ACTUALIZACIÓN DE CPGRAPH,UN FRAMEWORK PARA RESOLVER CSPs (PROBLEMAS DE SATISFACCIÓN DE RESTRICCIONES),QUE INVOLUCRAN GRAFOS,DE GECODE 1.0.0 A GECODE 2.1.1

Autores:

José Luis Pulido Burbano[1]

David Alejandro Przybilla[2]

Director:

Orlando Arboleda
Ingeniero de Sistemas

UNIVERSIDAD DEL VALLE

ESCUELA DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

PROGRAMA ACADÉMICO DE INGENIERÍA DE SISTEMAS

TULUA

[1]josel28_20@gmail.com
[2]paranoic.pum@gmail.com

"*Whatever you can do, or dream you can, begin it.*" - Johann Wolfgang von Goethe.

Queremos dar las gracias entre muchas personas a las siguientes:

- A nuestros padres por permitirnos la oportunidad de estudiar esta maravillosa carrera.
- A nuestro director y maestro el Ingeniero Orlando Arboleda, quien desinteresadamente nos guió durante el desarrollo del proyecto.
- A Gustavo Gutierrez quien nos colaboro, resolvió dudas, compartió parte de su trabajo e ideas con nosotros.
- A nuestros profesores de Univalle Tulua y Cali como Simena Dinas ,Victor Manuel Vargas y Juan Francisco Díaz quienes nos ayudaron a mejorar nuestro trabajo colaborandonos con sus consejos sobre distintos aspectos como investigación y presentación del informe de Trabajo de Grado.

Índice general

Capítulo 1

Introducción

El objetivo de éste capítulo es brindar una descripción del problema que aborda este trabajo de grado, justificar su solución y establecer los objetivos generales y específicos que se cumplirán para solucionar el problema.

1.1. Descripción del Problema

CP(Graph) es un librería de clases que proporciona una estructura de base para desarrollar aplicaciones basadas en restricciones(un framework, en inglés), resultado de la implementación de las conclusiones y estudios realizados por Gregoire Dooms en su tesis de doctorado[1], la implementación del mismo fue hecha tanto sobre la librería de restricciones Gecode 1.3.1 como Gecode 1.0.0, Gecode esta escrita en el lenguaje C++. Una de las tantas aplicaciones de CP(Graph) se da en problemas relacionados con Bioinformática[1].

CP(Graph) facilita el planteamiento y solución de problemas de satisfacción de restricciones que involucran grafos, pues permite definir variables del dominio de grafos y aplicar restricciones a estas variables desde un alto nivel, sin tener que definirlas de la manera tradicional, es decir, desde el dominio de los enteros[1][8].

La librería Gecode esta en constante desarrollo y ha sufrido cambios en su núcleo que hacen imposible usar CP(Graph) con la ultima versión de Gecode (2.1.1), los cambios introducidos a Gecode en sus ultimas versiones sin embargo mejoran el rendimiento(perfomance, en inglés), mejoran el uso de la memoria y ofrecen nuevas alternativas para la definición de problemas de satisfacción de restricciones. El registro de cambios(changelog, en inglés) que se lleva en la pagina de Gecode indica que los propagadores durante el cambio de versión 1.3.1 a 2.0.0 mejoraron el uso de memoria de un 20 % a un 40 % además los propagadores del dominio de los Booleanos son el doble de rápidos con respecto a la versión 1.3.1 y el uso de memoria se redujo a la mitad[11].

El equipo de Gecode maneja las contribuciones de manera externa, éste no asegura el soporte de las contribuciones para futuras versiones, entonces es responsabilidad de los autores de las mismas realizar los cambios pertinentes, sin embargo Gregoire Dooms (el autor de CP(Graph)) no ha realizado, ni realiza ninguna actualización a CP(Graph)[12].
La pregunta de investigación que se busca responder es entonces:
¿Es posible realizar una actualización de CP(Graph) de la versión 1.0.0 a la versión 2.1.1 de la librería Gecode, versión de la librería probada extensivamente, con mejoras en su arquitectura como los Advisores, además con un kernel más rápido ?

1.2. Justificación

CP(Graph) es una contribución a la librería Gecode que permite reducir el tiempo en análisis y la cantidad de memoria usada en la ejecución de un problema que involucre grafos y restricciones[8, 1].

El tiempo de modelamiento de un problema realizado en el dominio de los enteros, se reduce de manera significativa en CP(Graph), pues éste introduce en forma nativa variables que pueden tomar diferentes tipos de restricciones asociadas a grafos.

CP(Graph) además permite agregar extensiones para crear nuevos propagadores y nuevas restricciones y/o mejorar los propagadores.

Entre las diferentes aplicaciones de CP(Graph) tenemos:

- Problemas de Bioinformática donde se requiere estudiar las redes metabólicas, las cuales se llevan a una representación en grafos y sobre estos aplican ciertas restricciones[1].
- Grafos lingüísticos, los cuales son usados para representar lenguajes y por tanto administrar la organización de los datos en diferentes clases de campos científicos[9, 1].
- Comparaciones de patrones entre grafos diferentes y encontrar caminos de Hamilton usando menor cantidad de memoria.[1]

Es importante para la resolución de problemas actualizar CP(Graph), esto con el objetivo de brindar una opción de alto nivel que permita realizar el modelamiento de problemas como grafos, haciendo uso de una librería que involucre las ultimas técnicas de restricciones en cuanto a propagadores y motores de búsqueda como lo garantiza Gecode en su versión 2.1.1.

1.3. Objetivos

1.3.1. Objetivo General

Desarrollar cambios de implementación a CP(Graph) que le permitan ser funcional en la versión 2.1.1 de la librería Gecode.

1.3.2. Objetivos Específicos

- Listar las diferencias entre Gecode 1.0.0 y Gecode 2.1.1 que afectan el núcleo de CP(Graph).
- Analizar los propagadores propuestos por Gregoire en CP(Graph).
- Diagramar los módulos, clases e interfaces propuestas en CP(Graph).
- Registrar los cambios a realizar a CP(Graph) bajo gecode 2.1.1 .
- Implementar los cambios a CP(Graph) para que funcione bajo Gecode 2.1.1.
- Analizar y comprender los problemas solucionados por Gregoire Dooms con CP(Graph), con el fin de validar su funcionamiento en la nueva versión.
- Evaluar resultados obtenidos a problemas resueltos con CP(Graph) en gecode 1.0.0 con respecto a los obtenidos con CP(Graph) en gecode 2.1.1.

Capítulo 2

Marco de Referencia

El objetivo de éste capítulo es brindar el conocimiento necesario para entender el área en general donde se ubica este proyecto que es Programación por Restricciones y Matemáticas Discretas (particularmente teoría de grafos), además se presenta parte del marco de conocimiento que Gregoire Dooms contribuye en su documento de tesis para que el lector pueda entender la representación teórica de los componentes más importantes de CP(Graph).

2.1. Programación por Restricciones

2.1.1. Concepto de Restricción

Una restricción es una relación lógica entre variables, cada una de las cuales toma un valor dado. por ejemplo las pareja de variables X, Y pueden estar relacionadas por una restricción X <Y, las variables X, Y por si solas no están relacionas pero al introducir la restricción se introduce información que deben cumplir las variables, algo lógico es que una variable pueda tomar un conjunto determinado de valores a esto se le denomina el dominio de la variable[7].

2.1.2. Utilidad de Programación por Restricciones

La Programación por Restricciones (CP o Constraint Programming, en inglés) trata del estudio de sistemas computacionales basados sobre restricciones. La idea de la programación por restricciones es resolver problemas declarando restricciones (condiciones o propiedades) las cuales deben ser satisfechas por la solución[7].

Existen una gran cantidad de aplicaciones de Programación por Restricciones dado la cantidad de problemas que involucran restricciones, para más información[7].

2.1.3. Problemas de Satisfacción de Restricciones (CSPs)

En Programación por Restricciones el modelamiento de problema se hace convirtiendo la descripción del problema en un CSP(Problema De Satisfacción de Restricciones o Constraint Satisfaction Problem, en inglés), es decir identificando:

- Las variables del problema.
- El dominio de cada una de las variables del problema.
- Las condiciones que deben cumplir los valores solución para las variables que involucra el problema.

De manera formal un CSP P se define como:

$P =< X_s, D_s, C_s >$ *donde* $X_s = \{X_1 ... X_n\}$ es el conjunto de variables.

D_s es el conjunto de dominios $D_s =< D_1, D_2...D_n >$ de tal forma que $X_i \epsilon D_i$Siendo U el conjunto universal de todos los posibles valores. Cada variable $X \epsilon X_s$tiene un posible conjunto de valores $D(X) \subseteq U$que puede ser asignado a la variable y el cual es llamado el dominio de la variable.

$C_s =< C_1, C_2....C_t >$ es el conjunto de restricciones. Una restricción C_j es una tupla (R_{s_j}, S_j) donde R_{s_j} es una relación sobre las variables S_j, donde $S_i = Scope(C_i)$ (el conjunto de las variables afectadas por la restricción). Es decir R_i es un subconjunto del producto cartesiano de las variables en S_i.

Una solución al CSP P es una tupla $A =< a_1, a_2...a_n >$ donde $a_i \epsilon D_i$ y cada C_j es satisfecho de tal forma que$< a_1, a_2...a_n > \epsilon C_s$ [3, 1, 2].

2.1.3.1. Propiedades Importantes de Los CSPs

- **Equivalencia:**
  Considere una secuencia de Variables $X := X_1, ..., X_n$ con una secuencia de dominios $D_1, ...D_n$.Tome un elemento $d := (d_1, ..., d_n)$ de $D_1 x....x D_n$y una subsecuencia $Y = X_{i_1}...X_{i_l}$de X. Entonces se denota la secuencia $(d_{i_1}, ..., d_{i_l})$como $d[Y]$ y se lee como la proyección de d sobre Y[2].
  Con la proyección se define el primer concepto de equivalencia:

  Sean dos CSPs nombrados respectivamente P_1 y P_2se consideran CSPs equivalentes si:

  $\{d[X] \,|\, d \; es \; una \; solucion \; a \; P_1\} = \{d[X] \,|\, d \; es \; una \; solucion \; a \; P_2\}$

  El segundo concepto de equivalencia:
  Considere los CSPs $P_0..P_m$ donde $m \geq 1$ y una secuencia X de sus variables comunes. Se dice que la unión de $P_1..P_m$ es equivalente a P_0 con respecto a X Si:

  $\{d[X] \mid d \; es \; una \; solucion \; a \; P_0\} = \bigcup_{i=1}^{m} \{d[X] \mid d \; es \; una \; solucion \; a \; P_i\}$.

  Es decir un CSPs puede verse como la unión de muchos otros CSPs.

- **Consistencia:** Un CSP es consistente o no fallido,
  Si $Soluciones(X, D, C) \neq \emptyset$, es decir si existe al menos una solución[2].

- Existen varias representaciones en CSP de un mismo problema[2].

- Un CSP esta resuelto, si todas las restricciones han sido resultas y para cada $X_i$que pertenece a X, $Dominio(X_i) \neq \phi$[2].

2.1.4. Resolviendo CSPs

Una vez el CSP esta planteado se procede a encontrar su solución, para ello primeramente tiene que definirse si el CSP es consistente (Si existe una solución), en caso de que el CSP cumpla con la propiedad de Consistencia el paso a seguir seria identificar a que tipo de solución se pretende llegar:

- Encontrar una solución.
- Encontrar todas las soluciones.
- Encontrar la solución óptima.
- Encontrar todas las soluciones óptimas.

Una vez definida la solución que se espera, se escoge el método para llegar a la solución.Existen métodos específicos y generales. Los específicos son llamados Solucionadores de Restricciones (constraint solvers, en inglés)y se aplican en problemas con dominios específicos y características específicas, por ejemplo

existen Solucionadores de Restricciones para resolver problemas que involucran ecuaciones lineales, problemas de variables del dominio de los reales, grafos, etcétera[1, 2].

Una vez planteados el CSP y la solución a la que se pretende llegar, empieza la resolución del CSP.

1. En primera instancia un CSP inicial generara otros CSPs equivalentes al CSP inicial, al ser transformado por los propagadores y reglas de transformación. Los propagadores en la primera etapa podaran los dominios de las variables según las reglas que le hayan sido especificadas.
2. Una vez los propagadores han podado los dominios de las variables del CSP, si no existe aun una solución, el motor de búsqueda y la estrategia de distribución se encargaran de explorar los dominios de una de las variables, la estrategia de distribución (branching, en inglés) define en cual orden se exploran las variables (e.g la variable con más restricciones, la variable con el dominio mas pequeño, etcétera) y en que orden serán asignados los valores del dominio a esa variable (e.g el valor mas grande del dominio, el valor mas pequeño del dominio, etcétera).Esta exploración del árbol y de los dominios de las variables genera nuevos CSPs equivalentes.
3. Los propagadores podan los dominios de los nuevos CSPs equivalentes.
4. Se itera entre el numeral 2 y numeral 3 explorando el árbol, los dominios de las variables y propagando. Hasta que se encuentre una solución o el dominio de una de las variables del CSP sea vació, en caso tal no se ha encontrado una solución.

2.1.5. Propagadores (Filtering algorithm, en inglés)

Es un algoritmo que durante la búsqueda de las soluciones a un CSP, se encarga de reducir los dominios de las variables, sin descartar soluciones[1].
Un propagador siempre debe reducir el dominio, por tanto debe cumplir con las siguientes propiedades:

Sea f un propagador asociado a la restricción C. f debe satisfacer que:

1. **Contractante:** $f(D1) \subseteq D1$
2. **Correcto:** $D1 \cap C \subseteq f(D_1)$

Un propagador convierte un CSP P en un P' de tal forma que sean CSPs equivalentes.

2.1.6. Modelo de Ejecución.

Durante la búsqueda de las soluciones de un CSP se tiene que el espacio de búsqueda son todos los posibles y potenciales valores que pueden tomar las variables que involucra el CSP, sin embargo explorar cada una de las posibles combinaciones cuando se tiene un numero considerable de variables es costoso en complejidad.

El algoritmo 1 muestra el flujo desde un alto nivel para la solución de un CSP.

- El preprocesamiento hace referencia a que las restricciones y las variables son transformadas a la forma sintáctica en que los propagadores y los algoritmos pueden procesarlas.
- "*Happy*" significa que se ha llegado al objetivo esperado, en este caso se recuerda que el objetivo puede ser: encontrar todas las soluciones posibles, solo una solución o las soluciones óptimas entre otras.
- Atómico hace referencia a que los dominios de todas las variables son unitarios.
- La propagación de restricción (Constraint propagation, en inglés) hace referencia a generar CSPs equivalentes usando propagadores que recorten de manera significativa los dominios de las variables.

