in order to minimize the associated risk. In the present project, we solve the problem by using the model proposed by the economist Harry Markowitz and the Eclipse and Minizinc solvers.

Key Words: Portfolio Selection Problem, Solver, Constraint Programming, Constraint Satisfaction Problem.

Índice

Lista de Figuras

Lista de Tablas

1 Introducción

Actualmente las empresas, entidades y diversas instituciones buscan la mejor opción para poder satisfacer de forma rápida y efectiva las numerosas necesidades de los clientes. Por esto, las empresas se encuentran en una constante búsqueda para poder generar un mayor incremento en la productividad y eficiencia, pero por sobre todo poder responder efectivamente a las exigencias del mercado.

La fuerte competitividad que existe entre las empresas ha hecho que muchas de estas hayan incorporado el concepto de Optimización, el que ha ido ganando espacio de forma rápida y expansiva. Este concepto se ha masificado principalmente en el área de la Economía y específicamente en el sector Bancario, donde se trabaja con Portafolios.

Al hablar de Portafolio, nos referimos a una cartera de inversiones, la que es un conjunto de activos financieros que proporcionan el retorno esperado más alto para cualquier nivel de riesgo o el nivel más bajo, dependiendo de lo que se desee obtener. La clave para un Portafolio no consiste sólo en el número de acciones que lo componen, sino que también en la correlación de los retornos de las acciones que lo conforman.

El Problema de Selección de Portafolio Financiero con Restricciones, en inglés Constrained Financial Portfolio Selection Problem (CFPSP), tiene como finalidad encontrar la mejor cartera, dentro de un conjunto de ellas. Para esto, se utilizará el modelo propuesto por el economista Harry Markowitz, el que nos señala que se debe seleccionar la cartera que entregue el menor riesgo asociado. Por consiguiente, las carteras analizadas deberán cumplir las restricciones propuestas en este modelo, las que servirán para escoger la cartera óptima.

El objetivo de este proyecto es modelar y resolver el CFPSP utilizando programación con restricciones. La fase de modelamiento consistirá en formalizar el problema como un Constraint Optimization Problem (COP) y la fase de resolución se llevará a cabo utilizando distintos motores de búsqueda (solver).

Primeramente, la resolución se desarrolló con el solver Eclipse en donde se modeló el problema en sí. Posterior a esto, se realizó el mismo proceso utilizando el solver Minizinc. Una vez concluidos los procesos de traspasos a los solvers, se continuó con la fase de experimentación en donde se analizaron con distintas heurísticas y configuraciones de datos, con el fin de obtener una aproximación al momento de optar por la elección de uno u otro solver.

2 Objetivos

2.1 Objetivo General

Modelar y resolver el Constrained Financial Portfolio Selection Problem utilizando dos motores de búsqueda para programación con restricciones, con el fin de comparar y analizar los resultados obtenidos. Los motores de búsqueda utilizados fueron Eclipse y Minizinc.

2.2 Objetivos Específicos

- Modelar el Constrained Financial Portfolio Selection Problem como un COP.
- Resolver el Constrained Financial Portfolio Selection Problem utilizando los solvers Eclipse y Minizinc para COP.
- Comparar y analizar los resultados utilizando distintas estrategias de resolución.

3 Estado del Arte

Desde siempre, el inversor ha debido tomar precauciones al momento de invertir, ya que todos desean obtener la mayor rentabilidad posible, pero sin arriesgarse demasiado. Es por esto que la Selección de Carteras es un tema fundamental al momento de decidir cómo invertir nuestros bienes.

La Selección de Carteras consiste en, como su nombre lo indica, elegir la mejor cartera de activos, para poder conseguir, con el menor riesgo posible, la mayor rentabilidad asociada. Para esto se han generado diversos modelos, los que han ido evolucionando con el transcurso de los años. Estos modelos han ayudado a los inversores a escoger la mejor opción en temas de carteras de inversiones.

El modelo en el que se basa este trabajo corresponde al modelo propuesto por Harry Markowitz, el cual plantea que para escoger la mejor cartera de inversiones, se deben tener en cuenta ciertas restricciones, como por ejemplo, que la suma de los porcentajes a invertir de cada activo debe ser igual a 1.

En el transcurso de los años, este modelo ha sido desarrollado de diversos modos, utilizando distintas formas de resolución. Cramay Schyns [1] utilizan un enfoque Simulated Annealing, para dar solución al problema de la Selección de Carteras. Se consideró que el modelo de Markowitz era una excelente base, pero no consideraba todas las restricciones a las que se enfrentan los inversionistas, como por ejemplo, las limitaciones comerciales, el tamaño de las carteras, etc. Es por esto que deciden utilizar la metaheurística Simulated Annealing, incorporando estas restricciones al modelo.

Rubén Armañanzas y Jose A. Lozano [2] realizan una investigación, en la que analizan tres tipos de búsquedas para dar solución a la Selección de Carteras. Las técnicas de búsqueda analizadas son Greedy Search, Simulated Annealing y Ant Colony Optimization. La técnica Greedy Search es sencilla de utilizar y entrega resultado rápidamente, pero a su vez, su sencillez es su mayor desventaja, ya que una vez que encuentra una solución, es difícil que encuentre otra. Simulated Annealing ha sido explicado en el apartado anterior, por lo que no se harán referencias. El Ant Colony Optimization realiza una triple búsqueda. Es decir, busca solución al problema pero desde tres puntos de vistas distintos: realiza una búsqueda basándose en la alta rentabilidad, otra basándose en obtener un bajo riesgo, y por último, una búsqueda que combine ambos puntos mencionado recientemente.

Como nuestra base es el Modelo de Markowitz, una de las soluciones más simples para el problema de optimización, cuando no se considera la restricción de no-negatividad, es la utilización del Método de Lagrange, pues resulta muy útil para encontrar soluciones [3]. Pero, cuando se considera esta restricción, el modelo se vuelve más complejo y puede ser resuelto por algoritmos como la adaptación de Wolfe al Método Simplex [4].

Cuando el modelo incluye restricciones de comercio o de número máximo de activos en el Portafolio, el problema se vuelve de tipo entero mixto no-lineal. Uno de los investigadores que ha tratado de resolver este tipo de modelos es Perold, quien incluyó, un conjunto de restricciones al modelo de Markowitz [5].

Maringer y Kellerer lograron resolver el problema de optimización de carteras con restricciones, utilizando un algoritmo híbrido de búsqueda local, el que combina los principios de Simulated Annealing con Estrategias evolutivas [6].

4 Constraint Programming

4.1 Definición

Hoy en día existen un sinnúmero de problemas asociados a la vida real, los cuales presentan una serie de restricciones. Problemas comunes como fijar una reunión, adquirir una vivienda, o realizar un plato de comida dependen de muchos aspectos independientes, con sus respectivos problemas, sujetos a un conjunto de restricciones. Es aquí donde la programación con restricciones ha generado gran expectación entre los expertos, debido a su potencial de resolución de problemas cotidianos.

La programación con restricciones es un paradigma surgido en los años 90. Nace principalmente de la Inteligencia Artificial como también de la Investigación de Operaciones, resolviendo gran cantidad de problemas de diversa índole, como lo son problemas de calendarización, planificación y combinatoriales, entre otros.

La idea, es resolver problemas mediante la declaración de un modelo, el que está compuesto por restricciones, variables, dominios y en algunos casos una Función Objetivo de un problema dado y consecuentemente encontrar soluciones que satisfagan dichas restricciones.

4.2 Constraint Satisfaction Problem (CSP)

Como se señaló anteriormente, la programación con restricciones se encarga de resolver un problema a través de la formulación de un modelo, el que está compuesto por variables, dominio y restricciones.

Una vez que se tiene claro el concepto del problema, hay que traspasarlo a una formulación matemática la cual se conoce como Constraint Satisfaction Problem (CSP) en inglés, el cual es la formalización del problema que se utiliza para poder resolverlo.

Un problema de satisfacción de restricciones se define como una terna (X, D, R) donde [7]:

- $X = \{X_1, X_2,\ldots,X_m\}$es un conjunto finito de variables,
- $D = \{D_1,D_2,\ldots,D_m\}$ es un conjunto finito de dominios y
- $R = \{R_1,R_1,\ldots,R_n\}$ es un conjunto finito de restricciones.

Cada variable X_i toma valores en su correspondiente dominio D_i. Una restricción entre un subconjunto de variables $\{Xi_1,\ldots,Xi_k\}$ de X es un subconjunto del producto cartesiano que está formado por las tuplas de valores que satisfacen la restricción. Dicho de otra forma, contiene las combinaciones de asignaciones de valores a variables que no violan la restricción.

Los objetivos de un problema de satisfacción de restricciones son los siguientes:

- Consistencia del problema (que exista una solución).
- Obtener una o todas las soluciones del problema.
- Obtener una solución óptima, o al menos una buena solución, medida por alguna Función Objetivo.

Los ejemplos de CSP son variados, como los que se muestran a continuación.

4.2.1 Problema del "Send + More = Money"

Este problema consiste en reemplazar las letras (S,E,N,D,M,O,R,Y) por dígitos dentro de un dominio {0,…,9} de manera tal que esta suma se satisfaga (ver figura 4.1). Utilizando dígitos del 0 al 9 y distintos para cada una de las letras.

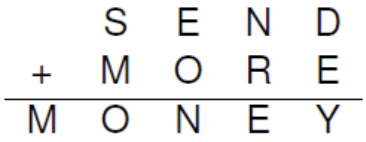

Figura 4.1 – Problema del "Send + More = Money"

Como se acaba de mencionar, este problema se modela por tener una variable para cada letra, donde la variable representa el dígito asociado a la letra. Las limitaciones son evidentes a partir de la especificación del problema.

La solución a este problema es multiplicar cada uno de los dígitos por el número que corresponde a la posición dentro de la suma. Todas las restricciones que representan la diferencia de los valores que pueden tomar cada uno de los dígitos porque si no a todas las variables se le asignan 0 y se tendría la solución inmediatamente. La idea es que cada uno adopte un valor distinto. La representación de las restricciones es la siguiente:

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

4.2.2 Coloración de Mapa

El problema de coloración de mapa, parte de la base que existe un mapa el que está dividido por secciones, las cuales están sin color. El problema consiste en colorear cada región del mapa sin que se topen los colores de regiones adyacentes. Para formular el problema como un CSP se define una variable por cada región del mapa, y el dominio de cada variable es el conjunto de colores disponible. Las restricciones son 5, las que consisten en que para cada par de regiones contiguas existe una restricción sobre las variables correspondientes que no permite la asignación de idénticos valores a las variables.

Para este ejemplo se tiene el mapa de la figura 4.2, en el cual se encuentran cuatro regiones x, y, z, w para ser pintadas con los posibles colores Rojo, Verde, Azul. La formulación CSP sería:

- Variables: {x, y, z, w}.
- Dominio: {Rojo, Verde, Azul}, único para las tres variables.
- Restricciones: {x ≠ y, x ≠ w, x ≠ z, y ≠ z, w ≠ z}.

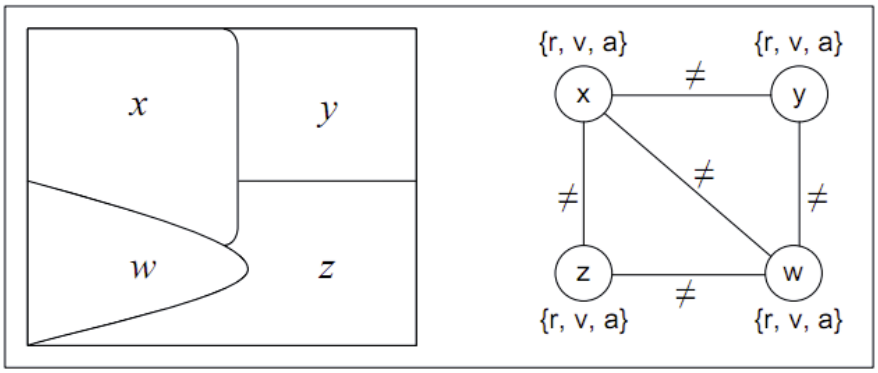

Figura 4.2 – Problema de Colorear Mapa

4.2.3 Problema de las N-Reinas

Consiste en ubicar N reinas en un tablero de dimensiones N x N de tal forma que cada reina, cuando realice un movimiento, no se interfiera o puedan ser capturadas por el desplazamiento de cada una de ellas. Para ejemplificar este problema, en este caso se utiliza N = 4, quedando un tablero 4 x 4 (ver figura 4.3).

