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

CHOICE FUNCTION BASADO EN TOP-K PARA AUTONOMOUS SEARCH

PABLO IVÁN FIGUEROA REYES
DANIELA ANDREA MALLEA ORELLANA

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

MARZO 2014

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

CHOICE FUNCTION BASADO EN TOP-K PARA AUTONOMOUS SEARCH

PABLO IVÁN FIGUEROA REYES
DANIELA ANDREA MALLEA ORELLANA

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

Carrera: Ingeniería de Ejecución en Informática.

MARZO 2014

Queremos agredecer a nuestros profesores, amigos y familiares que siempre estuvieron acompañándonos y ayudándonos en los momentos que más necesitábamos, por compartir sus conocimientos, vivencias y alegrías.

Índice

Resumen

El presente proyecto tiene como objetivo mejorar la eficiencia en la resolución de problemas de satisfacción y para esto se utiliza *Autonomous Search* que se encarga de remplazar las estrategias que muestren rendimientos deficientes por otras que sean más eficaces, por lo mismo es necesaria la utilización de un algoritmo llamado *Choice Function* que vaya escogiendo a medida que se resuelve el problema las heurísticas más adecuadas para cada momento, basándose en una serie de parámetros. Esta función forma parte de la arquitectura de *Autonomous Search*. En la actualidad se han probado muy pocas heurísticas basadas en esta arquitectura, es por esto que se ha planteado implementar una nueva *Choice Function* para *Autonomous Search* basado en *Top-k,* la cual se encarga de rankear las mejores estrategias con el fin de encontrar resultados que demuestren una mejoría en su arquitectura.

Palabras Claves: Autonomous Search (AS), Programación con Restricciones, Choice Function, Top-k.

Abstract

The objective of the present project is to enhance efficiency in the resolution of constraint problems. Therefore *Autonomous Search* is used to replace strategies showing poor performance by others that are more effective; hence it is necessary to use an algorithm called *Choice Function*, which chooses the most appropriate heuristics for each time as it solves the problem, based on a number of parameters. This function is part of the architecture of *Autonomous Search*. Nowadays, very few heuristics have been tested based on this architecture, thus we propose to implement a new *Choice Function* for *Autonomous Search* based on *Top-k*, which is responsible to rate the best strategies in order to find results that demonstrate an improvement for its architecture.

Keywords: Autonomous Search (AS), Constraint Programming, Choice Function, Top-k.

1 Introducción

La Programación con Restricciones o *Constraint Programming* en inglés (CP), es uno de los paradigmas de la programación, que se utiliza para la resolución de problemas de optimización y de satisfacción de restricciones. Esta tecnología tiene su origen en diversas áreas: la Inteligencia Artificial (IA), la Investigación de Operaciones (IO) y los lenguajes de programación; aunque las áreas de aplicación se extienden a muchas otras fuera de la parte informática, como lo son el análisis financiero, el diseño de circuitos o problemas de asignación, tomando un rol muy importante en la planificación, logística y en la toma de decisiones.

Los problemas de CP se representan mediante las variables, las cuales son expresadas por medio de restricciones y dominios. La idea es esencialmente buscar un estado en el cual las restricciones sean satisfechas simultáneamente para todas las variables. Un problema de CP abarca un gran árbol de variables y puede comprender una enorme cantidad de restricciones, lo que hace el programa es buscar y encontrar los valores para cada una de las variables basándose en las restricciones que limitan sus dominios.

La búsqueda de los valores usa muchos recursos de tiempo y procesamiento, para esto AS se encarga de adaptar los procesos de búsqueda cuando estos exhiban un rendimiento pobre en la resolución de un problema. Básicamente, lo que hace AS es evaluar y clasificar las heurísticas *al vuelo*, cuando estas muestren una baja eficiencia remplazarlas, o sea a medida que está ejecutando el algoritmo. La forma en que AS hace esta tarea es mediante una *Choice Function*, esta evalúa las estrategias en base a un conjunto de indicadores y parámetros que se van entregando el proceso de la resolución.