Algorithm 1 Procedimiento Solve(Representa el Ciclo Básico de Programación por Restricciones) [2].

```
Solve:
Var continue: Boolean;
continue ←true;
while continue And Not Happy do
    Preprocess;
    Constraint Propagation;
    If Not Happy
    then
        if Atomic
        then
            continue ← false;
        else
            Split;
            Proceed By Cases;
        end
    end
end
```

- La División (Slip, en inglés) hace referencia a partir el CSP P en subproblemas $P_1, P_2 ... P_n$ de tal forma de que $\bigcup_1^n P_i = P$, estos subproblemas son nuevos CSPs que partieron el dominio del CSP P inicial. Es aquí cuando se explora el espacio de búsqueda y los potenciales valores que pueden tomar las variables.
  El Split es de vital importancia en la solución del CSP, pues se pueden definir diferentes maneras de explorar el dominio de las variables, la estrategia adoptada para la partición del dominio o la manera en la que se explora el árbol de búsqueda es la Heurística de Búsqueda. Es elección del usuario escoger la Heurística que mejor se ajuste al problema que desea atacar.

  La División se representa mediante reglas de la forma:

- $$\frac{Guard}{Rule}$$

- Donde la guarda (guard, en inglés) representa una expresión booleana que debe cumplirse para que la regla sea aplicada.Algunas reglas usadas durante el split son:

  Etiquetamiento (labeling, en inglés)

  $$\frac{X \epsilon \{a_1, ... a_k\}}{X \epsilon \{a_1\} \mid ... \mid X \epsilon \{a_k\}}$$

  Bisección (bisection, en inglés)

  $$\frac{X \epsilon [a, b]}{X \epsilon \left[a, \frac{a+b}{2}\right] \mid X \epsilon \left[\frac{a+b}{2}, b\right]}$$

- La notación | indica que varios CSPs son generados.

2.2. Teoría de Grafos

Debido a que la base de este proyecto es restricciones sobre grafos y las propiedades de los mismos, es necesario por tanto entender los siguientes conceptos**:**

- **Grafo Dirigido:** *Sea* G un grafo dirigido, entonces $G = (V, E)$ donde V es el conjunto de los vértices y E el conjunto de aristas. Para una arista (u, v), u es llamado la cola (source o tail, en inglés) y v es la cabeza (target o head, en inglés)[1].
  - Vecinos de salida de vertice u: los vecinos de salida de un vértice u en un grafo G=(V, E) es el conjunto de vértices en los cuales una arista proviene de u:$\{v|\,(u, v) \in E\}$[1].
  - Vecinos de entrada de vertice v: los vecinos de entrada de un vértice v en el grafo G es el conjunto de vértices desde los cuales hay una arista incidente en v: $\{u|\,(u, v) \in E\}$[1].
  - Grado de salida de vertice u: el grado de salida de un vertice u, es el numero de aristas de salida del vertice u[1].
  - Grado de entrada de vertice u: el grado de entrada de un vertice u, es el numero de aristas que inciden en el vertice u[1].
  - Simetría en grafos dirigidos: un grafo dirigido simétrico es un grafo en el cual $(u, v)\in$ E $\iff$ $(v, u) \in$ E[1].
  - Circuito: Es una camino simple que empieza y termina en el mismo vértice[1].
  - Grafo acíclico dirigido: un grafo dirigido sin circuitos es llamado un grafo acíclico dirigido (direct aciclic graph, en el inglés)[1].
- **Grafo No dirigido:** *Sea* G un grafo no dirigido, entonces $G = (V, E)$, donde V es el conjunto de vértices de G y E es el conjunto de aristas de G .
  En un grafo no dirigido sus aristas son conjuntos de dos nodos, es decir:
  $E \subseteq \{\{u, v\} \mid u \in V \wedge, v \in V \ \wedge \ u \neq v\}$.
  - Vecinos de vertice u: los vecinos de un vertice u es el conjunto de todos los vértices que forman una
    arista con u: $\{v|\,(u, v)\} \in E$.Tales vértices se dice son adyacentes a u[1].
  - Grafo conectado: una grafo no dirigido es conectado si cada par de vértices tiene un camino que los une[1].
  - Ciclo en un grafo: es una camino simple que empieza y termina en el mismo vértice[1].
- Subgrafo inducido de un grafo: G1 es un grafo inducido de G2, si todas las aristas de G2 inciden en vértices de G1 [1] ejemplo:

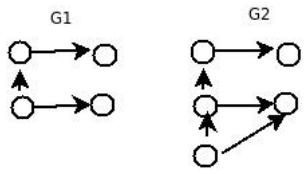

Figura 2.1: Subgrafo Inducido

- Grafo bipartito: un grafo simple es bipartito si su conjunto V de vértices se puede dividir en dos conjuntos disyuntos (conjuntos sin nodos en común) V1 y V2 tales que cada arista del grafo conecta un vertice de V1 con un vertice de V2 [5]ejemplo:

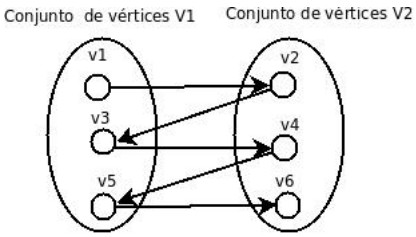

Figura 2.2: Grafo Bipartito

- Grafo completo: el grafo completo de orden n es un grafo con n nodos donde cada nodo esta conectado con los demás nodos.
- Subgrafo: un subgrafo de un grafo $G = (V, E)$es un grafo $H = (W, F)$ donde $W \subseteq V$ y $F \subseteq E$ [5].
- Complemento de un grafo: el grafo complemento de un grafo $G = (V, E)$ es un grafo $\bar{\mathrm{G}} = (\mathrm{V}, \bar{\mathrm{E}})$ el cual contiene todas las aristas las cuales no están presentes en E: $\bar{\mathrm{E}} = (\mathrm{V} \times \mathrm{V}) \setminus \mathrm{E}$ [1].
- Orden de un grafo: el numero de vértices que componen un grafo.
- Tamaño de un grafo: el numero de aristas que componen un grafo.
- Camino simple (path, en inglés): una secuencia de vértices y aristas con un vertice de inicio y fin en el cual no hay vértices repetidos [1].
- Camino (walk, en inglés): una secuencia de vértices y aristas con un vertice de inicio y fin [1].
- Circuito: es una camino simple que empieza y termina en el mismo vértice [1].
- Grafo Kneser:

  $\mathrm{KG}_{n,k}$ es el grafo cuyos vértices corresponden a los subconjuntos de k-elementos de un conjunto de n elementos, y donde dos vértices están conectados si y solo si los dos conjuntos correspondientes son disyuntos [13].

  Ejemplos:

  1. El grafo completo de n vértices es el grafo Kneser $\mathrm{KG}_{n,1}$.
  2. Grafo Kneser $\mathrm{KG}_{5,2}$.

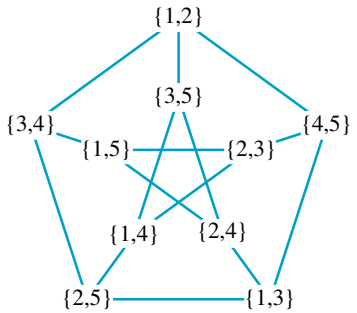

Figura 2.3: Grafo de Kneser[13]

- Clausura transitiva de un grafo dirigido G: la clausura transitiva (transitive closure, en inglés) de un grafo dirigido $G(V, E)$ es un grafo $G^+(V, E^+)$ tal que $(u, v) \in E^+$ si y solo si hay un camino de u a v en G [1], ejemplo:

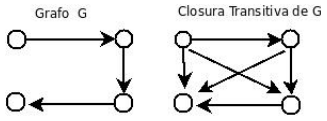

Figura 2.4: Clausura Transitiva

2.3. Dominio de Computación de CP(Graph)

2.3.1. Introducción[15]

Esta sección enumera las restricciones disponibles en CP(Graph), la representaciones de variables de grafos que se implementaron y otras generalidades de CP(Graph).
Una variable del dominio de grafos toma valores de Grafos, es decir el dominio de la variable es un conjunto de grafos. Si se intentara enumerar todos los posibles grafos que existen en el dominio de una variable de grafos el gasto en memoria seria excesivo, para evitar lo anterior se usan abstracciones.

2.3.2. Intervalo de Grafo

El primer cuestionamiento al plantearse un framework que permita definir restricciones sobre los grafos, es la representación que van a tener los grafos. Existen diferentes enfoques al plantearse la pregunta "*¿como representar una variable del dominio de los grafos?*". Para la definición de las restricciones a nivel teórico, el autor de CP(Graph) planteo una sola representación genérica, mientras que a nivel de implementacion existen dos representaciones diferentes.

Sea g un grafo, entonces $g = (sn, sa)$ donde sn es el conjunto de nodos que compone el grafo, sa es el conjunto de aristas que posee el grafo y por tanto se tiene que: $sa \subseteq sn\, x\, sn$.

Dada la definición anterior la representación de variables del dominio de grafos esta dada por un intervalo de grafos, es decir:

- un Least Upper Bound(LUB) (Cota Superior mas Pequeña): Es una cota superior que define el conjunto superior de nodos y el conjunto superior de aristas, es decir aquellos nodos y aristas que ***podrían*** ser incluidos en el grafo.
- Un Greatest Lower Bound (GLB) (Cota Inferior mas Grande): Es una cota inferior que define el conjunto inferior de nodos y el conjunto inferior de aristas, es decir aquellos nodos y aristas que se ***saben tienen*** que ser incluidos en el grafo.

por tanto se tiene que si $G_1 \subseteq G_2$, $[G_1, G_2]$ es un Intervalo de grafos. De tal forma que representan un Intervalo de Grafos.

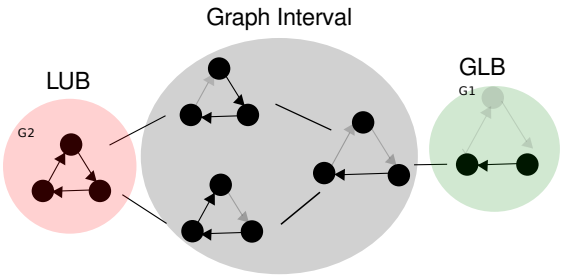

Figura 2.5: Intervalo de Grafo

En la Figura 6 el GLB (circulo verde) define los nodos y aristas obligatorios. El LUB (circulo rojo) define las aristas y nodos que podrian ser incluidos en el grafo. El circulo gris determina el conjunto de grafos que son abstraídos por el GLB y el LUB pero que no son enumerados.

Si G es una variable del dominio de grafos, se denota $dom(G) = [gL, gU]$ con $gL = glb(G)$ y $gU = lub(G)$. Si S es una variable de conjuntos finita, se denota $dom(S) = [sL, sU]$, con $sL = glb(S)$ y $sU = lub(S)$.

2.3.3. Variables en CP(Graph)

Para el modelamiento de problemas de grafos CP(Graph) ofrece dos alternativas distintas para la representación de los grafos del problema. El usar una u otra tiene impacto en la memoria y en el tiempo que toma para la solución del problema.

Las dos alternativas son: OutAdjSetsGraphView y NodeArcSetsGraphView. En la figura a continuación se muestra la diferencia de modelamiento con un ejemplo sencillo. Para mas información revisar el capítulo 6.5.2 de [15].

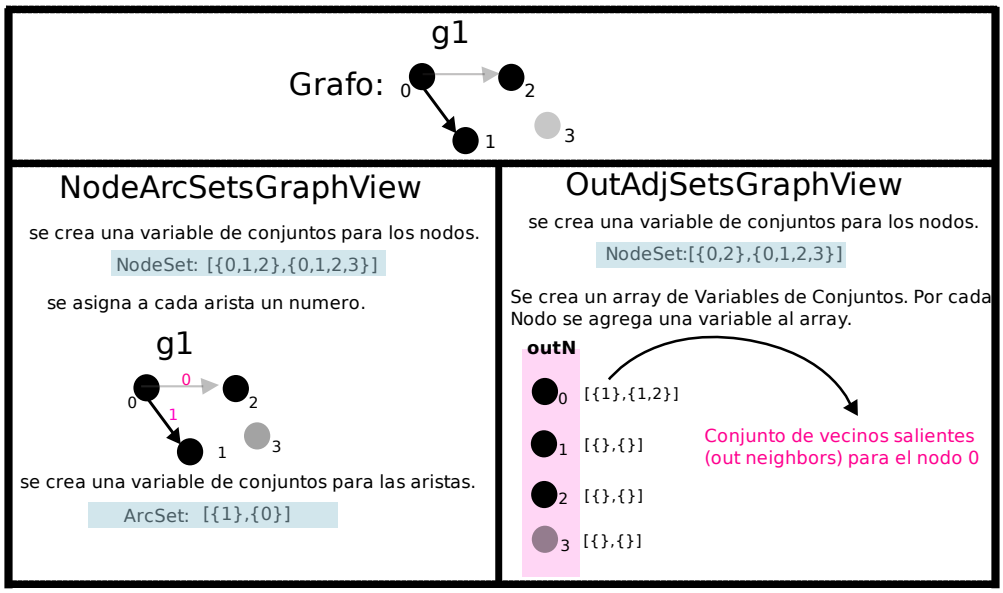

Figura 2.6: Representación de Grafos en CP(Graph)

2.3.4. Variables en CP(Graph)[1]

CP(Graph) usa distintos tipos de variables, entre estos están variables de nodo, variables de arista, variables de conjuntos de aristas y variables de conjuntos de nodos, además se tiene para problemas relacionados con pesos, funciones que asignan a una arista un peso.