El problema puede ser modelado de la siguiente forma:

- Variables: { Q_i }, i = 1...4.
- Dominio: {1,2,3,4}, para todas las variables.
- Restricciones: ($\forall Q_i, Q_j, i \neq j$):
  - $Q_i \neq Q_j$, para las filas
  - $Q_i + i \neq Q_j + j$, diagonal 1
  - $Q_i - i \neq Q_j - j$. diagonal 2

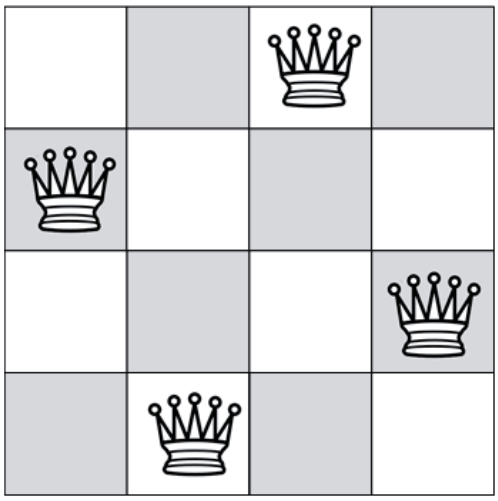

Figura 4.3 – Problema de las N-Reinas

No se modela como un matriz, si no que como un vector porque se asume que cada reina se mueve en su lugar. No se valida la columna porque ya se sabe que se mueven en la columna, por lo tanto solamente interesa validar las filas que no se ataquen hacia el lado; también que Qi sea distinto Qj si es que cada uno vale uno. Interesa que no estén en la misma posición y si no se van a atacar en la línea.

4.3 Constraint Optimization Problem (COP)

Como se explicó en el punto anterior (ver capítulo 4.2), CSP resuelve los problemas a través de la utilización de variables, dominios y restricciones. COP corresponde a una extensión de CSP, al cual se le agrega la utilización de una Función Objetivo, la que se puede maximizar o minimizar dependiendo del resultado esperado.

El objetivo de la programación con restricciones es solucionar los problemas mediante la declaración de restricciones sobre el dominio del problema, y encontrar soluciones que satisfagan dichas restricciones. Se espera obtener una solución óptima.

Un problema de optimización con restricciones, a menudo, se formula como [8]:

Minimizar: $f(x)$

$$g_j(x) \leq 0, \quad i = 1, \dots, p$$

$$h_i(x) = 0, \quad j = 1, \dots, m$$

En el siguiente ejemplo podemos ver de manera sencilla la declaración de un COP.

$$x \in [0, 2]$$

$$y \in [2, 4]$$

$$x + y \leq 5$$

$$\max(x + y)$$

Donde se espera encontrar los valores de X e Y los cuales, sumados, entreguen un valor menor o igual a 5. Al utilizar una Función Objetivo, ejemplificada con "max", lo que se espera obtener son los valores únicos que cumplen con esta condición. En este caso, X=2 e Y=3.

4.4 Algoritmos de Búsqueda

La mayoría de los algoritmos utilizados para resolver CSP buscan sistemáticamente entre las asignaciones posibles de valores a variables, siendo una gran ventaja el encontrar una solución (si es que existe), pero en otros casos siendo desventaja la ineficiencia de algunos algoritmos. Las combinaciones de la asignación de valores a las variables en un CSP generan un espacio de búsqueda que puede ser visto como un árbol de búsqueda. La búsqueda mediante *Backtracking* (ver capítulo 4.4.2) en un CSP corresponde a la tradicional exploración primero en profundidad en el árbol de búsqueda (ver figura 4.4).

En este punto, se explican los algoritmos de búsqueda, encargados de encontrar la solución óptima para el problema dado. Algunos algoritmos de búsqueda son los siguientes.

4.4.1 Generate and Test (GT)

Algoritmo de la década del 1940-1950. Su objetivo es buscar todas las combinaciones posibles, probando una combinación y después verificando si esta última es correcta. Entonces el solver crea un árbol de búsqueda y ve todas las combinaciones posibles.

Aquí el número de combinaciones posibles es el producto cartesiano de todos los dominios de las variables generando una muy ineficiente y básica búsqueda. A continuación se ilustra el *Generate and Test* (ver figura 4.4):

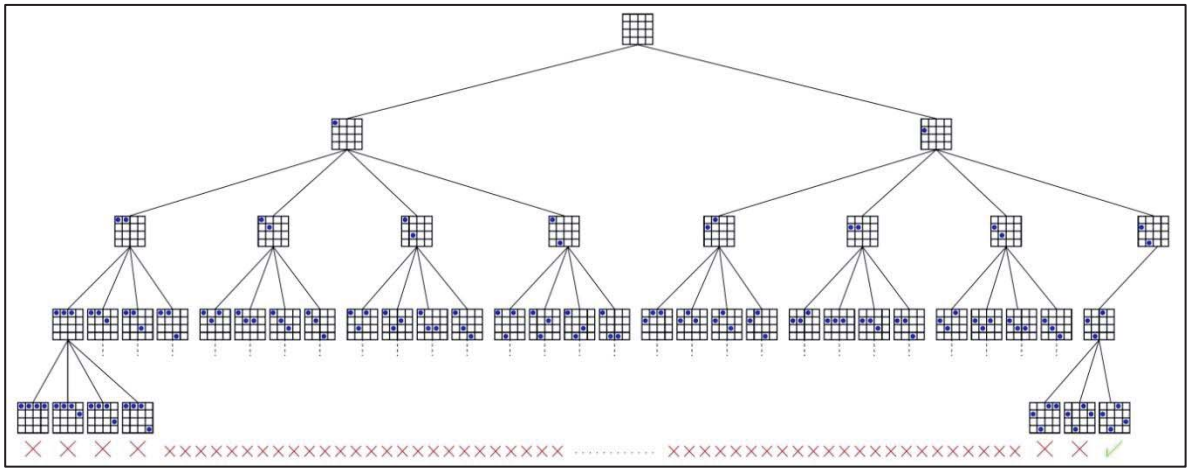

Figura 4.4 – Resolviendo el problema de las N-Reinas con Generate and Test

4.4.2 Backtracking (BT)

Algoritmo de la década del 1950. Mejora un poco el *Generate and Test*, construyendo soluciones parciales a medida que progresa el recorrido. Estas soluciones limitan las regiones en las que se puede encontrar una solución completa. En este caso el algoritmo puede bien detenerse, si lo único que se necesita es una solución del problema o bien seguir buscando soluciones alternativas, si es que se desea examinar todas.

En otras palabras, *Backtracking* comprueba si es que hay una solución candidata; si esta solución candidata realmente es una solución, se realiza alguna acción con ella (dependiendo del problema), construyendo todas las posibles extensiones de esta solución encontrada e invocar recursivamente al algoritmo con todas ellas.

A continuación, se ilustra el *Backtracking*:

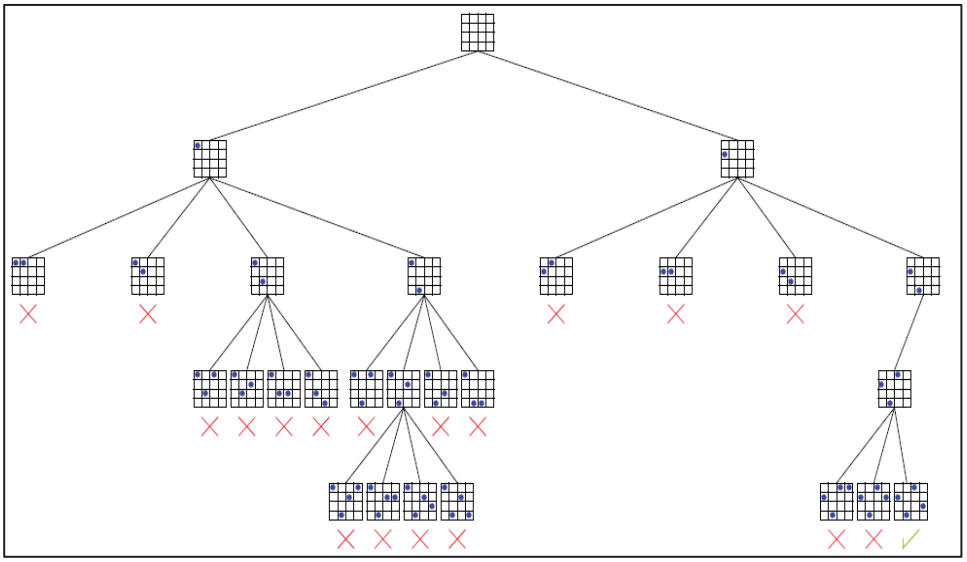

Figura 4.6 – Resolviendo el problema de las N-Reinas con Backtraking

4.4.3 Forward Checking (FC)

Algoritmo de la década del 1980. *El Forward Cheking* consiste en *Backtracking* más una técnica de consistencia, también llamados algoritmos *Look-Ahead*. Son algoritmos que hacen una comprobación hacia adelante en cada fase de la búsqueda, o sea, llevan a cabo ciertas comprobaciones para determinar si los futuros caminos a tomar pueden llevar o no a obtener inconsistencias en las variables futuras y pasadas. En cambio *Backtracking* es un tipo de algoritmo *Look-Back*, el cual comprueba solamente la consistencia de las variables hacia atrás.

El poder realizar una comprobación hacia adelante permite identificar antes situaciones sin salida y además, los valores inconsistentes se pueden descubrir y tener una mejor respuesta para las variables futuras.

Otra ventaja de este algoritmo es que en caso de que el dominio de una variable futura se quede vacío, la instanciación de la variable actual se descarta o deshace y se prueba con otro nuevo valor. En el caso de que ningún valor sea inconsistente entonces se llevará a cabo el *Backtracking* normal.

Entonces, *Forward Checking* garantiza que en cada etapa la solución parcial actual es consistente con cada valor de cada variable futura.

A continuación se ilustra el *Forward Checking*:

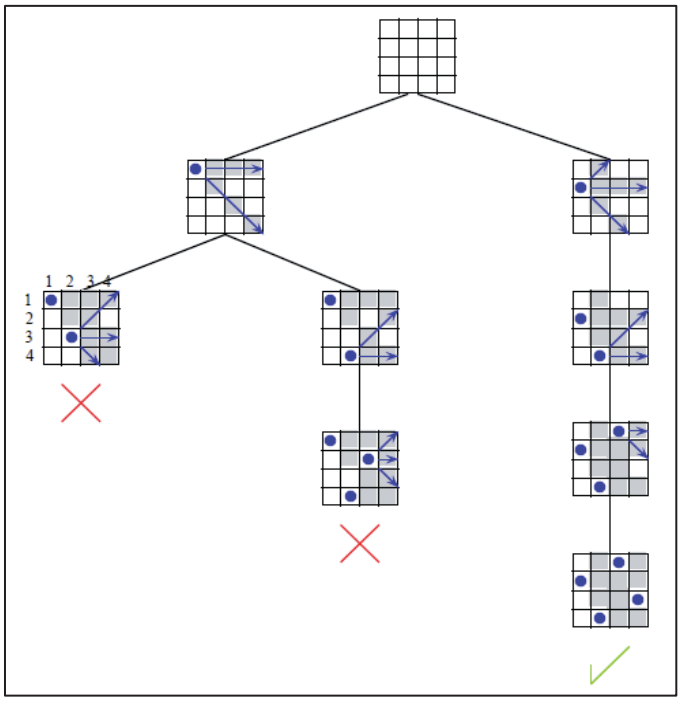

Figura 4.8 – Resolviendo el problema de las N-Reinas con Forward Checking

4.4.4 Maintaining Arc Consistency (Full Look Ahead) (MAC)

Algoritmo de la década del 1990. El *Maintaining Arc Consistency* revisa los conflictos entre las variables futuras y además prueba las variables actuales con las futuras variables en búsqueda de posibles conflictos. Los dominios actuales de cada variable tienen que ser consistentes con todas las restricciones.

Ciertamente, la reducción de los dominios de las variables tiene un costo, el número de comprobaciones a cada paso, incrementa el coste por iteración del algoritmo. Dependiendo del problema, el coste de propagación puede hacer que el coste de hallar una solución sea mayor que utilizar el algoritmo de *Backtracking* cronológico.

Esto hace que sea un algoritmo mucho más complejo de usar y eficiente, pero a la vez de un costo computacional mayor.

A continuación se ilustra el *Maintaining Arc Consistency:*

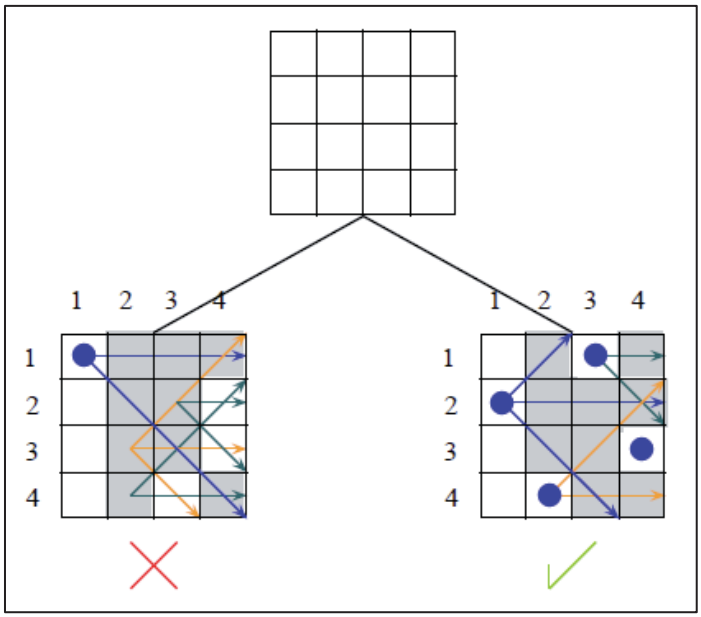

Figura 4.9 - Resolviendo el problema de las N-Reinas con Maintaining Arc Consistency

4.4.5 Branch and Bound (BnB)

Branch and Bound comienza con la representación del total de conjuntos de árboles para los que se necesita encontrar el árbol de menor coste. Este conjunto se parte repetidamente, eliminando todos los miembros de la partición para los que se puede demostrar que, después de su eliminación, todavía existe un árbol de mínimo coste en el resto de conjuntos no eliminados.

