IMPLEMENTACIÓN DE LOS MECANISMOS DE BÚSQUEDA PARA LA NUEVA MÁQUINA VIRTUAL DE MOZART

Autor

Miguel Angel Villanueva Sanclemente

Universidad del Valle

Escuela de Ingeniería de Sistemas y Computación

Ingeniería de Sistemas

Tuluá - Valle

2013

IMPLEMENTACIÓN DE LOS MECANISMOS DE BÚSQUEDA PARA LA NUEVA MÁQUINA VIRTUAL DE MOZART

Autor
Miguel Angel Villanueva Sanclemente
miavisa@gmail.com
Código. 200859247

Documento presentado como requisito parcial para
la obtención de grado de Ingeniero de Sistemas

Director
Juan Francisco Díaz Frias

Asesor
Gustavo Gutiérrez Sabogal

Universidad del Valle
Escuela de Ingeniería de Sistemas y Computación
Ingeniería de Sistemas
Tuluá - Valle
2013

Trabajo de grado presentado por:
Miguel Angel Villanueva Sanclemente
Como requisito parcial para la obtención del título de Ingeniero de Sistemas.

Juan Francisco Díaz Frias, Ph. D
Director

Jurado 1

Jurado 2

Agradecimientos

En el desarrollo de este trabajo de grado quiero agradecer a todas esas personas que fueron parte de este proceso: A mis padres, hermanas, novia y demás familiares por su confianza, amor y apoyo incondicional. A mi mentor y director Juan Francisco Díaz, por la confianza que depositó en mi; por su constante apoyo y valiosos aportes en el desarrollo de este trabajo de grado. A Gustavo Gutiérrez por sus comentarios y valiosos aportes para el desarrollo de este trabajo de grado. A Holmes Salazar por su ayuda y colaboración durante este proceso. A todos los miembros de AVISPA que de alguna u otra forma contribuyeron en el desarrollo de este trabajo de Grado: Andrés Felipe Barco, Carlos Martinez, Alejandro Cardona, Daniel Montenegro y Gustavo Gomez. A mis profesores y compañeros que a lo largo de mi carrera universitaria, contribuyeron en mi formación personal y profesional.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

Resumen

La programación por restricciones presenta estrategias de solución a problemas de optimización y combinatorios. La estrategia general para su uso es mediante la especificación de las variables del problema y las relaciones (restricciones) que entre ellas se deben mantener. Después de dicha especificación el usuario hace uso de *motores de búsqueda predefinidos* que se encargan de recorrer el espacio de soluciones del problema.

Gecode es un componente de software que implementa el *paradigma de programación por restricciones* de manera eficiente. Sin embargo su utilización es compleja para usuarios por el lenguaje de programación. Oz es un lenguaje de programación inventado en la década de los 90 que soporta entre otros el paradigma de programación por restricciones. Debido a su concepción multiparadigma ha sido adoptado por una comunidad de investigación importante. Este lenguaje interpretado ofrece mejores abstracciones y provee un ambiente de trabajo más confortable. Su implementación, Mozart hasta ahora ha carecido de soporte en la adopción de los avances en la investigación en esta área. Por esta razón se ha diseñado una nueva máquina virtual con Gecode como motor de restricciones, para tomar ventaja de estos avances.

En este documento se propondrán los mecanismos para la interacción de los motores de búsqueda implementados en Gecode con Mozart, como también los mecanismos que permitan al usuario implementar sus propios motores de búsqueda en Oz y utilizarlos. Esto es de gran importancia para los usuarios porque permitirá sacar ventaja del conocimiento que este posee sobre el problema obteniendo búsquedas más eficientes.

CAPÍTULO 1

Contexto y Objetivos

1.1. Planteamiento del Problema

1.1.1. Descripción del Problema

Mozart es la implementación del lenguaje de programación Oz. El cual soporta diferentes paradigmas de programación: declarativo (concurrente), lógico, por restricciones y distribuido. Mozart fue implementado por el Consorcio Mozart y es el resultado de más de dos décadas de investigación en las áreas mencionadas. Todo esto hace de Mozart una herramienta con un gran potencial expresivo y funcional.

Desde que Mozart fue liberado en 1995 no ha tenido cambios relevantes en su implementación y durante estos 17 años la evolución de las plataformas de cómputo propone nuevos desafíos. La aparición de procesadores multi-núcleo y dispositivos móviles, entre otros, presentan nuevas posibilidades para utilizar Oz. Por esta razón y para que Mozart pueda estar vigente una década más ha nacido la necesidad de realizar una nueva implementación de la máquina virtual. Entre otros aspectos, esta implementación deberá integrar los avances realizados en los últimos años en la programación por restricciones.

Desde el punto de vista de diseño, la nueva implementación tiene tres componentes principales, uno de los cuales es la máquina virtual en si y los otros dos soportan la progra-

mación distribuida y por restricciones. Este trabajo se enfoca únicamente en el subsistema de restricciones (CSS) y más específicamente en la integración de los motores de búsqueda implementados en Gecode.

1.1.2. Formulación del Problema

¿Cómo poder integrar los motores de búsqueda disponibles en Gecode con la nueva implementación de la máquina virtual de Mozart manteniendo compatibilidad y permitiendo a los usuarios definir (implementar) sus propios motores de búsqueda?

1.2. Justificación

En la programación por restricciones es de gran importancia tener la posibilidad de definir motores de búsqueda propios ya que estos permiten explotar el conocimiento que se tiene sobre el problema en pro de obtener soluciones más eficientes. Al momento de definir estos motores de búsqueda es más importante el conocimiento que el usuario tiene sobre el problema que el conocimiento que se tenga en programación, por esta razón es fundamental usar lenguajes de programación de alto nivel como Oz para la implementación de estos motores.

La importancia de este proyecto a nivel académico, es que permitirá la adquisición de nuevos conocimientos en áreas como la programación por restricciones, conociendo a fondo cómo se deben implementar los motores de búsqueda, distribuidores y propagadores utilizados en ésta.

Adicionalmente, como este proyecto es de colaboración entre AVISPA y el grupo de investigación en programación del profesor Peter Van Roy en la Universidad Católica de Lovaina, este proyecto afianza y fortalece esos lazos de cooperación y abrirá puertas para sus participantes en posibles proyectos de posgrado posteriores.

1.3. Objetivos

1.3.1. Objetivo General

Desarrollar una interfaz[1] que integre los motores de búsqueda genéricos de Gecode con la nueva máquina virtual de Mozart. Esta interfaz deberá también permitir la definición de nuevos motores de búsqueda por parte de los usuarios.

[1]Se refiere a un medio de comunicación entre dos sitemas

1.3.2. Objetivos Específicos

1. Escribir un reporte sobre las diferentes estrategias de búsqueda en la programación por restricciones y su diseño e implementación en Gecode.[15] (Ver Tabla 5.1)

2. Identificar los componentes de Mozart que permitan la integración con los motores de búsqueda implementados en Gecode. (Ver Secciones 3.3, 4.1 y 5.2)

3. Implementar una interfaz entre el módulo CSS y Gecode que permita la utilización de los motores de búsqueda existentes de manera eficiente. (Ver Capítulo 5)

4. Implementar una interfaz que permita utilizar los motores de búsqueda escritos en Oz por el usuario. (Ver Capítulo 4)

1.4. Alcance

El presente proyecto se limitará a la integración de los motores de búsqueda genéricos implementados en Gecode con el subsistema de restricciones (CSS) de la nueva máquina virtual de Mozart. Además de brindar las herramientas necesarias para el diseño de motores de búsqueda propios por el usuario.

Es importante aclarar que este proyecto no abarcará otros componentes de la máquina Virtual de Mozart diferentes a los motores de búsqueda como lo son: distribuidores, propagadores ni el subsistema de distribución (DSS).

Estructura del Documento Los dos siguientes capítulos conforman el *Marco Teórico*, el cual está conformado por una explicación sobre qué es la programación por restricciones en el Capítulo 2; continuando en el Capítulo 3 con una especificación del diseño de la implementación de la nueva máquina virtual de Mozart. El proceso de implementación se presenta en los Capítulos 4 y 5. Las pruebas realizadas se presentan en el Capítulo 6; terminando con las conclusiones y trabajo futuro en el Capítulo 7.

CAPÍTULO 2

Programación por Restricciones

La programación por restricciones es una tecnología de software emergente para la resolución de problemas combinatorios especialmente en áreas de planeación y scheduling.[6] Por la complejidad de estos problemas no es posible resolverlos con los paradigmas tradicionales como lo son el paradigma orientado a objetos, funcional, entre otros.

La programación por restricciones tiene sus inicios en los años 60's y 70's en la Inteligencia Artificial y en la Computación Gráfica; las cuales se enfocaban en la representación explícita y manipulación de las restricciones en un sistema de computación. Pero solo a finales de la década de los 80's se empezó a tomar conciencia de que estas ideas constituyen una herramienta poderosa para la programación, modelado y resolución de problemas.[7]

A diferencia de otros paradigmas como la programación funcional u orientada a objetos, en las cuales se debe especificar paso a paso, cómo se va a resolver el problema, en la programación por restricciones lo importante es especificar las restricciones que posee el problema, ya que los algoritmos que se encargan de resolverlo se encuentran incluidos en el lenguaje de programación.

En este capítulo se especificará en que consisten los *Problemas de Satisfacción de Restricciones* y cómo éstos se pueden solucionar de manera eficiente.

2.1. Problemas de Satisfacción de Restricciones

La programación por restricciones consiste, en que dado un problema basado en restricciones se debe encontrar una solución que satisfaga todas las restricciones del problema. A estos problemas se les conocen como Problemas de Satisfacción de Restricciones o CSP (*Constraint Satisfaction Problems*). Los CSP han sido un tema de investigación en la Inteligencia Artificial por muchos años[6].

Un CSP básicamente consiste en un conjunto de variables, cada una de estas tiene asociado un dominio finito de posibles valores que podrá tomar. Y por último un conjunto de restricciones que permiten acotar los valores del dominio de las variables; estas restricciones se pueden ver como las relaciones que existen entre las variables.

Formalmente un CSP P se puede ver como un conjunto de tres elementos. $P = \{X, D, C\}$, donde:

- $X = \{X_1, X_2, ..., X_n\}$ es un conjunto de variables.
- $D = \{D_1, D_2, ..., D_n\}$ es un conjunto de dominios, donde D_i es el dominio de X_i.
- $C = \{C_1, C_2, ..., C_t\}$ es un conjunto de restricciones, donde cada una de estas restricciones se pueden ver como una función $f : X' \to \{0, 1\}, X' \subseteq X$. Cuando el valor de la función es uno, se dice que los valores en cuestión satisfacen la restricción.

Una solución a un CSP consiste en asignar valores a cada una de las variables de X, los cuales satisfacen todas las restricciones de C, esta solución se puede representar como $((X_1, a_1), ..., (X_n, a_n))$, donde cada par ordenado (X_i, a_i) representa la asignación del valor a_i a la variable X_i y $a_i \in D_i$.

Los CSP se puede llegar a clasificar en tres categorías según lo que se desee obtener de él:

[Una Solución] Se obtiene una solución que satisface al CSP, esta solución no tiene preferencia alguna. Generalmente es la primera solución que se encuentra.

[Todas las Soluciones] Se obtiene todas las soluciones que satisfacen al CSP.

[Mejor Solución] Se obtiene la mejor solución que satisface al CSP, según una función objetivo definida en términos de algunas o todas las variables del problema.

2.2. Solución de CSP

Todo CSP se puede resolver por medio de *Fuerza Bruta*, probando todas las posibles combinaciones de asignación de valores para cada una de las variables, pero generalmente este método no es eficiente.

La programación por restricciones ofrece una solución *inteligente*, para resolver CSP, que reduce la cantidad de posibles combinaciones a evaluar. Esta solución es conocida como *Propagar y Buscar* o *Propagar y Distribuir* y está basada en tres importantes ideas:

- Obtener información parcial del CSP.
- Realizar deducciones locales.
- Realizar una búsqueda controlada de la solución.