La siguiente tabla resume los tipos de variables de CP(Graph) y sus características más importantes:

Tipo de variable	Representación	Restricción	Constantes	Variable
Integer	0, 1, 2...		$i_0, i_1, ...$	$I_0, I_1, ...$
Node	0, 1, 2, ...		$n_0, n_1, ...$	$N_0, N_1, ...$
Arc	(0, 1), (2, 4), ...		$a_0, a_1, ...$	$A_0, A_1, ...$
Set	$\{0, 1, 2\}, \{2, 4\}$		$s_0, s_1, ...$	$S_0, S_1, ...$
Set of nodes	$\{2, 3, 4\}, \{6, 7\}$		$sn_0, sn_1, ...$	$SN_0, SN_1, ...$
Set of arcs	$\{(0, 3), (1, 2)\}$		$sa_0, sa_1, ...$	$SA_0, SA_1, ..$
Graph	(SN, SA)	SA$\subseteq SNxSN$	$g_0, g_1, ...$	$G_0, G_1, ...$
Weight functions	$N \cup A \rightarrow \mathbb{Z}$		$w_0, w_1, ...$	$W_0, W_1, ...$

2.3.5. Restricciones del Núcleo de CP(Graph)

Las restricciones de grafos del núcleo de CP(Graph) definen un conjunto de restricciones las cuales permiten expresar CSPs de grafos y nuevas restricciones como una combinación de estas restricciones junto con restricciones enteras y de conjuntos. Estas restricciones enlazan variables de grafos con nodos, aristas y conjuntos de estas[1].

- *Nodos*(G, SN): esta restricción se cumple si los valores de la variable SN(un conjunto de nodos) es igual al conjunto de nodos de la variable de grafo G.
- *Arcs*(G, SA): esta restricción se cumple si el valor de la variable SA(un conjunto de arcos o aristas) es igual al conjunto de arcos de la variable de grafos G.
- *ArcNode*(A, N1, N2) esta restricción se cumple si el valor de la variable de arco A es igual a el arco (N1, N2) donde N1 y N2 son variables nodo.
- *Adjacency(G)*: esta restricción se cumple si el conjunto de aristas del grafo es igual o un subconjunto de todas las aristas posibles que puede contener el grafo, es decir si Arcs(G) $\subseteq$ Nodes(G) $\times$ Nodes(G).

La restricción *ArcNode* es independiente de las variables de grafos.Todos los grafos en un CSP en CP(Graph) son subgrafos de el grafo universo de el CSP.Por lo tanto el mapeo entre valores de variables nodo y variables arco codificado en la restricción ArcNode es independiente de las variables de grafos. Este necesita ser definido una vez para el grafo universo en el CSP.

2.3.6. Restricciones Globales de CP(Graph)

Las restricciones globales de CP(Graph) se crean a partir de las restricciones del núcleo de CP(Graph), estas son:

1. Restricciones binarias:
   a) *Subgraph*(G1, G2): esta restricción se cumple si G1 es un subgrafo de G2.
   b) *Complement*(G1, G2): esta restricción se cumple si G1 y G2 tienen el mismo conjunto de nodos y cada arista del grafo completo construido sobre aquellos nodos pertenece únicamente a uno de los dos grafos, si una arista esta en G1 esta no puede estar en G2, de igual forma una arista en G2 no puede estar en G1.
   c) *InduceSubgraph*(G1, G2): esta restricción se cumple si G1 es un grafo inducido de G2, es decir, si todas las aristas de G2 inciden en vértices de G1 .
2. Restricciones de Conexión:
   a) *Connected*(G): esta restricción se cumple si G es un grafo no dirigido conectado, es decir, entre cada par de nodos hay un camino.
3. Restricciones para soportar grafos no dirigidos:
   a) *Symmetric*(G): esta restricción se cumple si la relación de adyacencia es simétrica, es decir, que se puede verificar que si la arista (x, y) esta en G, entonces la arista (y, x) también esta en G.
   b) *Undirected*(G, Gu): esta restricción se cumple si Gu es el grafo no dirigido(simétrico) obtenido de ignorar la dirección de las aristas de G.
4. Restricciones de no ciclos:
   a) *DAG*(G): esta restricción se cumple si G es un grafo dirigido acíclico .Esta restricción prohíbe cualquier circuito en G.

b) *Bipartite*(G): esta restricción se cumple, si un grafo es bipartito, es decir, si y solo si su conjunto de nodos puede ser particionado en dos conjuntos N1 y N2 tales que cada arista tiene su punto final en ambos conjunto: E $\subseteq$(N1 $\times$ N2) $\cup$ (N2 $\times$ N1).

c) *Tree*(T): esta restricción se cumple si T es un árbol no dirigido, por ejemplo un grafo conectado acíclico no dirigido.

5. Restricciones de peso:

   a) *Weight*(G, W, I): esta restricción se cumple si I es el peso total asociado a la variable de grafo G de acuerdo a la función de peso W.

6. Restricciones de camino:

   a) *Path*(G, N1, N2): esta restricción se cumple, si G es un camino simple(es decir un camino sin repetición de vértices), de N1 a N2.

   b) *Path*(G, N1, N2, W, I): esta restricción se cumple si G es un camino simple de N1 a N2 cuyo peso total no es mas grande que I.

   c) *Undirected Path*(G, n1, n2): esta restricción se cumple si G es un camino simple no dirigido desde n1 a n2.

   d) *QuasiPath*(G, n1, n2): esta restricción se cumple si cada nodo de G excepto n1, n2 tienen grado 2.

7. Restricción de clausura transitiva(transitive closure, en inglés):

   a) TC(G, G^+) =esta restricción se cumple si G^+ es la clausura transitiva de G.

2.3.7. Expresividad de CP(Graph)

La investigación de un dominio de grafos en Programación por Restricciones, es decir, el desarrollo de CP(Graph), fue iniciada por un problema en análisis de redes bioquímicas, en este capítulo se pretende informar sobre el problema y mostrar como se soluciona en CP(Graph), adicionalmente se explica otro problema de grafos que pueden se resuelto utilizando las restricciones de CP(Graph).

2.3.7.1. Aplicación en Recuperación de Secuencias de Reacciones (Pathways, en inglés) Metabólicas.

El metabolismo es un conjunto de reacciones químicas que ocurren en los organismos vivos con el fin de mantener la vida, las reacciones químicas del metabolismo son organizadas en secuencias de reacciones metabólicas en las cuales un químico es convertido en otro, estas redes están compuestas de cientos de nodos los cuales son de dos tipos: compuestos y reacciones .

Redes Metabólicas

Las Redes Metabolicas están compuestas de entidades bioquímicas (compuestos) que participan en reacciones, tales entidades bioquímicas dependiendo de su papel pueden ser vistas como sustratos o productos[1].

En la Figura 2.8, se tiene una pequeña red metabólica en donde los compuestos están dentro de los círculos y las reacciones dentro de los rectángulos, Pyruvate (compuesto) es un sustrato ya que entra a la reacción 4.2.1.52 y dihydrodipicolinic acid(compuesto) es un producto ya que es producido por la reacción.

Las redes metabólicas pueden ser modeladas como un grafo dirigido bipartito cuyos nodos son las entidades bioquímicas y reacciones y cuyas aristas enlazan entidades y reacciones.

Secuencia De Reacciones[1]

Una secuencia de reacciones (pathway, en inglés) son subconjuntos específicos de una red metabólica los cuales son identificados como procesos funcionales de la célula, éstas pueden ser usadas para estudiar

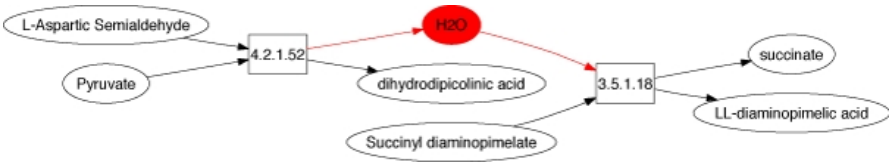

Figura 2.7: Pequeña Red Metabólica [25]

redes metabólicas.Como la mayoría de secuencias de reacciones tiene una estructura linear, un tipo de análisis de red metabólica consiste en encontrar caminos simples en un grafo metabólico.

En CP(Graph), se usa Programación por Restricciones para resolver problemas de búsqueda de caminos simples restringidos.El tipo general de análisis hecho con CP(Graph) consiste en una recuperación de secuencias de reacciones por búsqueda de subgrafo restringidos.

Configuración Previa a la Usada en CP(Graph)[1]

Para entender el modelo planteado por Gregoire Dooms para éste problema es importante explicar de manera somera la configuración en la que el se baso, es decir, que extendió.La primera configuración consistía en buscar el camino simple más corto punto a nodo a nodo en un grafo modelando la red bioquímica; esto puede ser hecho usando búsqueda primero en profundidad.Un problema en esta aproximación es que existen moléculas altamente conectadas como H_2O mostrada en la figura 6, las cuales generan gran cantidad de caminos que no corresponden a secuencias de reacciones reales .Una solución propuesta es remover estos nodos del grafo antes de realizar la búsqueda del camino simple.Un nuevo problema es entonces que algunas secuencias de reacciones, tales como glycolisis, usan algunos de estas moléculas como intermedios.Para reducir la posibilidad de seleccionar estos nodos mientras permiten seleccionarlos, cada nodo tiene asignado un peso proporcional a su grado.Un grupo de metabolitos tiene un grado muy alto así que son menos probables de seleccionar en el camino simple más corto.Este problema requiere un algoritmo de camino simple más corto positivo tal como Dijkstra.

La tasa de recuperación de secuencias de reacciones correctas debería ser incrementada: algunos caminos simples van a través de una reacción y su reacción inversa. Esta pareja de reacciones inversas modelan la reacción desde sustratos para producir y desde productos para sustraer.La mayoría del tiempo, estas reacciones son observadas en una sola dirección en cada especie. Por lo tanto caminos simples que contengan ambas reacciones deberían ser excluidas del conjunto resultado.

Configuración Usada en CP(Graph)[1]

El modelo extiende la configuración anterior y agrega nodos obligatorios en el camino simple.

Sea $n_1, ...n_m$los nodos obligatorios incluidas las reacciones (r_{i1}, r_{i2}) $1 \leq i \leq t$ las parejas mutuamente exclusivas de reacciones, el CSP es:
Minimizar Weight(G, w)

sujeto a:

$Subgraph\,(G, g) \wedge Path(G, n_1, n_m) \wedge \forall i \in [1, m] : n_i \in Nodes(G) \wedge \forall i \in [1, t] : r_{i1} \notin Nodes(G) \vee r_{i2} \notin Nodes(G)$

El problema de optimización presentado se resuelve usando una heurística first fail para búsqueda de caminos.Esta heurística selecciona el nodo con el más bajo grado de salida y escoge la arista de salida la cual maximiza el grado de entrada del el nodo al que apunta(targe node, en inglés). Este problema es resuelto para las tres secuencias metabólicas lineares más largas: lysine, glycolysis y heme. Todos los nodos que son reacciones son obligatorios.El peso de cada nodo es su grado.Cada reacción es doblada y su reacción inversa es mutuamente exclusiva (no puede aparecer dentro del mismo camino).

2.3.7.2. Problema del Recorrido del Caballo

Este es otro problema que es posible resolver usando las restricciones de CP(Graph), el problema consiste en a partir de las movidas (saltos) del caballo (en L) en un ajedrez, encontrar un camino que

recorra todos los cuadros del ajedrez exactamente una vez.Éste es un ejemplo del problema del camino hamiltoniano sobre un tipo especial de grafo llamado Grafo del Caballo[1].

Un modelo de CP(Graph) simple para el recorrido del caballo es:
Sea $KG_{8,8}$ el grafo del caballo para un tablero cuadrado de 8 x 8 cuadros.Este grafo se compone de un nodo para cada cuadro de el tablero de ajedrez y aristas conectando cada pareja de cuadros los cuales pueden ser enlazados por una movida del caballo.Sea $n_{0,0}$ un cuadro ubicado en la esquina superior izquierda del tablero y $n_{1,2}$la ubicación que resulta de una movida del caballo desde $n_{0,0}$, el problema es:

Subgraph(G, $KG_{8,8}$) $\wedge$ Nodes($KG_{8,8}$) = Nodes(G) $\wedge$ Path(G, n0, 0, n1, 2)

Capítulo 3

Identificación y Análisis del Problema

El objetivo de éste capítulo es presentar:

1. Los aspectos clave de Gecode que permitan entender los cambios realizados a CP(Graph) original.
2. Los cambios que tuvo Gecode que afectan el funcionamiento de CP(Graph) original en Gecode 2.1.1.

Para cumplir con el primer objetivo se hizo:

- Se aprendió sobre los diferentes componentes de Gecode los cuales se usan en CP(Graph) original, para esto se hicieron experimentos, se consulto en la lista de Gecode, se revisaron los ejemplos que vienen con Gecode 2.1.1 entre otras tareas.

Para cumplir con el segundo objetivo se uso:

- El registro de cambios entre versiones de Gecode[22].
- La documentación de Gecode 1.0 y Gecode 2.1.1.
- El Compilador de GNU.
- El GDB para rastrear problemas de segmentación y problemas en tiempo de ejecución.

3.1. Gecode

El ambiente de programación de restricciones Gecode fue diseñado utilizando el paradigma orientado a objetos, el modelo de propagación utilizado por Gecode es centrado en los propagadores, en éste modelo los propagadores son planificados y ejecutados, una desventaja del anterior modelo radica en que tiene un costo alto mantener información completa sobre los cambios hechos en el dominio de las variables modificadas, por el contrario, otros sistemas de restricciones utilizan un modelo propagación centrado en las variables, en éste la propagación se basa en planificar las variables modificadas, donde el costo de almacenar información completa de las modificaciones hechas al dominio de las variables es bajo.

Una vista gráfica general Gecode 2.1.1 es proporcionada en [17], la siguiente es la lista completa de tareas que se pueden realizar en Gecode 2.1.1, adicionalmente aquellas que fue necesario entender y aplicar están señaladas con un asterisco:

- Programación de modelos(*).
- Programación de motores de búsqueda.

- Programación de propagadores y estrategias de distribución(*).
- Programación de variables.
- Programación de vistas(*).
- Pruebas.
- Reflexión y serialización.

Dependiendo de la tarea que se desee realizar en Gecode se trabaja con un componente distinto, en la siguiente lista están los componentes principales de Gecode 2.1.1, adicionalmente los componentes manejados en CP(Graph) se distinguen con un asterisco:

1. Modelos y variables.
2. Vista(*) e implementación de variable(*).
3. Motor de búsqueda(*) y espacio(*).
4. Propagador(*), condición de propagación(*) y evento de modificación(*).
5. Estrategia de distribución(branching)(*).
6. Iterador(*)

A continuación se explican los componentes que hacen parte de CP(Graph):

1. **Modelos y variables**

La programación de un en CSP en Gecode 2.1.1 va de la mano de la utilización de objetos variable. Las variables son clases disponibles al usuario final para que lleve a cabo el modelamiento del problema.