El objetivo de BnB es seleccionar una solución que tenga un menor coste en la Función Objetivo. Al igual que BT, BnB poda, no solamente las instanciaciones parciales que son inconsistentes, sino que también los que son evaluados como inferiores a la mejor solución actual. Es decir, que cada valor del nodo de la actual solución parcial es estimado por medio de una evaluación de la función y comparación con la mejor solución actual; si ésta es inferior, la búsqueda por ese camino se termina. Cuando la evaluación de la función es correcta, BnB poda las ramas del árbol de búsqueda.

4.5 Estrategias de Resolución

Algunas técnicas de consistencias y/o heurísticas aplicadas a algoritmos de búsqueda, pueden ayudar notablemente en la resolución tanto de un CSP como un COP. Ayudas que se pueden ver reflejadas en tiempos de cómputos menores o ahorro de recursos.

También existen otros métodos de mejoras para una buena estrategia de resolución, como lo es el orden correcto de la instanciación de las variables y los valores de ellas, los cuales pueden mejorar de gran manera la eficiencia al momento de buscar soluciones.

A continuación se detallarán las estrategias de resolución utilizadas a lo largo del proyecto.

4.5.1 Heurísticas Ordenación de Variables

Existen análisis de muchos entendidos en la materia los cuales prueban que el orden de las variables en el transcurso de la búsqueda puede tener un gran impacto en el tamaño del espacio de búsqueda explorado.

Las heurísticas de ordenación de variables tratan de asignar lo antes posible las variables más restringidas y de esa manera determinar las situaciones sin salida lo antes posible y así reducir el número de vueltas atrás.

4.5.2 Heurísticas Ordenación de Variables Dinámicas

Abordan el problema del tratamiento dinámico del dominio de las variables, que va cambiando durante el proceso de búsqueda con la propagación de restricciones.

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

Existen varias heurísticas de ordenación de variables dinámicas que afrontan este problema. Las utilizadas en este proyecto son las siguientes:

- ***First Fail* (FF):** Este principio se basa en que para tener éxito, se debería intentar primero, probar las variables donde sea más probable que falle. De esta manera las situaciones sin salida pueden identificarse antes y además se ahorra espacio de búsqueda. Hay que seleccionar la variable con dominio más pequeño.

- ***Smallest* (S)*:*** Este principio se basa en que para tener éxito se debería seleccionar la entrada que posea el valor más pequeño, ya que con esto se encontrarán los errores de forma más rápida.

4.5.3 Heurísticas Ordenación de Valores

La idea de las heurísticas de ordenación de valores es seleccionar el valor de la variable actual que tenga más probabilidad de llevar a una solución, es decir, identificar la rama del árbol de búsqueda que sea más probable que obtenga una solución. Algunos ejemplos son:

- ***Indomain* (I):** Este principio se basa en que para tener éxito, los valores deben ser probados en orden creciente. En caso de fallos, los valores previos no son borrados.

- ***Indomain Split* (IS):** Este principio se basa en que para tener éxito, los valores son probados dividiendo varias veces el dominio, probando primero la mitad inferior del primer dominio. En caso de fallar, el intervalo probado es removido.

- ***Indomain Min* (IM):** Este principio se basa en que para tener éxito, los valores, al igual que *Indomain*, deben ser probados en orden creciente. En caso de fallo, los valores probados previamente son borrados.

5 Constrainted Financial Portfolio Selection Problem

El Constraint Financial Portfolio Selection Problem (CFPSP) nace como una necesidad al momento de escoger la mejor cartera para invertir. Cuando una persona desea entrar al mundo de las inversiones debe poseer activos (o acciones) en diversos sectores o compañías y el inversor debe calcular cual es la mejor forma de invertir estos activos para poder obtener la mayor ganancia con el menor riesgo asociado. Se pueden generar muchas combinaciones posibles, pero poder ver cuál de ellas es la más conveniente es un proceso lento. CFPSP nos permitirá obtener la cartera óptima, es decir, nos dirá cuál de todas las carteras es la que diversifica mejor el riesgo y a la vez poder obtener una mayor ganancia.

Cuando hablamos de diversificación, nos referimos al hecho de minimizar el riesgo. En términos de finanzas, la diversificación es una estrategia que consiste en dividir el dinero a utilizar en diferentes acciones, en vez de concentrarlo todo en un solo tipo de inversión. Con esto se puede disminuir el riesgo, o poder tener un mayor control de éste al momento de invertir [9].

5.1 Antecedentes de la Selección de Carteras

En 1952, Harry Markowitz publicó en la revista Journal of Finance un artículo llamado *Portfolio Selection*, en el cual plantea su modelo sobre la selección de carteras. Posteriormente, en 1959 publica el libro *Portfolio Selection, Efficient. Diversification of Investments* [10] en el que detalla y expone en totalidad su modelo.

La Selección de Cartera (*Portfolio Selection*) es un proceso que se basa en el modelo de Harry Markowitz, el cual consiste en seleccionar una cartera de inversiones, la que estará compuesta por un conjunto de títulos. Estos títulos poseen diversas restricciones con las que se deberá trabajar para poder obtener todas las estimaciones necesarias con lo cual se obtendrá la solución óptima para dicha cartera.

Para trabajar con la selección de carteras, debemos tener un Portafolio bien definido. Para ello, debemos considerar los siguientes puntos:

- Definir qué clase de activos van a ser incluidos en el Portafolio.
- Perfil de riesgo que posee el inversor.
- Principios de diversificación.
- Plazo de inversión, es decir, si será en uno o varios períodos.
- Definición del índice de referencia del Portafolio.
- Asignación estratégica de cada clase de activo en el largo plazo.
- Asignación táctica de cada clase de activo en el corto plazo.
- Estrategia de selección a usar dentro de cada clase de activo.

En el primer punto, sobre qué clase de activos serán incluidos en el Portafolio, se refiere a los activos financieros con los cuales se trabajará en el Portafolio. Estos deben ser bonos, efectivo, acciones, etc.

Cuando hablamos del perfil de riesgo que posee el inversor, nos referimos al grado de tolerancia al riesgo. Esto es de vital importancia, ya que servirá para saber que tanto se puede arriesgar al momento de realizar una inversión.

De acuerdo a lo surgido en el punto anterior, se considerará el tercer punto, o sea, los principios de diversificación que se utilizarán, o dicho de otra forma, en qué países se invertirá, bajo qué moneda, etc.

El cuarto punto nos permitirá ver cuán agresiva será nuestra inversión, ya que a mayor plazo de inversión, mayores posibilidades de cumplir con el objetivo de la inversión.

Al hablar del índice de referencia del Portafolio, lo que se está diciendo es que, se debe tener un punto de comparación al momento de ir efectuando nuestras inversiones, ya que esto nos ayudará a saber que tan bien se han producido nuestras elecciones.

Una vez que todos los puntos anteriores estén definidos, debemos pasar a la asignación estratégica, la cual refleja la actitud que se tomará para con el Portafolio en un largo plazo, entendiéndose este entre 4 a 7 años. Paralelamente, se trabajará en la asignación táctica, la cual se encargará de los resultados a corto plazo de cada activo, y que los posicionará en la parte superior o inferior del rango. Esto es lo que se conoce como *stock picking*. Por último, está la estrategia de selección, que se utiliza para saber cómo tratar cada activo y qué se puede esperar de cada uno de ellos.

Cuando ya tenemos nuestro Portafolio constituido, debemos tener en cuenta un punto muy importante; la Frontera Eficiente. Ésta, es una línea que se calcula teniendo un máximo retorno dado un nivel de riesgo, o dicho de otra forma, el mínimo riesgo dado un nivel de retorno esperado. Las carteras que se encuentran sobre esta línea, o Frontera Eficiente, son las carteras deseadas por los inversores.

Como se muestra en la figura 5.1, el riesgo se encuentra medido en el eje X y el retorno esperado se encuentra en el eje Y. La curva XYZ representa carteras eficientes. Las carteras que se encuentre bajo o a la derecha de esta curva serán ineficientes y entregarán retornos inferiores dado cada nivel de riesgo.

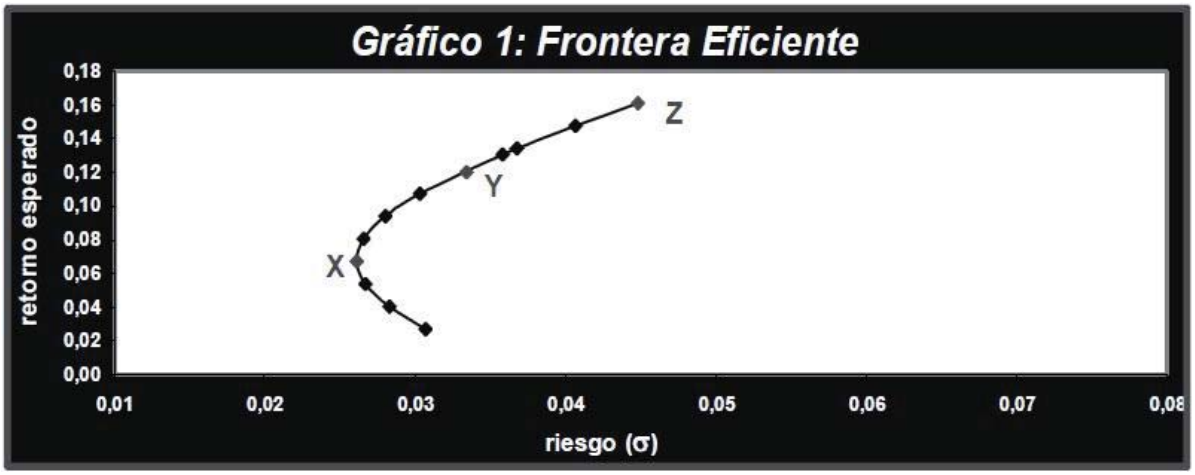

Figura 5.1 Ejemplo de Frontera Eficiente

La Selección de Cartera es un proceso que posee dos tendencias fuertemente diferenciadas entre ellas. La primera consta de la estrategia activa, la cual utiliza la información disponible y las técnicas de proyecciones para obtener rendimientos superiores a los de un Portafolio que se encuentra diversificado. Esta estrategia se basa en la posibilidad de lograr identificar los valores sobrevalorados o infravalorados con los cuales poder generar un rendimiento suficiente para cubrir los costos de transacción y riesgo asumido.

La segunda tendencia corresponde a la estrategia pasiva supone el cumplimiento de la hipótesis de eficiencia del mercado. En otras palabras, el precio de cotización de título refleja toda la información existente en el mercado [11].

En años posteriores, surgieron nuevos modelos paralelos al expuesto por Markowitz. En ellos podemos mencionar a William F. Sharpe con su publicación llamaba *A Simplified Model for Portfolio Analysis;* Elton, Gruber y Padberg; Konno y Yamazaki entre otros.

5.2 Markowitz y su Modelo

Markowitz desarrolla un modelo de acuerdo a las necesidades del inversor, las que son rechazar el riesgo y maximizar la rentabilidad. Es decir, el modelo será eficiente si entrega una cartera que maximice la rentabilidad o proporcione el menor riesgo posible para una rentabilidad. Para esto se debe resolver el siguiente programa cuadrático:

Minimizar $$f(X) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sigma_{ij} x_i x_j$$

Sujeto a $$\sum_{i=1}^{n} r_i x_i \geq R$$

$$\sum_{i=1}^{n} x_i = 1$$

$$0 \leq x_i \leq 1 \ (i = 1, \dots, n)$$

donde x_i corresponde al porcentaje de inversión de i con respecto del total de la cartera; f(x) corresponde a la varianza de la cartera p, y σ_{ij}, la covarianza entre los activos a_i y a_j.

La teoría de Markowitz se sustenta bajo las siguientes hipótesis:

- La rentabilidad de cualquier cartera es un valor aleatorio, obtenido a partir de las cotizaciones de dicho valor o Portafolio, y cuya distribución de probabilidad es conocida por el inversionista. El modelo acepta como rentabilidad, la esperanza matemática de dicho activo.
- La medida del riesgo corresponde a la dispersión de la rentabilidad de una cartera, medida por la varianza.
- El inversionista elegirá aquella cartera que posea una mayor rentabilidad y un menor riesgo.

- Si ocurriera el caso de existir dos carteras con el mismo riesgo, el inversionista elegirá la cartera que le aporte una mayor rentabilidad.