En esta investigación, y como una forma de mejorar la eficiencia de la *Choice Function* se desarrollará e implementará el algoritmo de rankeo *Top-k*. Luego de implementar el algoritmo se espera obtener respuestas positivas a través de pruebas, todo esto para posibles comparaciones con otros algoritmos. Para realizar esta tarea, se escogió usar el *solver* ECLiPSe en la resolución de tareas.

2 Estado del Arte

La programación con restricciones actualmente es usada como una tecnología de software para la descripción y resolución de problemas combinatorios particularmente difíciles, especialmente en las áreas de planificación y programación de tareas (como calendarización). Los primeros trabajos realizados con CP son en el área de inteligencia artificial. Un ejemplo de implementación de un problema de satisfacción de restricciones fue el desarrollo de Sketchpad [18] en la década del 60, este software fue el primer programa de dibujo para computador.

En la actualidad existe una variedad de trabajos de investigación en relación a la programación con restricciones, por ejemplo [12][14][15] entre otros. CP también ha sido utilizado últimamente para modelar y resolver el problema de Nurse Rostering (NRP) [17].

Los estudios de Autonomous Search son relativamente nuevos [16][19], es por esto que existe poca información asociada a él. Esto no quiere decir que sea menos importante, pues su investigación ha ido en aumento y su arquitectura ha ido cambiando, de esta forma se mejora la calidad de los resultados en la resolución de un problema de satisfacción con restricciones. Hace algunos años se completó una arquitectura modular para AS [1], esta es una herramienta completa que permite la resolución de CSP.

Top-k ha sido investigado y utilizado en consultas en bases de datos [11]. El algoritmo de rankeo Top-k puede ser usado en muchos ámbitos, permite tomar decisiones acertadas, ya que realiza una selección de las mejores variables de un dominio en concreto, el cual evalúa las diferentes combinaciones de parámetros para los problemas de restricciones de tipo satisfactorias, con el fin de encontrar la respuesta más eficiente. Además este algoritmo puede seguir siendo modificado y mejorado.

3 Definición de Objetivos

3.1 Objetivos Generales

- Diseñar una *Choice Function* basada en *Top-k* para *Autonomous Search*

3.2 Objetivos Específicos

- Comprender el contexto de la programación con restricciones y AS.
- Analizar la arquitectura para AS.
- Implementar una *Choice Function* para AS basada en *Top-k*.
- Probar la nueva *Choice Function* y analizar los resultados obtenidos.

4 Programación con Restricciones

4.1 Bases de la programación con restricciones

La programación con restricciones es el estudio de sistemas computacionales basados en restricciones, que se utilizan para describir y posteriormente resolver problemas generalmente de tipo combinatorio, como también de optimización por medio de la declaración de restricciones en el dominio del problema con el objetivo de encontrar soluciones que cumplan con todas estas.

Los inicios de CP datan de las décadas del ‘60 y ‘70, en el ámbito de la inteligencia artificial para el entendimiento del lenguaje natural. A través del tiempo se ha aplicado en diversas áreas del conocimiento como la investigación operativa, la planificación, bases de datos, sistema de recuperación de la información, etc. Entre sus principales campos de aplicación se encuentran por ejemplo: análisis financiero, diseño y construcción de circuitos integrados, la resolución de problemas geométricos, asignación de stands en los aeropuertos o asignación de pasillos de salida en aeropuertos de Hong Kong [20], desarrollo de turnos de enfermeras de un hospital [17].

Para la solución de los ejemplos anteriores, CP se puede extender como un problema de optimización o de satisfacción, por un lado ambas comparten las mismas terminologías pero sus técnicas de resolución son distintas. Los problemas de optimización se resuelven como un problema de satisfacción regular, pero el objetivo es generar una solución óptima a través de una función objetivo.