Si se desea programar un CSP en Gecode 2.1.1 se debe hacer:

- Crear una clase para modelar el problema la cual debe de extender de la clase Space de Gecode.
- Declarar las variables (los objetos variables permiten declarar un dominio inicial, fijas restricciones sobre éstas y consultar el estado de su dominio) del CSP y su respectivo dominio.
- Especificar las restricciones y relaciones entre las variables declaradas en el punto anterior
- Especificar la estrategia de distribución que desea aplicar para la solución del problema.
- Definir un constructor de copia (Gecode 2.1.1 explora bajo copying)
- Crear una instancia de la clase Options de Gecode 2.1.1 e indicar cuantas iteraciones desea realizar en el proceso de búsqueda u otras opciones de configuración pertinentes al problema
- Crear una instancia de la clase que modela el problema.
- Llamar el método doSearch() de la instancia anteriormente creada.

2. Vistas e implementación de variable

Las vistas a diferencia de las variables, son clases no disponibles directamente al usuario final que modela problemas de restricciones, si no a las personas que se encargan de implementar nuevas variables y dominios en Gecode. Las vistas e implementaciones de variable describen la funcionalidad interna de una variable que usa el usuario. La variable no implementa métodos, es simplemente una interfaz que esconde al usuario final métodos que le permitirían manipular directamente el dominio de una variable. El siguiente gráfico presenta una vista general de como encajan las vistas, variables e implementación de variables en Gecode 2.1.1:

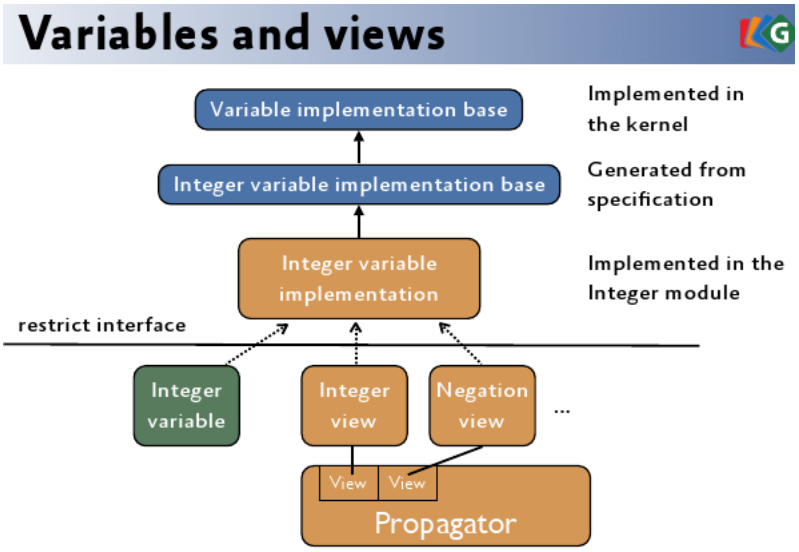

Figure 3.1: Vistas y Variables en Gecode [24]

Para llevar a cabo el corte del dominio de una variable, el propagador tiene dos opciones que son o usar un objeto implementación de variable de aquí en adelante llamado varimp o un objeto vista de aquí en adelante llamado vista.

Un varimp proporciona métodos para consultar y actualizar el dominio de una variable, una vista tiene una referencia a una varimp pero adicionalmente proporciona métodos para transformar la entrada del propagador (el dominio de la variable) y para transformar la salida que éste genera.

Si por ejemplo se implementara un propagador para la restricción $[Max\{x_1, ..., x_n\} = y]$, y se deseara implementar también otro propagador para la restricción $[Min\{x_1, ..., x_n\}]$, usando varimp habría que tener dos propagadores, cada uno con una referencia a una varimp distinta, ésto dado el ligero cambio en la restricción el cual genera un cambio en los tells que maneja cada varimp, pero estas 2 restricciones están relacionadas por la equivalencia:

$[Min\{x_1, ..., x_n\}] = [Max\{-x_1, ..., -x_n\} = -y]$.

Para resolver éste problema de redundancia de código, se introdujeron las vistas, la idea es que las vistas adecúen la entrada y salida para el propagador.El problema anterior visto desde el enfoque de las vistas se podría resolver usando un clase propagadorA que recibe al ser instanciada una referencia a una vista A para la restricción Max y un clase propagadorA que recibe al ser instanciada una referencia a una vista B la cual adecúa la entrada y salida para éste, este propagador se denomina derivado, a nivel de programación esto se logra usando lo que se denomina parametricidad.La abstracción de parametricidad a través de tipos permite que el mismo código pueda ser instanciado con diferentes parámetros de tipos compatibles (el mismo propagador sera instanciado con distintas vistas).

En gecode existen variables como IntVar, SetVar y de forma equivalente existen IntView, SetView.

IntVar, SetVar están disponibles a los usuarios para el modelamiento, mientras las segundas (IntView, SetView) están disponibles a los desarrolladores en Gecode.

El cuerpo de una vista esta compuesto de:

- **Métodos de acceso:** permiten ver el estado del dominio actual, permiten iterar sobre los valores del mismo y responder inquietudes sobre valores del dominio e.g Métodos para acceder el máximo valor del dominio, el tamaño del dominio, la distancia entre el valor más pequeño del dominio y el más grande del dominio.
- **Pruebas de dominio**: métodos que retornan bool y que normalmente es el resultado de alguna búsqueda, e.g probar si un numero se encuentra dentro del dominio
- **Tells**: permiten modificar el dominio actual de una Variable y retornan siempre un ModEvent. e.g métodos para restringir que el dominio sea menor o igual a un numero
- **Tells con iteradores**: tienen la misma funcion que los anteriores Tell, solo que el recorte de dominio se realiza con un iterador, el cual tiene una funcion interna que evalúa cada valor del dominio evaluando si el valor es parte o no del dominio.

3.**Motores de búsqueda y espacio**

El encargado de encontrar las soluciones a un CSP en Gecode es el motor de búsqueda, éste tiene 2 tareas que son: encontrar todas las soluciones al CSP y garantizar la correctitud de estas soluciones, para esto el motor de búsqueda utiliza dos técnicas que son la búsqueda y la inferencia, la búsqueda (implementada con *la estrategia de distribución*) asegura encontrar todas las soluciones al CSP, la inferencia (implementada con *propagadores*) asegura que cada solución cumpla las restricciones del CSP.

Durante la búsqueda de las soluciones a un CSP el motor de búsqueda genera un árbol de búsqueda, un nodo en el árbol de búsqueda representa un punto fijo (una foto del sistema en ejecución) calculado por la propagación, este punto fijo se denomina un espacio.Cada espacio se caracteriza por tener:

- Cola de propagadores: indica cuales propagadores deben ser corridos.
- Cola de estrategias de búsqueda .
- Dominios de variables.
- SpaceStatus.

El SpaceStatus de un espacio representa el estado del espacio actual, el cual puede ser: una solución al CSP, no ser una solución al CSP (por ejemplo una variable es vacía) o puede contener variables no asignadas (en el caso de variables enteras, variables con más de un valor), a nivel de implementación esto es:

- **SS_SOLVED**: Ningún propagador retorno ES_FAILED y ninguna distribución es permitida.
- **SS_FAILED**: Un propagador retornado es fallido.
- **SS_BRANCHED**: haga distribución, ya que las variables no están asignadas y el propagador no retorno falla así que pueden haber soluciones al CSP.

Algo importante sobre un espacio es que al tomar su estado SS_FAILED, se requiere restaurar éste a el estado anterior para continuar con la búsqueda, para esto se utilizan los métodos:

- status()
- clone(share)
- commit(desc, a)

Para profundizar en como Gecode utiliza un espacio se puede consultar [18].

4.Propagador, condición de propagación y evento de modificación

Un propagador es una abstracción la cual brinda la implementacion de una restricción, un propagador a nivel conceptual esta representado por unas reglas de filtrado, estas reglas de filtrado reflejan la forma en que el propagador va a podar los dominios de las variables y deben garantizar la correctitud del propagador, es decir, que genera las soluciones correctas a la restricción.En Gecode se pueden implementar nuevos propagadores que apliquen a las variables por defecto de Gecode, pero también para variables "nuevas", creadas a partir de las variables por defecto de Gecode.Cada propagador retorna un Status que indica al estado actual que ocurrió durante la propagación.

Desde el punto de vista del número de restricciones que toman los propagadores, los propagadores que Gecode proporciona se pueden clasificar en:

- **UnaryPropagators**: Se refieren a restricciones de una sola variable, e.g Sea X una variable entera, la restricción X>3 es una restricción unaria pues implica una sola variable de restricciones.
- **BinaryPropagators**: Se refieren a restricciones que involucran dos variables e.g: X && Y=0 donde X, Y son variables del dominio de los booleanos
- **TernaryPropagators**: Se refieren a restricciones que involucran tres variables e.g: X && Y=Z donde, X, Y, Z son variables del dominio de los booleanos
- **NaryPropagators**: Se refieren a restricciones que involucran arreglos de variables e.g la restricción lineal típica, expresada de la forma A+B+C=T donde T es una constante entera A, B, C son variables de restricciones Enteras en este caso, gecode podría tomar las variables A, B, C como un array de IntVars y el propagador recibiría el array de IntVar como la variable de entrada al propagador.
- **NaryOnePropagators**: Se refieren a restricciones que involucran un array de variables y una variable de restricciones, por ejemplo, la restricción lineal típica, expresada de la forma A+B+C=T donde T es una variable entera de restricciones y donde A, B, C son variables de restricciones enteras, en este caso, Gecode podría tomar las variables A, B, C como un array de IntVars y el propagador recibiría el array de IntVar como la variable de entrada al propagador y una variable entera de restricciones T.

Teniendo lo anterior claro, se tiene en Gecode para los diferentes tipos de propagadores mencionados clases de las cuales podemos extender (Heredar) de acuerdo al propagador que tengamos intención de implementar. Se debe tener en cuenta que todos los propagadores heredan de la clase Propagator.

Un propagador P depende de una vista X (Variable sobre la cual aplica la restriccion) con una *Condición De Propagación* P y es llamado (encolado para su ejecución) cuando una actualización de dominio sobre X retorne un evento de modificación que sea el evento que encaja en la Condición de Propagación. Las siguientes son las Condiciones de Propagación que Gecode incluye por defecto:

- *PC_INT_VAL* Indica que el propagador sera corrido cuando se realicen asignaciones de valor
- *PC_INT_BND* Indica que el propagador sera corrido cuando se realicen cambios sobre las cotas superiores e inferiores de la variable.
- *PC_INT_DOM* Indica que el propagador sera corrido cuando se realicen cambios de cualquier tipo sobre la variable.

Durante la aplicación de las reglas de podado de dominio dentro del propagador, se le tendrá que decir a una variable que asigne su cota superior a un determinado valor o que su cota inferior sea modificado a un valor determinado, estas funciones que modifican el dominio de una variable se llaman "Tells" dentro de la jerga de Gecode. Como regla en Gecode, siempre que se haga un llamado a un Tell se tendrá que revisar si el Tell ha retornado un valor diferente de fallo, para que en caso de fallo gecode se encargue de determinar que el problema no tiene solución. Los propagadores manejan el failure (cuando el dominio de una variable es vació) y controlan la propagación [1].

Un Propagador puede retornar:

- **ES_FAILED**: Indica que el propagador ha fallado y que el espacio debe por tanto fallar
- **ES_SUBSUMED**: Indica que el propagador no realizara más cambios al dominio durante el resto del CSP y por tanto el objeto puede ser destruido.
- **ES_FIX**: Indica que en el Espacio actual correr el propagador nuevamente no realizara ninguna poda en el dominio de las variables
- **ES_NOFIX**: Indica que el propagador no ha alcanzado un punto fijo y que en caso de que su PC se cumpla, entonces debe ser encolado nuevamente.

5.Iterador

Un iterador proporciona acceso a los elementos de una colección en un orden secuencial, uno al tiempo[23].En Gecode existen dos iteradores los cuales permiten acceder los elementos de el dominio de una variable, el iterador de valor permite acceder elemento por elemento del dominio de la variable y el iterador de rango que permite acceder una secuencia de rangos obtenida del dominio de la variable, por ejemplo para el conjunto de valores enteros $s = \{1, 2, 4, 5, 7\}$ el iterador de valor permitiría acceder a 1, 2, 4, 5, 7 pero el iterador de rango permitirá acceder a los rangos, es decir, $<[1, 2]\,[4, 5][7]>$, en este caso son 3 rangos y el iterador de rango avanzara sobre cada rango y solo permite hallar el mayor y menor de cada rango más no iterar sobre cada elemento dentro del rango.

Algunas veces se requiere de que un Tell haga por ejemplo un recorte de dominio sobre más de un valor, se requiere incluir varios valores dentro del dominio de la variable, o se requiere de métodos de acceso al dominio que devuelven mas de un valor, en estos casos se usan los iteradores para almacenar esos valores del dominio o para realizar un Tell precisamente sobre esos valores del dominio.

6.Estrategias de distribución

La estrategia de distribución hace parte de la búsqueda de la solución a un CSP, ésta determina la forma la forma del árbol de búsqueda escogiendo la próxima variable y valor.

La búsqueda de la solución comienza con los propagadores, una vez los propagadores han alcanzado un punto fijo, el proceso de distribución comienza.En este punto existen tres posibilidades:

- Todas las variables del CSP están asignadas.En este caso la búsqueda se detiene ya que una solución se ha obtenido.
- Existen variables que todavía no están asignadas.En este caso, se realiza un nuevo paso de distribución.Un paso de distribución consiste de:

  1. Seleccionar próxima variable a explorar.
  2. Seleccionar el valor (alternativa) de la variable escogida.

Una vez esta selección se ha hecho, se agrega la correspondiente restricción al almacén (store).Esta adición de restricción posiblemente active algunos propagadores, así que se debe esperar hasta que todos los propagadores queden fijos para realizar una nueva iteración de distribución.

Si hay al menos una variable cuyo dominio es vació, en este caso, se vuelve al estado anterior, y se prueba una alternativa distinta esperando por la estabilización de los propagadores (que alcancen un punto fijo).

3.2. Cambios Detectados

En esta sección se mencionan los cambios detectados entre Gecode 1.0.0 y Gecode 2.1.1 que hacen que CP(Graph) no funcione en Gecode 2.1.1.

3.2.1. Algoritmo de Propagación.

Como se ha mencionado en las secciones anteriores existen dos tipos de "*propagación*", propagación orientada en propagadores y propagación orientada en las variables.

