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

ARCO-CONSISTENCIA ADAPTATIVA PARA CSPs

CAMILA FERNANDA CISTERNAS PERINES
LEONARDO ESTEBAN ZAMORANO PINO

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

DICIEMBRE 2013

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

ARCO-CONSISTENCIA ADAPTATIVA PARA CSPs

CAMILA FERNANDA CISTERNAS PERINES
LEONARDO ESTEBAN ZAMORANO PINO

Profesor Guía: Ricardo Soto de Giorgis.
Profesor Co-referente: Wenceslao Palma Muñoz.

Carrera: Ingeniería de Ejecución en Informática.

DICIEMBRE 2013

Queremos agradecer a nuestros profesores, amigos y familiares, quienes siempre nos acompañaron, apoyaron y alegraron en todo momento, gracias por formar parte de esto.

Resumen

Un *Problema de Satisfacción de Restricciones* o *CSP* se representa por medio de variables, dominios y restricciones. La forma en que se encuentra una solución para un CSP, depende entre otros de las *Técnicas de Consistencia* que se utilicen. Las técnicas de consistencia ayudan a reducir la cantidad de nodos que se instancian en el proceso de búsqueda. La técnica utilizada en el presente proyecto es la *Arco-Consistencia*, la cual se aplicará a restricciones binarias. Existe una amplia gama de algoritmos de arco-consistencia, pero esta investigación se centrará en los algoritmos AC-1 y SAC-1. La idea es intercalar ambos algoritmos bajo un esquema de *Arco-Consistencia Adaptativa*, en el cual se aplicará la técnica más apropiada en las diferentes partes del proceso en base a indicadores de rendimiento..

Palabras Claves: Problemas de Satisfacción, Técnicas de Consistencia, Arco-Consistencia, Arco-Consistencia Adaptativa.

Abstract

A *Constraint Satisfaction Problem* or *CSP* is represented by means of variables, domains, and constraints. The way in which solutions are found for a CSP, depends among others on the *Consistency Techniques* employed. The consistency techniques help to reduce the amount of nodes that are instantiated in the search process. The consistency technique used in this thesis is *Arc-Consistency*, which is applied to binary constraints. There exist several arc-consistency algorithms, but this work focuses on the AC-1 and SAC-1 algorithms. The idea is to interleave both algorithms under an *Adaptive Arc-Consistency* schema, in which the more appropriate technique will be applied on the different parts of the process based on performance indicators.

Keywords: Satisfaction Problems, Consistency Techniques, Arc-Consistency, Adaptive Arc-Consistency.

Índice

Lista de Figuras

1 Introducción

En el mundo de la informática la *Programación con Restricciones* (en inglés *Constraint Programming*, CP) es considerada como un paradigma, que se utiliza para dar solución a problemas mediante el uso de restricciones. Existen problemas propiamente tales que se presentan recurrentemente en el ámbito ingenieril, se hace referencia a los *Problemas de Satisfacción de Restricciones*, los que se definen como problemas matemáticos capaces de llegar a una o más soluciones mediante el cumplimientos de restricciones. Esta clase de problemas se utilizan en demasía en áreas como la Investigación de Operaciones, Inteligencia Artificial, en los Lenguajes de Programación y en muchas más.

Para encontrar una solución a un problema de satisfacción con restricciones, se debe emplear algún tipo de algoritmo que permita llegar a dicha solución, siendo los Algoritmos de Búsqueda los más utilizados, estos se encargan de recorrer todas las posibles alternativas que se generan a partir de un problema (árbol de búsqueda). En el presente informe se abordarán cuatro algoritmos, estos son: Generate and Test, BackTracking, Forward Checking y Mantenimiento de Arco-Consistencia, con distintos enfoques, cada algoritmo pretende llegar a la o las soluciones deseadas de la forma más eficiente posible.

Como complemento a los algoritmos de búsqueda, existen técnicas que ayudan a que el árbol de búsqueda se pueda recorrer de una forma mucho más acotada, a lo que se hace referencia son las Técnicas de Consistencia, que a grandes rasgos se encargan de ver aquellos valores que presenten inconsistencias en los dominios, vale decir, que no permitan llegar a una solución, o que no sean parte de ella, con esto se simplifica el trabajo de los algoritmos de búsqueda, puesto que se reduce el árbol.

La Arco-Consistencia ha sido un tema muy estudiado en lo que se refiere a las técnicas de consistencia, la que tiene como función restringir el dominio de dos variable asociadas mediante las restricciones impuestas, existen variados procedimientos de arco-consistencia, pero se abordarán solo dos algoritmos en esta investigación, AC-1 y SAC-1, donde se especificará el funcionamiento y las características principales de cada uno.

Teniendo claro todo lo mencionado con anterioridad, se creó un algoritmo de arco consistencia adaptativa, lo que a grandes rasgos es la utilización de AC-1 y SAC-1 en forma alternada para lograr llegar a una o más soluciones en un problema determinado. Lo que permitirá la correcta utilización de esta técnica, serán los distintos indicadores que evaluarán el trabajo de AC-1 y SAC-1 en el árbol de búsqueda, permitiendo que se utilice el más adecuado según sea su rendimiento.

Finalmente se explica la implementación y funcionamiento del solver que se utiliza (RSSolver), mencionando los pasos que debe seguir el modelo para lograr una satisfactoria compilación y posteriormente una exitosa búsqueda de la solución de dicho modelo. Para lograr encontrar la solución, se implementó en el RSSolver el algoritmo AC-1 y SAC-1 junto al algoritmo de búsqueda BackTracking. Ambos algoritmos son utilizados en primera instancia de forma separada para ver su rendimiento individual y luego se utilizan de forma alternada para así ver cómo es su rendimiento en conjunto, el que se podrá apreciar con las diferentes pruebas que se desarrollaron.

2 Definición de Objetivos

2.1 Objetivos Generales

- Implementar arco-consistencia adaptativa para problemas de satisfacción de restricciones, en base a los algoritmos AC-1 y SAC-1.

2.2 Objetivos Específicos

- Comprender los conceptos básicos de la programación con restricciones.
- Implementar algoritmos AC-1 y SAC-1.
- Implementar un sistema de arco-consistencia adaptativa mediante AC-1 y SAC-1.
- Realizar experimentos con la implementación desarrollada.

3 Bases de la Programación con Restricciones

La programación con restricciones es una tecnología que tiene sus orígenes en los años sesenta y su principal objetivo es encontrar soluciones a problemas complejos mediante restricciones, dichas soluciones deben respetar todas las restricciones declaradas. En sus inicios CP era utilizada en el área de la inteligencia artificial, desde allí siguió desarrollándose durante veinte años, y es en los años ochenta cuando nace el primer lenguaje de programación basado en restricciones, el cual fue hecho en LISP, sin embargo, a medida que transcurre el tiempo CP se ha aplicado a una variedad de áreas como la resolución de problemas combinatorios, investigación operativa, sistemas de recuperación de la información, entre otros. A finales de las décadas 80' y 90' nacen dos nuevos conceptos, *Programación Lógica por Restricciones* (CLP) y *Programación Concurrente por Restricciones* (CCP) respectivamente, CLP se refiere al uso de CP como extensión de la programación lógica, donde se mezclan restricciones y métodos para solucionar un problema mediante un lenguaje basado en lógica [7], aunque en sus principios tenían la desventaja de ser lenguajes poco expresivos pero en la actualidad tal desventaja no existe, ya que los leguajes están enfocados para un propósito mucho más generalizado.[6]

La programación con restricciones se puede dividir en dos grandes ramas, las que comparten ciertas similitudes en la terminología utilizada, sin embargo, difieren en las técnicas que emplean para la resolución del problema, estas ramas son: Resolución de Restricciones y Satisfacción de Restricciones. La primera trata con problemas de dominios infinitos, su objetivo es encontrar una solución óptima a través de una función objetivo, ya sea maximizando o minimizando [3]; Por otro lado la Satisfacción de Restricciones se enfoca básicamente a dilemas que presentan restricciones con un dominio finito, en el cual las variables pueden tomar valores de ese dominio cumpliendo con la condición de que no se violen las restricciones del problema.

Un típico problema en el que se puede ocupar CP para llegar a una solución, es el de "N-reinas", clásico ejemplo el cual busca variables que cumplan con la restricción de que las reinas, independientemente del tamaño del tablero, no sean una amenaza entre ellas, es decir, si una reina está posicionada en la ubicación A_{ij}, no pueden haber mas reinas tanto horizontal, vertical o diagonalmente, si dos reinas coinciden en cualquiera de estas direcciones, no sería una solución factible, ya que no cumple con las restricciones mencionadas.

¿Cómo se programa con restricciones? Hay tres palabras claves para aclarar esta incógnita: *Modelar*, *Imponer* y *Explorar*. Agruparemos las dos primeras, en una primera fase donde se lleva a cabo el modelado del dominio ya sea con restricciones básicas o restricciones establecidas por el usuario, imponiendo las reglas que acotarán los posibles valores de las variables en un espacio definido. El tercer concepto, explorar, utiliza algoritmos que buscan soluciones o asignaciones de valores a variables que sean factibles tomando siempre en

consideración cada una de las restricciones; A todos estos problemas que cumplan con lo anterior y se resuelvan a través de CP se les denomina CLP.