2.2.1. Espacios de Computación

La noción de espacio de computación, es general para el modelo computacional. Un espacio de computación está compuesto de un número de tareas o hilos (*threads*) conectadas con un almacén de información (*store*). Un *thread* es una estructura de control de computación secuencial. El *store* contiene las estructuras de datos que van a utilizar los *threads* para realizar la computación. Los *threads* y el *store* están conectados por medio de variables [4, 12]

En programación por restricciones, estas tareas se conocen como *propagadores* que representan las restricciones y el almacén de información como el *almacén de restricciones*. En el almacén de restricciones se guarda información sobre los valores de las variables expresadas como una conjunción de restricciones básicas.[12]

De lo anterior podemos concluir que un CSP, se puede modelar como un espacio de computación, donde las variables y los dominios son la información almacenada en el *store* y las restricciones son los *threads*.

Los espacios de computación presentan dos estados, los cuales son: ejecutable (*runnable*) y estable (*stable*). Un espacio de computación *runnable*, es aquel que tiene *threads* ejecutables y un espacio *stable*, es aquel que no presenta *threads* ejecutables.

Los espacios de computación *stable* se clasifican en tres categorías, las cuales son de gran importancia al momento de solucionar CSP:

[Failed] Es aquel espacio que ya no tiene hilos ejecutables porque presenta inconsistencia en sus restricciones, esto quiere decir que el CSP no tiene solución.

[Succeeded] Es aquel espacio que ya no tiene hilos ejecutables, que sus restricciones son consistentes y se ha encontrado una solución.

[Distribuitable] Tiene las mismas condiciones de un espacio *succeeded* solo que aún no se ha encontrado una solución.

2.2.2. Propagación

El imponer los dominios y las restricciones de un CSP, no excluye posibles valores para ser asignados a las variables, por lo cual para encontrar una solución se deberían evaluar todas las posibles asignaciones. Esto es correcto, pero requiere un número exponencial de pasos para solucionar un CSP. La forma más eficiente para resolver CSP es utilizar la propagación.[13]

La propagación excluye valores que no pueden ser parte de alguna solución. Esta exclusión de valores se realiza por medio de *propagadores* de acuerdo a las restricciones del CSP.

Los *propagadores* son agentes de computación concurrentes que para la exclusión de los valores utilizan funciones que permiten reducir los dominios de las variables, a estas funciones se les conocen como *Algoritmos de Propagación*. Cada restricción del CSP, es impuesta por un *propagador*.

Los *propagadores* se organizan en una cola y se van ejecutando uno por uno hasta que no puedan contribuir con más cambios sobre el dominio de las variables o se detecte alguna inconsistencia o falla en las restricciones. Este proceso se puede ver en la Figura 2.1

Ejemplo 2.1. Supongamos que tenemos dos propagadores que imponen las restricciones $X + Y = 9$ y $2X + 4Y = 24$. Sobre X y Y se tiene la siguiente información $X \in \{0, ..., 9\}$ y $Y \in \{0, ..., 9\}$. La ejecución de la propagación se realiza de la siguiente manera:

- El primer propagador no realiza modificaciones y se retira de la cola, entonces se selecciona el segundo propagador el cual acota los dominios de X y Y: $X \in \{0, ..., 8\}$ y $Y \in \{2, ..., 6\}$. Se vuelve a activar el primer propagador porque sus variables relacionadas se modificaron.

- Como el segundo propagador ya no puede realizar más modificaciones se elimina de la cola, entonces se selecciona el primer propagador el cual acota el dominio de X: $X \in \{0, ..., 8\}$ y $Y \in \{2, ..., 6\}$. Se activa de nuevo el segundo propagador.

- Ahora el segundo propagador acota los dominios de X y Y: $X \in \{4, ..., 6\}$ y $Y \in \{3, ..., 4\}$.

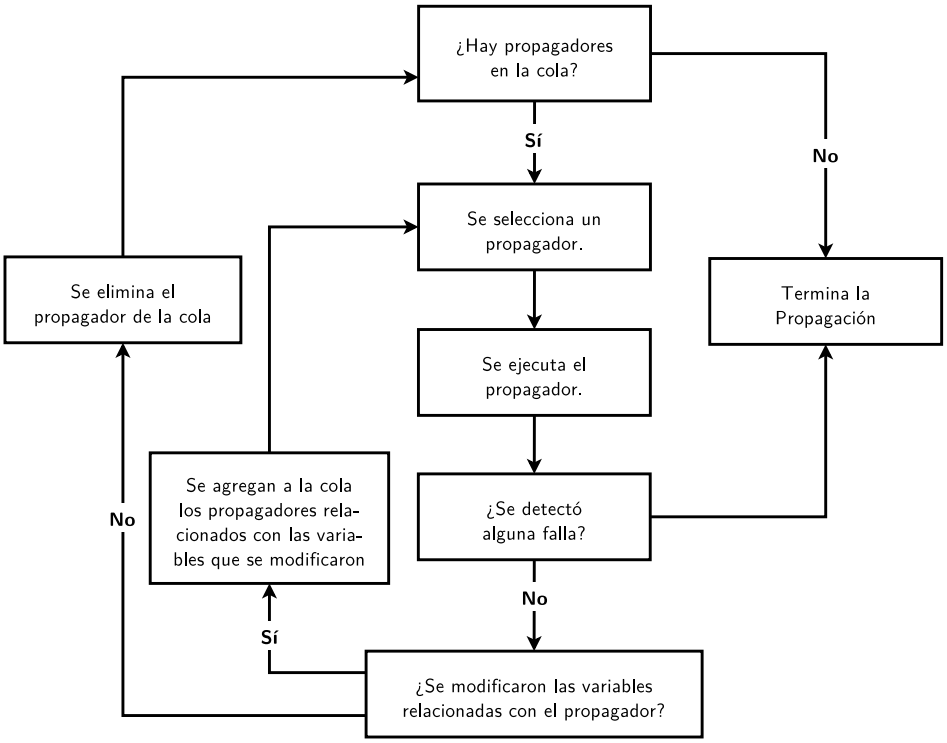


Figura 2.1: Ejecución de la propagación.

- Una vez más se activa el primer propagador que acota el dominio de X: $X \in \{5, ..., 6\}$ y $Y \in \{3, ..., 4\}$.
- Por último el segundo propagador determina los valores de X y Y: $X = 6$ y $Y = 3$.

Como se observa en el Ejemplo 2.1 se obtiene la solución al CSP solo utilizando la propagación, aunque ésta no siempre es suficiente.

Ejemplo 2.2. Supongamos que tenemos dos propagadores que imponen las restricciones $X < Y$ y $X + Y = 6$. Sobre X y Y se tiene la siguiente información $X \in \{0, ..., 9\}$ y $Y \in \{0, ..., 9\}$. La ejecución de la propagación se realiza de la siguiente manera:

- Se selecciona el primer propagador el cual acota los dominios de X y Y: $X \in \{0, ..., 8\}$ y $Y \in \{1, ..., 9\}$.
- Ahora se selecciona el segundo propagador el cual acota los dominios de X y Y: $X \in \{0, ..., 5\}$ y $Y \in \{1, ..., 6\}$.
- Por último se vuelve a activar el primer propagador, pero este no puede realizar modificaciones sobre X ni Y. En este punto los dos propagadores están desactivados por lo cual la propagación termina.

En el Ejemplo 2.2 se observa que la propagación no fue suficiente para obtener una solución al CSP, por esta razón es importante introducir un nuevo concepto, *La Distribución*.

2.2.3. Distribución

La distribución también conocida como *labelling* o *branching* es un proceso de especulación con variables. La distribución nos permite dividir un espacio de computación *distributable* como el del Ejemplo 2.2 para poder seguir resolviendo un CSP. A cada una de estas divisiones las podemos llamar *ramas* o *alternativas*.

La distribución consiste en dividir un espacio S en n espacios. Cada uno de estos nuevos espacios se consigue aplicando sobre S una nueva restricción C_i sobre un subconjunto X' de las variables de restricción de S, tal que:

$$C_1 \vee C_2 \vee ... \vee C_n = 1$$

Generalmente para solucionar CSP se buscar dividir el espacio S en dos *alternativas*, imponiendo la restricción C sobre uno de los espacios y en el otro se impone la restricción $\neg C$.

Ejemplo 2.3. Retomemos el Ejemplo 2.2 donde después de aplicar la propagación obtuvimos que: $X \in \{0, ..., 5\}$ y $Y \in \{1, ..., 6\}$. Supongamos que se define la estrategia de distribución de seleccionar la primera variable sin asignar y el valor mínimo del dominio de dicha variable. Para este caso, la estrategia de distribución escoge la variable X, el valor 0 y aplica las siguientes restricciones $X = 0$ para la primera alternativa y $X \neq 0$ en la segunda alternativa.

Después de la distribución en la primera alternativa tenemos: $X = 0$ y $Y \in \{1, ..., 6\}$, en este punto se vuelven a activar los propagadores y la propagación se ejecuta obteniendo como resultado: $X = 0$ y $Y = 6$.

En la segunda alternativa después de la distribución tenemos: $X \in \{1, ..., 5\}$ y $Y \in \{1, ..., 6\}$, igual que en la primera alternativa se activan los propagadores y se ejecuta la propagación obteniendo como resultado: $X \in \{1, ..., 4\}$ y $Y \in \{2, ..., 5\}$.

2.2.4. Búsqueda

La búsqueda es la encargada de coordinar la propagación y la distribución para solucionar un CSP. Inicia creando un espacio con la información de un CSP a resolver, propaga las restricciones del espacio hasta que este es estable, si el espacio es *failed* o *succeeded* la búsqueda

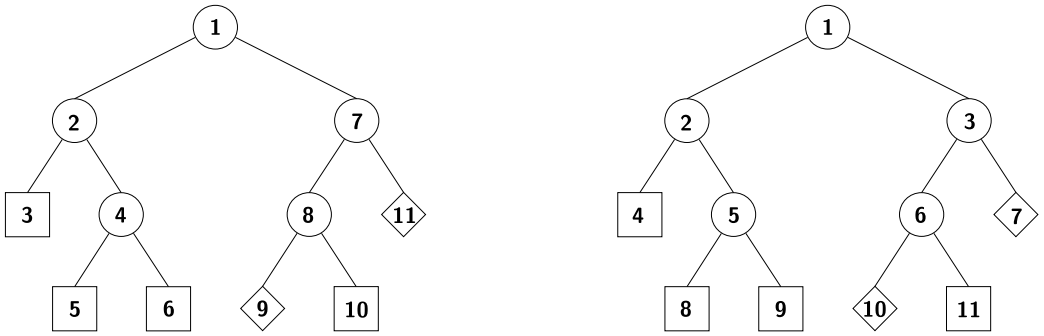
finaliza y si el espacio es *distribuitable* se procede a distribuir el espacio.[12] Este proceso se realiza iterativamente sobre los espacios que se crean al distribuir.

La búsqueda la podemos representar como un árbol, donde cada nodo del árbol representa un espacio estable, y los hijos de cada nodo, son los espacios generados al distribuir el espacio. Este árbol se conoce como *árbol de búsqueda*. Generalmente en la programación por restricciones los nodos ♢ representan espacios *succeeded*, los nodos □ representan espacios *failed* y los nodos ◯ representan espacios *distribuitable*.

Para encontrar la solución a un CSP, es necesario explorar el *árbol de búsqueda*. Existen diferentes estrategias de exploración, las más comunes son *depth-first* (profundidad) y *breadth-first* (anchura).

[depth-first] Esta estrategia consiste en tomar uno de los nodos hijos y explora todos sus caminos aplicando la misma estrategia. Después de terminar pasa a explorar los otros nodos hijos.

[breadth-first] Esta estrategia consiste en explorar todos los nodos de un nivel, sin importar si estos tienen nodos hijos o no. Después de explorar todos los nodos de un nivel, pasa a explorar los nodos del siguiente nivel.



(a) Exploración con depth-first **(b)** Exploración con breadth-first

Figura 2.2: Ejecución de la estrategias de exploración.

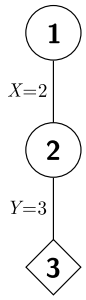
Estas estrategias de exploración son implementadas por *los motores de búsqueda*, que son los que se encargan de definir, cómo se va a explorar el árbol de búsqueda, qué tipo de solución se necesita y de coordinar la propagación y la distribución.

Ejemplo 2.4. Supongamos que tenemos un CSP con la siguiente información: $X \in \{2, 4\}$, $Y \in \{3, 5\}$, $Z \in \{2,4\}$, $X \leq Z$ y $Y > Z$. Se necesita encontrar **una** solución a este CSP, por lo cual se define un *motor de búsqueda* que me encuentra una solución y utiliza como

estrategia de exploración *depth-first*. Como estrategia de distribución se elige seleccionar la primera variable V sin asignar y el valor mínimo v del dominio de dicha variable y se imponen la restricción $V = v$ sobre la primera alternativa y la restricción $V \neq v$ sobre la segunda alternativa.