En cambio, los problemas de satisfacción de restricciones (*Constraint Satisfaction Problem, CSP*) constan de conjuntos finitos de variables, con un dominio de valores para cada una de estas y un conjunto de restricciones que acotan la combinación de valores que las variables pueden tomar, de esta forma se puede encontrar un valor para cada variable que satisfaga todas las restricciones impuestas en el problema [2], esto quiere decir que se pueden encontrar muchas soluciones siempre y cuando respeten las restricciones.

Cada una de estas formas de CP tiene como fases de resolución el modelado y la búsqueda. En el modelado se debe analizar el problema, luego debe ser planteado como un, CSP y por último se implementa el modelo en un lenguaje para la programación con restricciones. La búsqueda es parte de las técnicas de resolución la cual se lleva a cabo a través de algoritmos de búsqueda, que se encargan de buscar en un espacio definido las posibles asignaciones de valores a las variables considerando las restricciones del problema, con el fin de encontrar la solución si es que existe, en el caso contrario demostrar que no hay soluciones.

Para ayudar a mejorar la eficiencia de estos algoritmos se utilizan las técnicas de consistencia, las cuales verifican la consistencia entre variables actuales como futuras. Se utiliza además una heurística de selección de variables y valor, que establece el orden en

que se van a estudiar estas, ya que esto puede mejorar notablemente la eficiencia en el tiempo de resolución.

Por último CP cuenta con la ventaja de que el uso de las restricciones puede disminuir eficientemente el espacio de búsqueda, eliminando combinaciones de valores de variables que no cumplen con las condiciones antes mencionadas o simplemente no pueden aparecer juntas.

4.2 Problemas de satisfacción de restricciones

Los CSP se resuelven a través de algoritmos de búsqueda, que buscan las posibles asignaciones a las variables en un espacio que puede ser finito o infinito, las cuales tiene que satisfacer todas las restricciones para encontrar una solución al problema. Si es que existe la solución, consiste en la instanciación de un valor del dominio a cada variable con el fin de satisfacer cada una de las restricciones.

Para el proceso de resolución de un CSP se utilizan dos conceptos fundamentales que son parte de su metodología y que detallaremos a continuación:

- La modelación del problema como un CSP que formula el problema como un conjunto de variables, dominios y restricciones de este.
- Realizar un procesamiento del CSP, que utiliza los algoritmos de búsqueda y las técnicas de consistencia para procesar las restricciones.

La acción en conjunto de los algoritmos de búsqueda con las técnicas de consistencia nos permite reducir el espacio de solución y explorar solo ese espacio resultante, con el objetivo de obtener a una o más soluciones al problema.

4.2.1 Definición de un CSP

Un CSP se representa como una terna *(X, D, C)* donde:

- *X* es un conjunto de variables $x_1, x_2, \dots, x_n$.
- *D* es un conjunto de dominios $d_1, d_2, \dots, d_n$. donde d_i es el dominio de x_i, e $i = 1, \dots, n$.
- *C* es el conjunto finito de restricciones $c_1, c_2, \dots, c_m$, donde $j = 1, \dots, m$. tal que c_j es la relación sobre el conjunto de variables $\{x_1, x_2, \dots, x_n\}$ y $c(x_i, x_j)$ son restricciones entre x_i y x_j restringiendo los valores que pueden tomar las variables.

4.2.2 Modelado de un Problema de Satisfacción con Restricciones

En un CSP existen diferentes formas de modelar un problema para llegar a su resolución, debido a esto mostraremos a continuación 2 ejemplos de modelado:

4.2.2.1 N–reinas

Este problema consiste en colocar *N* reinas en un tablero de ajedrez de tamaño *N*x*N*, de tal manera que no se amenacen entre ellas, una reina amenaza a otra si está en la misma fila, columna o diagonal. Se considerará para el siguiente ejemplo cantidad de reinas *N* = 4.