3.1 Problema de Satisfacción de Restricciones

A todos los problemas modelados, de tal forma que contengan variables que pueden tomar valores de un dominio finito, para así encontrar una solución que sea consistente con cada una de las restricciones de dicho problema, se denominan *Problemas de Satisfacción de restricciones* (CSP). Una infinidad de problemas pueden ser modelados de esta forma, variados ámbitos como la investigación operativa, la recuperación de información, o incluso problemas más cotidianos como planificar un viaje o preparar un plato de comida están sujetos a restricciones por lo que se les podría modelar como CSP, sin embargo, hay ciertas soluciones que, aunque cumplan con el conjunto de restricciones, son más convenientes en algunas ocasiones y no tan conveniente en otras, por lo que se hace necesario tener más de una solución para lograr obtener una solución óptima.

La resolución de un CSP es un proceso divido en dos subprocesos fundamentales para la obtención de una solución. Para describir la fase inicial se ocupa el término *modelado*, la modelación es un paso importantísimo a la hora de realizar el proceso de resolución, ya que se ordena el problema estructurándolo de tal manera que sea compatible en la sintaxis de CSP, en otras palabras, expresa el problema como variables, dominio y restricciones. Luego de terminada la etapa de modelado, se avanza a la fase de resolución, en donde se toman variables, dominios y restricciones anteriormente definidas y se procesan utilizando algoritmos de búsqueda o técnicas de consistencia para encontrar una o múltiples soluciones, que respeten todas las restricciones, las que no necesariamente son óptimas. La acción en conjunto de los algoritmos de búsqueda con las técnicas de consistencia permite reducir el espacio de solución y explorar solo el espacio resultante, con el objetivo de obtener una solución al problema de forma más rápida. [4].

En cada una de las fases sobresalen términos que no se puede dejar de definir y serán detallados a continuación:

Asignación: también llamada instanciación, son valores del dominio que se le conceden a una variable, por ejemplo: (*x*,*a*) representa que el valor de *x* es asignado a la variable *a*, es decir , la variable *a* es instanciada con el valor *x*.

Solución: concepto muy utilizado en la presente investigación, es una asignación de valores a todas las variables de forma que satisfaga todas las restricciones.

3.1.1 Definición de un Problema de Satisfacción de Restricciones

En términos generales, un problema de satisfacción de restricciones se representa como una terna *(V, D, R)* donde:

- *V* es un conjunto de variables $v_1,\ v_2,\ v_3, \dots ,\ v_n$.
- *D* es un conjunto de dominios $d_1, d_2, \dots , d_n$. Donde d_i corresponde a los valores posibles que puede tomar la variable v_i, con $i = 1, \dots, n$.
- *C* es el conjunto finito de restricciones $c_1, c_2, \dots, c_m$ tal que c_j, donde $j = 1, \dots, m$, es la relación sobre el conjunto de variables $\{ x_1, x_2, \dots, x_n\}$. Una restricción es binaria si se relacionan únicamente dos variables como $x_i\ y\ x_j$ y se denotan como $c(x_i, x_j)$. *C* es la encargada de restringir los valores que pueden tomar las variables *V* [5].

3.1.2 Modelado de un Problema de Satisfacción con Restricciones

Existen diferentes formas de representar un problema de satisfacción de restricciones, incluso las restricciones pueden ser representadas ya sea como ecuaciones o como tuplas válidas y no válidas para los problemas [4]. En esta ocasión, se modela un ejemplo de CSP muy conocido, llamado N-reinas para plasmar el modelado de CSP.

3.1.2.1 N–reinas

La idea principal del problema de las N-reinas consiste en ubicar un tablero de ajedrez de con *n* unidades de largo y *n* unidades de ancho, y que a la vez posea igual número de reinas, en este caso *n.* La problemática radica en que una reina no debe amenazar a las restantes reinas dentro del tablero, en el juego de ajedrez la reina amenaza a todas las figuras que están en la misma fila, columna o diagonal, por lo que dos o más reinas no pueden coincidir horizontal, vertical, ni diagonalmente, tal como se muestra en la figura 1. En el presente ejemplo se ocupará un tablero de 4x4, por lo tanto el número de reinas será 4.

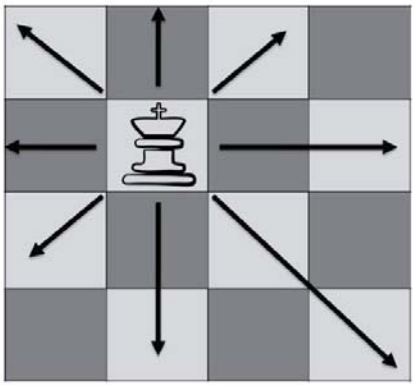

Figura 1 N- reinas.

El modelo sería el siguiente, definiendo en primer lugar las variables correspondientes a cada fila del tablero y el dominio del CSP, en este caso a las columnas del tablero.

$$R_1, R_2, R_3, R_4 \in [1,4].$$

Cada R_i tiene asociado un valor j perteneciente al dominio, es decir, una reina R se encuentra ubicada en la intersección de la fila *i* con la columna *j*.

Luego se definen las restricciones del CSP

- Las reinas deben estar en distintas filas:

$$(\forall\ i \in [1,3] y\ j \in [i+1,4])\ /\ R_i \neq R_j.$$

- Las reinas no deben colocarse en diagonal:

$$(\forall\ i \in [1,3] y\ j \in [i+1,4])\ /\ R_i + i \neq R_j + j \quad \text{y} \quad R_i - i \neq R_j - j.$$

Ya definidas las variables, los dominios y las restricciones del problema, simplemente se ordenan con la sintaxis adecuada y se procede a finalizar la etapa de modelado, el resultado de esta fase es el siguiente:

Variables: R = {R1, R2, R3, R4}, una reina ubicada en cada fila.

Dominios: D_i= {1, 2, 3, 4}, un valor por cada columna.

Restricciones:

$$(\forall\ i \in [1,3] y\ j \in [i+1,4])\ /\ R_i \quad \neq R_j\ , \quad R_i + i \neq R_j + j \quad \text{y} \quad R_i - i \neq R_j - j.$$

Ya terminada la etapa da modelado se procede a resolver el problema, al unir todos los elementos anteriores se llega a posibles soluciones del CSP de N-reinas observables en la siguiente ilustración, en ambas se respetan todas las restricciones ya que ninguna reina puede amenazar a otra.

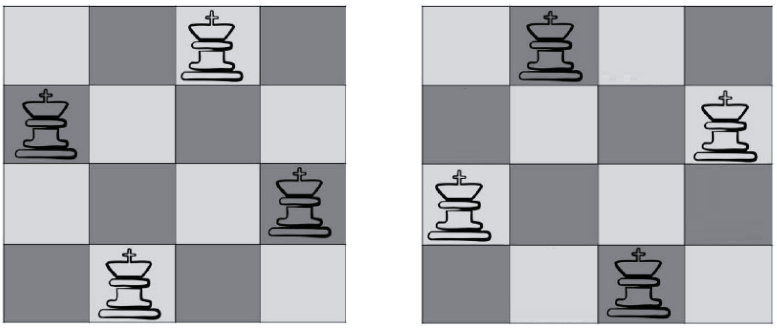

Figura 2 Soluciones N- reinas.

3.2 Algoritmos de Búsqueda

Hay algoritmos especializados para encontrar soluciones a CSPs, los llamados *Algoritmos de Búsqueda* recorren posibles soluciones en un conjunto de posibles instancias de variables, sin embargo, ocupan distintas metodologías para lograr encontrar soluciones que no infrinjan ninguna de las restricciones, esto se verá en detalle más adelante.

Todas las instanciaciones de variables y posibles soluciones generan un espacio de estados, se pueden representar en forma de árbol, es en este árbol donde los algoritmos de búsqueda realizan sus recorridos en busca de una solución [3].

Existen dos grandes tipos de algoritmos para realizar búsquedas[8], estos son, los *completos* e *incompletos*, la principal diferencia entre ambos radica en el área de búsqueda de soluciones; A los algoritmos incompletos también se les llama locales ya que realizan la búsqueda solo en una sección del espacio de estados por lo que no garantiza una solución, y mucho menos una solución óptima, sin embargo, tienen la ventaja de reducir costos al recorrer menos espacio que las búsquedas completas y al mismo tiempo son más veloces, su funcionamiento parte desde una solución inicial, la que itera varias veces dirigiéndose cada vez a una solución distinta, tratando de mejorar el valor de la función objetivo. El segundo grupo son los algoritmos completos, los cuales recorren la totalidad del espacio buscando una solución, mediante la asignación de valores a las variables en un árbol que representa cada una de las asignaciones realizables, el nodo raíz de dicho árbol representa el problema sin variables asignadas.

Con el objetivo de resolver CSP se han desarrollado una gran cantidad de algoritmos de búsqueda completa los que trabajan realizando búsquedas en la totalidad del espacio de estados siendo algunos de ellos más eficientes que otros.

Para simplificar la explicación de los algoritmos de búsqueda, se utilizará el mismo ejemplo que anteriormente se usó en CSP, de esta manera se hace más fácil comprender el funcionamiento de los diferentes algoritmos que buscan las posibles soluciones.