En la Figura 2.3 se muestra el proceso que realiza el motor de búsqueda para encontrar la solución al CSP, cada fila nos informa el estado del nodo después de aplicar la propagación.

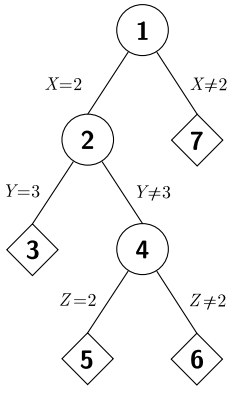
Podemos observar que en este caso solo se realiza una exploración parcial del árbol, ya que al encontrar una solución en el **Nodo 3** no es necesario seguir explorándolo.



Nodo	X	Y	Z
1	{2,4}	{3,5}	{2,4}
2	2	{3,5}	{2,4}
3	2	3	2

Figura 2.3: Resultado de realizar la búsqueda en el Ejemplo 2.4

Ejemplo 2.5. Retomemos el Ejemplo 2.4, pero en esta oportunidad queremos explorar todo el árbol para encontrar **todas** las soluciones del CSP. Por esta razón en esta ocasión utilizaremos un motor de búsqueda que encuentre todas las soluciones.



Nodo	X	Y	Z
1	{2,4}	{3,5}	{2,4}
2	2	{3,5}	{2,4}
3	2	3	2
4	2	5	{2,4}
5	2	5	2
6	2	5	4
7	4	5	4

Figura 2.4: Resultado de realizar la búsqueda en el Ejemplo 2.5

CAPÍTULO 3

Integración Mozart-Gecode

En este capítulo se dará una breve descripción de Mozart y Gecode. Se continuará explicando cómo se realiza la integración de Gecode como motor de restricciones de Mozart en la implementación de su nueva máquina virtual. También de ahora en adelante nos referiremos como *Mozart1.4* a la actual máquina virtual de Mozart y como *Mozart2* a la nueva máquina virtual de Mozart.

3.1. Mozart

Es una implementación multiplataforma del lenguaje de programación Oz. Originalmente desarrollado por Gert Smolka y su grupo de investigación en la Universität des Saarlandes[1] a principios de la decada de los 90s bajo el nombre *DFKI Oz*[4]. En 1999 su desarrollo continuó por parte de un grupo internacional llamado el Consorcio Mozart, el cual estaba compuesto por la Universität des Saarlandes (Alemania), el Swedish Institute of Computer Science (SICS)[2] (Suecia) y la Université catholique de Louvain[3] (Belgica). En 2005, la

[1] http://www.uni-saarland.de/
[2] http://www.sics.se/dsl/
[3] http://www.info.ucl.ac.be/ pvr/distribution.html

responsabilidad de la dirección del desarrollo de Mozart fue transferida a la Mozart Board[4]

3.2. Gecode

Es una librería distribuida como software libre, para el desarrollo de sistemas y aplicaciones basadas en restricciones. Gecode está implementado en C++ siguiendo los mas rigurosos estándares, los cuales garantizan su portabilidad. Gecode permite la implementación de nuevos propagadores (como implementación de restricciones), estrategias de distribución y motores de búsqueda, por lo cual se puede considerar como una librería abierta. Gecode tiene un excelente desempeño con respecto a ejecución, uso de memoria y escalabilidad, lo que lo convierte en una herramienta realmente eficiente[16]

3.3. Integración

Con Mozart2 se busca poder integrar los avances de los últimos años en la programación por restricciones y en la computación distribuida. En el nuevo diseño de la máquina virtual se busca desacoplar el soporte de estos dos de la máquina virtual y se van a considerar dos subsistemas (de restricciones y de distribución) que interactuarán con la máquina virtual. Esto permitirá una integración de los avances en estas dos áreas con menos esfuerzo que en la actualidad.

En el desarrollo de este documento nos enfocáremos solo en el subsistema de restricciones (CSS por sus siglas en ingles). El cual utiliza a Gecode como motor de restricciones.

Mozart y Gecode en esencia son sistemas muy similares respecto a la programación por restricciones, ambos utilizan la noción de espacios de computación para encapsular las variables de restricción y los propagadores.

Esta similitud es el punto de encuentro más natural que existe entre Mozart y Gecode, por esto podemos decir que cada espacio de Mozart debe estar asociado con un espacio de Gecode. La idea en la nueva máquina virtual de Mozart, es que el espacio de Gecode encapsule toda la información correspondiente a la programación por restricciones; por esta razón este espacio solo existe si se va a utilizar programación por restricciones de lo contrario este espacio es nulo.

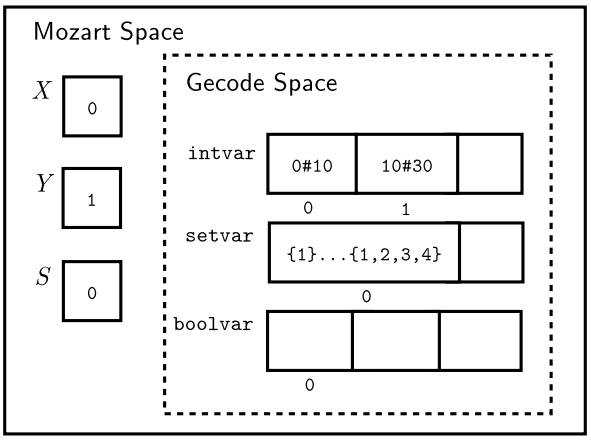
Lo anterior se ve reflejado en la implementación de la nueva máquina virtual, cuando se define un espacio de Gecode (`GecodeSpace`) como atributo de los espacios de Mozart. El

[4]http://www.mozart-oz.org/governance.html

diseño actual de los espacios de Mozart se muestran en la Figura 3.1.

Figura 3.1: Diseño espacios de Mozart en la nueva máquina virtual.

En ese `GecodeSpace` se encapsulará todo lo referente a la programación por restricciones como lo son las variables de restricción y los propagadores.




```
proc {CSP R}
  X :: 0#10    Y :: 10#30
  S = {FS.dom {1} {1,2,3,4}}
in
  R = [X Y S]
  % Some constraints
  % Distribution
end

{Space.new CSP Sp}
```

Figura 3.2: Representación de las variables de restricción en Mozart2.

En la Figura 3.2 observamos cómo se representan las variables de restricción de un CSP en Mozart2. En cada `GecodeSpace` tendremos tres vectores donde cada uno nos representa un tipo de variable de restricción (enteros, conjuntos y booleanos). En estos vectores se almacenará el valor de cada una de las variables de restricción definidas en nuestro CSP. En el espacio de Mozart se contará con una variable por cada variable de restricción definida en el CSP, pero esta no almacenará el valor sino que almacenará la posición en la que se encuentra almacenada en su correspondiente vector en el `GecodeSpace`.

A pesar de la gran similitud entre Gecode y Mozart, estos presentan algunas diferencias que debemos tener en cuenta. La diferencia más importante es cómo funciona la propagación, ya que en Mozart la propagación es *ansiosa*, es decir, esta se ejecuta siempre que se pueda ejecutar; mientras que la propagación en Gecode es *perezosa*,es decir, esta se ejecuta únicamente cuando se le dice que se ejecute, por medio del procedimiento `status` de Gecode. En Mozart2, la propagación se realiza por medio de un hilo de propagación encargado de decirle a Gecode que propague por medio del procedimiento `status`.

CAPÍTULO 4

Búsqueda en Mozart

En este capítulo se hablará de los mecanismos necesarios para escribir y utilizar motores de búsqueda escritos desde Oz. El poder escribir motores de búsqueda en Oz nos permitirá integrar los motores de búsqueda de Mozart1.4, los cuales están escritos en este lenguaje.

Para poder entender los mecanismos necesarios para la búsqueda en Mozart debemos tener claro los siguientes conceptos:

[árbol de espacios] Los espacios de computación en Mozart manejan una jerarquía, la cual se representa como un árbol; no hay que confundir el *árbol de espacios*, con el *árbol de búsquedas*, en el segundo los nodos de éste son espacios creados por la operación `clone`. Mientras que en el primero los nodos de este son espacios creados por la operación `new`.

[toplevel space] El *toplevel space* es la raíz del árbol de espacios, por lo tanto es el primer espacio que se crea al momento de ejecutar Mozart.

[current space] Es el espacio que se encuentra instalado en un momento específico en Mozart. Este espacio puede ser cualquier espacio del *árbol de espacios* incluido el *toplevel space*. El encargado de definir qué espacio es el *current space* es el *scheduler* quien define qué hilo se ejecuta en la máquina

virtual. Para que un hilo se ejecute en la máquina virtual el espacio al que pertenece debe estar instalado, es decir, debe ser el *current space*.

[rootvar] La variable raíz es el punto de encuentro de los espacios con el resto del sistema, la solución del CSP es almacenada en ésta.

[admisibilidad] Se dice que un espacio S_1 es admisible por un espacio S_2, si S_1 no pertenece al conjunto de espacios que son padres de S_2 incluyendo S_2.

4.1. Operaciones de los espacios

Para realizar la búsqueda los motores de búsqueda deben poder comunicarse con los espacios de computación. Esta comunicación es posible gracias a las operaciones de los espacios.

En *Mozart2* cada una de estas operaciones está escrita en C++ y está soportada en Mozart como Built-in. A continuación explicaremos cada una de las operaciones y su implementación en *Mozart2*.

4.1.1. Interfaz SpaceLike

Esta interfaz se implementa en las clases `ReifiedSpace`, `FailedSpace` y `MergedSpace`. Estas clases son abstracciones de los estado de los espacios, donde `FailedSpace` representa un espacio *failed*; `MergedSpace` representa un espacio mezclado; y `ReifiedSpace` representa los espacios en los que se puede realizar algún tipo de computación (*succeeded*, *distribuitable* o inestable).

Esta interfaz nos permite tener implementaciones distintas de las operaciones de los espacios, ya que estas presentan comportamientos distintos según el estado del espacio.

```
1  class SpaceLike;
2  template<>
3  struct Interface<SpaceLike>:
4    ImplementedBy<ReifiedSpace, FailedSpace, MergedSpace>,
5    NoAutoReflectiveCalls {
6
7    bool isSpace(RichNode self, VM vm);
8    UnstableNode askSpace(RichNode self, VM vm);
9    UnstableNode askVerboseSpace(RichNode self, VM vm);
10   UnstableNode mergeSpace(RichNode self, VM vm);
11   void commitSpace(RichNode self, VM vm, RichNode value);
12   UnstableNode cloneSpace(RichNode self, VM vm);
13   void killSpace(RichNode self, VM vm);
14   void injectSpace(RichNode self, VM vm, RichNode callable);
15   void waitStableSpace(RichNode self, VM vm);
```

```
16
17  #ifdef VM_HAS_CSS
18    void info(RichNode self, VM vm);
19    UnstableNode dataMergeSpace(RichNode self, VM vm);
20  #endif
21  };
```

Código 4.1: Interfaz SpaceLike

4.1.2. New

Para poder encontrar la solución a un CSP es necesario primero poder representar dicho CSP como un espacio de computación; para esto es necesario usar la operación `Space.new`.

```
S = {Space.new P}
```

Esta operación crea un espacio que posee un hilo que ejecuta el procedimiento `P`, y lo asigna a la variable `S`. En el procedimiento `P` se definen las variables, los propagadores y las estrategias de distribución del CSP; además debe recibir como parámetro la `rootvar`.

La implementación de esta operación en Mozart2, consiste en un built-in escrito en C++ en el módulo `Space`.

```
1   class New: public Builtin<New> {
2   public:
3     New(): Builtin("new") {}
4
5     static void call(VM vm, In target, Out result) {
6       // Create the space
7       Space* space = new (vm) Space(vm, vm->getCurrentSpace());
8
9       // Create the thread {Proc Root}
10      ozcalls::asyncOzCall(vm, space, target, *space->getRootVar());
11
12      // Create the reification of the space
13      result = ReifiedSpace::build(vm, space);
14    }
15  };
```

Código 4.2: Implementación Built-in Space.new

En la implementación del built-in `Space.new` en la linea 5 se definen las entradas (`In`) y las salidas (`Out`) del built-in; luego observamos que en la línea 7 se crea un espacio el cual tiene de hijo al *current space*, esto quiere decir que el espacio que se acaba de crear es un nodo hijo del *current space*. En la línea 10 se crea el hilo con el procedimiento `target` y se asocia al espacio `space` y a la variable raíz del espacio `*space->getRootVar()`.