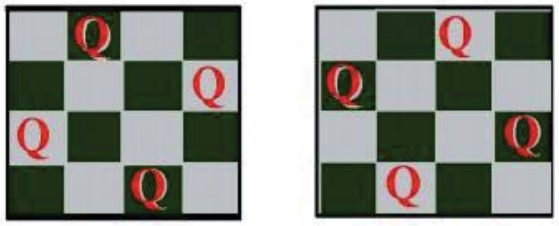

Figura 4.1 Soluciones del problema de 4-reinas

Modelo:

- Definición de las variables (una para cada reina) y el dominio que corresponde a la cantidad de filas.

$$Q_1, Q_2, Q_3, Q_4 \in [1,4].$$

- Restricciones para ($i \in [1,3] y\ j \in [i+1,4]$)
  - Donde las reinas deben estar en distintas filas: $Q_i \neq Q_j$.
  - Las reinas no deben colocarse en la diagonal noroeste, suroeste: $Q_i - Q_j \neq j - i$.
  - Las reinas no deben colocarse en la diagonal suroeste, noroeste: $Q_i - Q_j \neq i - j$.

4.2.2.2 Criptográfico

En este ejemplo (Figura 4.2) se presenta un problema criptográfico, en el cual se plantea una ecuación en la que se deben remplazar las letras *{s,e,n,d,m,o,r,y}* por dígitos distintos que pertenezcan al conjunto del [0,9] de forma que satisfaga el *send + more = money*.

Figura 4.2 Problema de send more money

Modelo:

- Definición de variables, una para cada letra: $s, e, n, d, m, o, r, y \epsilon [0,9]$.
- Restricciones:
  - Cumplir con la regla de las numeración decimal:

$$10^3(s+m) + 10^2(e+o) + 10(n+r) + d + e = 10^4 m + 10^3 o + 10^2 n + 10e + y.$$

  - Todas las letras deben tener valores diferentes:

$$alldiferent(s, e, n, d, m, o, ry)$$

4.3 Algoritmos de búsqueda

Los algoritmos de búsqueda son la base para la resolución de un CSP, porque buscan en el espacio de estados del problema, que es el conjunto de todas las posibles asignaciones de las variables, para encontrar una solución que satisfaga todas las restricciones del problema [3] y si en el caso contrario no se logra encontrar una solución se pruebe que no existe.

El espacio que contiene el conjunto de variables puede ser representado como un árbol de búsqueda, en el cual cada nodo es una asignación parcial de valores de las variables, el nodo raíz del árbol de búsqueda representa el caso en el que ninguna variable se encuentra instanciada, y los nodos hojas son todas las variables que se encuentran instanciadas.

Las búsquedas pueden ser de dos tipos completas si es que recorren todo el espacio y garantizan encontrar una solución y las incompletas que son aquellas que utilizan algoritmos de búsqueda local y sólo recorren cierto espacio de soluciones, estas son más usadas ya que tiene menos costos que las búsquedas completas. A continuación se detallarán los más utilizados.

4.3.1 Generate and Test (GT)

Este método instancia cada una de las variables verificando por cada una, si satisface todas las restricciones del problema, luego va probando cada posible solución desde el conjunto que corresponde a todas las posibles asignaciones de los valores de las variables. La primera combinación que cumpla con todas las restricciones es la solución al problema (Figura 4.3).