Propagación Orientada en los propagadores

El algoritmo de propagación de Gecode es orientado en los propagadores, lo que significa que los propagadores que aun no han alcanzado un punto fijo son encolados para su ejecución en cada espacio.

Propagación Orientada en las variables

La propagación orientada en las variables, no tiene una cola de los propagadores, si no de las variables cuyo dominio ha sido modificado, durante cada ejecución se correrán aquellos propagadores que están subscritos a esas variables.

La propagación orientada a las variables tiene ventajas sobre la propagación orientada en propagadores pues durante la propagación se puede saber exactamente que variable fue la que invoco el propagador, y además se puede saber cual fue el cambio en el dominio de esa variable. Esto permite hacer que el propagador cada vez que realiza su propagación no empiece a calcular desde cero (*Propagación Incremental*).

e.g: Supóngase que se tiene un propagador:

$$z = \sum_{i=1}^{k} x_i$$

En propagación orientada a las variables, cada vez que el propagador sea ejecutado, sabría exactamente cual variable x_l cambio y cual fue cambio en el dominio de la variable de x_l. Normalmente en este propagador el valor de la banda inferior de Z es la suma de las bandas inferiores de cada x_i, en este caso para saber el nuevo valor de la banda inferior de Z solo se tendría que hacer la diferencia entre el dominio nuevo y antiguo dominio de x_l.

En Gecode 2.0, se agrego un nuevo componente llamado *Advisor* para permitir a la propagación

orientada a los propagadores realizar propagación Incremental. Este cambio introdujo algunas modificaciones al algoritmo de ejecución de Gecode.[14]

El cambio del algoritmo de propagación tuvo impacto en la forma como se programan los Propagadores y las Views.

Algoritmo de Propagación Gecode 1.0

Algorithm 2 Algoritmo de Propagación Centrado en los Propagadores (Gecode 1)[14]

```
Propagate(P,s)
begin
    N ← P;
    while N ≠ ∅ do
        remove p from N;
        ⟨l, σ⟩ ← p(d, state[p]);
        s' ← s; state[p] ← σ;
        foreach x ∼ n ∈ l do
            s' ← s'[x ∼ n];
        N ← N ∪ ⋃_{x∈dis(s,s')} prop[x];
        s ← s';
    return s;
end
```

P es una cola donde se encuentran los propagadores que deben ser corridos, S es un store donde se almacenan las variables y sus dominios. El algoritmo consiste de un ciclo que se mantiene mientras existan propagadores en la cola. Cada propagador p dentro de P tiene un estado representado por la instrucción $state[p]$, además cada propagador recibe un dominio y un estado, devolviendo:

- l, un listado de los tells (cambios de dominio) que el propagador ordena hacer.
- q, un nuevo estado, arrojado después de la propagación.

El comando $prop[x]$ devuelve los propagadores subscritos a la variable x.

Luego de propagar, se explora el listado de tells, y se ejecuta cada tell, lo que da como resultado un nuevo Store s, y por cada variable x cuyo dominio haya sido modificado, se agregaran a la cola aquellos propagadores que estén subscritos a la variable x.

Algoritmo de Propagación Gecode 2.1.1

El algoritmo de propagación de Gecode 2.1.1 Introduce los Advisores.

Un Advisor es una funcion que esta relacionado con un propagador, el Advisor recibe que variable en particular ha cambiado y el cambio del dominio sobre esa variable en particular, según esta información el advisor puede hacer dos cosas:

- Indicarle al propagador si debe correr o no debe correr.
- Cambiar el estado del propagador para indicarle al propagador como debe propagar y de esta forma propagar más eficientemente.

Un Advisor a se define como una funcion que tiene como entrada un store s, un estado q y un tell t y retorna una pareja $a(s, q, t) =< \sigma', Q >$donde σ'es un estado y Q es un conjunto de propagadores. Un advisor da concejo a un solo propagador denominado $prop[a]$[14].

La idea detrás de los Advisores radica en que es una funcion simple y que dentro de esta funcion no se intentara propagar(hacer tells). En caso de indicar que no se debe propagar el conjunto seria equivalente a $Q = \emptyset$.

Un advisor a depende de una y una sola variable denominada $var(a)$, por tanto el advisor a sera ejecutado solamente cuando la variable de la que depende fue modificada. el comando $adv[x]$denomina la lista de Advisores que depende de la variable x.

Teniendo en cuenta lo anterior el algoritmo propuesto para la propagación con advisores es el siguiente:

Algorithm 3 Algoritmo de Propagación Centrado en los Propagadores con Advisores (Gecode 2.1.1)[14]

```
Propagate(P,s)
begin
    N ← P;
    while N ≠ ∅ do
        remove p from N;
        ⟨l, s⟩ ← p(s, state[p]);
        s' ← s; state[p] ← s;
        foreach x ∼ n ∈ l do
            s' ← s'[x ∼ n];
            foreach a ∈ adv[x] do
                ⟨s, Q⟩ ← a(s', x ∼ n, state[prop[a]]);
                state[prop[a]] ← s; N ← N ∪ Q;
        N ← N ∪ ⋃_{x∈dis(s,s')} prop[x];
        s ← s';
    return s;
end
```

El algoritmo en general es el mismo, excepto en el ultimo segmento, en donde por cada variable cuyo dominio ha sido modificado se explorara su lista de Advisores, llamándose a cada advisor, esto cambiara internamente los estados de los propagadores y determinara cuales propagadores deben encolarse.

El cambio en el algoritmo de propagación tiene como consecuencia:

- Nuevos elementos a nivel de implementacion Objetos Deltas, Advisores, Councils.
- Cambio de responsabilidades de algunas tareas.

3.2.2. Propagadores

- Los propagadores ahora son encargados de terminar la subscripción de las variables que depende de ellos.
- Los propagadores tienen nuevas PC (condiciones de propagación, PropagationConditions en inglés).
- La función Propagate recibe nuevos parámetros (Deltas), los objetos delta permiten en el método de la propagación saber que tipo de cambio activo la PC (sobre quien se realizo el cambio de manera general. e.g la banda inferior o superior)
- Los propagadores son encargados de implementar la funcion advise en caso de que se quiera usar los advisores
- Cambiaron la definición de algunas constantes de Gecode
- Algunas restricciones sobre los booleanos disponibles en Gecode 1, no están en Gecode 2, sin embargo restricciones equivalentes son expresables.

3.2.3. Variables

En Gecode 1 las Variables BoolVar e IntVar dependían de la misma VarImp, esto quiere decir que dentro del kernel las variables booleanas eran manejadas como variables enteras cuyo dominio estaba entre 0 y 1. Esto permitía imponer restricciones de enteros sobre variables Booleanas y restricciones Booleanas sobre variables Enteras.

En Gecode 2.1.1 las BoolVar tienen su propia VarImp lo que no permite imponer restricciones de Enteros sobre las BoolVar, e igualmente no permite imponer restricciones booleanas sobre variables Enteras.

3.2.4. Distribución

- Las constantes para elegir las variables de distribución cambiaron de identificador.

3.2.5. Otros Problemas Encontrados

A continuación se enumeran otros problemas encontrados durante la etapa de análisis del problema. Durante esta etapa se concluyo que:

- La implementacion de CP(Graph) no contiene todos los propagadores y/o vistas expresados en la tesis de Gregoire Dooms
- Los usuarios de CP(Graph) realizan el modelamiento directamente sobre objetos que no tienen encapsulados los métodos que permiten realizar los tells.
- El propagador de PathDegree esta incompleto para la vista *NodeArcSetsGraphView*, lo que al mismo tiempo hace que la restriccion de Path no funcione correctamente para todos las instancias, dado que la restriccion de Path internamente esta basada en la restriccion de PathDegreee. Sin embargo PahtDegree funciona correctamente para la vista *OutAdjSetsGraphView.*
- Se encontró otra versión de CP(Graph) con más propagadores en [21], sin embargo al no coincidir con la versión del repositorio Oficial, ni tener documentación de ayuda, además de tener un código fuente desordenado se descarto el trabajar sobre esta versión. Este trabajo de grado y las modificaciones hechas se hicieron sobre la versión encontrada en [20]. Sin embargo, se usaron algunos aportes encontrados en [21].

Capítulo 4

Modificaciones

Éste capítulo pretende mostrar las modificaciones más importantes realizadas a CP(Graph).
El procedimiento que se aplico para realizar las modificaciones fue:

- Identificar mediante el registro de cambios de Gecode los cambios a nivel de implementación [22].
- El uso del compilador de GNU para rastrear errores de compilación.
- El uso del GDB para rastrear problemas de segmentación y depuración.
- El uso intensivo de la documentación de Gecode 1.0 y Gecode 2.1.1.

4.1. Propagadores

4.1.1. Cambios Generales

Sobre los Propagadores se realizaron cambios a nivel de implementación que involucraban:

- Incluir nuevos parámetros en los métodos de propagación
- Cambiar identificadores de constantes
- Involucrar restricciones de Channeling para permitir imponer restricciones de las IntVar a variables del dominio de los Booleanos.
- Evitar la implementación de código, haciendo que los Propagadores hereden de las clases UnaryPropagator y BinaryPropagator.

4.1.2. Propagadores con Visitors

Durante la propagación existen varios Propagadores que usan visitors, los visitors son clases que a su vez usan un scanner. En el CP(Graph) original la idea es iterar durante todo el scan, sin importar si se ha encontrado un fallo, luego se revisa si se encontró un fallo. La idea fue implementar un método que permitiera determinar cuando habría encontrado un fallo y de esa forma evitar completamente el scaneo posterior, pues ya se sabe de antemano que el Propagador generara un fallo y no sirve de nada seguir el scan.

4.1.3. Propagadores Agregados

Se agregaron tres Propagadores encontrados en la versión no oficial [21] de CP(Graph), para ambos se comprobó su funcionamiento con ambas Vistas. La lógica de su funcionamiento es la misma que la expresada teóricamente en la tesis de Gregoire Dooms.

Los propagadores que se agregaron fueron:

- Symmetric[15]
- Undirected[15]
- Subgraph[15]

4.2. Advisores

Se experimento con la implementación de Advisores para varios de los Propagadores disponibles. Finalmente se adoptaron dos advisores para los Propagadores Symmetric, Subgraph y Complement. .

Los Advisores en los Propagadores propuestos por Dooms en CP(Graph) tienen gran utilidad y pueden reducir el tiempo de ejecución notablemente.Normalmente a nivel de implementación durante la propagación, los Propagadores de CP(Graph) tienen que usar un visitor y un scanner. El scanner es un objeto encargado de explorar las aristas/nodos de un grafo y clasificar cada arista/nodo, esta clasificación es usada por el visitor para almacenar cuales aristas/nodos deben ser quitados o agregados del Intervalo de grafos. e.g el Propagador de Complement (Impone la restriccion g1 es el complemento de g2) tiene un scanner que recibe dos grafos, este scanner tiene que generar cada arista del grafo completo, y clasificarlas para el ComplementVisitor, quien le informara al Propagador que aristas agregar al LB y cuales quitar del UB. La figura 9 tiene un esquema del funcionamiento más detallado.

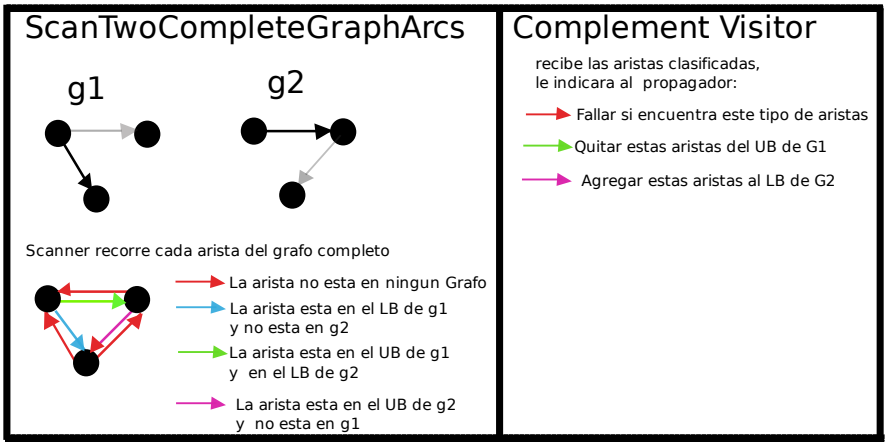

Figura 4.1: Scanner y Visitor usados por el Propagador de Complemento

Durante cada propagación del Propagador de complemento, se debe iterar por cada arista del grafo completo, sin importar si el cambio sobre uno de los dos grafos fue el más mínimo, el Propagador usara el visitor, el visitor usara el scanner y el scanner realizara una iteración completa para entregarle al visitor la clasificación de las aristas.

El funcionamiento es similar para todos los Propagadores, la única diferencia entre los Propagadores son los visitores y scaners que usan, es claro que existen scaneres más costosos que otros. En Gecode 1.0 esto era necesario puesto que no había forma de saber cual variable cambio, ni cual fue el cambio realizado, por ello durante cada propagación el Propagador tenia que “empezar de cero”.

En Gecode 2.1.1 se tiene el componente extra, los advisores que pueden informar que variable cambio y cual fue el cambio del dominio, aunque lo anterior tiene una limitación, los advisores solo pueden informar de cambios pequeños sobre una variable, si existieron muchos cambios entonces el advisor no puede dar un informe de los cambios.

Retomando el ejemplo anterior incluyendo advisores, el advisor podría cambiar el estado interno del Propagador diciéndole que el ultimo cambio fue que una arista se agrego al LB del grafo g1, en caso tal, un scaneo completo no es necesario, bastaría con que el Propagador haga un tell diciéndole al grafo g2 que remueva la arista del UB de g2.