4.1.3. Ask

Durante la búsqueda es necesario saber el estado del espacio, ya que éste es el que nos va a informar si hemos encontrado una solución, si no hay solución o si todavía podemos seguir explorando. Para conocer el estado de los espacios utilizamos la operación `Space.ask`.

```
E = {Space.ask S}
```

Esta operación espera hasta que el espacio `S` sea estable, después de esto obtiene el estado del espacio y lo almacena en la variable `E`. Si el espacio es *succeeded* entonces `E = succeeded`, si el espacio es *failed* entonces `E = failed` y por ultimo si el espacio es *distribuitable* entonces `E = alternatives(N)`, donde `N` es el número de alternativas en las que se puede dividir el espacio.

La implementación de esta operación se divide en 2 partes: la implementación del built-in en el módulo `Space` y la implementación del método `askSpace` de la interfaz `SpaceLike`.

La implementación del built-in solo consiste en el llamado al método `askSpace` de la interfaz `SpaceLike` y asignar el valor que retorna en la variable de salida.

La interfaz `SpaceLike` está implementada sobre las clases `ReifiedSpace`, `FailedSpace` y `MergedSpace`. En las clases `FailedSpace` y `MergedSpace` el método `askSpace` retorna el átomo `failed` y `merged` respectivamente. La implementación de `askSpace` en la clase `ReifiedSpace` es la siguiente:

```
1  UnstableNode ReifiedSpace::askSpace(RichNode self, VM vm) {
2    using namespace patternmatching;
3
4    Space* space = getSpace();
5
6    if (!space->isAdmissible(vm))
7      raise(vm, vm->coreatoms.spaceAdmissible, self);
8
9    RichNode statusVar = *space->getStatusVar();
10
11   if (matchesTuple(vm, statusVar, vm->coreatoms.succeeded, wildcard())) {
12     return Atom::build(vm, vm->coreatoms.succeeded);
13   } else {
14     return { vm, statusVar };
15   }
16 }
```

Código 4.3: Implementación ReifiedSpace::askSpace

En la anterior implementación observamos que en la línea 9 se obtiene la variable de estado del espacio; luego en la línea 11, esta variable se pasa a la función `matchesTuple`, que

compara dos tuplas; en esta función se espera hasta que la variable estado tenga asignado un valor, es decir, que el espacio sea estable y luego se compara con el átomo `succeeded`. Si son iguales entonces se retorna el átomo `succeeded` de lo contrario se retorna el valor asignado a la variable de estado.

Durante el desarrollo de este trabajo de grado se presentó un problema con ésta operación, ya que retornaba espacios *succeeded* cuando estos eran *failed* o *distribuitable*. Esto era debido a que el método encargado de determinar la estabilidad de un espacio estaba incompleto. Faltaba verificar que en el espacio no existieran hilos suspendidos.

La verificación de estabilidad se realiza cada que un hilo termina de ejecutarse, por medio del método `isStable()` de la clase `Space`. Si el espacio es estable entonces procede a ligar la variable de estado del espacio con el valor correspondiente (`succeeded`, `failed` o `alternatives(n)`).

Para solucionar el anterior problema se encontró una solución temporal, la cual permite el desarrollo de este trabajo de grado. Esta solución consistía en tener un contador de hilos suspendidos en el espacio la cual incrementaba cada que un hilo se creaba o suspendía y disminuía cuando un hilo se reanudaba; y por último se puso una condición en el método `isStable()` en la que se verificaba que ese contador fuera 0 para que el espacio fuera estable. Esta solución es temporal ya que a pesar de que soluciona el problema para la búsqueda, la estabilidad es de gran importancia para otros aspectos en la máquina virtual, por esta razón es necesario un estudio más detallado de este problema, el cual se encuentra por fuera del alcance de este trabajo de grado.

4.1.4. Clone

Para poder seleccionar las alternativas que se presentan al momento de la distribución es necesario realizar copias del espacio a distribuir para poder aplicar a cada una de estas las distintas alternativas a tomar. Estas copias las realiza la operación `Space.clone`.

```
C = {Space.clone S}
```

Esta operación espera hasta que el espacio `S` sea estable, luego realiza una copia de `S` y la almacena en `C`.

Al igual que la operación `Space.ask`, esta operación está implementada en dos partes, la implementación del built-in y la implementación del método `cloneSpace` de la interfaz `SpaceLike`.

El built-in de ésta operación solo consiste en un llamado al método `cloneSpace` de la interfaz `SpaceLike` y asignar el valor que retorna en la variable de salida.

Al igual que `askSpace`, `cloneSpace` está implementado sobre las clases `ReifiedSpace`, `FailedSpace` y `MergedSpace`. En la clase `MergedSpace` el método `cloneSpace` lanza una excepción y en la clase `FailedSpace` retorna un espacio fallido. La implementación de `cloneSpace` en la clase `ReifiedSpace` es la siguiente:

```
1  UnstableNode ReifiedSpace::cloneSpace(RichNode self, VM vm) {
2    Space* space = getSpace();
3
4    if (!space->isAdmissible(vm))
5      raise(vm, vm->coreatoms.spaceAdmissible);
6
7    RichNode statusVar = *space->getStatusVar();
8    if (statusVar.isTransient())
9      waitFor(vm, statusVar);
10
11   Space* copy = space->clone(vm);
12   return ReifiedSpace::build(vm, copy);
13 }
```

Código 4.4: Implementación ReifiedSpace::cloneSpace

Ésta implementación consiste en obtener la variable de estado del espacio en la línea 7, en la línea 8 se pregunta si la variable de estado está ligada; sino está ligada, se espera hasta que se ligue. En la línea 11 se realiza la copia del espacio; por último se retorna la copia del espacio.

4.1.5. Commit

Para imponer una de las alternativas de la distribución sobre un espacio, es necesaria la operación `Space.commit`.

{`Space.commit S A`}

Esta operación espera hasta que el espacio `S` sea estable; luego aplica la alternativa `A` sobre el espacio.

Esta operación esta implementada en dos partes, la implementación del built-in y la implementación del método `commitSpace` de la interfaz `SpaceLike`.

El built-in de esta operación consiste únicamente en el llamado al método `commitSpace` de la interfaz `SpaceLike`.

Al igual que todos los métodos de la interfaz **SpaceLike**, `commitSpace` está implementado sobre las clases `ReifiedSpace`, `FailedSpace` y `MergedSpace`. En la clase `FailedSpace`

el método `commitSpace` no hace nada y en la clase `MergedSpace` lanza una excepción. El comportamiento en la clase `ReifiedSpace` se mostrará a continuación.

```
1  void ReifiedSpace::commitSpace(RichNode self, VM vm, RichNode value) {
2    using namespace patternmatching;
3
4    Space* space = getSpace();
5
6    if (!space->isAdmissible(vm))
7      raise(vm, vm->coreatoms.spaceAdmissible);
8
9  #ifdef VM_HAS_CSS
10   if(space->hasConstraintSpace()){
11     GecodeSpace& home = space->getCstSpace(true);
12     if(home.branchers()!=0){
13       nativeint alt= getArgument<nativeint>(vm,value, "integer");
14       const Gecode::Choice *ch = home.choice();
15       home.commit(*ch, (unsigned int)alt-1);
16       return;
17     }
18   }
19 #endif
20
21   if (!space->hasDistributor())
22     raise(vm, vm->coreatoms.spaceNoChoice, self);
23
24   nativeint left = 0, right = 0;
25
26   if (matches(vm, value, capture(left))) {
27     int commitResult = space->commit(vm, left);
28     if (commitResult < 0)
29       raise(vm, vm->coreatoms.spaceAltRange, self, left, -commitResult);
30   } else if (matchesSharp(vm, value, capture(left), capture(right))) {
31     raise(vm, "notImplemented", "commitRange");
32   } else {
33     raiseTypeError(vm, "int or range", value);
34   }
35 }
```

Código 4.5: Implementación ReifiedSpace::commitSpace

El método `commitSpace` se divide en dos partes, la primera trabaja sobre distribuidores escritos en Gecode (líneas 9-19) y la segunda sobre los distribuidores escritos en Mozart (líneas 21-34). En este trabajo de grado sólo se utilizarán los distribuidores escritos en Gecode, debido a que a la fecha el soporte para distribuidores escritos en Mozart todavía esta en desarrollo.

El método `commitSpace` usando los distribuidores de Gecode consiste en: primero saber si el espacio posee un espacio de restricciones o `GecodeSpace` (línea 10); Si es asi y se verifica que el espacio tenga distribuidores por medio del método `branchers()` soportado por Gecode (línea 12). Por último se aplica la alternativa por medio de la operación `commit` soportada por

Gecode (línea 15), es importante hacer `alt-1` debido a que en Mozart la primera alternativa se identifica con 1 y en Gecode esta se identifica con 0.

4.1.6. Merge

Para poder extraer la solución del espacio que contiene el CSP, y hacerla visible desde el *toplevel space*, espacio donde se realiza la búsqueda, es necesario traer toda la información del espacio solución al *toplevel space*, lo cual se realiza con la operación `Space.merge`.

```
M = {Space.merge S}
```

Esta operación consiste en tomar toda la información que hay en el espacio `S` (variables de restricción, propagadores, distribuidores e hilos) y almacenarla en el *current space*, luego obtiene la variable raíz del espacio `S` y la almacena en la variable `M`.

Space.dataMerge

Al usar programación por restricciones en *Mozart2* se presenta un problema al usar la operación `Space.merge` debido a que según la definición de la operación al mezclar se deben llevar al *current space* los propagadores, las variables de restricción y los distribuidores, pero estos en *Mozart2* se encuentran almacenados en un `GecodeSpace` y en Gecode no existe la operación *merge* lo que impide mezclar esta información con el *current space*.

En la búsqueda la operación `Space.merge` es necesaria para poder obtener la solución de un CSP, por esta razón se plantea hacer una versión simplificada de `Space.merge`, que solo almacene en `M` la variable raíz del espacio `S`, la cual almacena la solución al CSP. A esta versión simplificada del `Space.merge` la llamaremos `Space.dataMerge`.

```
M = {Space.dataMerge S}
```

La implementación de esta operación esta dividida en 2 partes, la implementación del built-in en el módulo `Space` y la implementación del método `dataMergeSpace` en la interfaz `SpaceLike`.

La implementación del built-in de esta operación consiste únicamente en el llamado al método `dataMergeSpace` y asignar el valor que esta retorna a la variable de salida.

El método `dataMergeSpace` en la clase `FailedSpace` hace que el *current space* falle; en la clase `MergedSpace` lanza una excepción. El comportamiento en la clase `ReifiedSpace` se explica a continuación.

```
1  UnstableNode ReifiedSpace::dataMergeSpace(RichNode self, VM vm) {
2    Space* currentSpace = vm->getCurrentSpace();
3    Space* space = getSpace();
4
5    if (!space->isAdmissible(currentSpace))
6      raise(vm, vm->coreatoms.spaceAdmissible);
7
8    // Update status var
9    RichNode statusVar = *space->getStatusVar();
10   if (statusVar.isTransient()) {
11     UnstableNode atomMerged = Atom::build(vm, vm->coreatoms.merged);
12     DataflowVariable(statusVar).bind(vm, atomMerged);
13   }
14
15   // Extract root var
16   auto result = mozart::build(vm, *space->getRootVar());
17
18   // Become a merged space
19   self.become(vm, MergedSpace::build(vm));
20
21   return result;
22 }
```

Código 4.6: Implementación ReifiedSpace::dataMergeSpace

Primero se verifica que el espacio `S` sea admisible por el *current space* (línea 5); luego se actualiza la variable de estado del espacio (línea 12); despues se extrae el valor de la variable raiz y se almacena en la variable `result` (línea 16); luego el se transforma el espacio en un `MergedSpace` (línea 19); por último se retorna la variable `result`.

4.1.7. Inject

Cuando necesitamos encontrar una solución óptima a un CSP, cada que encontremos una posible solución, debemos aplicar un función objetivo sobre nuestro CSP que nos permitirá solo tener en cuenta soluciones que sean mejor a la solución ya encontrada. Ésta función objetivo está compuesta por una o varias restricciones. Para poder aplicar la función objetivo sobre nuestro CSP, es necesario utilizar la operación `Space.inject`.

```
{Space.inject S P}
```

Esta operación añade al espacio `S` un hilo que ejecuta el procedimiento `P` aplicado sobre la `rootvar` del espacio. En este procedimiento se define la función objetivo.