3.2.1 Generate and Test (GT)

Es la técnica de búsqueda más simple, ya que instancia cada uno de los posibles valores a las variables y recorren sistemáticamente todo el árbol de búsqueda, su nombre deriva de las dos principales acciones que lleva acabo, Generar la instanciación de una variable a la vez, hasta llegar al nodo hoja, luego procede a comprobar si estas cumplen con las restricciones planteadas en el problema. Este método tiene la desventaja de ser muy costoso y lento al momento de buscar una solución, ya que genera asignaciones que no cumplen con las restricciones lo que se traduce en pérdida de tiempo y costo.

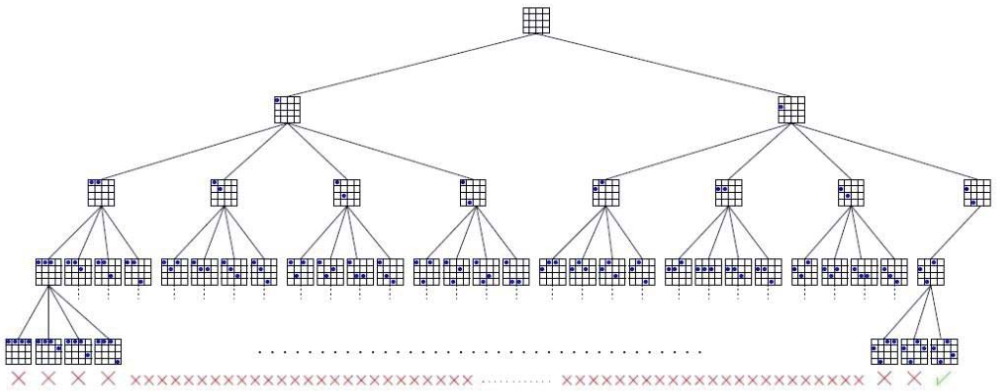

Figura 3 Generate and Test en el problema de 4-reinas.

Aplicando GT en el problema de las N-reinas, se concibe una gran cantidad de instanciaciones innecesaria, ya que cuando se llega al término de una rama (nodo hoja), como se muestra en la Figura 3, teniendo todas las variables instanciadas comprueba si se satisfacen todas las restricciones, Demostrando la ineficiencia de este método.

3.2.2 Backtracking (BT)

Otro algoritmo de búsqueda de CSP es el conocido Bakctracking, su forma de trabajo es similar a Generate and Test, pero con la gran diferencia que logra eliminar la ineficiencia de GT. Cabe mencionar que el algoritmo de Backtracking asume dos cosas: por un lado, que el CSP es binario, y por otro lado asume que tanto las variables como sus dominios son estáticos. Este algoritmo abarca dos procesos, el primero es hacia adelante en el cual se selecciona la siguiente variable y se instancia con el próximo valor, comprobando que no se entre en conflicto con las restricciones del problema, en caso de que eso ocurra, se realiza el segundo proceso (hacia atrás) donde el algoritmo deshace la asignación actual e intenta instanciar la variable con otro valor, si no se encuentra un valor que satisfaga la inconsistencia, entonces retrocede tomando en cuenta la variable anterior y evitando el recorrido innecesario de la rama que cuelga de la última asignación. Los dos procesos se repiten hasta completar el recorrido del árbol donde las variables toman valores consistentes con las restricciones planteadas.

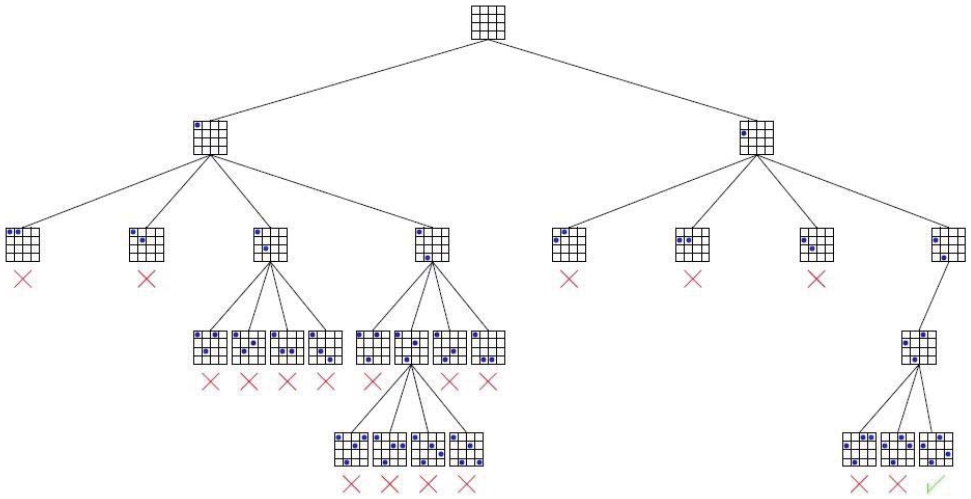

Figura 4 Backtracking el problema de 4-reinas.

3.2.3 Forward Checking (FC)

A medida que se avanza en la revisión de algoritmos de búsqueda se encuentran nuevos algoritmos cada vez más eficientes como el Forward Checking, el cual tiene un enfoque Look-Ahead, en otras palabras, va chequeando hacia adelante para obtener las inconsistencias de variables futuras involucradas, además de las variables actuales. Su forma de trabajo es básicamente instanciar una variable y luego chequear hacia adelante todos los valores futuros que producen alguna inconsistencia, estos valores son temporalmente eliminados del dominio y solo quedan los que si son potenciales solucionas al CSP. Si el dominio queda vacío, se instancia la variable con otro valor y si ningún valor logra ser consistente, entonces el algoritmos retrocede y deshace la instanciación volviendo a la anterior.

En el ejemplo de las N-reinas se instancia una variable R_{ij}, para este ejemplo se tomará *i=1* y *j=1*, eliminándose temporalmente del dominio los valores horizontales, verticales y diagonales, permitiendo que la próxima reina solo pueda tomar valores que no amenacen a la primera, en la figura 5, *i* toma el valor tres y *j* el valor dos, nuevamente se eliminan las variables conflictivas a futuro y la columna *j* queda totalmente eliminada, por lo que el dominio queda vacío y al no poder tomar otro valor retrocede a la instanciación anterior. El algoritmo continúa realizando los mismos procesos hasta encontrar una solución.

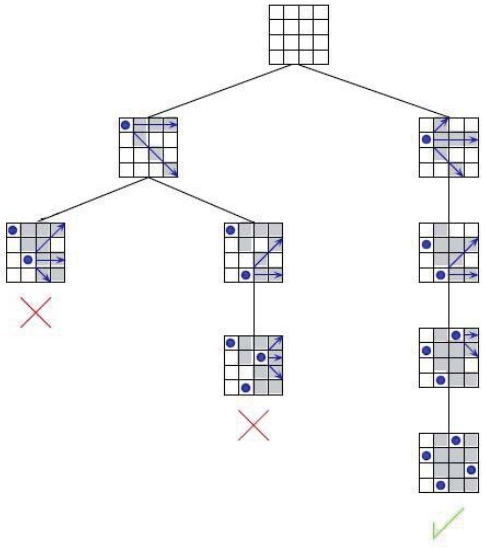

Figura 5 Forward Checking utilizado en el problema de N-reinas.

3.2.4 Mantenimiento de Arco-Consistencia (MAC)

Gracias a su rapidez y eficiencia el mantenimiento de arco-consistencia es de lo más utilizado en la actualidad para problemas de CSP. Esta técnica, en palabras simples, mezclan Algoritmos de Búsqueda con Arco-Consistencia.

La principal ventaja que presenta MAC, además de verificar inconsistencia futuras, es que también puede ver las de variables pasadas, con esto se pretende reducir el espacio de búsqueda más rápidamente que en forward checking. Cuando se intenta la asignación de una variable, MAC aplica arco-consistencia al sub-problema formado por todas las futuras variables. Si el dominio de alguna variable futura se queda vacío, la instanciación realizada a la variable actual se deshace, el valor probado se elimina del dominio de la variable actual y se prueba la arco-consistencia de nuevo [1]. Luego, como en FC se prueba instanciar la variable con otro valor y en el caso de que no queden valores en el dominio, se lleva a cabo el backtracking. El trabajo extra que MAC realiza al aplicar arco-consistencia puede eliminar más valores de las futuras variables y como consecuencia logra podar más el árbol de búsqueda que FC [1].

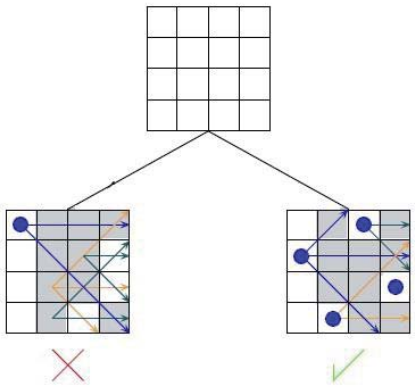

Figura 6 Utilización de MAC en el problema de 4-reinas.

3.3 Técnicas de Consistencia

Los algoritmos de búsqueda por si solos son insuficientes para la resolución óptima de un CSP, por lo que se hace necesaria la utilización de técnicas que permitan mejorar su funcionamiento a la hora de resolver un problema. Estas técnicas se utilizan como etapas de pre-proceso donde se detectan y se eliminan las inconsistencias locales, antes de empezar la búsqueda o durante el proceso de búsqueda en sí, con el fin de reducir los nodos a instanciar en el árbol de búsqueda [1].