Markowitz nos demuestra que la clave para diversificar un Portafolio no está sólo en el número de acciones que componen la cartera, sino que también y más importante aún, en la correlación de los retornos de los activos que la componen. Si los retornos están fuertemente correlacionados, el Portafolio no se podrá diversificar, ahora, si la correlación entre los activos es baja, si se podrá diversificar y el riesgo será mucho menor.

Un inversionista puede calcular las covarianzas entre las acciones que componen la cartera. Con esta información, Markowitz demostró que se pueden construir Portafolios que sean eficientes. Debemos entender como Portafolio eficiente a todas aquellas carteras que en el pasado presentaron un retorno mayor al nivel de riesgo.

En cuanto a la Frontera Eficiente, en ella se sitúan las mayores rentabilidades para un riesgo asociado. Cabe destacar que a mayor riesgo, mayor es la rentabilidad esperada. Es por esto que, de acuerdo al grado de aversión al riesgo que posea el inversor, éste debe situarse en un punto de la Frontera. Apostar a cualquier punto fuera de esta Frontera Eficiente sería irracional.

Por todo esto, cuando un inversor posee una cartera y desea saber cuál es la mejor forma de invertir, debe utilizar este modelo, ya que le permitirá conocer los distintos resultados dependiendo de las proporciones a invertir de cada activo y de su nivel de aceptación al riesgo.

5.3 Formalización como un COP

El principal objetivo del CFPSP en un conjunto de carteras es seleccionar un Portafolio y obtener, junto a las restricciones propias de dicha cartera y guiándose por el modelo de Markowitz, los resultados necesarios para poder elegir la cartera óptima. Para conseguir esto, se trabajó con dos solver los que nos proporcionaron los resultados necesarios para poder realizar la comparación.

En este proyecto, el problema de la selección de cartera se consideró como un problema de optimización con restricciones. Dicho esto, se procederá a especificar las variables, constantes, Función Objetivo, restricciones y dominio:

Constante

- N, número de títulos que componen la cartera.

Cuando hablamos de los títulos o activos que componen una cartera, nos referimos a las acciones, bonos, opciones financieras, etc. que posee una persona, en este caso, el inversor.

La cartera corresponde al conjunto de títulos con los cuales se realizará la inversión.

Variables y Dominio

- x_i: Cada x_i representa la fracción invertida en los activos a_i.
- σ_{ij}: Corresponde a la covarianza de los activos (a_i, a_j).
- f(x): Es la varianza de la cartera.
- R: Retorno esperado de la cartera.
- r_i: Es el retorno esperado del activo a_i.

El dominio utilizado fue discreto. Cuando se habla de xi, se refiere a una variable que se utilizará dentro de la función matemática. Esta cartera estará compuesta por n títulos, donde cada título tendrá un porcentaje de inversión dependiendo del grado de ganancia que se pueda obtener de cada uno. x_i contendrá el porcentaje a invertir de cada título, es decir, si nuestra cartera está compuesta por 5 títulos, del primer activo (x_1) se invertirá un 15%, del segundo activo (x_2) se invertirá un 10%, y así sucesivamente hasta llegar a x_5.

La covarianza es el grado en que dos variables se mueven en la misma dirección, o en direcciones opuestas. En otras palabras, si dos variables se mueven en el mismo sentido diremos que poseen una covarianza positiva, en caso contrario, tendrán una covarianza negativa. Si la covarianza es positiva, lo que se conseguirá es aumentar el riesgo al momento de invertir. Por lo tanto, lo que se busca es que la covarianza sea negativa o cercana a 0.

La varianza mide la variabilidad de la rentabilidad de un título. Si, por ejemplo, se posee una cartera con 5 títulos, se debe tener en cuenta la varianza de cada título.

El retorno corresponde al porcentaje de ganancia que se espera conseguir con la inversión.

La variable r_i busca obtener los porcentajes de ganancias que se esperan de cada activo. La suma de estos porcentajes corresponde al retorno esperado de la cartera.

Función Objetivo

La Función Objetivo del CFPSP tiene como fin minimizar los riesgos al momento de elegir una cartera. Es decir, una cartera será óptima si los riesgos de seleccionar esta cartera son los mínimos posibles. Para esto, se debe trabajar con la siguiente función:

Minimizar $$f(X) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sigma_{ij} x_i x_j$$

Restricciones

$$\sum_{i=1}^{n} r_i x_i \geq R$$

$$\sum_{i=1}^{n} x_i = 1$$

La primera restricción permite obtener la rentabilidad esperada para poder conocer cuál es la cartera más rentable, es decir, obteniendo los porcentajes de inversión de cada activo de la cartera sabremos si esta cartera nos proporcionará una rentabilidad igual o mayor a la esperada, en caso contrario la cartera es desechada ya que no cumple con el retorno esperado para dicha inversión. La segunda restricción proporciona el presupuesto del inversor. Para obtener esto, necesitamos conocer cuántos títulos componen la cartera, donde dicho número nos entregará el límite de estimaciones a realizar. Se debe tener en cuenta que estas restricciones tienen como objetivo limitar el acceso de las carteras a la Función Objetivo, es decir, si una cartera no cumple con una de las restricciones, no es candidata para optar a ser la cartera óptima.

La covarianza es un indicador de riesgo. Cuando hablamos de covarianza, nos referimos al grado de relación o asociación entre dos variables, en este caso *i* y *j*; además indica el sentido de la correlación entre las variables.

Si $\sigma_{ij} > 0$ la correlación es directa.

Si $\sigma_{ij} < 0$ la correlación es inversa.

En el caso del Modelo de Markowitz es importante el riesgo de la correlación entre éstos. Aquí, hay que buscar correlaciones negativas o menores a 0, ya que esto hará disminuir el riesgo. En otras palabras, se espera que la rentabilidad tenga un comportamiento inverso.

Cabe destacar que este modelo está hecho para cualquier tipo de cartera. Cuando uno evalúa las carteras más eficientes en términos de retorno, en la teoría están sustentadas en carteras con muchos activos que tienen correlaciones negativas, o sea si se buscan correlaciones positivas lo que se hará es concentrar el riesgo.

La minimización se realiza con el objetivo de disminuir el grado de riesgo que hay. Para esto se debe tener en cuenta la covarianza existente entre los activos, es decir, cómo interactúan los activos entre sí. Cuando la covarianza es mayor a 0, las rentabilidades estarán asociadas positivamente y por tanto habrá un comportamiento directo, ocasionando que se concentre el riesgo; cuando la covarianza es menor que 0 el comportamiento será indirecto. Es por esto que cuando uno minimiza, lo que se busca es que la covarianza entre los activos se acerque a -1, es decir, que se relacionen inversamente, de tal forma que no importa cómo se mueva un activo, éste no afectará a los demás activos que compongan la cartera.

6 Eclipse

La primera generación de lenguajes de programación con restricciones se centra en una sola técnica: propagación de restricciones. Esta técnica ha demostrado su eficacia en una variedad de aplicaciones.

Eclipse está diseñado para ser más que una implementación de CLP (Constraint Logic Programming, Programación Lógica con Restricciones en español), también es compatible con la programación matemática. La ventaja fundamental de Eclipse es que permite al programador utilizar una combinación de algoritmos apropiados para la aplicación en cuestión. Éste solver está diseñado para resolver complejos problemas combinatoriales en las áreas de planificación, programación y asignación de recursos. Además, entrega la posibilidad de enfocarse en dos aspectos: la búsqueda y las restricciones.

Dentro de las bibliotecas de Eclipse se incluyen la propagación finita de dominio, la propagación y el intervalo de resolución de restricción lineal [12].

Eclipse ha sido diseñado para resolver problemas combinatoriales, los que se caracterizan por tener un conjunto de decisiones que realizar y un conjunto de restricciones entre las decisiones. Cada decisión se basa en una variable y cada elección tiene un valor para esa variable.

Los requisitos de Eclipse son de dos tipos:

- El tipo de problema a modelar sea simple y natural.
- Permitir que el modelo del problema resultante sea resuelto de manera eficiente.

La programación lógica es apta para el modelado de problemas combinatoriales, ya que se basa en las relaciones y es compatible con las variables lógicas.

6.1 Prolog

La programación lógica tiene sus inicios en la automatización de la demostración de teoremas basados en el método de resolución del matemático, científico y profesor de Universidad de Siracusa, Estados Unidos (EEUU), Alan Robinson, quien en su obra principal, propuesta el año 1965 [13], introduce el principio de resolución, la noción y el algoritmo de unificación. Usando su resolución, se pueden probar teoremas dados como fórmulas de lógica de primer orden, con el fin de obtener un “sí” o “no” como respuesta a una pregunta, aunque bajo este modelo, no era posible realizar cálculos con una respuesta de este tipo [14].

Para resolver este problema Robert Kowalski, científico estadounidense, propuso una versión modificada de la resolución, que consistía en un subconjunto de la lógica de primer orden que permitía generar substituciones que en últimas satisfacían la fórmula original. Esta substitución puede ser interpretada como el resultado del cálculo. Este acercamiento finalmente se llamó Programación Lógica.

Por otro lado, en Marsella Francia, un científico de computación y profesor de la universidad de Montreal, Alain Colmerauer trabajaba en un lenguaje de programación para el procesamiento del lenguaje natural basado en la demostración del teorema automatizado, lo que lo llevó a la creación de Prolog en la década del ’70. Colmerauer, en su experimentación con Prolog, solucionó algunas de sus más importantes limitaciones, por ejemplo, podía resolver ecuaciones entre términos, pero no entre cadenas. Usando la terminología actual, Prolog solo soporta un solucionador de restricciones. Esto lo llevó a diseñar los sucesores de Prolog, Prolog II [14], Prolog III y Prolog IV [15].

Prolog III fue uno de los primeros sistemas de tipo CLP, el cual fue pionero en reemplazar los procedimientos unificados de la Programación Lógica por lineamientos más generales para resolución de restricciones. En Prolog III, se pueden plantear restricciones sobre listas, valores booleanos y reales, siendo considerado como la primera realización de la Programación Lógica con Restricciones.

6.1.1 Diferencias entre el lenguaje CLP y Prolog

- En los programas de Prolog, los dominios de variables son declarados implícitamente por elementos de las listas dispersas en varios predicados en diferentes lugares del programa. Los programas CLP contienen, en su parte superior, dominios declarados explícitamente para todas las variables utilizadas en el programa.

- Prolog puede manejar sólo las variables definidas sobre dominios de los términos. Los lenguajes CLP son capaces de manejar las variables en un grado más amplio, por ejemplo, dominios enteros, dominios reales y los dominios simbólicos.

- En Prolog, la propagación de restricciones se ha hecho a través de la unificación. Los lenguajes CLP utilizan métodos más eficientes para la propagación de restricciones, conocidas como *consistency techniques* (técnicas de consistencia en español).

- Los lenguajes CLP utilizan métodos de búsqueda más eficientes en comparación con la búsqueda “primero en profundidad” que con *Backtracking Standart*. Los métodos más eficientes son, entre otros *Forward Checking* y *Forward Checking-Lookingahead*.

- En los programas Prolog, una búsqueda se inicia automáticamente cada vez que una consulta se invoca. Para los lenguajes CLP una búsqueda se inicia por un predicado especial (generalmente incorporado) que sitúa las variables en un cierto orden, más conocido como labeling/ 1. Las propiedades básicas de este predicado labeling corresponden a la regla:

```
1   labeling([H|T]) : -
2      indomain(H).
3       labeling(T).
4   labeling([]).,
```

Donde el predicado indomain (variable) sitúa sucesivamente los valores de su dominio, en el orden que aparecen en la definición de dominio, de izquierda a derecha. Este orden a veces no es la más eficiente, de este modo los lenguajes CLP (como Eclipse) ponen a disposición una serie de búsquedas heurísticas diferente a la realizada por el labeling/ 1.

6.2 Sintaxis en Eclipse

Eclipse basa su sintaxis en Prolog, este último utilizado para resolver problemas en los que existen objetos y relaciones entre objetos. La programación en Eclipse consistirá en declarar hechos sobre los objetos y sus relaciones, definir reglas sobre dichos objetos y relaciones y por último hacer preguntas.

6.2.1 Sistema Lógico

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

• **Hechos:** Conjunto de elementos que son lógicamente verdad.

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

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

6.2.1.1 Hechos

Un Hecho es una relación entre objetos. Su sintaxis en Eclipse es:

relación (objeto, objeto,…).