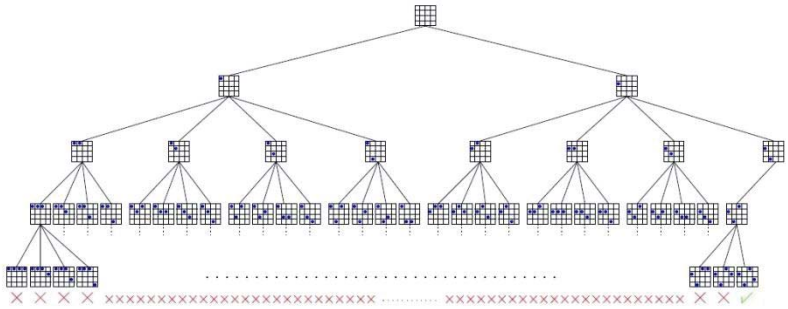

Figura 4.3 Generate and Test en el problema de 4-reinas

4.3.2 Backtracking (BT)

Este algoritmo va construyendo soluciones parciales a medida que va asignando un nuevo valor a la variable actual con el fin de ir comprobando si este nuevo valor es consistente con los valores de las asignaciones pasadas, si no es consistente esa asignación se deshace y se asigna un nuevo valor a la variable. En el caso de no encontrar ningún valor consistente y no haya otra opción (*dead-end*) el algoritmo debe retroceder y probar un nuevo valor en la variable anteriormente instanciada. Estos pasos se pueden volver a repetir si se cayera nuevamente en un *dead-end* (Figura 4.4).

Este método tiene una gran ventaja sobre el Generate and Test, porque para ver si la asignación es consistente no necesita ir instanciando todas las variable y luego ver si cumple con las restricciones recorriendo todo el árbol, ya que apenas detecta una inconsistencia en la asignación parcial la abandona de inmediato.

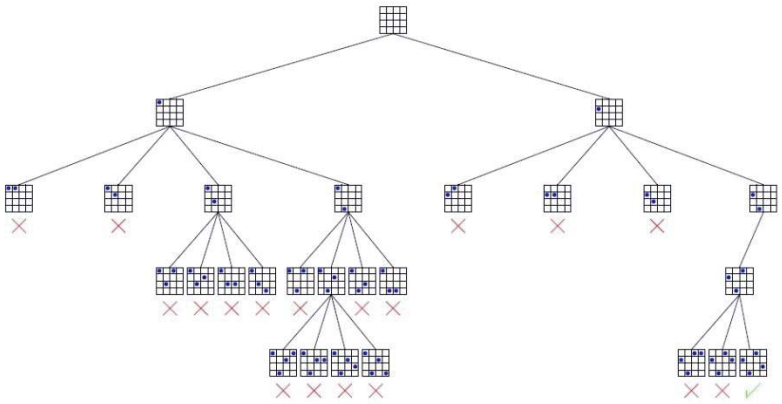

Figura 4.4 BT en el problema de 4-reinas

4.4 Técnicas de consistencia

Los algoritmos de búsqueda son un método bastante sencillo, sin embargo, frecuentemente sufren de una explosión combinatoria del espacio de búsqueda, y por lo tanto por si solos no son un método suficientemente eficiente para resolver un CSP. Una de las principales dificultades con las que se enfrentan estos algoritmos es la continua aparición de inconsistencias locales. Las inconsistencias locales son valores individuales o combinación de valores de las variables que no pueden participar en la solución.

Las técnicas de consistencia (o inferencia) pretenden reducir el espacio de búsqueda con un proceso de filtrado que elimina los valores inconsistentes de los dominios de las variables o inducen restricciones implícitas entre las variables, obteniendo así un nuevo CSP, igual al inicial, pero donde se han explicitado las restricciones que antes estaban implícitas. A continuación se explicarán las técnicas más usadas en los algoritmos de búsqueda:

4.4.1 Consistencia de Nodo

El funcionamiento de la consistencia de nodo (o nodo-consistencia) se basa en asegurar que todos los valores en el dominio de una variable satisfagan las restricciones unarias sobre la variable. Un CSP es nodo-consistente si por cada valor del dominio de la variable x, cada restricción unaria sobre x es satisfecha. Si el dominio de la variable x, contiene valores que no son satisfechos, entonces la inconsistencia del nodo puede ser eliminada, ya que no estaría contenida en ninguna solución. Este proceso se realiza eliminando los valores del dominio de X.