Gracias a las técnicas de consistencia, se evita que se busque una solución que no existe, lo que permite ahorrar tiempo y que no se entorpezca el buen funcionamiento de los algoritmos. La continua aparición de valores individuales o conjunto de ellos que no pueden participar de ninguna solución, esto es lo que se conoce como inconsistencia local, es por esto que estas técnicas son tan necesarias, ya que eliminan valores inconsistentes de los dominios de las variables, con lo que se acota el universo de soluciones en un problema. Si bien estas técnicas ayudan en demasía a la resolución de un problema, no quiere decir que todo lo restante que queda en el árbol son soluciones, pero si se reduce la complejidad del problema.

3.3.1 Consistencia de Nodo

La consistencia de nodo o también conocida *nodo-consistencia* corresponde al nivel más básico de consistencia, donde se eliminan los valores inconsistentes con las restricciones unarias donde participa la variable [1]. Una variable *xi* es nodo-consistente si y sólo si todos los valores de su dominio *Di* son consistentes con las restricciones unarias sobre las variables. Un *CSP* es nodo-consistente si y sólo si todas sus variables son nodo-consistentes: $\forall X_i \in X, \forall R_i \in R : X_i$ satisface R_i. De lo contrario, si los valores contenidos en el dominio de la

variable *x* no pueden satisfacer las restricciones, se está frente a una inconsistencia, pudiéndose eliminar los valores del dominio de la variable *x*, ya que no serían parte de ninguna solución.

3.3.2 Consistencia de Arco

La *consistencia de arco* tiene como principal característica, el que trabaja con restricciones binarias en vez de unaria como lo hacía la nodo-consistencia [1]. Un *CSP* es arco-consistente si para todo par de dominios D_i y D_j, $y \neq z$, y una restricción R_n se cumple que:

$$\forall x_i \in D_y, \exists x_j \in D_z(x_i, x_j) \text{ Cumple restricción}$$

Si se cumple con lo antes mencionado, se podrá eliminar cualquier valor del dominio D_y de la variable x_i que no sea arco-consistente, esto porque no será parte de ninguna solución.

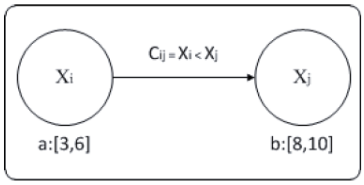

Figura 7 Ejemplo de consistencia de Arco.

Como se muestra en la figura 7, se tienen dos variables cada una con dominio propio; La restricción indica que $X_i < X_j$, en este caso, para cada variable a existe un b que cumple con la restricción por lo tanto es arco-consistente, sin embargo, si la restricción fuera $X_i = X_j$, el problema no sería arco consistente, ya que ningún *n* cumple con la restricción planteada.

4 Algoritmos de Arco Consistencia

Como ya se mencionó anteriormente, lo que se logra con las técnicas de arco-consistencia es el poder eliminar algunos valores que jamás formarán parte de una solución, es decir, restringir los dominios de dos variables asociadas a una restricción, esto último es lo que se conoce como restricciones binarias[1].

El proceso secuencial mediante el cual se realizan las técnicas de arco-consistencia, se llama *propagación* y el cómo los algoritmos desarrollan esta tarea, se denomina *granulidad*. Los algoritmos de arco-consistencia a utilizar en el presente estudio serán dos, AC-1 y SAC-1. A continuación se explicarán brevemente en que consisten ambos algoritmos, puesto que los dos serán de trascendental importancia para la investigación que se pretende.

4.1 Algoritmo AC-1

Algoritmo que tiene su origen en los años 70', y en su proceso principal contiene un sub-proceso llamado REVISAR, que también es ocupado en otros AC como AC-2 y AC-3, este sub-proceso se encarga de eliminar valores del Dominio *i*, en el caso de eliminar algún valor, este algoritmo devuelve *VERDADERO* y en caso contrario devuelve *FALSO* al algoritmo principal de AC-1, éste a su vez devuelve un nuevo CSP equivalente y consistente, que contiene los mismos dominios pero eliminando valores que no forman parte de una solución [8]. El siguiente algoritmo (Figura 8) es el detalle del sub-proceso REVISAR:

Algoritmo 1 Procedimiento REVISAR

```
1  inicio
2  change ← falso
3  para cada a ∈ D_i hacer
4      si no existe b ∈ D_i tal que (a, b) ∈ R_ij
5          D_i ← D_i − a
6          change ← verdadero
7      fin si
8  fin para
9  retorna change
10 fin inicio
```

Figura 8 Pseudocódigo Procedimiento REVISAR.

Como se muestra en el algoritmo anterior, se tienen los dominios D_i y D_j para dos variables v_i y v_j. Si existe un $x \in D_i$, y un valor y perteneciente al dominio D_j que viole la

restricción $R_{ij}\,(x, y)$, entonces se elimina x de D_i, cuando se recorre D_i por completo, entonces tenemos un arco (i, j) consistente en la dirección desde i a j, es decir:

$$\forall\,(x \in D_i\,)\;\exists\,\big(y \in D_j\big)\;/\;(a, b) \in R_{ij},$$

Entonces *(i,j)* es arco-consistente direccional (de *i* a *j*).

Se repite el proceso REVISAR cuantas veces sea necesario dentro del algoritmo principal, para que AC-1 devuelva arcos consistentes [4].

Cabe destacar que el AC-1 se aplica a restricciones binarias, ya que sin estas no sería posible formar los arcos que son la base de AC-1 [8]. En la figura 9 se muestra el algoritmo principal de AC-1 empleando del sub-proceso REVISAR, eliminando los valores inconsistentes o redundantes y luego retornando un CSP arco-consistente.

Algoritmo 2 Pseudocódigo AC-1

```
1  inicio
2  Q ← {(i, j)|(X_i, X_j) es un arco de (G), i ≠ j }
3  repetir
4       change ← falso
5            para cada (i, j) ∈ Q hacer
6            change ← REVISAR (i, j) o change
7  fin para
8  hasta no change
9  fin inicio
```

Figura 9 Pseudocódigo AC-1.

En cada repetición, se inicializa la variable CHANGE en *falso*, luego en cada arco se realiza el algoritmo REVISAR, si este eliminó algún valor de D_i, el valor de CHANGE cambia a *verdadero*, este paso se realiza hasta que se termina de recorrer *Q*.

4.2 Algoritmo SAC-1

Este algoritmo fue desarrollado y presentado en Japón el año 1997 [9] por Chiristian Bessiere y Romuald Debruyne. Singleton Arc Consistency (SAC) se caracteriza por ser un algoritmo denominado de "fuerza bruta", esto debido a que la forma en que trabaja es más minuciosa que por ejemplo AC-1. Por otro lado, SAC puede trabajar con diversos algoritmos de arco consistencia, para efectos de este estudio será junto al algoritmo AC-1, por lo que se denominará SAC-1. Como se mencionó con anterioridad, este es un algoritmo de fuerza bruta, esto debido a que SAC-1 además de ver si un problema es arco consistente, también verifica si es que cada valor de un dominio es consistente por sí solo, lo que permite una evaluación mucho más específica. A medida que se elimina un valor del dominio se debe ir actualizando dicho dominio.

Definición 1: *Para un problema del tipo* $P = (X, D, C)$ *se dice que es inconsistente si y sólo si* (P) *tiene dominios o restricciones vacías.*

Definición 2: *Para un problema del tipo* $P = (X, D, C)$ *es singleton arc consistent si y sólo si* $\forall i \in X, \forall a \in Di$, *la red* $P|_{i=a}$ *se obtiene reemplazando* D_i *por singleton* $\{a\}$ *y este no sea arco inconsistente, o tenga dominios o restricciones vacías.*

Para entender de forma más clara lo recién mencionado, se desarrollará un ejemplo en el cual se verá claramente cómo funciona SAC-1:

Se tienen dos dominios, D_w y D_z con sus variables y restricción:

variables:

x en [1,2];
y en [5,6];

restricción:

$x+y>6$;

Al evaluar el ejercicio con AC-1 se tiene lo siguiente,

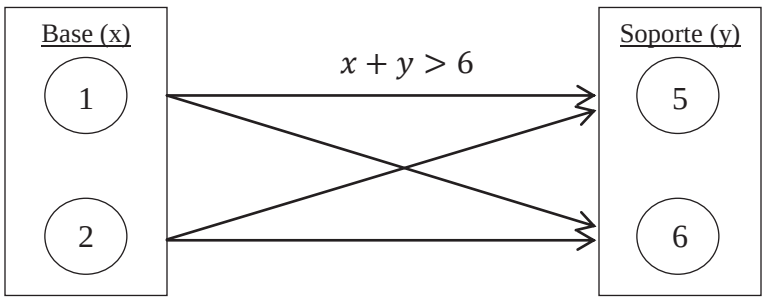

Figura 10 Modo de Trabajo AC-1.