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

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

Por ejemplo, un hecho puede ser: persona **(camila, 23).**

6.2.1.2 Reglas

Un Regla es cuando la verdad de un hecho depende de la verdad de otro hecho o de un grupo de hechos. Una regla consiste en una cabeza y un cuerpo. El cuerpo puede estar formado por varios hechos u objetivos [16]. Su sintaxis es:

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

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

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

6.3 Ejemplos de Eclipse

6.3.1 Problema de "N-Reinas"

Como se describe en la sección 4.2.3 el problema de las N-reinas posee ciertas restricciones que se deben cumplir para poder encontrar la solución óptima. A continuación se mostrará la solución a este problema, desarrollado en Eclipse.

```
1  :-lib(ic).
2  queens(N,Board):-
3  dim(Board, [N]),
4  Board[1..N] :: 1..N,
5  ( for(I,1,N), param(Board,N) do
6    ( for(J,I+1,N), param(Board,I) do
7        Board[I] #\= Board[J],
8        Board[I] #\= Board[J]+J-I,
9        Board[I] #\= Board[J]+I-J
10   )
11  ),
12  labeling(Board).
```

Figura 6.1 Ejemplo del problema de N-reinas en Eclipse

6.3.2 Problema del "Send + More = Money"

Otro clásico ejemplo es el conocido Send+ More = Money, explicado en la sección 4.2.1. A continuación se entregará una solución por medio de Eclipse.

```
1  :-lib(ic).
2  send(L) :-
3  L = [S,E,N,D,M,O,R,Y],
4  L :: [0..9],
5  S #\= 0,
6  M #\= 0,
7  alldifferent(L),
8  1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E
9  #= 10000*M + 1000*O + 100*N + 10*E + Y,
10 labeling(L).
```

Figura 6.2 Problema "Send + More = Money" en Eclipse

7 Modelado en Eclipse

Un modelo, es una representación matemática simplificada de una realidad compleja. Frecuentemente, los sistemas o conjuntos de procesos y subprocesos, son difíciles de comprender, por lo cual el modelado nos da la oportunidad de organizar y documentar la información existente sobre nuestro sistema.

En el caso de los problemas de optimización, existen variadas herramientas que nos permiten dar una mejor solución a nuestros problemas. Pero a pesar de esto, los métodos de optimización no han sido totalmente explorados en el ámbito financiero. Un ejemplo es poder escoger la mejor cartera de inversión, la cual nos ofrezca un mayor retorno con el menor riesgo posible. A raíz de eso surge CFPSP, como solución a este tipo de problemas; donde el modelado de éste se realizará bajo el solver Eclipse.

7.1 Representación del CFPSP en Eclipse

La representación del CFPSP consiste en una representación codificable bajo los términos y sintaxis del solver con restricciones en Eclipse, es decir, dada la formulación del CFPSP como un problema de optimización, establecer el modelo en base a variables, constantes, dominios y restricciones bajo los términos del solver Eclipse. Dado esto, primero se presentará el código sujeto a sus restricciones. Posteriormente, se explicará cada parte del código expuesto anteriormente.

7.1.1 Codificación del CFPSP en Eclipse

- **Vector_RI:** En este vector se encuentran los porcentajes (en valores enteros) correspondientes a los retornos esperados de cada activo, es decir, en cada casilla del vector se encontrará el porcentaje de retorno que se espera conseguir por activo.

```
13  cfpsp(Vector_XI, R):-
14
15  Activos = 4,
16  Sum = 100,
17  Vector_RI = [](16, 40, 20, 20),
18
19  Matriz_covarianzas = []([]( 28, 20, 13, 20),
20                          []( 20, 16, 20, 28),
21                          []( 13, 20, 71, 19),
22                          []( 20, 28, 19, 40)),
23
24  dim(Vector_XI,[Activos]),
25  Vector_XI[1..Activos] :: 1..100,
```

Figura 7.1 Declaración del Vector_RI

- **Matriz_covarianza:** En esta matriz se encuentran los valores obtenidos mediante la covarianza entre los activos, es decir, en la posición 1,2 de la matriz, se encuentra la covarianza existente entre el activo 1 y el activo 2. En el caso de la diagonal, ésta se calcula mediante la varianza de activo correspondiente, es decir, en la posición 1,1 se encuentra el valor de la varianza obtenido con los activos 1 y 1.

```
17  Vector_RI = [](16, 40, 20, 20),
18
19  Matriz_covarianzas = []([]( 28, 20, 13, 20),
20                          []( 20, 16, 20, 28),
21                          []( 13, 20, 71, 19),
22                          []( 20, 28, 19, 40)),
23
24  dim(Vector_XI,[Activos]),
25  Vector_XI[1..Activos] :: 1..100,
```

Figura 7.2 Declaración de la Matriz_covarianza

- **Vector_XI:** Este vector contendrá la proporción de cada activo, es decir, se encontrarán los porcentajes (en valores enteros) que se debe invertir en cada activo. Cuando hablamos de invertir en activos (títulos), se debe considerar qué porcentaje de cada activo se va a invertir, ya que no es recomendable invertir todos los activos en una sola cartera (ver capítulo 6).

- **Primera Restricción:** Esta restricción consiste en que la suma de los valores obtenidos en el Vector_XI debe ser 1. Si esta restricción no se cumple, se debe buscar otro Vector_XI que si cumpla con lo pedido. Cabe destacar que para poder realizar la suma de los valores del vector, éste debe ser pasado a una lista. A continuación veremos la restricción en su expresión matemática, para luego verla traspasada a código.

Expresión Matemática	Codificación

$\sum_{i=1}^{n} x_i = 1$	`24  dim(Vector_XI,[Activos]),`
	`25  Vector_XI[1..Activos] :: 1..100,`
	`26`
	`27  flatten_array(Vector_XI, Vector_V),`
	`28  sum(Vector_V) #= Sum,`

- **Segunda Restricción:** Esta restricción considera que la suma de la multiplicación de los vectores (Vector_XI y Vector_RI) sea igual o superior al retorno esperado de la cartera (R). En caso de no cumplirse esta restricción, se debe volver a buscar un Vector_XI el cual debe cumplir la primera restricción para poder pasar a la segunda restricción. A continuación veremos la expresión matemática de esta restricción, acompañada de su codificación.

Expresión Matemática	Codificación

$\sum_{i=1}^{n} r_i x_i \geq R$	```
	(for(J,1,4),
	  fromto(0,In,Out,Total_1),
	  param(Vector_RI, Vector_XI) do
	  X is Vector_RI[J],
	  Z is Vector_XI[J],
	  Out #= In + X*Z
	),
	R :: 10..25,
	R is 11,
	Total_1 #>= R,
	``

- **Función Objetivo:** Para llegar a la Función Objetivo, primero se deben cumplir las dos restricciones mencionadas anteriormente. La Función Objetivo es realizada por un doble for en el cual se realiza la multiplicación de la Matriz_covarianza y el Vector_XI en doble instancia. Esta función se debe minimizar, para disminuir el riesgo al momento de invertir.

Expresión Matemática	Codificación

Minimizar
$f(X) = \sum_{i=1}^{n}\sum_{j=1}^{n} \sigma_{ij} x_i x_j$
	``
	(for(I,1,Activos) * for(J,1,Activos),
	  fromto(0,In,Out,Total_2),
	  param(Matriz_covarianzas,Vector_XI) do
	  A is Matriz_covarianzas[I,J],
	  B is Vector_XI[I],
	  C is Vector_XI[J],
	  Out #= In + A*B*C
	),
	``

Acabamos de ver las reglas que componen el CFPSP en Eclipse, por lo tanto a continuación se especificarán las variables y constantes del modelo, además del dominio utilizado.

- **Constante Activos**: Corresponde al número de activos (títulos) que compondrán la cartera.

Esta constante estará presente en casi todo el programa. Primeramente se encuentra en la declaración del Vector_XI, y en la declaración de uno de los for de la Función Objetivo. Esto se puede apreciar en las siguientes figuras:

```
24  dim(Vector_XI,[Activos]),
25  Vector_XI[1..Activos] :: 1..100,
```

Figura 7.3 Representación de la utilización de la constante

```
41  ( for(I,1,Activos) * for(J,1,Activos),
42     fromto(0,In,Out,Total_2),
43     param(Matriz_covarianzas,Vector_XI) do
44          A is Matriz_covarianzas[I,J],
45          B is Vector_XI[I],
46          C is Vector_XI[J],
47          Out #= In + A*B*C
48 ),
```

Figura 7.4 Representación de la constante en el modelo

```
27  flatten_array(Vector_XI, Vector_V),
28  sum(Vector_V) #= Sum,
```

Figura 7.5 Representación del uso de la constante en el modelo

Variables y Dominio

- **x_i:** Es el porcentaje de *i* con respecto al total de la cartera, donde *i* corresponde al activo financiero (1,…, n), con dominio (1,..., Activos), donde Activos está definido dentro del programa.

Esta variable es utilizada en casi todo el programa, ya que corresponde a los porcentajes de inversión de cada activo (título) que compone la cartera con la cual se está trabajando.

```
24  dim(Vector_XI,[Activos]),
25  Vector_XI[1..Activos] :: 1..100,
```

Figura 7.6 Utilización de la variable x_i

- **σ_{ij}:** Corresponde a la covarianza de los activos (a_i, a_j), con dominio (1,…, 100).

La variable σ_{ij} se utiliza en la Función Objetivo, y en la creación de la Matriz_covarianza, la cual es calculada fuera del programa y no es ingresada por pantalla al sistema, sino que una vez calculada, se ingresa directamente en el código.

```
19  Matriz_covarianzas = []([]( 28, 20, 13, 20),
20                          []( 20, 16, 20, 28),
21                          []( 13, 20, 71, 19),
22                          []( 20, 28, 19, 40)),
```

Figura 7.7Utilización de la variable σ_{ij}

- **f(x):** Es la varianza de la cartera, utilizada en la Función Objetivo.

Esta variable, que en el programa es llamada Total_2, se utiliza únicamente para guardar el resultado obtenido en la Función Objetivo; resultado que debe ser mostrado por pantalla una vez finalizado el programa.

```
41  ( for(I,1,Activos) * for(J,1,Activos),
42     fromto(0,In,Out,Total_2),
43     param(Matriz_covarianzas,Vector_XI) do
44           A is Matriz_covarianzas[I,J],
45           B is Vector_XI[I],
46           C is Vector_XI[J],
47           Out #= In + A*B*C
48  ),
```

Figura 7.8 Utilización de la variable f(x) (Total_2 en el programa)

- **R:** Retorno esperado de la cartera.

Utilizaremos esta variable en la segunda restricción, donde tendrá como función poner un límite inferior a la suma de los porcentajes de retornos esperados para cada activo.

```
30  ( for(J,1,4),
31     fromto(0,In,Out,Total_1),
32     param(Vector_RI, Vector_XI) do
33          X is Vector_RI[J],
34          Z is Vector_XI[J],
35          Out #= In + X*Z
36  ),
37  R :: 10..25,
38  R is 11,
39  Total_1 #>= R,
```

Figura 7.9 Uso de la variable R en el programa

- **r_i:** Es el retorno esperado del activo a_i, con dominio (1,…, Activos).

El uso de esta variable se verá en la segunda restricción y en la Función Objetivo. Los valores que se almacenarán en el vector que utiliza a esta variable no serán ingresados al programa, ya que se encontrarán definidos en el código.

```
17  Vector_RI = [](16, 40, 20, 20),
```

Figura 7.10 Declaración del vector que utiliza a la variable r_i

Restricciones

La definición de las restricciones en CFPSP en Eclipse, al igual que en otros solvers, debe tener un tratamiento especial ya que nos ayudan en la resolución de cualquier problema de optimización con restricciones. La sintaxis utilizada en Eclipse para las restricciones es la siguiente:

- Menor que, se escribe como #<
- Mayor que, se escribe como #>
- Menor o igual que, se escribe como #=<
- Mayor o igual que, se escribe como #>=
- Igual, se escribe como #=
- No igual, se escribe como #\=

7.2 Software de desarrollo

Se utilizaron 2 programas para el desarrollo del programa; TkEclipse y Notepad++

• **TkEclipse:** Es el compilador del código generado en un editor. Se utilizó por ser libre, de manejo más accesible y gratuito. Utilizando versión 6.0 #159 (i386_nt). Descargada directamente del sitio web de los desarrolladores [17].

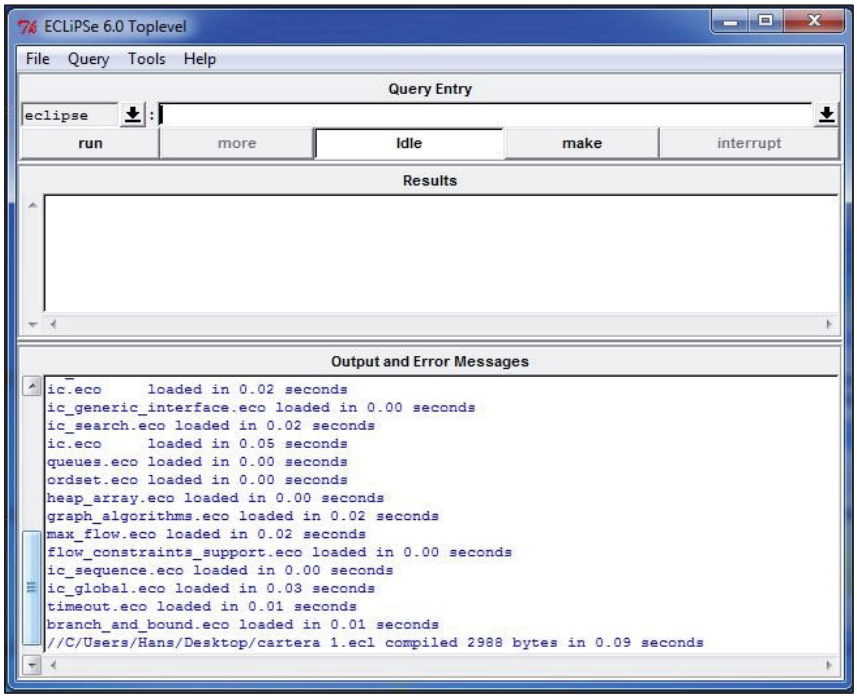

Figura 7.11 – Compilador para Eclipse, TkEclipse

• **Notepad++:** Se utilizó este editor de texto por su distribución libre y bajo licencia GNU. La versión fue la 5.9.2. Éste, es un editor que ayuda al desarrollador, destacando distintos colores o tipografías, sentencias, valores numéricos propios del lenguaje Prolog, lo que ayuda a mejorar la comprensión del código fuente.