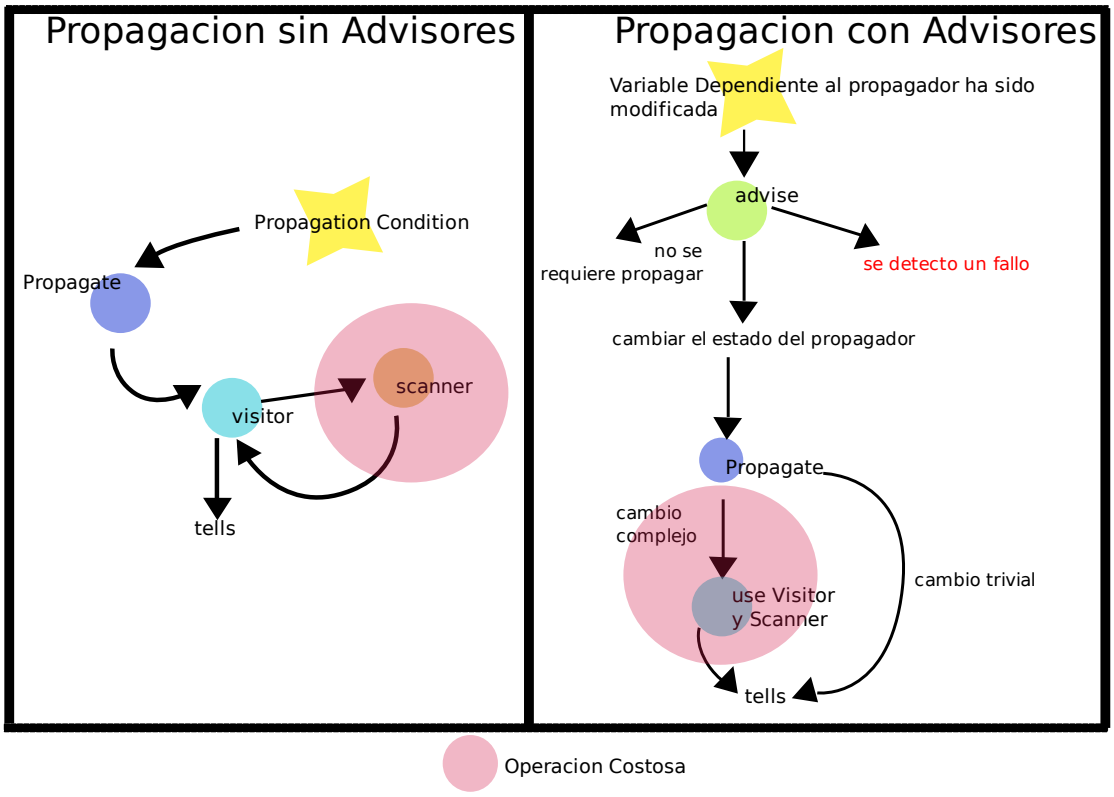

Figura 4.2: Esquema general del cambio en la propagación con advisores

4.2.1. Modificaciones Generales Hechas Sobre los Propagadores que Usan Advisores

Aquí se mencionan las modificaciones que se realizaron sobre todos los Propagadores que usan advisores.

El manejo de advisores en Gecode es relativamente sencillo, consiste en implementar una funcion virtual en los Propagadores, esta funcion se llama advise, allí se modificara el estado interno del Propagador y se le indicara si debe o no propagar. Sin embargo hay que hacer unos cambios para adecuar un Propagador de forma que trabaje conjuntamente con los advisores.

- La PC a las que están subscritos es PC_GEN_NONE, quiere decir que ningún Propagador que use advisores sera ejecutado por una condición de propagación, la única forma de que la funcion de propagación sea ejecutada es por que el advisor ha determinado que el Propagador debe ser corrido.

- Cada Propagador contiene una variable entera “cuenta” que se encarga de contar cuantas veces ha sido corrido, y cuyo objetivo es garantizar que la primera vez que el Propagador es corrido se realiza una propagación con scaneo. Existe otra forma de lograr este mismo efecto, y es subscribiendo manualmente el Propagador al inicio en el CSP planteado.

- Cada Propagador contiene una variable entera “activo” cuyo trabajo es ser una bandera. Cuando el Propagador con advisores realiza la propagación se realizan un conjunto de Tells, que generan al mismo tiempo ModEvents sobre las variables cuyos dominios están siendo modificadas, cuando estos ModEvents Ocurren, los advisores que están suscritos a estas variables, empezaran a ejecutarse y a encolar los respectivos Propagadores a quienes están suscritos. La idea con la bandera es indicar cuando el advisor llamado pertenece al Propagador que actualmente esta corriendo y de esta forma evitar que lo encole nuevamente.

- Cada Propagador contiene una variable "co" la cual es un objeto Council y sera encargado de mantener los advisores. Esta variable es usada en el constructor del Propagador para subscribir las variables a los advisores.
- En el constructor del Propagador aquellas variables que dependan del Propagador deben subscribirse a un advisor.
- Cada Propagador usa la clase nameAdvisor para saber en el método advise, cual variable ha cambiado y que es lo que cambio (Una arista o Un nodo), además para identificar si se agrego al LB o se quito del UB.
- Cada Propagador contiene una variable entera "indicador" que indica el estado actual del Propagador. Esta variable determina la forma de propagar.

NameAdvisor
La clase NameAdvisor fue creada, esta clase extiende de un viewAdvisor<SetView>. Un objeto de esta clase llega como parámetro al método Advise, este objeto contiene toda la información concerniente a que fue lo que cambio en el dominio (Una arista o Un Nodo) y si fue agregado al LB o removido del UB.

Esta clase extiende de un SetView puesto que internamente las variables de grafos están construidas a partir de SetView, lo que significa que a bajo nivel, el quitar o poner una arista/nodo es una modificación a una o más variables de conjuntos.

- El atributo graph_id, sirve como referencia para los Propagadores que tienen varias variables de grafos, y de esta forma saber en el método advise cual grafo fue el que genero el llamado a la funcion Advise.
- El atributo nombre sirve como referencia para saber si el cambio se efectuó sobre una arista o un nodo.
- El atributo índice sirve para saber exactamente cual variable cambio dentro de un arreglo de variables de conjuntos.

4.3. Variables & Views

En esta sección se explicaran aquellos cambios relevantes hechos sobre las Vistas de CP(Graph)

4.3.1. Inclusión de Advisores

La inclusión de advisores o al menos la posibilidad de tener advisores genero cambios importantes en las dos Vistas disponibles en CP(Graph). Debe existir un método capaz de suscribir un advisor a una variable de grafos. Para ello se incluyo un nuevo método en cada Vista, el método Subscribe que recibe un Council, un Propagador y un identificador de grafo (nombre para el grafo).

Este método se encarga de suscribir un advisor a una Vista de grafos. La razón por la que este método se incluye en la Vista es para abstraer del Propagador la forma como cada Vista subscribe los advisores, de esta forma cada Vista tiene una manera diferente de realizar la subscripción, siendo para el Propagador indiferente la forma como este proceso se haga.

Dado que Cada implementación de Variable subscribe a los advisores de manera diferente, se explicara como se realiza la subscripción para las Vistas disponibles en CP(Graph) ya que esto tiene un impacto importante en el rendimiento de los Propagadores que usen los advisores. En el capítulo 2.3.3 se explica detalladamente las características de las Vistas disponibles en CP(Graph)

OutAdjSetsGraphView

La subscripción de esta Vista se hace subscribiendo las variables internas que la conforman:

- **NodeSet**: La variable de conjuntos, que determina el conjunto de nodos del grafo resultante.
- **OutN**: el arreglo de variables de conjuntos, que determina para cada nodo el conjunto de vecinos de salida (out neighbors, en inglés).

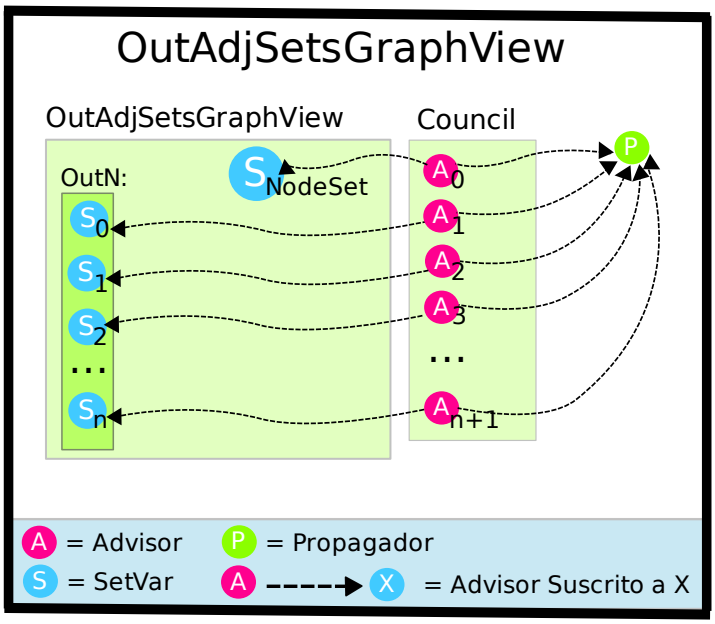

Figura 4.3: Suscripción de Advisores en OutAdjSetsGraphView

La figura 12, muestra como un Propagador P que usa advisores suscribe una de sus variables que es una Vista OutAdjSetsGraphView a los advisores. Existe un advisor por cada variable de conjuntos, quiere decir que cada vez que una de estas variables sea modificada el método advise del Propagador sera llamado y la información sobre el cambio del dominio viajara en un objeto tipo Advisor.

NodeArcSetsGraphView

La subscripción de esta Vista se hace subscribiendo las variables internas que la conforman:

- **NodeSet**: La variable de conjuntos, que determina el conjunto de nodos del grafo resultante.
- **ArcSet**: La variable de conjuntos, que determina el conjunto de aristas del grafo resultante.

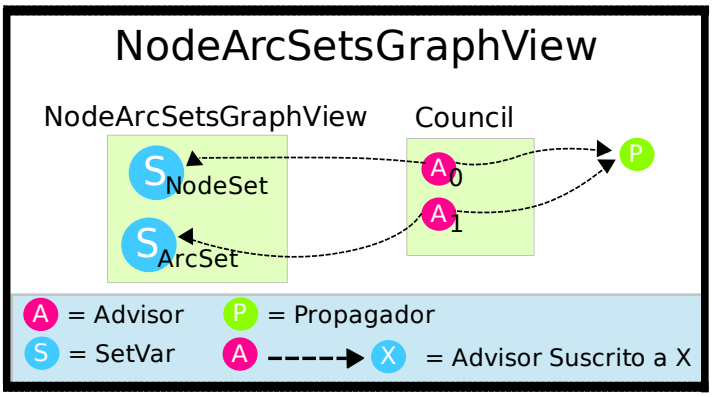

Figura 4.4: Suscripción de Advisores en NodeArcSetsGraphView

La figura 13 muestra como un Propagador P que usa advisores suscribe una de sus variables la cual es una Vista NodeArcSetsGraphView a los advisores. Existe un advisor por cada variable de conjuntos, es decir solo dos advisores.

Capítulo 5

Pruebas y Discusión de Resultados

Este capítulo pretende informar sobre el desarrollo de las pruebas realizadas sobre CP(Graph) actualizado, los resultados obtenidos en éstas y las conclusiones a las que se llego luego de analizar los resultados.

5.1. Pruebas Realizadas

Las pruebas realizadas se hicieron con 2 objetivos: evaluar la correctitud de CP(Graph) actualizado y estudiar su rendimiento.

1. **Correctitud**

   Partiendo de que la lógica de los propagadores no se modifico, es decir, las reglas de podado que maneja el propagador son las adecuadas y de que se introdujeron mejoras a nivel de programación, el objetivo fue probar que estas mejoras generaban los resultados esperados, para esto:

   a) Se reviso que CP(Graph) actualizado generara respuestas semejantes y validas (respuestas que cumplen las restricciones del problema) ante los mismos escenarios que CP(Graph) original.

   b) Para los propagadores que se trajeron de CP(Graph) versión no oficial, en el caso de los que no implementan advisores se probo que sus resultados fueran validos y en el caso de algunos que manejan advisores, se probo la versión con y sin advisores y se reviso que los resultados fueran validos e iguales en una versión del propagador con respecto a la otra.

   c) Se reviso que en problemas que involucraban propagadores de CP(Graph) versión no oficial y de CP(Graph) original los resultados fueran correctos, por ejemplo, problemas donde se aplicaban las restricciones Symmetric (obtenida de versión no oficial) y Path (obtenida de versión oficial).

   d) Se probaron dos experimentos que no estaban en CP(Graph) original sino en la versión no oficial, pero que resultaron de mucha utilidad para presentar problemas reales que se pueden solucionar con CP(Graph).

   e) Para todos los ejemplos probados en CP(Graph) actualizado, se reviso que ejecutaran con Gist (Una herramienta de búsqueda gráfica e interactiva que sirve para explorar cualquier parte del árbol de búsqueda generado para un CSP) y que el Gist diera los mismos resultados que la consola.

   Para cumplir este objetivo se hizo lo siguiente:

   a) Se diseñaron los ejemplos 1, 2, 3, 4, 5, 8 y 21, estos funcionan sobre tanto CP(Graph) actualizado como CP(Graph) original.

b) Se diseñaron los ejemplos 6, 7 y 9, estos evalúan los propagadores para las restricciones de Subgraph, Symmetric y Undirected.

c) Diseñaron ejemplos 13, 14, 15, 16, 17, 18, 19 y 20, estos evalúan tanto los propagadores que se trajeron de CP(Graph) original como de CP(Graph) no oficial.

d) Utilizaron los ejemplos 11 y 12 los cuales son experimentos creados por Gregoire Dooms.

Como sustento de las pruebas hechas se puede revisar la carpeta PRUEBAS_CORRECTITUD, en esta se encuentra el código fuente de las pruebas y los resultados obtenidos, estos se editaron para que fueran fácilmente comparables usando una herramienta para comparar archivos como por ejemplo diff.

2. **Rendimiento**

A nivel de tiempo y memoria consumida se estudio el comportamiento CP(Graph) actualizado en una seria de escenarios.En detalle se hizo lo siguiente:

a) Se evaluó el rendimiento de CP(Graph) actualizado con una serie de ejemplos que usaban restricciones propias de CP(Graph) original y se evaluó estos ejemplos también en CP(Graph) original.

b) Se evaluó el rendimiento de CP(Graph) actualizado con una serie de ejemplos que usaban restricciones propias de CP(Graph) versión no oficial.

c) Se evaluó el rendimiento de CP(Graph) actualizado con una serie de ejemplos que usaban restricciones tanto de CP(Graph) original como de CP(Graph) versión no oficial, como por ejemplo Path y Undirected.