Se puede apreciar en la figura 10 que al evaluar cada valor x_i del dominio D_w (base) con un valor y_i del dominio D_z (soporte), si se cumple la consistencia de arco, ya que al evaluar en la restricción $x + y > 6$ se tiene con la primera base que $1 + 6 > 6$ por lo que al menos un valor del soporte si permite cumplir la restricción. Luego con la segunda base ambos soportes satisfacen a la base 2. Siguiendo con la primera evaluación, ya se comprobó que el valor 1 de la $Base\ (x)$ tiene al menos un $Soporte\ (y)$ que cumple la restricción, por lo que también se debe ver si al implementar la *bidireccionalidad* sigue siendo consistente.

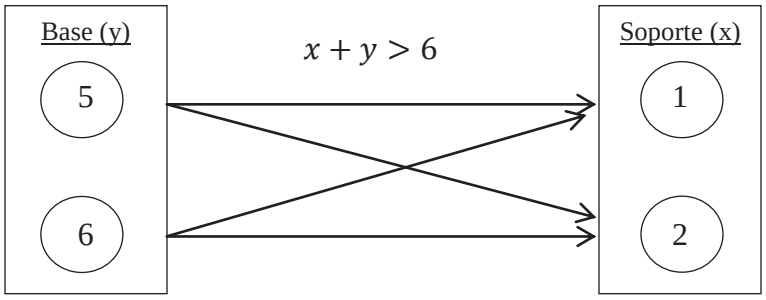

Figura 11 Modo de Trabajo Bidireccional AC-1.

En la figura 11 se mantiene la consistencia de arco, ya que al evaluar cada base con los soportes, si hay valores que satisfagan la restricción. Ahora bien, luego de ver cómo trabaja AC-1, se apreciará la forma en que realiza esto mismo el algoritmo SAC-1. En primer lugar se debe tratar cada valor de un dominio como un dominio aparte, es decir, dividiremos en sub problemas el ejercicio completo. Antes se utilizó el dominio D_w completo, ahora ese dominio se dividirá para ser tratado cada valor individualmente como un nuevo dominio.

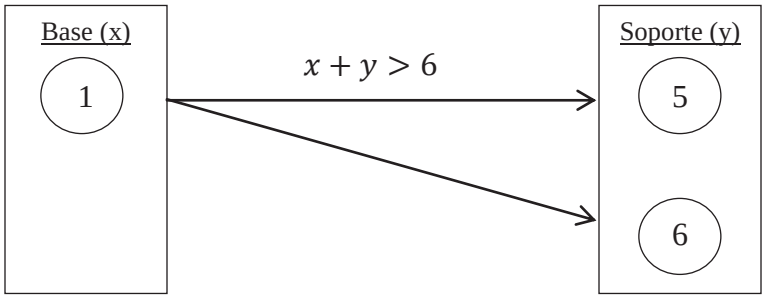

Figura 12 Modo de Trabajo SAC-1.

Se puede observar en la figura 12 que se sigue manteniendo la consistencia, ya que el valor de la $Base\ (x)$ sigue teniendo un $Soorte\ (y)$ que permite cumplir la restricción. Pero ahora al aplicar la bidireccionalidad se notará la diferencia.

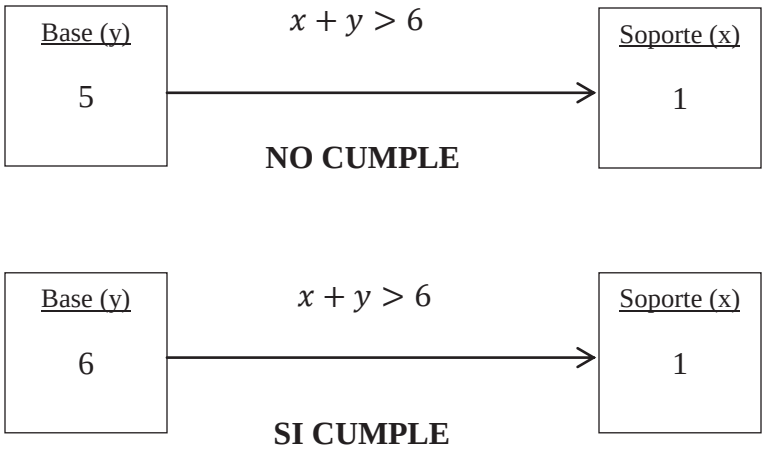

Figura 13 Modo de Trabajo Bidireccional SAC-1.

Finalmente se aprecia en la figura 13 como SAC-1 realiza la bidireccionalidad y se observa que el valor 5 de la $Base\ (y)$ no es consistente con el valor del $Soorte\ (x)$, ya que $1 + 5 > 6$ no satisface la restricción, por lo que SAC-1 en este caso eliminaría el valor 5 de la $Base\ (y)$. Por otro lado el valor 6 de la $Base\ (y)$ si cuenta con un $Soporte\ (x)$ que satisfaga la restricción.

SAC-1 permite encontrar inconsistencias que por sí solo AC-1 no detecta, esto gracias a la descomposición que hace de los dominios que permite trabajarlos de forma individual, por eso se dice que todo algoritmo SAC es de fuerza bruta, ya que trabaja de forma más específica y minuciosa un determinado problema.

5 Arco-Consistencia Adaptativa

Se entiende por arco-consistencia adaptativa para efectos de este informe, como la acción de poder alternar los algoritmos AC-1 y SAC-1 de manera tal, que al momento de recorrer el árbol de búsqueda si no se obtiene la poda esperada se cambiará de una estrategia a otra, esto quiere decir que dicha estrategia deberá cumplir con un porcentaje determinado por el usuario del programa, en caso de no cumplir este porcentaje se procederá a trabajar de inmediato con la otra estrategia.

Lo que se pretende lograr mediante la *adaptabilidad*, es el poder utilizar más de un algoritmo de arco-consistencia para la resolución de problemas y también comprobar si afecta negativamente esta alternancia. En la teoría se desea mejorar el tiempo de búsqueda de la o las soluciones. A priori se planea utilizar arco-consistencia adaptativa junto al algoritmo de búsqueda *Backtracking*. A continuación se presentarán de forma más clara lo que se pretende realizar en la presente investigación.

Se comenzará por definir un *porcentaje esperado de poda* (%PEP) para cada AC, a modo de ejemplo será de un 10% para ambos algoritmos, lo que se establece acá, es que tanto AC-1 como SAC-1 al momento de actuar en el árbol deberán podar un 10% como mínimo de los nodos recorridos. Si AC-1 está en funcionamiento y poda un 8% del árbol, ese ocho corresponde al $8\%RA_1$ de AC-1 (RA_1 corresponde al rendimiento logrado por AC-1), lo que es menor en comparación a lo esperado, provocando un cambio de algoritmo en el siguiente nodo, pasando a trabajar SAC-1.

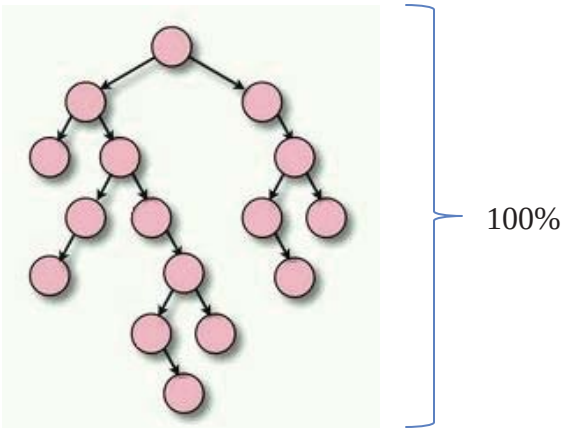

Figura 14 Representación del 100% del árbol inicial.

Luego, al cambiar a SAC-1, este seguirá recorriendo y podando el árbol, esta vez con SAC-1 se logró podar el árbol en un $9\%RA_2$ (RA_2 corresponde al rendimiento logrado por SAC-1). Como los dos porcentajes son menores al 10% esperado, se procede a compararlos

entre ellos, es decir se compara RA_1 con RA_2, como el 9% de RA_2 es mayor, se continúa con SAC-1. Posteriormente se repite lo anterior hasta llegar al final del árbol, chequeando cada vez que corresponda (dependiendo del porcentaje de revisión %*PREV* que se otorgue) los $\%RA_i$ y comparándolos con el porcentaje esperado de poda y con el porcentaje de rendimiento actual del otro algoritmo. Es necesario mencionar que cuando uno de los algoritmos obtiene o supera el porcentaje de rendimiento actual, en este caso 10% RA_i, la comparación sólo se realiza con él %*PEP* que se estableció, ya que si el algoritmo logra cumplir con la poda esperada, se seguirá trabajando con este.

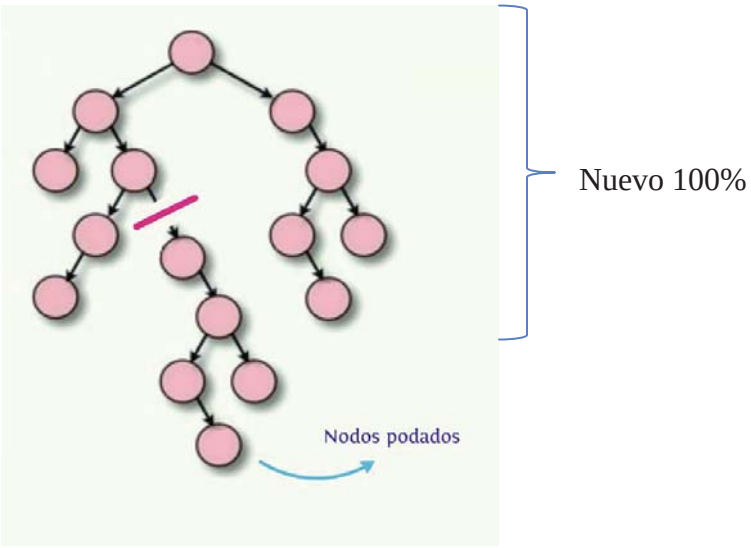

Figura 15 Representación del árbol podado.