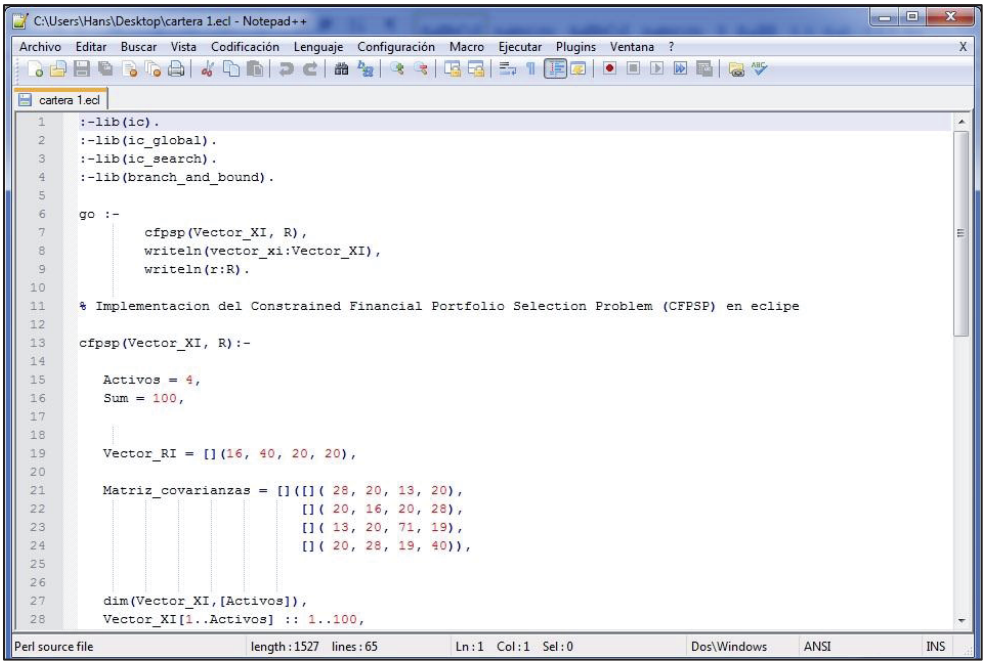

Figura 7.12 – Editor de texto, Notepad++

8 Minizinc

Minizinc es un solucionador el cual ha sido desarrollado para la resolución de optimización con restricciones y problemas de decisión sobre números enteros y reales [18].

Los problemas resueltos en Minizinc, se componen principalmente, de dos partes. La primera corresponde al modelo, que es donde se describe la estructura de una clase de problema; la segunda parte son los datos, que especifican un problema en particular dentro de esta clase. El emparejamiento de un modelo con un conjunto de datos en particular es un ejemplo del modelo, a veces llamado instancia [19].

Los datos y el modelo pueden estar en archivos separados. Los archivos de datos sólo pueden contener asignaciones a los parámetros declarados en el modelo. Un usuario especifica los archivos de datos en la línea de comandos, en lugar de nombrarlos en el archivo de modelo, por lo que éste no está vinculado a ningún archivo de datos en particular.

8.1 Operadores relacionales

Minizinc, al igual que todos los lenguajes de modelado, posee operadores relacionales. Estos son:

- Igual =
- No !=
- Estrictamente menor que <
- Estrictamente mayor que >
- Menor o igual que <=
- Mayor o igual que >=

8.2 Parámetros

Los parámetros se asemejan las variables en un lenguaje de programación estándar. Se les debe asignar un valor y esto se realiza durante la etapa de compilación. Los parámetros se declaran con un tipo o rango. Se pueden declarar de las siguientes formas:

- int: i=3;
- par int: i=3;
- int: i; i=3;

8.3 Variables de decisión

Las variables de decisión son como las variables en matemáticas. Se declaran con un tipo y una palabra clave *var*. Su valor es calculado por Minizinc, con el fin de satisfacer el modelo. Por lo general, se declaran mediante una serie o un conjunto en lugar de un nombre de tipo. El rango da el dominio de la variable. Se pueden declarar de las siguientes formas:

- var int: i; constraint i >=0; constraint I <=4;
- var 0..4: i;
- var {0,1,2,3,4}: i;

8.4 Ejemplos en Minizinc

8.4.1 Problema de las N-Reinas

En la figura 10.1 se puede ver la codificación del problema de las N-Reinas bajo sintaxis Minizinc, en donde la explicación del problema se realizó en la sección 4.23.

```
1  int: n;
2  array [1..n] of var 1..n: q;
3  predicate
4  noattack(int: i, int: j, var int: qi, var int: qj) =
5     qi     != qj     /\
6     qi + i != qj + j /\
7     qi - i != qj - j;
8
9  constraint
10  forall (i in 1..n, j in i+1..n) (
11    noattack(i, j, q[i], q[j])
12  );
13
14  solve ::
15  int_search(
16             q,
17             "first_fail",
19             "indomain_min",
20             "complete"
21  )
22  satisfy;
23
24  output  [show(n)] ++ [" queens, CP version:"] ++
25        [       if j = 1 then "\n" else "" endif ++
26  show_cond(q[i] = j, "Q ", ". ")
27        |       i, j in 1..n
28  ];
```

Figura 8.1 Problema de las N-Reinas bajo sintaxis Minizinc

8.4.2 Problema del “Send + More = Money”

Otro clásico ejemplo es el conocido Send + More = Money, explicado en la sección 4.2.1. A continuación se entregará una solución por medio de Minizinc

```
1  include "globals.mzn";
2  var 0..9: S;
3  var 0..9: E;
4  var 0..9: N;
5  var 0..9: D;
6  var 0..9: M;
7  var 0..9: O;
8  var 0..9: R;
9  var 0..9: Y;
10  constraint
11  all_different([S,E,N,D,M,O,R,Y]) /\
12     1000*S + 100*E + 10*N + D +
13     1000*M + 100*O + 10*R + E =
14     10000*M + 1000*O + 100*N + 10*E + Y
15  /\
16  S > 0 /\
17  M > 0;
18  solvesatisfy;
19  output [" ", show(S), show(E), show(N), show(D),"\n",
20  " + ", show(M), show(O), show(R), show(E),"\n",
21  " =
22  ];
```

Figura 8.2 Problema del “Send+More=Money” bajo sintaxis Minizinc

9 Modelado en Minizinc

Como se explicó en el capítulo 7, el modelado de un problema corresponde a una herramienta de ayuda para poder entender dificultades de mayor envergadura. Un modelo de un sistema es, básicamente, una herramienta que nos ayuda a resolver algunas interrogantes del sistema, sin tener la necesidad de recurrir al experimento con él mismo.

El modelado corresponde a una representación simplificada de una realidad el cual, por medio de herramientas, organización y documentación de la información nos permiten poder entender a cabalidad el problema que se está tratando que, a diferencia del capítulo 9, se desarrollará bajo la sintaxis y estructura del solver Minizinc.

9.1 Representación del CFPSP en Minizinc

Tal como se explicó en el capítulo 9, la representación del CFPSP consiste en una transferencia de términos matemáticos, a una codificación bajo sintaxis y términos de un solver en específico, en este caso Minizinc, con el fin de poder entregar al lector otra representación del CFPSP pero basado en otra plataforma.

9.2 Codificación del CFPSP en Minizinc

- **Vector_ri:** En este vector estarán contenidos los retornos esperados para cada activo. Es decir, cuánto se espera recibir por cada acción que compone la cartera. Estos retornos estarán expresados en números enteros.

```
1  include "globals.mzn";
2
3  int: activos = 4;
4  int: suma = 100;
5
6  array[1..num activos] of int: vector ri = [7,8,9,10]
7  array[1..activos, 1..activos] of int:
8  Matriz_covarianzas =
9  array2d(1..activos, 1..activos,
10       [
11        40, 4, 6, 2,
12        4, 60, 3, 1,
13        6, 3, 85, 8,
14        2, 1, 8, 100,
15       ]);
```

Figura 9.1 Declaración del vector_ri

- **Matriz_covarianza:** Esta matriz tendrá almacenados los valores correspondientes a las varianzas y covarianzas obtenidas del cálculo del vector_ri y vector_xi.

```
1  include "globals.mzn";
2
3  int: activos = 4;
4  int: suma = 100;
5
6  array[1..num activos] of int: vector ri = [7,8,9,10]
7  array[1..activos, 1..activos] of int:
8  Matriz_covarianzas =
9  array2d(1..activos, 1..activos,
10       [
11        40, 4, 6, 2,
12        4, 60, 3, 1,
13        6, 3, 85, 8,
14        2, 1, 8, 100,
15       ]);
```

Figura 9.2 Declaración de la Matriz_covarianza

- **Vector_xi:** En este vector se encontraran los porcentajes que se debe invertir por cada activo que compone la cartera. Es decir, que porcentaje de la acción se debe invertir. Estos valores estarán en números enteros y serán entregados por el solver.

- **Primera Restricción:** Esta restricción consiste en que la suma obtenida de la multiplicación de cada posición de los vectores xi y ri sea mayor o igual a R. El valor de R se declara en el código y corresponde al retorno esperado de la cartera. A continuación veremos la restricción en su expresión matemática, para luego ver su codificación.

Expresión Matemática	Codificación

$\sum_{i=1}^{n} r_i x_i \geq R$	`constraint`
	`total_1 = sum(i in 1..activos) (`
	`vector_xi[i]*vector_ri[i] )`
	`/\`
	`total_1 >= 0`
	`/\`
	`total_1 >= r`
	`;`

- **Segunda Restricción:** Esta restricción tiene como finalidad que la suma de los valores del vector_xi sea igual a suma. La variable suma tiene valor 100 y está definido en el código.

Expresión Matemática	Codificación
$\sum_{i=1}^{n} x_i = 1$	34 constraint 35 sum(vector_xi) = suma 36 ;

- **Función Objetivo:** Para llegar a la Función Objetivo, la cartera debe cumplir las dos restricciones anteriores. La Función Objetivo está compuesta por una doble sumatoria, en las cuales se multiplica la Matriz_covarianza y el Vector_xi en doble instancia.

Expresión Matemática	Codificación
Minimizar $f(X) = \sum_{i=1}^{n}\sum_{j=1}^{n} \sigma_{ij} x_i x_j$	37 constraint 38 total_2 = sum(i,j in 1..activos) 39 (Matriz_covarianzas[i,j]*vector_xi[i]*vector_xi[j]) 40 /\ 41 total_2 >= 0 42 ;

- **Constante Activos**: Corresponden a los títulos que componen la cartera.

  Esta constante estará presente en casi todo el programa. Primeramente se encuentra en la declaración del Vector_ri, y en la declaración de uno de los for de la Función Objetivo.Esto se puede apreciar en las siguientes figuras.

```
1  include "globals.mzn";
2
3  int: activos = 4;
```

Figura 9.3 Declaración de la constante

```
6  array[1..num_activos] of int: vector_ri = [7,8,9,10]
7  array[1..activos, 1..activos] of int:
8  Matriz_covarianzas =
9   array2d(1..activos, 1..activos,
10        [
11         40, 4, 6, 2,
12         4, 60, 3, 1,
13         6, 3, 85, 8,
14         2, 1, 8, 100,
15        ]);
```

Figura 9.4 Utilización de Activos

Variables de decisión

- **x_i:** Corresponde al porcentaje a invertir del activo i con respecto del total de la cartera, donde i corresponde al activo financiero (1..n) con dominio (1..Activos).

```
16  array[1..activos] of var 1..100: vector_xi;
```

Figura 9.5 Utilización de la variable de decisión x_i

- **Total_1:** Esta variable almacenará el valor obtenido en la primera restricción. Es decir, en total_1 se guardará el resultado de la suma resultando de la multiplicación del vector_ri por el vector_xi. Luego, r se comparará con Total_1, donde esta variable debe ser mayor o igual al valor de R.

```
17  var int: total_1;
```

Figura 9.6 Declaración de la variable de decisión Total_1

- **Total_2:** En esta variable se almacenará el valor obtenido de la Función Objetivo, el que posteriormente se deberá minimizar, para poder disminuir el riesgo al momento de invertir.

```
18  var int: total_2; % to minimize
```

Figura 9.7 Declaración de la variable de decisión Total_2

9.2.1 Restricciones

Como en todo lenguaje de modelado, las restricciones utilizadas por CFPSP nos ayudan en la resolución de cualquier problema de optimización con restricciones, por lo que deben ser tratadas de forma especial. La sintaxis utilizada en Minizinc es la siguiente:

- Menor que, se escribe como <
- Mayor que, se escribe como >
- Menor o igual que, se escribe como <=
- Mayor o igual que, se escribe como >=
- Igual, se escribe como =
- No igual, se escribe como !=

10 Pruebas en Eclipse y Minizinc

En este capítulo podremos apreciar la fase de pruebas y experimentación que se realizaron sobre la programación en el solver Eclipse y Minizinc.