Para cumplir éste objetivo se hizo lo siguiente:

a) Se diseñaron los ejemplos 1, 2, 3, 4, 5, 8, 21, 22, 23, 24, 25, 26, 29 y 30 los cuales evalúan los propagadores y distribución por defecto de cada vista tanto en la distribución de CP(Graph) actualizado y CP(Graph) original.

b) Se diseñaron los ejemplos 6, 7, 9, 27, 28, 31 y 32 los cuales evalúan los propagadores para las restricciones de Subgraph, Symmetric y Undirected.

c) Se diseñaron ejemplos 13, 14, 15, 16, 17, 18, 19 y 20 los cuales evalúan tanto los propagadores traídos tanto de CP(Graph) original como de CP(Graph) no oficial.

Como sustento de las pruebas hechas se puede revisar la carpeta PRUEBAS_RENDIMIENTO, en esta se encuentra el código fuente de las pruebas y los resultados obtenidos.

5.2. Resultados Obtenidos

Los resultados permitieron determinar en entre varias cosas:

- En que casos los advisores resultaron más convenientes en tiempo y/o memoria.
- Que impacto tuvo en memoria y tiempo la actualización del sistema en general.

Para la obtención de estos resultados se desarrollaron 2 herramientas:

1. La primer herramienta permitió automatizar la compilación, ejecución y obtención de los resultados de los ejemplos para ser posteriormente procesados por la herramienta 2.
2. La segunda herramienta permitió procesar los resultados obtenidos de la anterior herramienta

Cada ejemplo fue corrido 4 veces y se promedio el tiempo y memoria consumido por cada vista.

Abreviaciones y símbolos

- Versión original: v.o
- Versión actualizada sin advisores: v.a.s.a
- Versión actualizada con advisores: v.a.c.a
- el símbolo ”-” se uso cuando no aplicaba la versión.
- Nota:Al lado de cada valor se agregó su desviación estándar en parentesis, por ejemplo “4-(3.2)” quiere decir para el valor promedio 4 la desviación estandar fue 3.2, en el caso de promedios para memoria , para todos los casos la desviación estandar fue 0 esto dado que para cada ejemplo el uso de memoria no vario entre ejecuciones del mismo por ésto implicitamente la desviación estandar de los promedios para memoria es 0.

Descripción de los cuadros 5.1 y 5.2

Las primeras 3 columnas después de la columna con el nombre del ejemplo, se refieren al tiempo promedio consumido por el ejemplo para cada versión, las ultimas 3 columnas se refieren a la memoria promedio consumida por el ejemplo para cada versión,

Ejemplo	v.o (s)	v.a.s.a (s)	v.a.c.a(s)	v.o(KB)	v.a.s.a (KB)	v.a.c.a (KB)
Ejemplo1(branching)	2.09-(0.02)	2.55-(0.12)	2.42-(0.04)	62	43	43
EJemplo2(pathdegree)	1.62-(0.03)	3.67-(0.01)	3.67-(0.02)	643	1156	1156
Ejemplo3(path)	117.85-(1.08)	117.52-(4.40)	103.84-(0.04)	771	1284	1284
Ejemplo4(path-heuristic)	392.06-(1.21)	371.28-(22.35)	353.64-(9.3)	642	1285	1285
Ejemplo5(path-cost)	49.68-(0.38)	54.81-(1.57)	49.69-(0.17)	642	1028	1028
Ejemplo6(subgraph)	-	14.21-(0.09)	5.48-(0.05)	-	40	40
Ejemplo7(symmetric)	-	166.02-(0.86)	28.56-(0.23)	-	48	48
Ejemplo8(complement)	7.16-(0.08)	10.76-(0.10)	3.30-(0.02)	97	67	67
Ejemplo9(undirected)	-	0.0075-(0.00829)	0.0075-(0.00433)	-	18	18
Ejemplo11(expe1)	-	-	-	-	-	-
Ejemplo12(expe2)	-	0.06-(0.00433)	0.06-(0.00433)	-	1155	1155
Ejemplo13(combinadas)	-	6.01-(0.07)	4.07-(0.04)	-	116	164
Ejemplo14(combinadas)	-	622.36-(32.22)	388.2-(14.55)	-	324	324
Ejemplo15(combinadas)	-	1.32-(0.04)	1.22-(0.01)	-	643	643
Ejemplo17(combinadas)	-	5.88-(0.06)	1.97-(0.00829)	-	63	67
Ejemplo18(combinadas)	-	51.93-(2.43)	38.88-(0.39)	-	108	108
Ejemplo19(combinadas)	-	141.90-(6.33)	133.59-(6.05)	-	44	80
Ejemplo20(combinadas)	-	13.93-(0.52)	12.12-(0.09)	-	139	139
Ejemplo21(combinadas)	14.38-(0.14)	18-(0.51)	18.12-(0.13)	119	227	292
Ejemplo22(complement)	0.0475-(0.00433)	0.0625-(0.00433)	0.04-(0.0)	50	43	43
Ejemplo23(complement)	0.15-(0.0)	0.22-(0.0082)	0.17(0.005)	79	75	83
Ejemplo25(complement)	8.60-(0.038)	7.34-(0.008)	2.68-(0.02)	88	67	67
Ejemplo26(combinada)	0.58-(0.01)	0.73-(0.0)	0.75-(0.0)	171	163	355

Cuadro 5.1: Resultados para la Vista OutAdjSetsGraphView

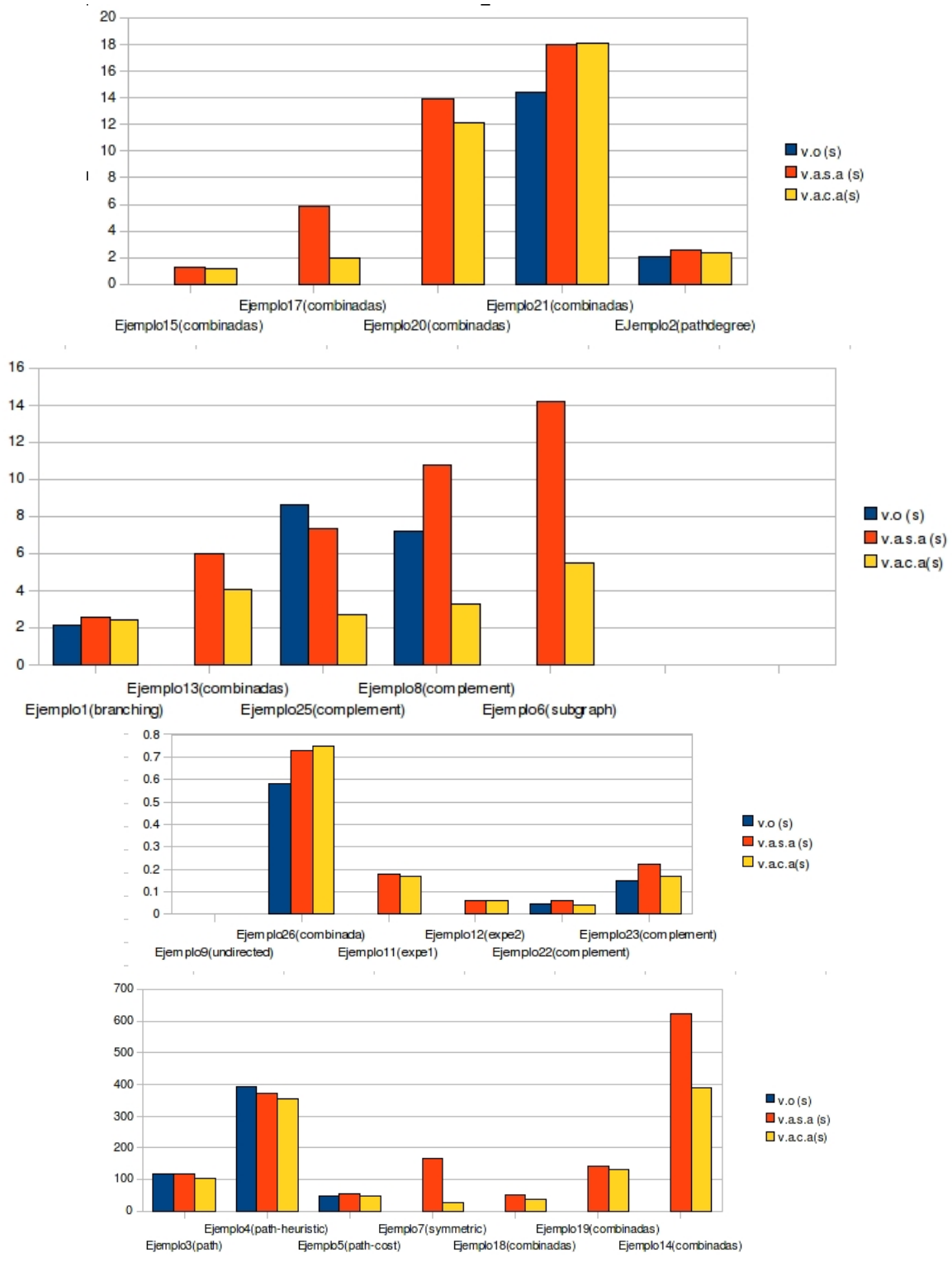

Figura 5.1: Resultados Comparativos de los Ejemplos Usando *OutAdjSetsGraphView*

Ejemplo	v.o (s)	v.a.s.a (s)	v.a.c.a (s)	v.o (KB)	v.a.s.a (KB)	v.a.c.a (KB)
Ejemplo1(branching)	29.68-(188.0)	84.26	76.66-(0.42)	32	48	48
EJemplo2(pathdegree)	-	-	-	-	-	-
Ejemplo3(path)	-	-	-	-	-	-
Ejemplo4(path-heuristic)	-	-	-	-	-	-
Ejemplo5(path-cost)	-	-	-	-	-	-
Ejemplo6(subgraph)	-	56.50-(0.15)	60.10-(0.29)	-	40	40
Ejemplo7(symmetric	-	640.81-(10.2)	603.55-(1.98)	-	48	48
Ejemplo8(complement)	20.77-(0.10)	28.49-(0.21)	24.28-(0.27)	47	35	43
Ejemplo9(undirected)	-	0.0075-(0.00433)	0.01-(0.00433)	-	18	18
Ejemplo11(combinadas)	-	0.18-(0.01)	0.17-(0.01)	-	27	27
Ejemplo12(combinadas)	-	-	-	-	-	-
Ejemplo13(combinadas)	-	-	-	-	-	-
Ejemplo14(combinadas)	-	-	-	-	-	-
Ejemplo15(combinadas)	-	-	-	-	-	-
Ejemplo17(combinadas)	-	17.77-0.61	14.53-(0.15)	-	39	39
Ejemplo18(combinadas)	-	105.74-(2.65)	79.61-(1.01)	-	56	56
Ejemplo19(combinadas)	-	475.73-(23.48)	493.43-(12.27)	-	48	48
Ejemplo20(combinadas)	-	33.16-(1.03)	28.21-(0.15)	-	55	55
Ejemplo21(combinadas)	-	-	-	-	-	-
Ejemplo22(complement)	0.12(0.0)	0.17-(0.007)	0.15-(0.0)	30	27	27
Ejemplo23(complement)	0.28-(0.0)	0.48(0.0043)	0.45-(0.0043)	34	23	23
Ejemplo25(complement)	14.89-(0.03)	20.22-(0.13)	18.21-(0.04)	41	35	35
Ejemplo26(combinada)	-	-	-	-	-	-

Cuadro 5.2: Resultados para la Vista NodeArcSetsGraphView

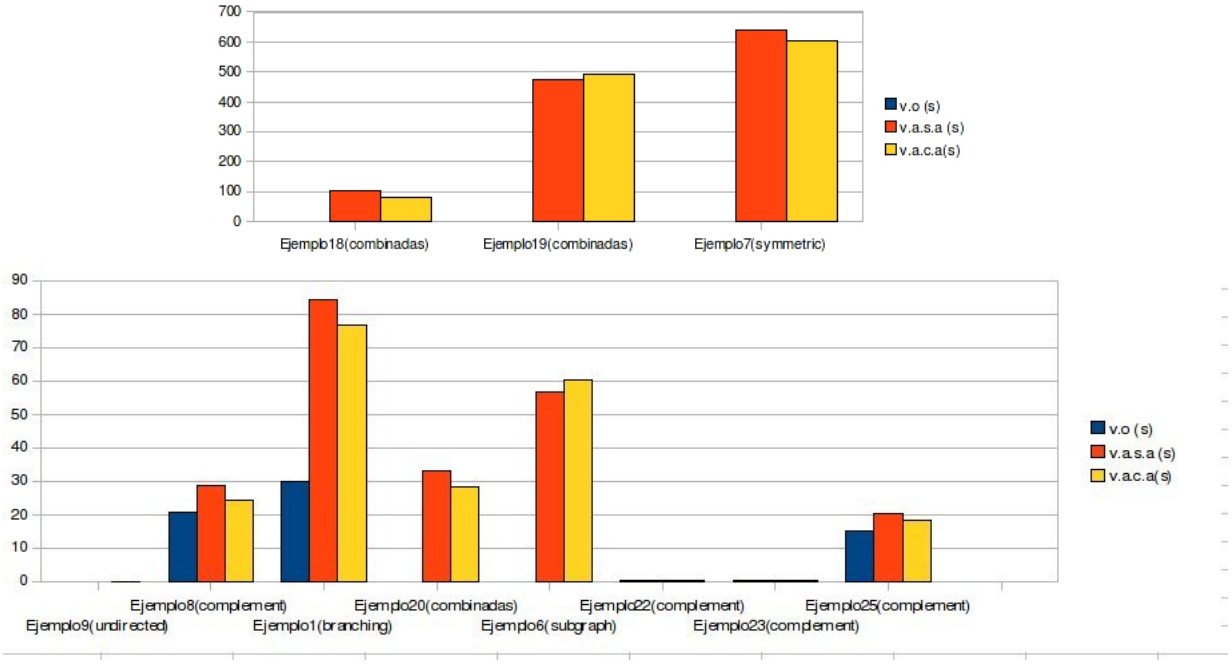

Figura 5.2: Resultados comparativos de los ejemplos usando NodeArcSetsGraphView

De las anteriores tablas y gráficas, se puede afirmar que los advisores afectaron negativamente los tiempos de los ejemplos corridos bajo la vista de *NodeArcSetsGraphView,* y mejoraron los tiempos de la vista *OutAdjSetsGraphView.* La explicación al resultado anterior se da en la Figura 13.