La implementación de esta operación está dividida en 3 partes, la implementación del built-in en el módulo `Space`, la implementación del método `injectSpace` de la interfaz `SpaceLike` y la implementación del método `inject` en la clase `Space`.

La implementación del built-in de esta operación consiste únicamente en el llamado al método `injectSpace`.

Al igual que los otros métodos de la interfaz `SpaceLike`, `injectSpace` está implementado sobre las clases `ReifiedSpace`, `FailedSpace` y `MergedSpace`. En cada una de estas clases el método `injectSpace` tiene un comportamiento diferente. En la clase `ReifiedSpace` se aplica el método `inject` de la clase `Space` sobre el espacio; en la clase `FailedSpace` no se realiza nada y en la clase `MergedSpace` se lanza una excepción.

```
1 void Space::inject(VM vm, RichNode callable) {
2   Space* src = this;
3   src->clearStatusVar(vm);
4   ozcalls::asyncOzCall(vm, src, callable, *src->getRootVar());
5 }
```

Código 4.7: Implementación Space::inject

La implementación del método `inject` consiste primero en desligar la variable de estado del espacio (línea 3), esto es necesario porque después de aplicar la operación `Space.inject` la estabilidad del espacio se ve afectada. Luego se crea un hilo en el espacio con la función objetivo.

4.2. Implementación Motores de Búsqueda desde Mozart2

Al contar con todas las operaciones de los espacios en *Mozart2*, el usuario puede escribir sus propios motores de búsqueda desde *Mozart2*. Al escribir un motor de búsqueda, este se debe implementar en dos partes: *el motor* y *la exploración*, esto es necesario ya que la exploración de un CSP es recursiva.

[el motor] Es el encargado de crear un espacio dado un CSP, llamar a *la exploración* y retornar la solución, según lo que se necesite, un espacio solución, la variable raíz del espacio o alguna otra forma de solución que necesite el usuario.

[la exploración] Es la encargada de explorar el árbol de búsqueda, para encontrar la solución al CSP.

A continuación se muestra como se implementa un motor de búsqueda que encuentra una solución y utiliza como estrategia de exploración *depth-first* en *Mozart2*.

```
1 fun {Exploracion S}
2   case {Space.ask S}
3   of failed then nil
```

```
4     [] succeeded then [S]
5     [] alternatives(2) then C={Space.clone S} in
6        {Space.commit S 1}
7        case {Exploracion S}
8        of nil then {Space.commit C 2} {Exploracion C}
9        elseof [O] then [O]
10       end
11    end
12 end
```

Código 4.8: Implementación de *la exploración* de un motor de búsqueda

La implementación de *la exploración* consiste en preguntar por el estado del espacio `S` por medio de la operación `Space.ask` (línea 2); si el estado del espacio es `failed` se retorna el átomo `nil`; si el estado del espacio es `succeeded` se retorna una lista con el espacio. Si el estado del espacio es `distribuitable` (línea 5) entonces se crea un clon del espacio `S` por medio de la operación `Space.clone` y se almacena en la variable `C`; se realiza por medio de la operación `Space.commit` el *commit* de la alternativa `1` sobre el espacio `S` (línea 6) y se aplica recursivamente *la exploración* sobre este. Si *la exploración* de `S` no encuentra solución (línea 8) se aplica la segunda alternativa al espacio `C`; de lo contrario se retorna la solución (línea 9).

```
1  fun {Motor P}
2    case {Exploracion {Space.new P}} of nil then nil
3    [] [S] then [{Space.dataMerge S}]
4    end
5  end
```

Código 4.9: Implementación de *el motor* de un motor de búsqueda

La implementación de *el motor* consiste en crear un espacio a partir del procedimiento `P` utilizando la operación `Space.new` y aplicar sobre este *la exploración*. Si el resultado de *la exploración* es el átomo `nil`, es decir, no se encontró solución, se retorna este mismo valor; de lo contrario se retorna una lista con la solución. La forma en que se retorna la solución depende del motor de búsqueda que se implemente, en este caso se retorna la variable raíz del espacio por medio de la operación `Space.dataMerge`.

4.3. Motores de Búsqueda de Mozart

El poder escribir motores de búsqueda desde *Mozart2* permite integrar los motores de búsqueda ya existentes en *Mozart1.4*. Estos motores de búsqueda están descriptos en la Tabla 4.1.

Motor	Tipo de Exploración	Tipo de Solución
Depth	*depth-first*	Una solución
Bound	*depth-first*	Una solución
Iter	*depth-first*	Una solución
All	*depth-first*	Todas las soluciones
BAB	*depth-first*	Solución óptima
Restart	*depth-first*	Solución óptima

Tabla 4.1: Motores de búsqueda disponibles en *Mozart1.4*

[Depth] Motor de búsqueda para una solución, utiliza como estrategia de exploración *depth-first*.

[Bound] Motor de búsqueda para una solución, utiliza como estrategia de exploración *depth-first*. Recibe como parámetro un entero que indica el limite en profundidad en el *árbol de soluciones* que alcanzará para encontrar una solución.

[Iter] Motor de búsqueda para una solución, ejecuta el motor de búsqueda *Bound*, primero con limite igual a 1; si no encuentra solución realiza ahora la búsqueda con limite igual a 2 y así sucesivamente hasta encontrar una solución.

[All] Motor de búsqueda para todas las soluciones, utiliza como estrategia de exploración *depth-first*.

[BAB] (*Branch and Bound*) Motor de búsqueda para soluciones óptimas, cada que encuentra una solución añade sobre los nodos restantes una restricción para optimizar la solución.

[Restart] Motor de búsqueda para soluciones óptimas, cada que encuentra una solución añade sobre el nodo raíz una restricción para optimizar la solución y reinicia la búsqueda.

Para cada uno de estos motores de búsqueda existen tres versiones de ellos, los cuales difieren en la forma como la solución es retornada. En la primera forma se retorna es la variable raíz del espacio, aplicando la operación `Space.merge`. En la segunda versión se retorna es el espacio solución, aplicando la función `wrapS` (Código 4.10); a esta versión se le añade al nombre del motor la letra S. La ultima versión consiste en retornar la solución encapsulada en un procedimiento, aplicando la función `wrapP` (Código 4.11); a esta versión se le añade al nombre del motor la letra P.

```
fun {WrapS S}
   S
end
```

Código 4.10: Procedimiento wrapS

```
fun {WrapP S}
  proc {$ X}
    {Space.dataMerge {Space.clone S} X}
  end
end
```

Código 4.11: Procedimiento wrapP

En la Tabla 4.2 se pueden observar los motores de búsqueda disponibles en *Mozart1.4* con sus diferentes versiones según la forma en que retornan la solución.

Nombre	`Space.dataMerge`	`wrapS`	`wrapP`
Depth	`Search.one.depth`	`Search.one.depthS`	`Search.one.depthP`
Bound	`Search.one.bound`	`Search.one.boundS`	`Search.one.boundP`
Iter	`Search.one.iter`	`Search.one.iterS`	`Search.one.iterP`
All	`Search.all`	`Search.allS`	`Search.allP`
BAB	`Search.best.bab`	`Search.best.babS`	`Search.best.babP`
Restart	`Search.best.restart`	`Search.best.restartS`	`Search.best.restartP`

Tabla 4.2: Motores de búsqueda disponibles en *Mozart1.4* y sus diferentes versiones.

Existen tres funciones de acceso directo a los motores de búsqueda:

[SearchOne] Ejecuta el motor de búsqueda `Search.one.depth`.

[SearchAll] Ejecuta el motor de búsqueda `Search.all`.

[SearchBest] Ejecuta el motor de búsqueda `Search.best.bab`.

CAPÍTULO 5

Búsqueda en Gecode

Una de las ventajas del uso de Gecode como motor de restricciones de *Mozart2*, es que nos permite contar con todas las herramientas que posee Gecode para programación por restricciones en *Mozart2*, una de estas herramientas son los motores de búsqueda.

En este capítulo se hablará sobre los mecanismos necesarios para realizar la integración de los motores de búsqueda de Gecode con *Mozart2*.

5.1. Motores de Búsqueda en Gecode

Gecode ofrece tres motores de búsquedas, los cuales nos permiten encontrar soluciones simples o soluciones óptimas. Estos motores de búsqueda son: *DFS*, *BAB* y *Restart*.

Motor	Tipo de Exploración	Tipo de Solución
DFS	*depth-first*	Todas las soluciones
BAB	*depth-first*	Solución óptima
Restart	*depth-first*	Solución óptima

Tabla 5.1: Motores de búsqueda disponibles en Gecode

5.2. Integración de los Motores de Búsqueda de Gecode con Mozart2

Para integrar los motores de búsqueda de Gecode con *Mozart2* es necesario implementar un built-in por cada uno de estos. Adicionalmente se creó la interfaz `ConstraintSpace` la cual se implementa sobre la clase `ReifiedSpace`. Esta interfaz nos permite saber si un espacio tiene un `GecodeSpace`, obtener el `GecodeSpace` de un espacio y actualizar el `GecodeSpace` de un espacio.

```
1  #ifdef VM_HAS_CSS
2  class ConstraintSpace;
3  template<>
4  struct Interface<ConstraintSpace>:
5    ImplementedBy<ReifiedSpace>,
6    NoAutoReflectiveCalls {
7
8    bool isConstraintSpace(RichNode self, VM vm);
9    GecodeSpace* constraintSpace(RichNode self, VM vm);
10   void updateConstraintSpace(RichNode self, VM vm, GecodeSpace* gs);
11 };
12 #endif
```

Código 5.1: Interfaz ConstraintSpace

En la Figura 5.1 se muestra el proceso que se realiza al ejecutar desde *Mozart2* un motor de búsqueda escrito en Gecode.

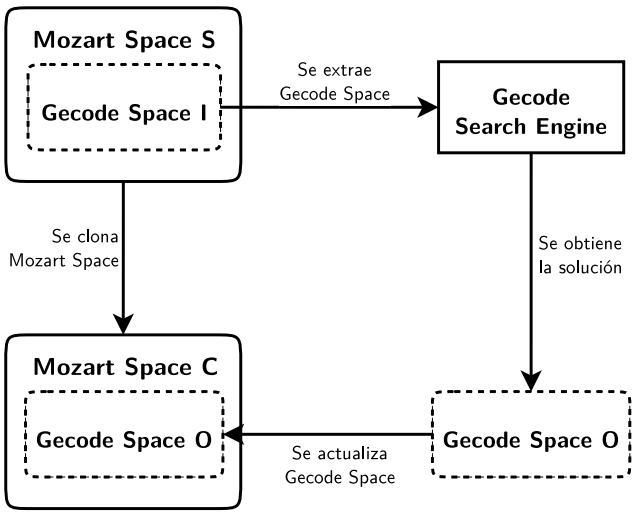


Figura 5.1: Ejecución general de un motor de búsqueda de Gecode en *Mozart2*.

Para ejecutar un motor de búsqueda de Gecode desde *Mozart2* primero debemos extraer el `GecodeSpace` de nuestro espacio de Mozart; luego se realiza la solución del CSP usando el motor de búsqueda deseado y se le pasa como parámetro el `GecodeSpace`; cuando se obtenga la solución al CSP la cual es un nuevo `GecodeSpace` se procede a realizar un clon del espacio de Mozart y se actualiza el `GecodeSpace` de este con el `GecodeSpace` solución obtenido después de aplicar el motor de búsqueda.

A continuación se explicará detalladamente la implementación de cada uno de los built-in encargados de realizar la ejecución de los motores de búsqueda de Gecode.

5.2.1. Búsqueda de una solución

Para realizar la búsqueda de una solución de un CSP, se utiliza el motor de búsqueda `DFS` de Gecode.

```
1  class OneDFS: public Builtin<OneDFS> {
2  public:
3    OneDFS(): Builtin("oneDFS") {}
4
5    static void call(VM vm, In space, Out result) {
6      SpaceLike(space).waitStableSpace(vm);
7      if (ConstraintSpace(space).isConstraintSpace(vm)) {
8        GecodeSpace* gs = ConstraintSpace(space).constraintSpace(vm);
9        Gecode::DFS<GecodeSpace> e(gs);
10       if (GecodeSpace *sol = e.next()) {
11         result = SpaceLike(space).cloneSpace(vm);
12         ConstraintSpace(result).updateConstraintSpace(vm, sol);
13       }
14       else
15         result = build(vm, vm->coreatoms.nil);
16     }
17   }
18 };
```

Código 5.2: Built-in DFS una solución

Lo primero que se debe tener en cuenta al integrar un motor de búsqueda de Gecode, es que debemos hacer el llamado al método `waitStableSpace` (línea 6), el cual suspende la ejecución del built-in hasta que el espacio sea estable; esto se debe hacer para garantizar que la representación del CSP es completa. Si no se realiza es posible que al ejecutar el motor de búsqueda algunos propagadores no hayan sido impuestos.