Ya claro el cómo trabajarán los AC, nace la siguiente interrogante ¿Cómo definir cuantos nodos recorrer en caso de que el porcentaje de recorrido no arroje un número entero? Para entender mejor la situación, se considerará un pequeño ejemplo:

Se tiene un árbol de cinco nodos y el usuario definió que cada 30% de recorrido del árbol, se debe chequear el rendimiento del algoritmo como se muestra en la figura 16. Según el cálculo (y tomando en cuenta que en este ejemplo se empezará en el nodo uno utilizando AC-1), AC-1 debería recorrer 1,5 nodos (1,5 es el 30% de cinco), y luego chequear porcentaje de rendimiento del algoritmo; Como los nodos se cuentan con números enteros se tienen dos posibilidades: recorrer un nodo, o bien dos nodos. Se optó por la primera opción, es decir que se recorrerá un nodo (se aproximará hacia abajo).

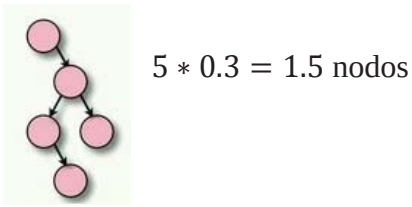

Figura 16 Ejemplo de nodos no coincidentes con porcentaje de poda.

Mediante todo este proceso se busca que el sistema sea capaz de adaptarse frente a lo que se considera como bajo rendimiento, cambiando el algoritmo que se emplea en ese momento por uno que se espera pueda superar de mejor forma esa situación. Es necesario mencionar que no hay forma de saber si el cambio de algoritmo será mejor o peor, ya que los problemas se generan de forma aleatoria.

6 Implementación

Como se ha mencionado con anterioridad, el objetivo principal del presente proyecto es implementar una técnica de arco-consistencia adaptativa en un algoritmo de búsqueda que está incluido en un prototipo de Solver. A continuación se hará una breve descripción del Solver a utilizar denominado *RSSolver,* el cual fue desarrollado por el Dr. Ricardo Soto.

6.1 RSSolver

En primer lugar se debe comprender que un *Solver* es una herramienta que se utiliza para la resolución CSPs, esto quiere decir, que se involucran variables, dominios y restricciones. En lo que respecta a la presente investigación se ha realizado en el prototipo de Solver *RSSolver,* desarrollado por el Dr. Ricardo Soto de Giorgis. Este Solver incorpora un lenguaje desarrollado con el propósito de crear los modelos correspondientes a un problema en particular, en donde se deben incluir los tres elementos claves mencionados con anterioridad. A continuación se muestra un ejemplo del modelo escrito en el lenguaje creado para RSSolver:

variables:

x in [1,1000];
y in [12,53];
z in [1,45];

constraints:

x+*y*>750;
y+*z*>60;
x-*z*>=860;

Como se puede apreciar en el ejemplo anterior, lo primero es definir de forma clara las variables (x, y, z) en el lenguaje con sus respectivos dominios, los que se especifican luego de la palabra reservada *in* dentro de los paréntesis cuadrados. Posterior a la definición de variables, se deben escribir las restricciones las cuales tienen que estar asociadas a las variables ya declaradas. Cabe destacar que en este lenguaje de modelado, la cota inferior y superior del dominio debe ser definida, es decir de forma consecutiva como se aprecia en el ejemplo anterior, lo que es un impedimento a la hora de querer definir un dominio no consecutivo como podría ser [3, 48, 105].

6.1.1 Compilación del RSSolver

Luego de la breve explicación de la herramienta en la cual se lleva a cabo el proyecto, hace falta detallar la estructura detrás del funcionamiento Solver. Mediante el lenguaje de modelado se crea el modelo que tiene relación con el problema que se desea resolver y este modelo debe ir pasando por diferentes etapas. En primer lugar está la etapa de compilación (rsSolver.compiler), la que comienza con el análisis léxico (rsSolverLexer.g). Este análisis tiene la tarea de revisar cada palabra reservada y símbolos que se utilizan en el modelo.

Ya superada la primera etapa de forma exitosa, se comienza con el análisis sintáctico (rsSolverParser.g), este recibe los tokens que se definieron en el léxico, es decir se revisa la sintaxis con respecto a las reglas que se establecieron en el modelo y se genera un árbol denominado AST, el que corresponde a una estructura de análisis. Posterior a esto, sigue la tercera etapa, en la cual se recorre el modelo a través del árbol que se generó, dicha etapa es la del análisis semántico (rsSolverTreeParser.g). En esta última etapa, el objetivo principal es encontrar incoherencias con respecto a las reglas semánticas definidas.

6.1.2 Comprensión y funcionamiento

En el proceso de compilación existen tres clases principales, *MODEL, VARIABLES y CONSTRAINS*. La primera es la clase madre, *MODEL* contiene 3 atributos que se encargan de transformar la estructura de los datos ingresados para que estos sean entendidos por el compilador:

- **variables:** este atributo se encarga de recibir las variables, a cada una de las variables que recibe le asigna un ID distinto y luego las almacena en un *HashMap* con objetos de tipo *VARIABLE*.
- **orderedVars:** El atributo orderedVars tiene como función almacenar los nombre de cada variable existente en un arreglo de datos tipo String.
- **constraints**: Corresponde a un arreglo donde se guardan las restricciones del modelo. Este es un arreglo con objetos de tipo constraints.

La segunda clase se denomina *VARIABLE*, desde esta clase se originan las instanciaciones u objetos almacenados en el *HashMap* variable de la clase *MODEL*, dichos objetos cuentan con atributos donde se almacena la información de cada uno:

- **id:** Atributo tipo String que contiene el identificador o nombre de la variable almacenada.
- **max bound:** Corresponde a la cota superior del dominio de la variable. Este atributo se almacena como un número entero.
- **min bound:** Corresponde a la cota inferior del dominio de la variable. Este atributo se almacena como un número entero.

- **value:** Cada una de las variables en algún momento tomarán un valor, este será almacenado en el atributo value.

Por último, la clase *CONSTRAINT*, contiene un arreglo de objetos llamado *EXPRESION*, donde se guardan las restricciones de la siguiente forma:

$$[[x = null;\ y = null;+;\ 10;>=];\ [x = null;\ y = null;+;\ 10;<=]]$$

Las tres clases descritas anteriormente forman parte del proceso de compilación, sin embargo, éste tiene su raíz en la clase *TOOL* la que se encarga del manejo general del proceso y da lugar al inicio del programa; Entre sus funciones se encuentran: Facilitar la interacción entre clases y analizadores y mostrar información al usuario como resultados y tiempo de compilación.

6.2 Software final

Para el desarrollo del software final, se utilizó RSSolver con funciones adicionales realizadas por Sebastián Lizama y Alexis Muñoz [4], por lo que en el presente proyecto se utilizarán algunas estructuras de datos, métodos y clases que no estaban presentes en el Solver original, pero variando la finalidad con las que son utilizadas, ya que para implementar la adaptabilidad de la forma planeada se necesitan dichos elementos:

1. ArrayList < Integer > llamado Arcs.

2. ArrayList < Integer > llamado toErase.

Se utililizaron los algoritmos de Arco-consistencia AC-1 y SAC-1 para poner en práctica la Arco-consistencia Adaptativa, por este motivo, ambos algoritmos se incorporaron al RSSolver. Ya comprendido el funcionamiento de este en secciones anteriores, se procedió a separar el trabajo en tres fases, comenzando con la incorporación del algoritmo AC-1, luego la incorporación de SAC-1 y finalmente la definición de parámetros necesarios para lograr la adaptabilidad.

Los parámetros aludidos anteriormente corresponden al porcentaje esperado de poda (%*PEP)*, el cual estará definido con anterioridad y no se tiene pensado un porcentaje fijo, este puede ser modificado, ya que no se puede establecer un porcentaje genérico y que asegure que será el mejor. El otro parámetro es el rendimiento actual de cada algoritmo (%RA_i), el que tampoco será fijo y podrá ser modificado, un alto porcentaje le exige un mejor rendimiento al algoritmo y puede provocar más alternancia en los algoritmos, por el contrario un bajo porcentaje no le exige tanto rendimiento a un algoritmo y puede provocar una baja alternancia. Y el siguiente parámetro es el porcentaje de revisión (%*PREV*), el que también se puede ir modificando, este parámetro es el que indica cada cuanto tiempo se deben hacer las

comparaciones de rendimiento, un bajo %*PREV* genera más comparaciones y por ende puede generar más alternancia, un alto %*PREV* genera pocas comparaciones y fuerza que actúe mayor tiempo un algoritmo.