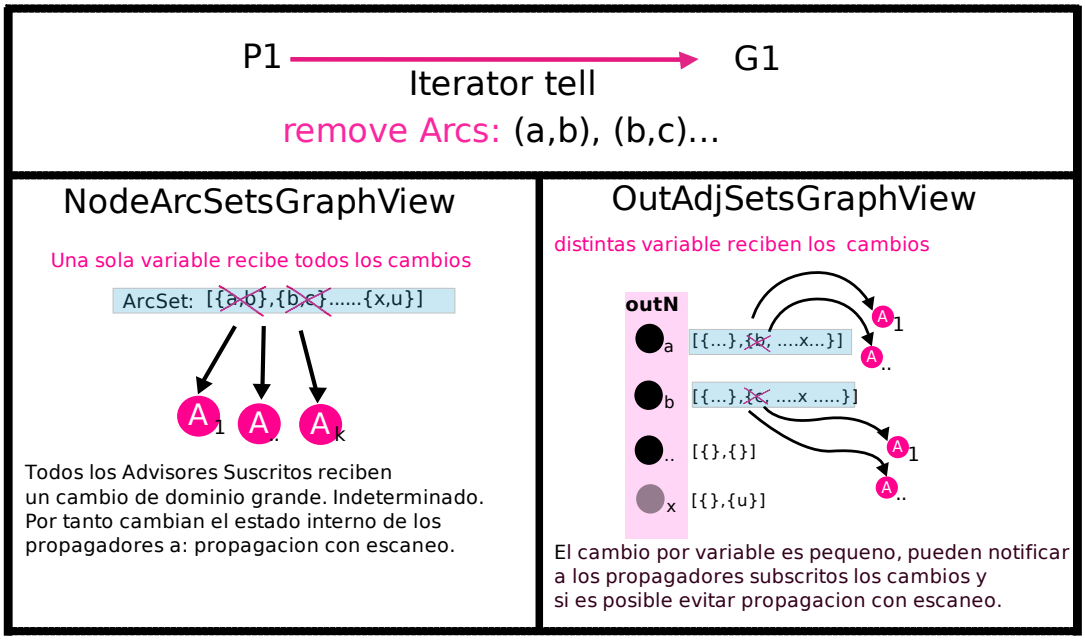

Figura 5.3: Ejecución de Advisores en las vistas

En NodeArcSetsGraphView una sola variable recibe todos los cambios haciendo para el propagador imposible determinar todos los cambios en el método advise y por tanto aconsejando una propagación con escaneo completo. En cambio en la vista OutAdjSetsGraphView dado que se compone de más

variables, el cambio se realiza sobre variables diferentes, siendo los cambios simples para cada variable sus advisores son capaces de aconsejar más adecuadamente a los propagadores subscritos, por tanto teniendo un mejor desempeño en tiempo.

Con respecto a la memoria, la versión original presenta en la mayoría de los casos un menor uso de memoria que la versión modificada sin Advisores y con Advisores.

La versión modificada con Advisores presenta un uso mayor de la memoria, esto se puede explicar, debido a los nuevas instancias de Advisores que deben estar subscritos a los propagadores, también por las variables agregadas a los propagadores que usan advisores y que se encargan de mantener el estado interno del propagador.

La versión modificada sin Advisores presenta un uso mayor en memoria lo cual podría estar explicado por la introducción de nuevos objetos que deben ser mantenidos en memoria durante la ejecución de un problema usando Gecode.Las estadísticas mostradas en [22]que comparan estadísticamente el uso de memoria y tiempo entre las diferentes versiones de Gecode, muestra claramente que la versión 2.1.1 usa más memoria que Gecode 1.0.

Descripción del cuadro 5.3

Este cuadro muestra los resultados de los ejemplos 27, 28, 29, 30, 31, 32 para ambas vistas, los ejemplos tienen una sola restriccion y grafos completos. El nombre del ejemplo indica la restriccion y el numero indica la cantidad de nodos que tienen los grafos. El factor de mejora abreviado con f.m, evalúa la mejora de CP(Graph) modificado con advisores frente a CP(Graph) original, en el caso de que el ejemplo no se hubiese podido correr en CP(Graph) original entonces la comparación realizada es entre CP(Graph) modificado con Advisores frente a CP(Graph) modificado sin Advisores.

	OutAdjSetsGraphView				NodeArcSetsGraphView			
Restriccion:nodos	v.o(s)	v.a.s.a(s)	v.a.c.a(s)	f.m	v.o(s)	v.a.s.a(s)	v.a.c.a(s)	f.m
Subgraph:2	-	0.005-0.005	0.005-0.005	1	-	0.01-0.0	0.01-0.005	1
Subgraph:3	-	0.66-0.00433	0.36-0.005	1.83	-	2.3-0.0707	2.42-0.01	0.95
Complement:3	0.02	0.03-0.00433	0.01-0.00433	2	0.05	0.07 -0.005	0.07-0.00433	0.71
Complement:4	2.62	3.77-0.01	1.51-0.005	1.73	9.57	12.59-0.05	12.21-0.03	0.78
Symmetric:3	-	0.0025-0.00433	0.0025-0.00433	1	-	0.01-0.0	0.01-0.00707	1
Symmetric:4	-	0.06-0.00433	0.02-0.0	3	-	0.18-0.00433	0.18-0.005	1

Cuadro 5.3: Resultados en Tiempo para Restricciones Basadas en Propagadores con Advisores

Restriccion:nodos	v.o(KB)	v.a.s.a(KB)	v.a.c.a(KB)	f.m	v.o(KB)	v.a.s.a(KB)	v.a.c.a(KB)	f.m
Subgraph:2	-	18	18	1	-	18	18	1
Subgraph:3	-	31	31	1	-	31	31	1
Complement:3	-	22	22	1	-	27	27	1
Complement:4	-	35	35	1	-	35	35	1
Symmetric:3	-	18	18	1	-	22	22	1
Symmetric:4	-	22	22	1	-	27	27	1

Cuadro 5.4: Resultados en Espacio en Memoria para Restricciones Basadas en Propagadores con Advisores

	OutAdjSetsGraphView			NodeArcSetsGraphView			
Restriccion:nodos	v.o(s)	v.a.s.a(s)	v.a.c.a(s)	f.m	v.o(s)	v.a.s.a(s)	v.a.c.a(s)	f.m
Knight:8	-	0.15-0.00433	0.15-0.00433	1	-	-	-	-
Knight:10	-	0.46-0.00829	0.45-0.00829	1	-	-	-	-
Knight:12	-	0.95-0.01	0.95-0.00707	1		-	-	-

Cuadro 5.5: Resultados en Tiempo para El Problema del Recorrido del Caballo

	OutAdjSetsGraphView				NodeArcSetsGraphView			
Restriccion:nodos	v.o(KB)	v.a.s.a(KB)	v.a.c.a(KB)	f.m	v.o(KB)	v.a.s.a(KB)	v.a.c.a(KB)	f.m
Knight:8	-	1028	1028	1	-	-	-	-
Knight:10	-	2183	2183	1	-	-	-	-
Knight:12	-	3908	3908	1	-	-	-	-

Cuadro 5.6: Resultados en Espacio en Memoria para El Problema del Recorrido del Caballo

Como lo muestra la Tabla 3, los propagadores con Advisores y la vista OutAdjSetsGraphView pueden mejorar notablemente los tiempos. Para ejemplos como el ejemplo26(muchas restricciones) puede no haber necesariamente mejora, esto es debido a que probablemente el problema obtiene una solución durante la primera propagación completa. Por otro lado cuando el ejemplo es lo suficientemente grande y requiere de varios branchings y propagaciones los advisores pueden mejorar notablemente los tiempos(Tabla 3).

La vista NodeArcSetsGraphView no mejora los tiempos con el uso de advisores, incluso en la mayoría de los casos el tiempo de solución del problema se ve afectado aumentando notablemente.

Los propagadores de Complemento, Subgraph y Symmetric pudieron mejorar notablemente sus tiempos mediante el uso de advisores y la vista OutAdjSetsGraphView con respecto a la versión Original y la versión Sin Advisores. Como lo muestra la tabla 3, esta mejora en tiempo Aumenta con el tamaño del problema.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

- Se Actualizo satisfactoriamente Cp(Graph) de Gecode 1 a Gecode 2.1.1. Durante la migración de versión se incluyeron restricciones no incluidas ni soportadas con anterioridad en la versión oficial, pero soportadas teóricamente en la tesis de Dooms. Estas restricciones son: Subgraph, Symmetric y Undirected.
- Se llevo acabo el diseño e implementaron de Advisores para algunos de los propagadores. La experimentación realizada permitió introducir la propagación incremental para alguno de los propagadores, lo que permitió a su vez la reducción de tiempos de ejecución para algunos casos. Sin embargo el uso de Advisores implica necesariamente el uso de más memoria para la solución de los problemas.
- Los Advisores permitieron la reducción de tiempos en la vista OutAdjSetsGraphView, pero ocurrió lo contrario en la vista NodeArcSetsGraphView.
- La implementacion de Advisores se hizo con un diseño Orientado a Objetos para evitar encapsular la forma de subscripción de las vistas a los advisores, de esta forma a los propagadores les es transparente la forma como las vistas realizan la subscripción.
- Se evaluaron los resultados obtenidos sobre ejemplos previamente disponibles y también sobre variedades de ejemplos creados. Se realizaron pruebas de Correctitud. Pero también se compararon resultados en tiempo, memoria de tres versiones: Cp(Graph) original, Cp(Graph) con Advisores, Cp(Graph) Sin advisores.
- Encontrar soluciones en el dominio de los grafos toma mucho tiempo. Un cambio muy pequeño puede ocasionar que tome tiempos muy prolongados el encontrar una solución, la implementacion de Advisores puede ayudar a reducir notablemente estos tiempos, pero no solo es posible con la implementacion de Advisores si no, con un replanteamiento de la representación de las variables de grafos.
- Se utilizaron y adecuaron herramientas como el autconf y automake para asegurar la instalación y deployment de CP(Graph) en diferentes plataformas.
- Al llevar Cp(Graph) a Gecode 2.1.1 con Advisores se redujeron en algunos casos notablemente los tiempos, sin embargo se necesita más experimentación, conocimiento y mejoras a nivel de diseño para mejorar aun más los tiempos.

6.2. Trabajo Futuro

En esta sección se plantean posibles mejoras que pueden ser introducidas en el futuro a CP(Graph)

Propagadores

A pesar de que existe un soporte teórico para muchas restricciones, muchas de ellas aun necesitan ser implementadas en Cp(Graph), actualmente la diversidad de restricciones es limitada.

Advisores

La implementacion de más Advisores para los propagadores existentes, además de GraphDeltas. La inclusión de GraphDeltas ayudaría a tener un modelamiento más limpio durante la implementacion de nuevos Advisores.

La identificación de cuando no son necesarios los escaneos completos puede mejorar notablemente los tiempos de los propagadores.

Copia Perezosa

Durante la clonacion de variables de Grafos de un espacio a otro, si la variable tiene un conjunto de aristas y nodos muy grande, la copia es muy costosa. Para estos casos es mejor tener una estructura adicional que describa los cambios que ocurrieron entre un espacio y otro. Cuando la estructura adicional ya tenga un tamaño determinado, entonces si se realiza la copia de la variable de grafos. (Propuesta e Idea por Gustavo Gutierrez)

Bibliografía

[1] Gregoire Dooms.The CP(Graph) Computation Domain in Constraint Programming 2005-2006. Tesis(Doctorado en Ciencias Aplicadas). Universite catholique de Louvain. www.info.ucl.ac.be/~dooms/thesis.pdf

[2] Krzysztof R. Apt. Principles of constraint programming. CWI, Amsterdam, The Netherlands.Cambridge University Press, 2003. ISBN 978-0-521-82583-2.

[3] F. Rossi, P.Van Beek, T.Walsh. Handbook of constraint programming. Elsevier, 2006. ISBN 978-0-444-52726-4.

[4] Programming Constraint Services.

[5] http://eisc.univalle.edu.co/materias/Matematicas_Discretas_2/pdf/introd_grafos_01.pdf

[6] José Juan Carreño Carreño, Introducción de Grafos. Universidad politecnica de madrid. www.eui.upm.es/~jjcc/md200506personal/material/Introduccion_Grafos.pdf

[7] http://ktiml.mff.cuni.cz/~bartak/constraints/intro.html

[8] Luis Quesada, Solving Constrained Graph Problems using Reachability Constraints based on Transitive Closure and Dominators, 2006.Tesis(Doctorado en Ciencias Aplicadas).Université catholique de Louvain. http://edoc.bib.ucl.ac.be:81/ETD-db/collection/available/BelnUcetd-11272006-164211/unrestricted/thesis.pdf

[9] Equipo de Gecode,Benchmark de Gecode con respecto a otras alternativas Similares. http://www.gecode.org/benchmarks.html

[10] Sven Lämmermann,Christian Schulte,Thomas Sjöland.Constraints. Royal Institute of Technology Stockholm, Sweden. http://web.it.kth.se/~cschulte/talks/Constraints@KTH.pdf

[11] Cambios de Gecode 2.0.0 con respecto a Gecode 1.3.1 http://www.gecode.org/gecode-doc-latest/PageChanges_2_0_0.html

[12] Equipo de Gecode,Sitio Web Oficial de Gecode. http://www.gecode.org

[13] http://en.wikipedia.org/wiki/Kneser_graph.

[14] Tack G,Constraint Propagation -Models , Techniques , Implementation Saarland University , Germany ,2009

[15] CP(Graph): Introducing a Graph Computation Domain in Constraint Programming ,Gregoire Dooms, Yves Deville, Pierre Dupont,Department of Computing Science and Engineering Université catholique de Louvain .

[16] Gecode, An Open Constraint Solving Library,Guido Tack, Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP).

[17] Gecode 2.1.1 Components, Raphael Reischuk , Marzo , 2008.

[18] Space::status, Raphael Reischuk, http://www.ps.uni-sb.de/~raphael/bachelor/downloads/gecode-space_status.pdf.

[19] Techniques for Efficient Constraint Propagation - Mikael Lagerkvist , Stockholm Sweden 2008.

[20] Pagina Oficial de CP(Graph) - http://cpgraph.info.ucl.ac.be/

[21] CP(Graph) Version No Oficial - http://www.cs.brown.edu/~gdooms/Software.html

[22] Gecodes ChangeLog - http://www.gecode.org/changes.html.

[24] Gecode's Architecture, Raphael Reischuk,Graduate Seminar PS Lab,Marzo, 2008,http://www.ps.uni-sb.de/~raphael/bachelor/downloads/talk2-80313.pdf.

[23] http://foldoc.org/

[25] Didier Croes, http://www.bigre.ulb.ac.be/Users/didier/pathfinding/images/badconnection.png.