En la línea 7 se verifica que el espacio posea un espacio de restricciones (`GecodeSpace`), si es así se crea un puntero al `GecodeSpace` (línea 8); luego se crea una instancia del motor de búsqueda `DFS` (línea 9), esta recibe como parámetro el `GecodeSpace` a solucionar.

Después de tener la instancia del motor de búsqueda `DFS` se procede a encontrar una solución al CSP por medio del método `next()` soportado por el motor de búsqueda de Gecode (línea 10); si se encuentra alguna solución entonces se crea un clon del espacio de Mozart (línea 11), a este clon se le actualiza el `GecodeSpace` usando la solución encontrada y se almacena el clon en la variable de salida; de no encontrar solución se almacena en la variable de salida el átomo `nil`.

5.2.2. Búsqueda de todas las soluciones

Para realizar la búsqueda de todas las soluciones de un CSP, se utiliza el motor de búsqueda `DFS` de Gecode.

```
1  class AllDFS: public Builtin<AllDFS> {
2  public:
3    AllDFS(): Builtin("allDFS") {}
4
5    static void call(VM vm, In space, Out result) {
6      SpaceLike(space).waitStableSpace(vm);
7      if (ConstraintSpace(space).isConstraintSpace(vm)) {
8        GecodeSpace* gs = ConstraintSpace(space).constraintSpace(vm);
9        Gecode::DFS<GecodeSpace> e(gs);
10       result = build(vm, vm->coreatoms.nil);
11       while (GecodeSpace *sol = e.next()) {
12         UnstableNode temp = SpaceLike(space).cloneSpace(vm);
13         ConstraintSpace(temp).updateConstraintSpace(vm, sol);
14         result = buildCons(vm, temp, std::move(result));
15       }
16     }
17   }
18 };
```

Código 5.3: Built-in DFS todas las soluciones

La implementación del built-in para todas las soluciones es similar al built-in de una solución; la principal diferencia es en el tipo de salida que cada una retorna; para el built-in de una solución se retorna un espacio de Mozart con la solución al CSP y en el built-in de todas las soluciones se retornara una lista con un espacio de Mozart por cada solución que se encuentre.

Como consecuencia de lo anterior la otra diferencia entre estos dos built-in está al momento de encontrar la solución (línea 11); en el built-in de todas las soluciones no se pregunta si existe una solución sino que mientras se hallen soluciones se creará un clon del espacio de Mozart (línea 12), el `GecodeSpace` de este clon se actualizará con la solución encontrada (línea 13) y por último se añade el clon a la lista de soluciones.

5.2.3. Búsqueda solución óptima

Para realizar la búsqueda de la solución óptima de un CSP, se utilizan los motores de búsqueda `BAB` y `Restart` de Gecode. Para poder utilizar estos motores de búsqueda en Gecode se debe implementar el método `constraint()` en el CSP que se va a solucionar, este método es el que va a contener la función objetivo a evaluar cada que se encuentre una posible solución. En *Mozart2* este método se debe implementar sobre la implementación de la clase `GecodeSpace`.

Pero como los usuarios de Mozart no pueden editar este método, ellos deben escribir su función objetivo desde Mozart como un procedimiento, de la misma manera que se ha realizado siempre en Mozart utilizando sus motores de búsqueda. Entonces el método `cosntraint()` debe realizar la ejecución de este procedimiento cada que se encuentre una solución.

En el estado actual de *Mozart2* no es posible integrar los motores de búsqueda para soluciones óptimas. Esto es debido a que la única forma de poder ejecutar la función objetivo es agregando un hilo al espacio de Mozart por medio de la operación `Space.inject` que ejecute el procedimiento que contiene esta función. Como la operación `Space.inject` trabaja sobre espacios de Mozart es necesario poder obtener el espacio de Mozart desde un `GecodeSpace` para poder ejecutar la operación `Space.inject` desde el método `constraint()`, pero en este momento la comunicación entre el espacio de Mozart con el `GecodeSpace` es en una sola dirección, es decir, únicamente se puede obtener el `GecodeSpace` desde un espacio de Mozart pero no al revés.

En el desarrollo de este trabajo de grado se planteó modificar la comunicación entre el espacio de Mozart y el `GecodeSpace` para que fuera bidireccional, pero esta modificación afectaba partes criticas de *Mozart2*, como lo es la recolección de basura entre otras. Por esta razón este problema necesita un análisis más profundo, el cual se sale del alcance de este trabajo de grado.

A pesar de este problema, se puede plantear una posible implementación de los built-in de los motores de búsqueda `BAB` y `Restart` debido a que el llamado al método `constraint()` está encapsulado en los motores de búsqueda.

```
1  class BAB: public Builtin<BAB> {
2  public:
3    BAB(): Builtin("bab") {}
4
5    static void call(VM vm, In space, Out result) {
6      SpaceLike(space).waitStableSpace(vm);
7      if (ConstraintSpace(space).isConstraintSpace(vm)) {
8        GecodeSpace* gs = ConstraintSpace(space).constraintSpace(vm);
9        Gecode::BAB<GecodeSpace> e(gs);
```

```
10        if (GecodeSpace *sol = e.next()) {
11          result = SpaceLike(space).cloneSpace(vm);
12          ConstraintSpace(result).updateConstraintSpace(vm, sol);
13        }
14        else
15          result = build(vm, vm->coreatoms.nil);
16      }
17    }
18  };
```

Código 5.4: Posible built-in BAB

```
1   class Restart: public Builtin<Restart> {
2   public:
3     Restart(): Builtin("restart") {}
4
5     static void call(VM vm, In space, Out result) {
6       SpaceLike(space).waitStableSpace(vm);
7       if (ConstraintSpace(space).isConstraintSpace(vm)) {
8         GecodeSpace* gs = ConstraintSpace(space).constraintSpace(vm);
9         Gecode::Restart<GecodeSpace> e(gs);
10        if (GecodeSpace *sol = e.next()) {
11          result = SpaceLike(space).cloneSpace(vm);
12          ConstraintSpace(result).updateConstraintSpace(vm, sol);
13        }
14        else
15          result = build(vm, vm->coreatoms.nil);
16      }
17    }
18  };
```

Código 5.5: Posible built-in Restart

5.3. Implementación de la Interfaz

Para que los usuarios de *Mozart2* puedan utilizar los motores de búsqueda de Gecode, aparte de los built-in, se deben implementar unos procedimientos en el módulo de búsqueda los cuales sirven de interfaz entre los usuarios y los built-in de los motores de búsqueda de Gecode. A continuación se mostrará la implementación de dichos procedimientos.

```
1  fun {SearchDFS S Ns}
2    case Ns
3    of one then {Search.oneDFS S}
4    [] all then {Search.allDFS S}
5    end
6  end
```

Código 5.6: Implementación función SearchDFS

En el Código 5.6 se muestra la implementación de la función `SearchDFS`, esta función nos permite usar el motor de búsqueda `DFS`. Recibe como parámetros de entrada un espacio `S` y un átomo `Ns` el cual puede ser `one` o `all`. Si el átomo `Ns` es igual a `one` se ejecuta el built-in `Search.oneDFS`; si el átomo `Ns` es igual a `all` se ejecuta el built-in `Search.allDFS`.

Para poder usar esta función desde *Mozart2* se debe exportar con un átomo que la identifique. Esta función esta identificada con el átomo `dfs`. Desde *Mozart2* esta función se usaría de la siguiente manera:

```
{Search.dfs S Ns}
```

Normalmente los usuarios de Mozart cuando van a resolver CSP, no manipulan los espacios directamente, ellos escriben su CSP como un procedimiento y luego este se pasa como parámetro a los motores de búsqueda. Por esta razón se brinda una interfaz que le permita a los usuarios ingresar como parámetro un procedimiento.

Esta interfaz se define como una tupla de funciones, donde cada elemento de la tupla representa una función encargada de ejecutar un motor de búsqueda.

```
1   SearchGecode = gecode(dfsOne: fun {$ P}
2                                   case {OneDFSGecode P}
3                                   of nil then nil
4                                   elseof S then [{Space.dataMerge S}]
5                                   end
6                                 end
7                         dfsAll: fun {$ P}
8                                   case {AllDFSGecode P}
9                                   of nil then nil
10                                  elseof S then {AuxAllSol S Space.dataMerge nil}
11                                  end
12                                end
13                        )
```

Código 5.7: Interfaz motores de búsqueda de Gecode

Las funciones en esta tupla consisten en llamar una función auxiliar que llama al motor de búsqueda especifico con un procedimiento como parámetro (líneas 2 y 8); luego según el resultado del motor de búsqueda se aplica el tratamiento adecuado a la solución para poder retornarla (líneas 4 y 10).

Las funciones anteriores utilizan unas funciones auxiliares, estas funciones son:

[OneDFSGecode] Recibe como entradas un procedimiento `P`, con el cual crea un espacio `S` y aplica la función `SearchDFS` sobre el espacio `S` con el átomo `one`. (Ver Código 5.8)

[AllDFSGecode] Recibe como entradas un procedimiento `P`, con el cual crea un espacio `S` y aplica la función `SearchDFS` sobre el espacio `S` con el átomo `all`. (Ver Código 5.9)

[AuxAllSol] Recibe como parámetro una lista de espacios `L`, un procedimiento O, y una lista `S` (en la primera ejecución `S = nil`). Si la lista es de la forma `H|T` entonces hace el llamado recursivo de ella misma pero con los siguientes valores: `L` ahora es `T`, la operación sigue siendo la misma, y `S` es ahora la lista formada por ejecutar O H y por `S`. Si la lista es vacía entonces se retorna a `S`. (Ver Código 5.10)

```
1  fun {OneDFSGecode P}
2    S
3  in
4    S={Space.new P}
5    {SearchDFS S one}
6  end
```

Código 5.8: Implementación función OneDFSGecode

```
1  fun {AllDFSGecode P}
2    S
3  in
4    S={Space.new P}
5    {SearchDFS S all}
6  end
```

Código 5.9: Implementación función AllDFSGecode

```
1  fun {AuxAllSol L O ?S}
2    case L
3    of H|T then {AuxAllSol T O {O H}|S}
4    [] nil then S
5    end
6  end
```

Código 5.10: Implementación función AuxAllSol

De las anteriores funciones las únicas disponibles para los usuarios de *Mozart2* son las funciones de la tupla de funciones (Código 5.7). Las cuales se pueden usar de la siguiente manera:

```
{Search.gecode.dfsOne P}
{Search.gecode.dfsAll P}
```

CAPÍTULO 6

Experimentación y Pruebas

En este capítulo se mostrarán las pruebas que se realizaron para verificar el correcto funcionamiento de los motores de búsqueda en *Mozart2*. Estas pruebas se dividirán en tres partes, primero se mostrarán las pruebas con las que se evalúan los motores de búsqueda de una solución; luego se mostrarán las pruebas para los motores de búsqueda de todas las soluciones y por ultimo las pruebas de los motores de búsqueda para soluciones óptimas.

Para la realización de estas pruebas se han seleccionado algunos problemas clásicos de la programación por restricciones, estos problemas son: *Safe*, *N Queens*, *Send More Money* y *Send Most Money*.

6.1. Pruebas Motores de Búsqueda para una solución

Primero evaluaremos los motores de búsqueda para una solución de Mozart, es decir, los que ya se soportaban en *Mozart1.4*, estos están explicados en la Sección 4.3. Esta prueba consiste en seleccionar un CSP, implementarlo en *Mozart2* y *Mozart1.4* y verificar que al aplicar el motor de búsqueda en ambas versiones de Mozart se obtengan los mismos resultados.

Para realizar esta prueba se ha seleccionado el CSP *Send More Money*, este problema tiene una única solución la cual es: `sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)` ($9567 + 1085 = 10652$).