Por otro lado si él %*PEP*, $\%RA_i$ y %*PREV* dan como resultado un número decimal, siempre se aproximará hacia el entero inferior.

En cuanto a la incorporación del algoritmo AC-1 utilizado en el software final, este contiene un sub-proceso llamado "*revise*" y dos atributos, el primero de los atributos es un *ArrayList* de números enteros llamado "*Toerase*", es en este arreglo donde se almacenan los valores que son eliminados del dominio por no cumplir con alguna restricción. En segundo lugar se tiene un atributo booleano que se encarga de indicar si se ha eliminado o no algún valor del dominio. Luego se itera el siguiente procedimiento hasta que la variable "*search*" cambie su valor de 1 a 0.

```
1 public void AC1(int k, ArrayList<Integer> Arcs_evaluation, int
  chosenConstraint_AC ){
2   ArrayList<Integer> Toerase = new ArrayList<Integer> ();
3   int retorno=-1;
4   boolean Ev = false;
5      for (int base= D.get(Arcs_evaluation.get(k)).minBound;
       base<=D.get(Arcs_evaluation.get(k)).maxBound; base++){
6            for (int support= D.get(Arcs_evaluation.get(k+1)).minBound;
             support<=D.get(Arcs_evaluation.get(k+1)).maxBound; support++){
7                  Ev = (evaluate(base, support, chosenConstraint_AC,
                    Arcs_evaluation.get(k), Arcs_evaluation.get(k+1))||Ev);
8                  if(Ev) break;
9            }
10           if(!Ev) {
11                 Toerase.add(base);
12                 exitPreExecute++;
13           }
14           Ev=false;
15     }
16     delete(Toerase,Arcs_evaluation ,k);
17     chosenConstraint_AC=chosenConstraint_AC+1;
18     if((k+1)!=(Arcs_evaluation.size()-1)) revise((k+2),Arcs_evaluation);
       model().getVar(D.size()-1).reinitializeDom()
19
20    }
```

Figura 17 Algoritmo AC-1

En las líneas 5 y 7 del algoritmo se muestran dos "*for*" los que permiten el recorrido del árbol, dentro de ellos el algoritmo llama al método "*evaluate*", si este método retorna la variable "*Ev*" con valor "*true*", AC-1 almacena en el arreglo "*Toeraser*" el valor eliminado del dominio (Línea 11) y nuevamente asigna "*falso*" a la variable "*Ev*", acto seguido se asigna el valor "0" a la variable "*search*", que es la encargada de seguir iterando en el caso de que existan variables sin recorrer.

El método *evaluate* realiza la asignación correspondiente a cada variable con sus respectivos valores, a su vez, implementa el submétodo "*evaluateConstraints*" y asigna su resultado a la variable "*Ev*" (línea 7), si la respuesta es "*true*". En la línea 7, el método *evaluate* retorna como parámetro un argumento de tipo booleano (false o true) al algoritmo AC-1 (Figura 17) indicando si se ha eliminado alguna variable del dominio. Luego de terminada la iteración, el algoritmo aumenta en una unidad la variable *chosenConstraint_AC* y en dos unidades la variable *k* (esta variable representa la posición en que se encuentra la variable dentro de un arreglo, al trabajar con restricciones binarias, *k* aumenta 2 veces en cada iteración) y nuevamente se manda a llamar a sí misma, hasta que se acaben las restricciones a evaluar.

```
1  public void getArc(){
2    for(Constraint con:model().getConstraints()){
3          for(int i=0;i<2;i++){
4                for(int j=0;j<model().getOrderedVars().size();j++){
5                      if(con.getExpression().get(i).toString().
6                      contains(model().getOrderedVars().get(j).toString())){
7                            Arcs.add(j);
8                            break;
9                      }
10               }
11          }
12    }
13 }
```

Figura 18 Método getArc

El método *getArc*() es el encargado de devolver los arcos que sestan involucrados en la restricción, define por ejemplo el arco (x, y), que forma parte de la restricción $x - y > 3$, este método no contiene atributos y comienza con dos *for* , el primero se encarga de recorrer las restricciones almacenadas en el modelo, luego, en el segundo *for* define las dos variables involucradas (en esta caso x e y) y el tercer *for* recorre todas las variables contenidas en el modelo. Por último el *if* tiene la finalidad de comprobar si la variable en la que está posicionada el último *for* se encuentra incluida en la restricción. Si la respuesta es *true* lo guardara en el *ArrayListArcs*, si la respuesta es *false*, continuará con la búsqueda.

Finalmente en la incorporación del algoritmo SAC-1, luego de recibir las variables y la restricción a evaluar, SAC1 llama a AC1SAC (línea 2), algoritmo que se encarga de evaluar el arco correspondiente. Dentro de AC1 se revisa si las variables (k y k+1) cumplen con la restricción, si la base no tiene al menos un soporte que cumpla con la restricción, esta es eliminada. Después de que AC1 revisa completamente la restricción 0, el algoritmo retorna a SAC1 donde sigue el proceso de revisión, pero esta vez tomando cada valor de la base como un dominio individual, y examinando los soportes con un dominio que solo tiene un valor. Al igual que en AC1, existen dos "*for*" los que permiten el recorrido de los dominios, dentro de ellos el algoritmo llama al método "*evaluate*", si este método retorna la variable "*Ev2*" con valor "*false*", el algoritmo agrega el valor inconsistente al arreglo "*Toerase2*"(línea 10), para más tarde ser eliminado mediante la función "*delete*" (línea 15). Cabe destacar que en esta parte del proceso a diferencia de AC1 se elimina el soporte, y no la base, detectando inconsistencias que AC1 no puede detectar por sí solo. .

Luego se aumentan los valore de las variables *chosenConstraint_AC* y *k* para avanzar en la revisión e iterar nuevamente.

```
1    public void SAC(int k, ArrayList<Integer> Arcs_evaluation){
2     AC1SAC(k, Arcs_evaluation);
3     boolean Ev2 = false;
4     int sizedom=0;
5        for (int base= D.get(Arcs_evaluation.get(k)).minBound;
          base<=D.get(Arcs_evaluation.get(k)).maxBound; base++){
6              ArrayList<Integer> Toerase2 = new ArrayList<Integer> ();
7              for (int support= D.get(Arcs_evaluation.get(k+1)).minBound;
               support<=D.get(Arcs_evaluation.get(k+1)).maxBound; support++){
8                Ev2 = (evaluate(support, base, chosenConstraint_SAC,
                       Arcs_evaluation.get(k+1), Arcs_evaluation.get(k))||Ev2);
9                      if(!Ev2) {
10                         Toerase2.add(support);
11                         exitPreExecute++;
12                     }
13                     if(Ev2) break;
14             }
15             delete(Toerase2, Arcs_evaluation, k+1);
16    }
17    chosenConstraint_SAC=chosenConstraint_SAC+1;
18    if((k+1)!=5) SAC((k+2),Arcs_evaluation);
19    model().getVar(D.size()-1).reinitializeDom();
20   }
```

Figura 19 Algoritmo SAC1

7 Pruebas

Ya implementados y funcionando los algoritmos AC-1 y SAC-1, corresponde realizar pruebas aleatorias, las cuales como orden tendrán: En primer lugar se utilizará AC-1 para ver el tiempo en el que encuentra la solución, en segunda instancia se utilizará SAC-1 para resolver el mismo problema y finalmente se resolverá utilizando ambos algoritmos de forma adaptativa.

El planteo del problema será de forma aleatoria. El procedimiento consiste en lo siguiente:

Se comenzará trabajando con el problema número 1 (Test1), se utilizará AC-1, el cual se compilará en diez ocasiones, en cada compilación se anotará su tiempo en un borrador para así obtener el peor, mejor y resultado promedio. Se lleva a cabo la misma acción pero ahora con SAC-1, y se continúa de la misma forma hasta el último test (Test20). Cabe señalar que todos los TEST que se llevaron a cabo se encuentran adjunto en el anexo de este proyecto.

Ya con todos los datos en el borrador, se deben traspasar estos a la tabla (figura 20).

TEST18	Peor Tiempo de Búsqueda (MS)	Mejor Tiempo de Búsqueda (MS)	Tiempo Promedio de Búsqueda (MS)
Prueba con AC-1	81411	71362	77382
Prueba con SAC-1	132267	149052	137774

Adaptativo Test18	Porcentaje de Revisión (%PREV)	Porcentaje Esperado Poda (%PEP)	Peor Tiempo de Búsqueda (MS)	Mejor Tiempo de Búsqueda (MS)	Tiempo Promedio de Búsqueda (MS)
Prueba 1	15	20	81801	76797	80538
Prueba 2	15	40	82880	80886	81613
Prueba 3	20	60	82520	81846	82218
Prueba 4	20	40	83977	80122	81514
Prueba 5	30	20	81218	79087	80278
Prueba 6	50	60	82484	79983	81298

Figura 20 Tabla de Resultados

En la tabla de resultados se tienen distintos campos que se detallan a continuación:

- Nombre del Test: Es el nombre del archivo .txt que contiene las pruebas, este se encuentra en la esquina superior izquierda.