En la primera parte, se presentan las pruebas realizadas a carteras (instancias) compuestas de 4, 6 y 8 activos, con distintas matrices covarianzas, vectores RI y retornos, resultados obtenidos, como también sus tiempos de respuesta para finalizar con el análisis de éstos. La data de las carteras utilizadas, se obtuvo del programa financiero ECONOMÁTICA (programa utilizado por los alumnos de Ingeniería Comercial de la PUCV), el que posee toda la información referente a las finanzas y acciones de las empresas chilenas. Esta información es obtenida de la Bolsa de Comercio de Chile.

En el capítulo 10.2 se presentan los resultados obtenidos sin la utilización de heurísticas, mostrando cada una de las instancias con sus respectivos datos de entradas y de salidas. Finalizando este capítulo, se presentan pruebas realizadas con distintas heurísticas de Selección de Variable y de Selección de Valor, comparando los tiempos de resolución de cada uno de éstos y realizando las observaciones pertinentes a cada instancia mostrada.

10.1 Software y configuración de parámetros

Se utilizaron diferentes programas para el desarrollo y prueba del CFPSP. Entre los que destacan TkEclipse, Notepad++ y la distribución de Minizinc G12. A continuación, se detallarán las principales características de cada uno de ellos:

- **TkEclipse:** Es el compilador del código generado en un editor. Se utilizó por ser libre, de manejo más accesible y gratuito. Utilizando versión 6.0 #159 (i386_nt). Descargada directamente del sitio web de los desarrolladores [17].

- **Notepad++:** Se utilizó este editor de texto por su distribución libre y bajo licencia GNU. La versión fue la 5.9.2. Éste, es un editor que ayuda al desarrollador, destacando distintos colores o tipografías, sentencias, valores numéricos propios del lenguaje Prolog, lo que ayuda a mejorar la comprensión del código fuente.

- **Distribución de Minizinc G12**: Esta distribución sirve de plataforma de software para la solución de problemas de optimización combinatoria. Ésta distribución está licenciada bajo una simple licencia estilo BSD.

- **Entorno de desarrollo**: En lo que respecta a las pruebas realizadas, se utilizó el computador: Notebook Dell XPS 14 con las siguientes especificaciones técnicas:

    - Intel® Core (TM) i5 CPU, M 460 @ 2.53 GHz. Turbo Boost a 2.9 GHz.

    - Memoria Ram 6 GB.

- Sistema Operativo Windows 7 Ultimate, Service Pack 1.

10.2 Desarrollo de Pruebas y datos utilizados

Como se mencionó anteriormente, se utilizaron distintos datos para realizar las pruebas. Se utilizaron 7 instancias diferentes, las cuales están compuestas por Vector RI, Matriz de Covarianza, retorno de la cartera, además del Vector XI y Total 2, que corresponden a los datos de salida entregados por los solvers. Estas carteras, junto a sus componentes, se detallarán a continuación:

Datos de Entrada:

- **Vector RI:** En este vector se encuentran los porcentajes (en valores enteros) correspondientes a los retornos esperados de cada activo, es decir, en cada casilla del vector se encontrará el porcentaje de retorno que se espera conseguir por cada activo.

- **Matriz_covarianza:** En esta matriz se encuentran los valores obtenidos mediante la covarianza entre los activos, es decir, en la posición 1,2 de la matriz, se encuentra la covarianza existente entre el activo 1 y el activo 2. En el caso de la diagonal, ésta se calcula mediante la varianza de activo correspondiente, es decir, en la posición 1,1 se encuentra el valor de la varianza obtenido con los activos 1 y 1.

- **R:** Es el retorno que se espera obtener de la cartera. Cuando se multiplique el Vector RI por el Vector XI, R tendrá que ser mayor o igual.

Datos de Salida:

- **Total 2:** Valor de la Función Objetivo a minimizar.

- **Vector XI:** Vector que contiene la proporción de cada activo, es decir, los porcentajes que se deben invertir en cada activo.

Otros

- **Instancia:** Nombre con el cual se hará referencia a la cartera y su respectiva configuración con la que se está trabajando.

A continuación se muestran los resultados de las pruebas realizadas sobre las instancias

Instancias	Matriz Covarianza	Vector RI	R	Vector XI	Total 2
Instancia 1	(28, 20, 13, 20) (20, 16, 20, 28) (13, 20, 71, 19) (20, 28, 19, 40)	[16,14,20,20]	11	[1,97,1,1]	163979
Instancia 2	(6, 5, 3, 1) (5, 12, 3, 4) (3, 3, 8, 1) (1, 4, 1, 6)	[15,20,17,5]	13	[35,1,21,43]	32030
Instancia 3	(100, 12, 32, 32) (12, 100, 15, 11) (32, 15, 100, 24) (32, 11, 24, 100)	[31, 3, 34, 21]	14	[21, 32, 23, 24]	401768
Instancia 4	(100, 99, 99, 98) (99, 100, 100, 95) (99, 100, 100, 95) (98, 95, 95, 100)	[8, 8, 8, 9]	13	[1, 48, 1, 50]	975202
Instancia 5	(100, 31, 21, 22) (31, 100, 24, 23) (21, 24, 100, 17) (22, 23, 17, 100)	[21, 19, 19, 15]	12	[24, 22, 27, 27]	420886
Instancia 6	(12, 2, 1, 6, 4, 1) (2, 1, 15, 11, 15, 11) (1, 15, 20, 6, 4, 1) (6, 11, 6, 15, 0, 0) (4, 15, 4, 0, 34, 4) (1, 11, 1, 0, 4, 19)	[86, 9, 6, 1, 7, 3]	15	[1,95,1,1,1,1]	19439
Instancia 7	(32,3,1,1,39,18,6,19) (3,20,3,2,16,35,3,18) (1,3,1,0,17,6,2,0) (1,2,0,5,2,7,1,0) (39,16,17,2,29,61,17,29) (18,35,6,7,61,65,12,80) (6,3,2,1,17,12,61,8) (19,18,0,0,29,80,8,18)	[6,3,2,13,2,15,3, 2]	14	[1,1,76,18,1,1, 1, 1]	13225

Tabla 10.1 -- Instancias utilizadas.

La tabla anterior muestra las carteras utilizadas y sus respectivos resultados. Primeramente, se pueden apreciar los porcentajes de inversión de cada activo y el valor de la Función Objetivo a minimizar. Es un detalle no menor, mencionar que la programación del modelo matemático fue realizada sin la utilización de heurísticas y por tanto no se ve el tiempo de resolución de cada solver, sin embargo, en el siguiente capítulo (ver capítulo 10.3) se tomará en cuenta.

10.3 Presentación de Pruebas

En esta sección se verán las pruebas realizadas a las instancias anteriormente mencionadas. Estas pruebas se realizaron sobre los dos solvers utilizando diversas heurísticas y por tanto mostrando como métrica fundamental los tiempos de resolución aplicadas a las distintas combinaciones de heurísticas de Selección de Variable y de Selección de Valor.

Las pruebas se muestran a continuación:

Instan-cias	Eclipse						Minizinc					
	FF-I	**FF-IS**	**FF-IM**	**S-IS**	**S-I**	**S-IM**	**FF-I**	**FF-IS**	**FF-IM**	**S-IS**	**S-I**	**S-IM**
Instan-cia 1	26,02	5,76	42,04	**5,68**	25,65	42,63	3,36	1,07	3,71	**0,78**	3,29	3,44
Instan-cia 2	46,35	27,61	64,55	**25,79**	46,75	63,18	2,7	1,18	3,27	**0,95**	2,46	4,13
Instan-cia 3	46,75	20,70	78,44	**20,47**	47,00	76,10	3,30	1,63	4,98	**1,61**	3,30	5,57
Instan-cia 4	**73,34**	138,75	146,06	138,78	74,02	140,60	4,07	6,73	6,65	**1,50**	6,75	6,41
Instan-cia 5	50,01	24,79	80,33	26,63	50,97	**8,90**	3,45	2,00	**0,29**	1,88	3,22	7,31
Instan-cia 6	1589,62	320,16	1343,70	**261,21**	1506,91	1112,38	120.02	25,3	97,44	**7,43**	115,7	30,54
Instan-cia 7	6341,83	1576,0	3424,25	**1569,09**	8270,16	3833,65	355,98	83,43	221,85	460,36	418,74	**47,84**

Tabla 10.2 – Resultados de las pruebas realizadas en Eclipse y Minizinc

La tabla recién expuesta contiene los tiempos de respuesta obtenidos en cada una de las pruebas realizadas en ambos solvers. Como se puede apreciar, se utilizaron distintas heurísticas, siendo éstas *FirstFail* (FF), *Indomain* (I), *Indomain Split* (IS), *Smallest* (S) e *Indomain Min* (IM). Los valores que se encuentran en color rojo corresponden al menor tiempo obtenido por instancia, tanto en Eclipse como en Minizinc.

Estos tiempos se pueden ver reflejados en la siguiente tabla:

Instancias	Eclipse	Minizinc
Instancia 1	5,68	0,78
Instancia 2	25,79	0,95
Instancia 3	20,47	1,61
Instancia 4	73,34	1,50
Instancia 5	8,90	0,29
Instancia 6	261,21	7,43
Instancia 7	1569,09	47,48

Tabla 10.3 Mejores tiempos de respuesta en Eclipse y Minizinc

Como se puede apreciar en las pruebas anteriores, se utilizaron distintas matrices, vectores y retornos como datos de entrada, en donde los dos solver arrojaron exactamente los mismos resultados, referentes al Vector XI y la variable Total 2. Esto se debe a que el traspaso de la programación de un solver a otro se hizo en forma correcta.

En lo que respecta a los tiempos de resolución, Minizinc fue ampliamente más rápido que Eclipse, obteniendo en todas las pruebas un menor tiempo de resolución. A la vez, se puede apreciar que el uso de la heurística de Smallest combinada con Indomain Slip fue la que tomo menos tiempo de resolución.

10.4 Gráficos de las Pruebas Heurísticas

A continuación, se muestran 2 gráficos, en los que se verá representado los mejores tiempos para las instancias analizadas. Estos gráficos se separarán de la siguiente manera. El primer gráfico mostrará los mejores tiempos para las carteras de 4 activos y el segundo gráfico corresponderá a las carteras de 6 y 8 activos. Cabe destacar que se decidió mostrar estos tiempos en dos gráficos, ya que, como se mencionó anteriormente, las carteras analizadas están compuestas por 4, 6 y 8 activos, por lo que, para poder apreciar de mejor manera las diferencias en los tiempos de respuesta, se agruparon las carteras de acuerdo a la cantidad de activos que las componen. Es por esto que el primer gráfico muestra los mejores tiempos de respuesta para las carteras de 4 activos, y el segundo gráfico agrupa las instancias de 6 y 8 activos. Los tiempos mostrados a continuación corresponden a los resultados obtenidos tanto en el solver Eclipse como Minizinc.