Ejemplo 6.1 (Send More Money[1]). Este problema consiste en encontrar dígitos distintos para las letras D, E, M, N, O, R, S, Y; tal que S y M sean diferentes de 0 y que la siguiente ecuación se satisfaga.

$$SEND + MORE = MONEY$$

A continuación se muestra la implementación de *Send More Money* en *Mozart2* y *Mozart1.4*.

```
proc {More Root}
  S E N D M O R Y
in
  Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)
  {FD.dom 0#9 Root}
  {FD.distinct post(Root)}
  {FD.rel post(S '\\=:' 0)}
  {FD.rel post(M '\\=:' 0)}
  {FD.linear post([1000  100  10  1
                   1000  100  10  1
            ~10000 ~1000 ~100 ~10 ~1]
                  [S E N D
                   M O R E
                 M O N E Y] '=:' 0)}
  {FD.branch Root none val_min}
end
```

Código 6.1: Implementación *Send More Money* en *Mozart2*

```
proc {More Root}
   S E N D M O R Y
in
   Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)
   Root ::: 0#9
   {FD.distinct Root}
   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
   {FD.distribute naive Root}
end
```

Código 6.2: Implementación *Send More Money* en *Mozart1.4*

Estas dos implementaciones son similares, es decir, usan las mismas restricciones y la misma estrategia de distribución; se diferencian en la forma en que se escriben cada una de ellas, debido a que *Mozart2* aun no cuenta con *azúcar sintáctico*[2].

[1] http://mozart.github.io/mozart-v1/doc-1.4.0/fdt/node15.html

[2] Provee una abstración nueva de la sintaxis que reduce el tamaño del programa y mejora su legibilidad

SearchOne

Se evalúa el acceso directo para le ejecución del motor de búsqueda de una solución:

```
1 {Show {SearchOne More}}
```

En ambas versiones obtuvimos los siguientes resultados:

```
Mozart Engine 1.4.0 (20110605105756) playing Oz 3

[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
```

```
Mozart Engine 2.0.0-alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 -0500) playing Oz 3

[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
```

Se observa que en ambas versiones se obtiene la solucion correcta y con el mismo formato (una lista de soluciones) al usar `SearchOne`.

Search.one.depth

Se evalúa el motor de búsqueda de una solución *Depth*:

```
1 {Show {Search.one.depth More 1 _}}
```

En ambas versiones obtuvimos los siguientes resultados:

```
Mozart Engine 1.4.0 (20110605105756) playing Oz 3

[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
```

```
Mozart Engine 2.0.0-alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 -0500) playing Oz 3

[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
```

Se observa que en ambas versiones se obtiene la solución correcta y con el mismo formato (una lista de soluciones) al usar `Search.one.depth`.

Search.one.bound

Se evalúa el motor de búsqueda de una solución *Bound*, para este motor de búsqueda lo evaluaremos para los primeros seis niveles de profundidad en el árbol de solución:

```
1 {Show {Search.one.bound More 1 1 _}}
2 {Show {Search.one.bound More 2 1 _}}
3 {Show {Search.one.bound More 3 1 _}}
4 {Show {Search.one.bound More 4 1 _}}
5 {Show {Search.one.bound More 5 1 _}}
6 {Show {Search.one.bound More 6 1 _}}
```

En ambas versiones obtuvimos los siguientes resultados:

```
Mozart Engine 1.4.0 (20110605105756) playing Oz 3

cut
cut
cut
cut
cut
[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
```

```
Mozart Engine 2.0.0-alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 -0500) playing Oz 3

cut
cut
cut
cut
cut
[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
```

Se observa que en ambas versiones se obtienen las mismas soluciones al usar `Search.one.bound`. Del hecho de que la solución se encontró en el mismo nivel de profundidad nos da un indicio de que el árbol de búsqueda es el mismo.

Search.one.iter

Se evalúa el motor de búsqueda de una solución *Iter*:

```
1 {Show {Search.one.iter More 1 _}}
```

En ambas versiones obtuvimos los siguientes resultados:

```
Mozart Engine 1.4.0 (20110605105756) playing Oz 3

[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
```

```
Mozart Engine 2.0.0-alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 -0500) playing Oz 3

[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
```

Se observa que en ambas versiones se obtiene la solución correcta y con el mismo formato (una lista de soluciones) al usar `Search.one.iter`.

De las pruebas anteriores podemos concluir que el funcionamiento de los motores de búsqueda de Mozart funcionan correctamente en *Mozart2* y se puede brindar un soporte por lo menos en la búsqueda a CSP escritos en *Mozart1.4*.

Ahora evaluaremos que la versión para una solución del motor de búsqueda de Gecode *DFS*, obtenga soluciones correctas.

Para realizar esta prueba además de usar el Ejemplo 6.1 se ha seleccionado el CSP *Safe*, este problema solo tiene una única solución la cual es: `code(4 3 1 8 9 2 6 7 5)`.

Ejemplo 6.2 (Safe[3])**.** El código de seguridad del Profesor Smart es una secuencia de 9 dígitos $C_1...C_9$ distintos y diferentes de 0. Estos dígitos cumplen las siguientes ecuaciones e inecuaciones:

$$C_4 - C_6 = C_7$$

$$C_1 * C_2 * C_3 = C_8 + C_9$$

$$C_2 + C_3 + C_6 < C_8$$

$$C_9 < C_8$$

¿Cúal es el código de seguridad del Profesor Smart?

A continuación se muestra la implementación de *Safe* en *Mozart2*.

```
proc {Safe C}
  T
in
  {FD.tuple code 9 1#9 C}
  {FD.tuple temp 3 1#1000 T}
  {FD.distinct post(C)}
 % C.4 - C.6 =: C.7
  {FD.linear post([1 ~1 ~1]
                  [C.4 C.6 C.7] '=:' 0)}
 % C.1 * C.2 * C.3 =: C.8 + C.9
  {FD.mult post(C.1 C.2 T.1)}
  {FD.mult post(T.1 C.3 T.2)}
  {FD.linear post([1 1 ~1]
                  [C.8 C.9 T.3] '=:'0)}
  {FD.rel post(T.2 '=:' T.3)}
 % C.2 + C.3 + C.6 <: C.8
  {FD.linear post([1 1 1 ~1]
                  [C.2 C.3 C.6 C.8] '<:' 0)}
 % C.9 <: C.8
  {FD.linear post([1 ~1]
                  [C.9 C.8] '<:' 0)}
  {For 1 9 1
    proc {$ I}
      {FD.rel post(C.I '\\=:' I)}
    end}
  {FD.branch C size_min val_min}
  {FD.branch T size_min val_min}
end
```

Código 6.3: Implementación *Safe* en *Mozart2*

[3] http://mozart.github.io/mozart-v1/doc-1.4.0/fdt/node19.html

Search.gecode.dfsOne

Se evalúa la versión para una solución del motor de búsqueda de Gecode *DFS* buscando la solución para los problemas *Send More Money* y *Safe*:

```
1 {Show {Search.gecode.dfsOne More}}
2 {Show {Search.gecode.dfsOne Safe}}
```

Se obtuvo el siguiente resultado:

```
Mozart Engine 2.0.0-alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 -0500) playing Oz 3

[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
[code(4 3 1 8 9 2 6 7 5)]
```

Se observa que se obtuvo la solución correcta para cada uno de los problemas después de aplicar `Search.gecode.dfsOne`.

Por último se evaluará el motor de búsqueda descrito en la Sección 4.2. Este se evaluará de igual manera que el motor de búsqueda `Search.gecode.dfsOne`.

```
1 {Show {Motor More}}
2 {Show {Motor Safe}}
```

Al evaluar los problemas *Send More Money* y *Safe* con el motor descrito en la Sección 4.2, se obtuvo el siguiente resultado:

```
Mozart Engine 2.0.0-alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 -0500) playing Oz 3

[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
[code(4 3 1 8 9 2 6 7 5)]
```

Se observa que el motor de búsqueda obtiene la solución correcta.

6.2. Pruebas Motores de Búsqueda para todas las soluciones

Al igual que en la sección anterior empezaremos evaluando los motores de búsqueda para todas las soluciones de Mozart. Para esta prueba seleccionaremos el problema *N Queens*, el cual es un problema que posee soluciones múltiples según un parámetro N. En la Tabla 6.1 se presentan las distintas soluciones para la entrada $N \in \{1, ..., 10\}$

Ejemplo 6.3 (N Queens[4])**.** Se necesita ubicar N reinas en un tablero de ajedres de $N \times N$, de tal manera que ningún par de reinas se ataquen entre ellas.

[4]http://mozart.github.io/mozart-v1/doc-1.4.0/fdt/node25.htm

N	1	2	3	4	5	6	7	8	9	10	11
Soluciones	1	0	0	2	10	4	40	92	352	724	2680

Tabla 6.1: Soluciones al problema *N Queens*

A continuación se mostrará la implementación del problema *N Queens* en *Mozart2* y *Mozart1.4*.

```
fun {Queens N}
  proc {$ Row}
    L1N = {MakeTuple c N}
    LM1N = {MakeTuple c N}
  in
    {FD.tuple queens N 1#N Row}
    {For 1 N 1
      proc {$ I}
        L1N.I = I
        LM1N.I = ~I
      end}
    {FD.distinct post(Row)}
    {FD.distinct post(LM1N Row)}
    {FD.distinct oist(L1N Row)}
    {FD.branch Row size_min val_min}
  end
end
```

Código 6.4: Implementación *N Queens* en *Mozart2*

```
fun {Queens N}
  proc {$ Row}
    L1N ={MakeTuple c N}
    LM1N={MakeTuple c N}
  in
    {FD.tuple queens N 1#N Row}
    {For 1 N 1
      proc {$ I}
        L1N.I=I
        LM1N.I=~I
      end}
    {FD.distinct Row}
    {FD.distinctOffset Row LM1N}
    {FD.distinctOffset Row L1N}
    {FD.distribute ff Row}
  end
end
```

Código 6.5: Implementación *N Queens* en *Mozart1.4*

SearchAll

Se evalúa el acceso directo para la ejecución del motor de búsqueda de todas las soluciones. Esta prueba consistirá en mostrar la solución para el problema *N Queens* donde $N \in \{1, ..., 6\}$ y mostrar el tamaño de la solución para los valores de $N \in \{7, ..., 10\}$ y comparar que los resultados obtenidos sean iguales entre *Mozart2* y *Mozart1.4*

```
{Show {SearchAll {Queens 1}}}
{Show {SearchAll {Queens 2}}}
{Show {SearchAll {Queens 3}}}
{Show {SearchAll {Queens 4}}}
{Show {SearchAll {Queens 5}}}
{Show {SearchAll {Queens 6}}}
{Show {Length {SearchAll {Queens 7}}}}
{Show {Length {SearchAll {Queens 8}}}}
{Show {Length {SearchAll {Queens 9}}}}
{Show {Length {SearchAll {Queens 10}}}}
```

En ambas versiones obtuvimos los siguientes resultados:

```
Mozart Engine 1.4.0 (20110605105756) playing Oz 3

[queens(1)]
nil
nil
[queens(2 4 1 3) queens(3 1 4 2)]
[queens(1 3 5 2 4) queens(1 4 2 5 3) queens(2 4 1 3 5) queens(2 5 3 1 4) queens(3 1 4 2 5) ...
    queens(3 5 2 4 1) queens(4 1 3 5 2) queens(4 2 5 3 1) queens(5 2 4 1 3) queens(5 3 1 4 2)]
[queens(2 4 6 1 3 5) queens(3 6 2 5 1 4) queens(4 1 5 2 6 3) queens(5 3 1 6 4 2)]
40
92
352
724
```

```
Mozart Engine 2.0.0—alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 —0500) playing Oz 3

[queens(1)]
nil
nil
[queens(2 4 1 3) queens(3 1 4 2)]
[queens(1 3 5 2 4) queens(1 4 2 5 3) queens(2 4 1 3 5) queens(2 5 3 1 4) queens(3 1 4 2 5) ...
    queens(3 5 2 4 1) queens(4 1 3 5 2) queens(4 2 5 3 1) queens(5 2 4 1 3) queens(5 3 1 4 2)]
[queens(2 4 6 1 3 5) queens(3 6 2 5 1 4) queens(4 1 5 2 6 3) queens(5 3 1 6 4 2)]
40
92
352
724
```

Se observa que las soluciones obtenidas aparte de que son las mismas, también se encuentran en el mismo orden lo que nos indica que `SearchAll` trabaja igual en ambas versiones.