- AC-1 y SAC-1: Estos campos corresponden al nombre de los algoritmos que ejecutaron la prueba, cada uno muestra tres resultados, el peor, mejor y resultado promedio.
- ADAPTATIVO: Es el resultado de utilizar AC-1 y SAC-1 alternadamente.
- Porcentaje de Revisión: Este indicador corresponde al porcentaje de la cantidad de nodos que debe recorrer cada algoritmo, permitiendo esto que se haga una revisión periódica. Se utiliza solo en el algoritmo adaptativo.
- Porcentaje Esperado de Poda: Corresponde a los nodos que se espera que pode cada algoritmo.
- Resultado Tiempo de Búsqueda: Es el tiempo que demora en ejecutarse un algoritmo. Estas son las tres últimas columnas, en las cuales se muestra el peor, mejor y tiempo promedio de cada ejercicio.

Se llevaron a cabo más de 25 ejercicios, en estos se pueden observar resultados individuales diversos, pudiéndose observar que cada algoritmo logró al menos una vez obtener un mejor resultado que el resto. Esto hace pensar que si existen problemas en los que el algoritmo adaptativo pueda obtener el mejor tiempo de búsqueda con respecto a los otros dos algoritmos. Así mismo se aprecia que viendo los ejercicios de forma individual, AC-1 logra ser el mejor en la mayoría de las veces, seguido por SAC-1.

También se pudo notar que cuando SAC-1 trabajaba mejor que AC-1, obtenía un resultado en la búsqueda distinto, lo que hace pensar que su forma distinta de trabajar si cumple su objetivo, no así cuando SAC-1 demoraba más que AC-1, ya que en este caso el resultado de la búsqueda que arrojaba era el mismo que AC-1 pero demoraba más tiempo en encontrarlo.

Por lo general AC-1 o SAC-1 obtenían una ventaja considerable entre sí, mostrando claramente que en algunos ejercicios era mucho más conveniente uno u otro algoritmo. Por otro lado, el algoritmo Adaptativo si bien no lograba ser el mejor en la resolución de cada problema, si mostraba una constante en el tiempo de búsqueda, ya que este por lo general se mantenía cercano al algoritmo que obtenía el mejor tiempo.

Ya analizados los ejercicios de forma individual, corresponde ver el promedio de los mejores resultados en cada ejercicio con los distintos algoritmos, dichos resultados se pueden apreciar en la figura 21. Acá se registran los mejores tiempos (el mejor tiempo entre las diez veces que se compilaba cada ejercicio) en cada Test y con cada algoritmo. Por ejemplo, si se ve el Test1, el tiempo de AC-1 (*2034 ms*) corresponde al mejor tiempo que se registró en las diez compilaciones de ese ejercicio. Pasa lo mismo para el valor de SAC-1 y Adaptativo. Esto se hizo con el fin de poder obtener el promedio de los mejores resultados y ver que algoritmo logra un mejor rendimiento.

Finalmente el resultado promedio, es distinto al del obtenido de manera individual, ya que el algoritmo adaptativo fue el de mejor rendimiento y a pesar de que tuvo una diferencia cercana con el algoritmo AC-1, la distancia fue algo más considerable con SAC-1. Esto además muestra que Adaptativo fue más regular en su trabajo en relación a los otros dos algoritmos y que si bien no es el más rápido en un determinado ejercicio, tiene la ventaja de tener buenos resultados en un conjunto de pruebas debido a su regularidad. Esto último se observa en la figura 21, en donde se tiene que el Adaptativo por lo general es el segundo mejor tiempo y los otros dos algoritmos alternan entre el mejor y peor tiempo.

Test	AC-1	SAC-1	Adaptativo	Test	AC-1	SAC-1	Adaptativo
Test1	2034	2418	2076	Test11	27081	45521	35112
Test2	48082	44206	45104	Test12	47190	54456	49596
Test 3	2527	2871	2543	Test 13	4057	7560	5481
Test 4	21887	56472	22575	Test 14	41632	39714	40125
Test 5	44117	51263	43939	Test 15	3916	8798	4084
Test 6	23322	57672	27306	Test 16	2137	1393	1716
Test7	10951	15601	11213	Test17	76213	60032	63981
Test8	52573	16380	18667	Test18	71362	149052	76797
Test9	49613	53892	50388	Test19	7153	18891	7838
Test10	3557	11840	5476	Test20	9188	16014	10721
				Promedio	27429	35702	26236

Figura 21 Tabla Final de Resultados

Un problema que se presentó, fue el hecho de no poder encontrar un porcentaje adecuado para los indicadores, ya que se obtenían resultados variados y no se logró encontrar un patrón a seguir. Por otro lado se decidió probar con un porcentaje de revisión mínimo de 15% y un porcentaje esperado de poda mínimo de 20%, esto porque al realizar más comparaciones se aumenta considerablemente el tiempo e impide un trabajo más tranquilo de un algoritmo y al exigirle poca poda es más probable que sólo trabaje un algoritmo. También se estableció como porcentaje de revisión máximo un 50% y un porcentaje esperado de poda de 60%, esto ya que con un 50% se asegura al menos una comparación y 60% porque un porcentaje mayor exige una gran cantidad de poda lo que podría obligar a cambiar de algoritmo aun cuando este pudiese estar trabajando de buena forma.

8 Conclusiones

Con toda la teoría ya clara, en el presente proyecto se implementó el software final, para esto se llevaron a cabo diversas tareas las que permitieron realizar lo que se planificó en el comienzo de este proyecto.

En primer lugar se partió por corregir todos los errores presentes en el avance de este trabajo, así como también replantear si se contaban con todas las herramientas necesarias para llevar a cabo una arco consistencia adaptativa, que en la teoría funcione de manera eficiente. Es por lo anterior que se decidió cambiar uno de los algoritmos de arco consistencia (AC-3), ya que su impacto es similar a AC-1. Por este motivo se optó por la utilización de SAC-1.

Ya claros los algoritmos a utilizar para la búsqueda de soluciones, se procedió con el análisis del RSSolver, es decir que se comprendió su funcionamiento, el como un modelo debe pasar por diferentes fases para terminar su compilación. Esto se lleva a cabo a través de los distintos analizadores como lo son el léxico, sintáctico y semántico.

Por otro lado, se logró implementar en el solver el algoritmo AC-1 y SAC-1 para que trabajen en conjunto en la resolución de problemas. En el software final que se efectuó, AC-1 y SAC-1 son capaces de resolver problemas de forma individual y de manera alternada. Si bien la eficiencia no es la esperada, ya que los tiempos no fueron los que se deseaban individualmente, sino que por el contrario los tiempos fueron mayores por lo general a utilizar los algoritmos de forma separada.

Para finalizar se cree que el trabajo realizado se logró desarrollar de forma correcta, ya que se pudo implementar la arco consistencia adaptativa. Además de implementarla, se lograron resultados exitosos, ya que el algoritmo Adaptativo obtuvo el mejor resultado promedio en el tiempo de búsqueda, esto en las veinte pruebas realizadas. Lo que muestra que si bien los algoritmos AC-1 y SAC-1 logran tener mejores tiempos en un ejercicio en particular, esto no lo reflejan en el promedio de tiempos cuando hay más de un ejercicio. Por otro lado se pudo experimentar con una técnica novedosa y de la cual aún no existe demasiada información, como lo es la Arco Consistencia Adaptativa, así como también se estudiaron dos técnicas diferentes (AC-1 y SAC-1) que permiten conseguir resultados y tiempos de búsqueda distintos dependiendo del ejercicio que se resuelva.

9 Referencias

[1] Marlene Alicia Arangú Lobig and Miguel Ángel Salido Gregorio. *Modelos y Técnicas en Problemas de Satisfacción con Restricciones.* PhD thesis, Universidad Politécnica de Valencia, 2011.

[2] Broderick Crawford, Carlos Castro and Erick Monfroy. Programación con restricciones dinámica. Technical report, Pontificia Universidad Católica de Valparaíso, Universidad Técnica Santa María, Université de Nantes, 2009.

[3] Federico Baber and Miguel A. Salido. Introducción a la programación de restricciones. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 20:13-30, 2003.

[4] Sebastián Andrés Lizama Avilés and Alexis Iván Muñoz Concha. Algoritmos de consistencia para programación con restricciones. *Informe final de proyecto para optar al título profesional de ingeniero de ejecución en informática.* Pontificia Universidad Católica de Valparaíso, 2013.

[5] Roman Barták, Miguel A. Salido, and Francesca Rossi. New trends in constraint satisfaction, planning and scheduling: A survey. *Knowledge Eng. Review*, 25:249-279, 2004.

[6] Wendy Yaneth García Martínez. Análisis de sistemas de Programación Lógica por Restricciones. *Ensayo.* Universidad Tecnológica de la Mixteca, 1998.

[7] Anatoli Koulinitch, Wendy Yaneth García Martínez. Análisis de Sistemas de Programación Lógica por Restricciones. *Ensayo.* Universidad Tecnológica de la Mixteca, 1998.

[8] Felip Manyà, Carla Gomes. Técnicas de resolución de problemas de Satisfacción de Restricciones. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 7:169-180, 2003.

[9] Christian Bessiere, and Romuald Debruyne. Some practicable filtering techniques for the constraint satisfaction problem. In Proceedings. *IJCAI'97*, Nagoya, Japan, 1997.

[10] Christian Bessiere, and Romuald Debruyne. Theoretical Analysis of Singleton Arc Consistency. *Journal Artificial Intelligence*, pp. 2-6 2007.