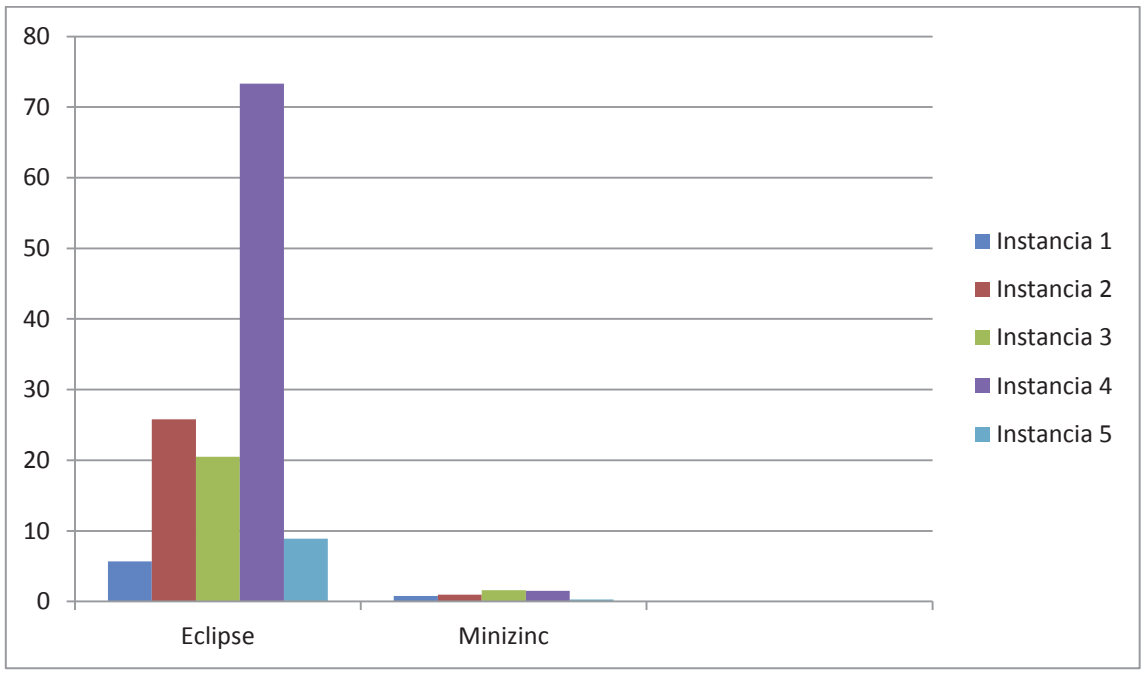

Figura 10.1 Mejores tiempos para las Instancias de 4 activos, tanto en Eclipse como en Minizinc

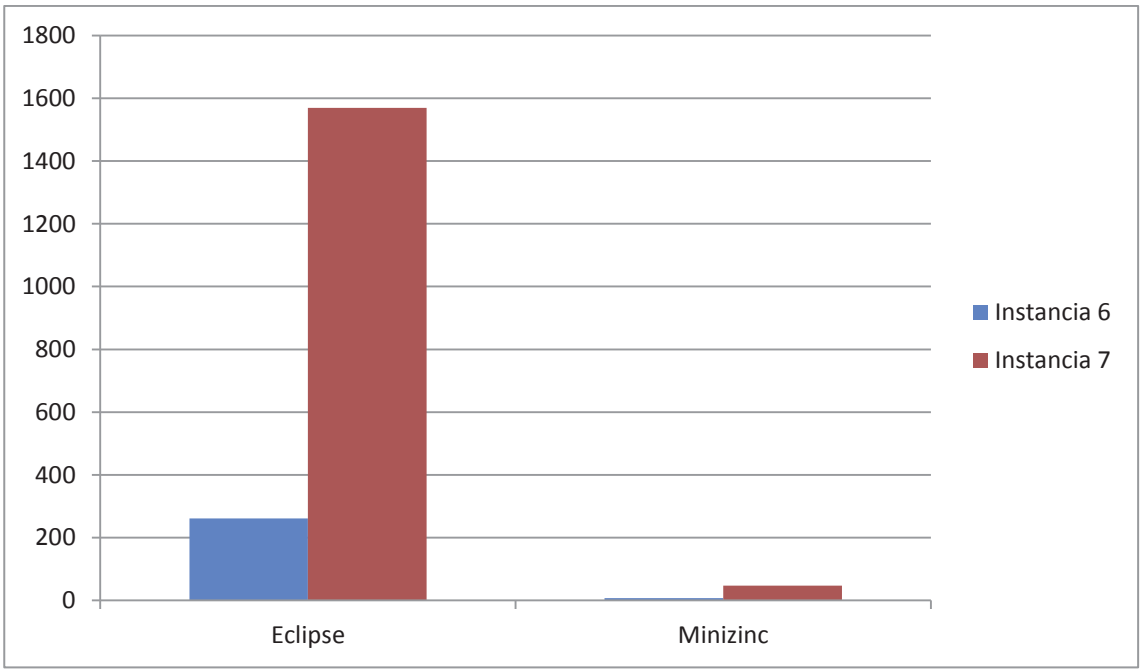

Figura 10.2 Mejores tiempos para las Instancias de 6 y 8 activos, tanto en Eclipse como en Minizinc

De acuerdo a estos gráficos, se puede apreciar de manera más visual y elocuente que Minizinc y su distribución G12 ha resultado ser más rápido que Eclipse en todas las pruebas, para estos tipos de problemas y no queriendo generalizar para todos los casos. La utilización de heurísticas permitió aumentar aún más la brecha de tiempos de resolución en todas las pruebas realizadas.

11 Conclusiones

Todo el trabajo anteriormente expuesto ayudó a conocer, comprender y experimentar con técnicas informáticas o lenguajes de programación menos populares a los conocidos lenguajes C, C++, Java o PHP, entre otros. En donde, la orientación no está enfocada al cómo hacer, si no a la declaración del modelo en un lenguaje específico de CP y la posterior experimentación por medio de un solver.

La programación en CP tiene su complejidad en el hecho de lograr entender que no se debe pensar en "cómo" resolver el problema, sino que en "qué" se debe hacer para que el problema sea resuelto. Tema que en un principio resulta difícil de comprender, ya que, nos encontramos acostumbrados a trabajar de forma imperativa, y no declarativa, que es la forma propuesta por CP.

Actualmente, se logró realizar la programación del modelo matemático propuesto por Markowitz, modelándolo en el solver Eclipse y Minizinc, obteniendo los resultados buscados; el que es obtener una cartera óptima. Llámese cartera óptima a la que entregue los porcentajes a invertir de cada acción y además obtener mayor retorno a un menor riesgo dado.

El modelo de Markowitz es considerado como el pionero en lo que respecta a la Selección de Cartera, pero a la vez, es un modelo el cual puede ser modificado, y por tanto ser un gran tema de investigación, desarrollo e implementación de mejoras. Tanto en su diseño, como en su implementación final, y por consiguiente una tentativa opción a un trabajo futuro.

Las pruebas realizadas en los dos solvers, ayudaron en gran medida a vislumbrar que cada uno tiene tiempos de respuestas distintos, y que no solo basta con utilizar un solver o un algoritmo de búsqueda en específico para obtener resultados de cómputos mínimos, si no que junto a una adecuada heurística puede ayudar notablemente en los tiempos de resolución. Con esto no se quiere decir que un solver sea mejor que otro si no que, depende exclusivamente del tipo de problema, heurística y o los datos a experimentar.

El presente trabajo no pretende inclinarse por un solver en específico, sino que, ejemplificar que para las instancias consultadas, con sus respectivas configuraciones de valores de entrada, Minizinc ha sido el solver que resolvió el problema de manera más rápida.

No cabe duda que es fundamental seguir explotando las capacidades que ofrece CP, dado su naturaleza de poder resolver prácticamente cualquier problema de la vida real. Además de su eficacia y eficiencia para resolver problemas, en relación a otra técnica informática más comúnmente usada, mencionada en el primer párrafo de estas conclusiones. Sin duda, que el aumento y complejidad de los problemas de hoy en día, genera la necesidad imperiosa, de optar por lenguajes menos populares como lo es CP.

12 Referencias

[1] Y. Crama a, M. Schyns (2003), Simulated annealing for complex portfolio selection problems, *European Journal of Operational Research, 150 (3),* 546-571.

[2] Rubén Armañanzas (2005), Jose A. Lozano, A multiobjective approach to the portfolio optimization problem, *Congress on Evolutionary Computation 2005*, 1388-1395.

[3] P. Gálvez, M. Salgado, M. Gutiérrez (2010), *Optimización de Carteras de Inversión. Modelo de Markowitz y Estimación de Volatilidad con Garch*, Bio Bio, Chile.: Horizontes Empresariales.

[4] G. Denga, W. Lin, C. Lo (2012), Markowitz-based portfolio selection with cardinality constraints using improved particle swarm optimization, *Expert Systems with Applications: An International Journal, 39 (4),* 4558-4566.

[5] A. Perold (1984), Large-Scale Portfolio Optimization, *Management Science 30 (10),* 1143-1160.

[6] D. Maringer, H. Kellerer (2003), Optimization of Cardinality Constrained Portfolios with a Hybrid Local Search Algorithm, *OR Spectrum 25(4)*, 481-495.

[7] F. Manyá, C. Gomes (2003), Solution Techniques for Constraint Satisfaction Problems, Jaume II, 69, E-25001 Lleida, Spain, pp, 2-4.

[8] S. Dong, (2066), Methods for Constrained Optimization. Available at http://web.mit.edu/dongs/www/publications/projects/2006-05-23_18.086_ConstrainedOptimization.pdf (visitada 15/05/2011).

[9] Página de Finanzas. Available at http://www.crecenegocios.com/la-diversificacion/ (visitada 17/05/2011).

[10] H. Markowitz (1959), Portfolio Selection, Efficient Diversification of Investments, John Wiley & Sons, New York.

[11] L. Di Gaspero, G. Di Tollo, A.Roli and A. Schaerf, (2007), Hybrid Local Search for Constrained Financial Portfolio Selection Problems. Available at http://www.sci.unich.it/~ditollo/works/ddrsCPAIOR07.pdf (visitada 15/08/2011).

[12] M. Wallace, S. Novello, J. Schimpf (1997), Eclipse: A Platform for Constraint Logic Programming. Available at http://Eclipseclp.org/reports/Eclipse/Eclipse.html (visitada 21/08/2011).

[13] C. Montelongo (1999), La Robótica Como Herramienta Del Hombre. Available at http://montelpz.htmlplanet.com/robot/refbiblio.html (visitada 02/09/2011).

[14] B. González, J. Quintero, (2010), "Análisis diseño e implementación de un sistema informático para el apoyo al proceso de asignación de la carga académica usando programación con restricciones", Universidad Tecnológica de Pereira, Colombia, 39-49.

[15] J. Gutiérrez, A. López, (2010), Modelado "Resolución del Manufacturing Cell Design Problem utilizando Programación con Restricciones", Informe final de Proyecto de Ingeniería de Ejecución en Informática, PUCV, Chile, 28-30.

[16] K. Apt, M. Wallace, (2006), Constraint Logic Programming using Eclipse, Cambridge University Press.

[17] Official website of Eclipse Solver. Available at http://www.eclipseclp.org (visitada 10/09/2011).

[18] K. Marriott, P. Stuckey, L. De Koninck, H.Samulowitz (2011), "An Introduction to MiniZinc", 3-7.

[19] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, G. Tack, (2007), Minizinc Towards A Standard CP Modelling Language, *Principles and Practice of Constraint Programming – CP 2007, 474*, 529-543.

13 Anexo

13.1 Código fuente del CFPSP en Eclipse

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

go :-
cfpsp(Vector_XI, R),
writeln(vector_xi:Vector_XI),
writeln(r:R).

 % Implementación del CFPSP en Eclipse

 cfpsp(Vector_XI, R):-

 Activos = 5,
 Sum = 100,

 Vector_RI = [](27, 30, 38, 17, 17),

 Matriz_covarianzas = []([]( 100 , 11, 24, 13, 19),
                      []( 11, 100, 17, 17, 19),
                      []( 24, 17, 100, 14, 42),
                      []( 13, 17, 14, 100, 26),
                      []( 19, 19, 42, 26, 100)),

 dim(Vector_XI,[Activos]),
 Vector_XI[1..Activos] :: 1..100,

 flatten_array(Vector_XI, Vector_V),
 sum(Vector_V) #= Sum,

 ( for(J,1,5),
 fromto(0,In,Out,Total_1),
 param(Vector_RI, Vector_XI) do
       X is Vector_RI[J],
       Z is Vector_XI[J],
       Out #= In + X*Z
 ),

 R #>= 0,
 Total_1 #>= R,

 ( for(I,1,Activos) * for(J,1,Activos),
   fromto(0,In,Out,Total_2),
     param(Matriz_covarianzas,Vector_XI) do
           A is Matriz_covarianzas[I,J],
           B is Vector_XI[I],
           C is Vector_XI[J],
           Out #= In + A*B*C
  ),

 term_variables([Vector_XI,Total_1, Total_2, R], Vars),
```

```
53  minimize(search(Vars,0,occurrence,indomain_median,complete,
54   [backtrack(Backtracks)]), Total_2),
55
56  % output:
57  writeln(total_2:Total_2),
58  writeln(vector_xi:Vector_XI),
59  writeln(r:R),
60  writeln(backtracks:Backtracks).
```

13.2 Código fuente del CFPSP en Minizinc

```
1  include "globals.mzn";
2
3  int: activos = 4;
4  int: suma = 100;
5
6  array[1..activos] of int: vector_ri = [9,10,10,10];
7  array[1..activos, 1..activos] of int: Matriz_covarianzas =
8  array2d(1..activos, 1..activos,
9        [
10       24, 36, 24, 22,
11       36, 54, 24, 24,
12       24, 24, 24, 24,
13       24, 24, 24, 24,
14       ]);
15
16  array[1..activos] of var 1..100: vector_xi;
17
18  var int: total_1;
19  var int: total_2; % to minimize
20  int: r = 11;
21
22   solve :: int_search(vector_xi, first_fail, indomain_split, complete)
23            minimize total_2;
24
25  constraint
26     total_1 = sum(i in 1..activos) ( vector_xi[i]*vector_ri[i] )
27     /\
28     total_1 >= 0
29     /\
30     total_1 >= r
31  ;
32
33  constraint
34     sum(vector_xi) = suma
35  ;
36
37  constraint
38 total_2 = sum(i,j in 1..activos)( Matriz_covarianzas[i,j] *
vector_xi[i] * vector_xi[j])
39       /\
40       total_2 >= 0
41  ;
42
43  output [
44    "total_2: " ++ show(total_2) ++ "\n" ++
45    "x      : " ++ show(vector_xi) ++ "\n"
46  ]
47  ++ ["\n"]
48  ;
```