Search.all

Para evaluar el motor de búsqueda *All* de Mozart, mostrará las soluciones cuando: $N = 2$ y $N = 6$; también se mostrará el tamaño de la solución para $N = 10$; y se comprobará que los resultados obtenidos sean iguales en ambas versiones.

```
1  {Show {Search.all {Queens 2} 1 _}}
2  {Show {Search.all {Queens 6} 1 _}}
3  {Show {Length {Search.all {Queens 10} 1 _}}}
```

En ambas versiones obtuvimos los siguientes resultados:

```
Mozart Engine 1.4.0 (20110605105756) playing Oz 3

nil
[queens(2 4 6 1 3 5) queens(3 6 2 5 1 4) queens(4 1 5 2 6 3) queens(5 3 1 6 4 2)]
724
```

```
Mozart Engine 2.0.0—alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 —0500) playing Oz 3

nil
[queens(2 4 6 1 3 5) queens(3 6 2 5 1 4) queens(4 1 5 2 6 3) queens(5 3 1 6 4 2)]
724
```

Observamos que el motor de búsqueda `Search.all` trabaja de igual manera en ambas versiones.

De las pruebas anteriores podemos concluir que el funcionamiento de los motores de búsqueda de Mozart funciona correctamente en *Mozart2* y se puede brindar soporte por lo menos en la búsqueda a CSP escritos en *Mozart1.4*

Search.gecode.dfsAll

A continuación evaluaremos el motor de búsqueda de Gecode *DFS* en su versión para obtener todas las soluciones. Este motor de búsqueda se evaluará de la misma forma que el motor de búsqueda `Serch.all`.

```
1  {Show {Search.gecode.dfsAll {Queens 2}}}
2  {Show {Search.gecode.dfsAll {Queens 6}}}
3  {Show {Length {Search.gecode.dfsAll {Queens 10}}}}
```

```
Mozart Engine 2.0.0—alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 —0500) playing Oz 3

nil
[queens(2 4 6 1 3 5) queens(3 6 2 5 1 4) queens(4 1 5 2 6 3) queens(5 3 1 6 4 2)]
724
```

Se observar que se obtuvieron las respuestas correctas para cada una de las versiones del problema *N Queens*.

6.3. Pruebas Motores de Búsqueda para soluciones óptimas

Por último evaluaremos los motores de búsqueda de Mozart para encontrar soluciones óptimas. Estas pruebas consistirán en ejecutar los motores de búsqueda *BAB* y *Restart* en *Mozart2* y *Mozart1.4* y verificar que las soluciones sean iguales en ambas versiones.

Para realizar estas pruebas se ha seleccionado el problema *Send Most Money*, el cual es una variante del problema *Send More Money* en el que se busca maximizar el valor de $MONEY$. La solución a este problema es: `sol(d:2 e:7 m:1 n:8 o:0 s:9 t:4 y:6)(MONEY = 10876)`.

Ejemplo 6.4 (Send Most Money)**.** Este problema consiste en encontrar dígitos distintos para las letras D, E, M, N, O, T, S, Y; tal que S y M sean diferentes de 0, que la siguiente ecuación se satisfaga:

$$SEND + MOST = MONEY$$

Ademas se búsca que el valor de $MONEY$ sea máximo.

A continuación se muestra la implementación del problema *Send Most Money* en *Mozart2* y *Mozart1.4*.

```
proc {Most Root}
  S E N D M O T Y
in
  Root = sol(s:S e:E n:N d:D m:M o:O t:T y:Y)
  {FD.dom 0#9 Root}
  {FD.distinct post(Root)}
  {FD.rel post(S '\\=:' 0)}
  {FD.rel post(M '\\=:' 0)}
  {FD.linear post([1000  100  10  1
                   1000  100  10  1
            ~10000 ~1000 ~100 ~10 ~1]
                  [S E N D
                   M O S T
                 M O N E Y] '=:' 0)}
  {FD.branch Root size_min val_min}
end

proc {Obj O N}
  {FD.linear post([10000  1000  100  10  1
                  ~10000 ~1000 ~100 ~10 ~1]
                  [N.m N.o N.n N.e N.y
                   O.m O.o O.n O.e O.y] '>:' 0)}
end
```

Código 6.6: Implementación *Send Most Money* en *Mozart2*

```
proc {Most Root}
  S E N D M O T Y
in
  Root = sol(s:S e:E n:N d:D m:M o:O t:T y:Y)
  Root ::: 0#9
  {FD.distinct Root}
  S \=: 0
  M \=: 0
              1000*S + 100*E + 10*N + D
           + 1000*M + 100*O + 10*S + T
  =: 10000*M + 1000*O + 100*N + 10*E + Y
  {FD.distribute naive Root}
end

proc {Obj O N}
```

```
16        10000*N.m + 1000*N.o + 100*N.n + 10*N.e + N.y
17      >: 10000*O.m + 1000*O.o + 100*O.n + 10*O.e + O.y
18    end
```

Código 6.7: Implementación *Send Most Money* en *Mozart1.4*

SearchBest

Se evalúa el acceso directo para le ejecución del motor de búsqueda para soluciones óptimas.

```
1  {Show {SearchBest Most Obj}}
```

En ambas versiones se obtuvieron los siguientes resultados:

```
Mozart Engine 1.4.0 (20110605105756) playing Oz 3

[sol(d:2 e:7 m:1 n:8 o:0 s:9 t:4 y:6)]
```

```
Mozart Engine 2.0.0-alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 -0500) playing Oz 3

[sol(d:2 e:7 m:1 n:8 o:0 s:9 t:4 y:6)]
```

Se observa que en ambas versiones se obtiene la solucion correcta y con el mismo formato (una lista de soluciones) al usar `SearchBest`.

Search.best.bab

Se evalúa el motor de búsqueda *BAB* de Mozart.

```
1  {Show {Search.best.bab Most Obj 1 _}}
```

En ambas versiones se obtuvieron los siguientes resultados:

```
Mozart Engine 1.4.0 (20110605105756) playing Oz 3

[sol(d:2 e:7 m:1 n:8 o:0 s:9 t:4 y:6)]
```

```
Mozart Engine 2.0.0-alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 -0500) playing Oz 3

[sol(d:2 e:7 m:1 n:8 o:0 s:9 t:4 y:6)]
```

Se observa que en ambas versiones se obtiene la solucion correcta y con el mismo formato (una lista de soluciones) al usar `Search.best.bab`.

Search.best.restart

Se evalúa el motor de búsqueda *BAB* de Mozart.

```
1 {Show {Search.best.restart Most Obj 1 _}}
```

En ambas versiones se obtuvieron los siguientes resultados:

```
Mozart Engine 1.4.0 (20110605105756) playing Oz 3

[sol(d:2 e:7 m:1 n:8 o:0 s:9 t:4 y:6)]
```

```
Mozart Engine 2.0.0—alpha.0+build.5854.ea52996 (Thu, 26 Sep 2013 14:17:15 —0500) playing Oz 3

[sol(d:2 e:7 m:1 n:8 o:0 s:9 t:4 y:6)]
```

Se observa que en ambas versiones se obtiene la solucion correcta y con el mismo formato (una lista de soluciones) al usar `Search.best.restart`.

CAPÍTULO 7

Conclusiones y Trabajo Futuro

7.1. Conclusiones

- El contar ya con motores de búsqueda en la nueva versión de la máquina virtual de Mozart, permite tener un primer *release* de Mozart que incluya el subsistema de restricciones, ya que a pesar de que aun faltan cosas por desarrollar en este subsistema como: soporte para variables dominio de conjuntos, recomputación, recolección de basura, entre otros; se brinda soporte a la parte más importante de la programación por restricciones, la cual es, permitir resolver un CSP.

- Para brindar un soporte completo de la búsqueda en la nueva máquina virtual de Mozart, es necesario un trabajo más a fondo en algunos aspectos necesarios para la búsqueda en la nueva máquina virtual de Mozart; como lo son: la estabilidad de los espacios, la comunicación entre el espacio de Mozart y el espacio de Gecode.

- Un aspecto importante a tener en cuenta como resultado de este trabajo de grado, es el poder soportar los motores de búsqueda de la versión anterior, lo que permite que los programas hechos por los usuarios en versiones anteriores en Mozart tengan soporte en la nueva máquina virtual de Mozart.

- El desarrollo de este trabajo de grado ha permitido identificar problemas importantes que presenta la implementación de la nueva máquina virtual de Mozart, los cuales afectan otros aspectos de Mozart.

- Los resultados de este trabajo hicieron parte de la publicación de un artículo en *TRICS13*[1] workshop de *The 19th International Conference on Principles and Practice of Constraint Programming.*[9]

7.2. Trabajo Futuro

- Investigar más a fondo la estabilidad en la nueva máquina virtual de Mozart para poder plantear una solución que corrija el problema identificado en el desarrollo de este trabajo de grado.

- Diseñar una solución que permita obtener desde un espacio de Gecode el espacio de Mozart a que corresponde, afectando lo menos posible el diseño actual de la nueva máquina virtual de Mozart. Lo que permitirá contar con los motores de búsqueda de Gecode para soluciones óptimas.

- Diseñar una solución que permita poder realizar la operación *merge* de forma correcta, es decir, poder mezclar toda la información encapsulada en el espacio de Gecode.

- Integrar el motor de búsqueda gráfico de Mozart (*Explorer*) el cual permite hacer un seguimiento paso a paso de cómo se realiza la exploración del *árbol de soluciones.*

- Integrar el motor de búsqueda gráfico de Gecode (*GIST*) el cual permite hacer un seguimiento paso a paso de cómo se realiza la exploración del *árbol de soluciones.*

[1] http://cp2013.a4cp.org/workshops/trics

Referencias

[1] Rossi F., Van Beek P. y Walsh T. Handbook of Constraint Programming. Italia: Elsevier; 2006.

[2] Apt K. Principles of constraint programming. UK: Cambridge University Press; 2003.

[3] Schulte C., Tack G. y Lagerkvist M. Z. Modeling and Programming whit Gecode; 2012.

[4] Mehl M., Scheidhauer R. y Schulte C. An Abstract Machine for Oz. PLILP 1995. p 151–168

[5] Diehl S., Hartel P. y Sestoft P. Abstract machines for programming language implementation. Future Generation Computer Systems 2000. p 739-751

[6] Barták R. Constraint Programming: In Pursuit of the Holy Grail. Proceedings of WDS99 (invited lecture) 1999.

[7] Saraswat V. y Van Hentenryck P. Constraint Programming: Strategic Directions. Constraints: An International Journal 1997; Volumen: 2. p 7–33

[8] Barber F. y Salido M. Introducción a la Programación de Restricciones. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial 2003; Volumen: 20. p 13-30.

[9] Gutiérrez G., Barco A. F., Villanueva M. A., Cardona A., Montenegro D., Miller S., Díaz J. F., Rueda C. and Van Roy P. The Mozart Constraint Subsystem. TRICS 2013.

[10] Gómez Farhat G. Caracterización del desempeño de GeOz e implementación de estrategias de mejoramiento [Trabajo de Grado]. Santiago de Cali: Universidad del Valle. Escuela de Ingeniería de Sistemas y Computación; 2010. 76p.

[11] Barco Santa A. Intercalando Ejecución De Hilos De Mozart-Oz Y Propagadores De Gecode [Trabajo de Grado]. Santiago de Cali: Universidad del Valle. Escuela de Ingeniería de Sistemas y Computación; 2009. 50p.

[12] Schulte C. Programming Constraint Services [Tesis de Doctorado]. Saarbrücken: Universität des Saarlandes. Naturwissenschaftlich-Technische Fakultät I; 2000. 194p.

[13] Reischuck R. Reconciling Copying and Trailing for Constraint Programming Systems [Bachelor's Thesis]. Saarbrücken: Universität des Saarlandes. Department of Computer Science; 2008. 70p.

[14] Mehl M. The Oz Virtual Machine - Records, Transients, and Deep Guards [Tesis de Doctorado]. Saarbrücken: Universität des Saarlandes. Technische Fakultät; 1999. 209p.

[15] Villanueva M. A. Reporte Motores de Búsqueda de Gecode. 2013.

[16] GECODE - An open, free, efficient constraint solving toolkit. URL: `http://www.gecode.org/`. [Consultado: 2012-06-11].

[17] The GeOz Projec. URL: `http://cic.javerianacali.edu.co/wiki/doku.php?id=grupos:avispa:geoz`. [Consultado: 2012-06-11].