Sea $P = (X, D, C)$, se dice que P es nodo-consistente si y sólo si:

$$\forall x_i \in X, \forall c_j, \exists a \in d_i \text{ Tal que el elemento } a \text{ satisface } c_j.$$

4.4.2 Consistencia de Arco

La consistencia de arco trabaja con restricciones binarias. Un CSP es arco consistente si para cada uno del par de variables restringidas x_i y x_j, para cada valor de a en d_i, existe por lo menos un valor b en d_j, tal que las asignaciones (x_i, a) y (x_j, b) satisfacen la restricción entre x_i y x_j. Lo anterior conduce a eliminar la consistencia de arco de cualquier valor en el dominio d_i de la variable x_i, que no es arco-consistente, ya que no formaría parte de ninguna solución. El dominio de una variable es arco-consistente si todos los valores de dicha variable son arco-consistentes.

Sea $P = (X, D, C)$, se dice que P es arco-consistente si y sólo si:

$\forall c(x_i, x_j) \in C, \forall a \in d_i, \exists b \in d_j$ Tal que el elemento b es un soporte para el elemento a en $c(x_i, x_j)$.

Un problema es arco-consistente, si y sólo si todos sus arcos son arco-consistentes.

4.5 Algoritmos de Búsqueda usando Técnicas de Consistencia

Habiéndose explicado las técnicas de consistencia, se procederá a explicar los algoritmos de búsqueda que utilizan estas técnicas. Estos algoritmos son algunos de los llamados *look-ahead*, los cuales hacen una comprobación hacia adelante en cada etapa de la búsqueda. O sea, estos algoritmos de cierta forma miran hacia el futuro para determinar si los caminos que se van a tomar pueden llevar o no a obtener inconsistencias en las variables. A diferencia de los algoritmos descritos anteriormente, que se dan cuenta de la inconsistencia una vez instanciadas todas las variables.

4.5.1 Forward Checking (FC)

Forward Checking es el algoritmo de look-ahead más común. En este, cada vez que se asigna una variable se comprueban todos los valores posibles de las variables que están restringidas con la variable actual. Si los valores de las futuras variables son inconsistentes con la asignación de la variable actual son temporalmente eliminados de sus dominios. Si el dominio de una futura variable se queda sin valores, la instanciación de la variable actual se deshace y se prueba con el siguiente valor, y si ningún valor es consistente se lleva a cabo el Backtracking cronológico.

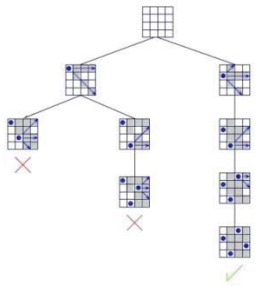

Figura 4.5 Forward Checking en el problema de 4-reinas

Cada vez que una nueva variable es asignada, se sabe que todos los valores de su dominio son consistentes con los valores de las variables asignadas anteriormente, por lo que no es necesario comprobar cada nueva asignación con todas las variables, solo se hace hacia adelante con las futuras. De esta manera, FC puede identificar antes las situaciones sin salida y "podar" el espacio de búsqueda. [5]

4.5.2 Maintaining Arc Consistency (MAC)

El algoritmo FC (Figura 4.6) comprueba únicamente las restricciones entre la variable que se instancia y el resto de las variables por instanciar, lo que quiere decir que mantiene la arco-consistencia entre la variable asignada y el resto de las variables. Este algoritmo (MAC) es llamado así porque intenta mantener la arco-consistencia también entre el resto de dichas variables.

Este algoritmo comprueba que, tras la poda en los dominios de las variables por instanciar realizadas en FC, cada variable que queda por instanciar tenga al menos un valor en sus dominios que sean consistentes con la restricción que exista entre ellas. Lo bueno de este algoritmo frente al FC es que también detecta las inconsistencias entre las variables futuras. Por lo que MAC permite podar muchos más dominios de las variables por instanciar, pero esto conlleva, evidentemente, un mayor costo.

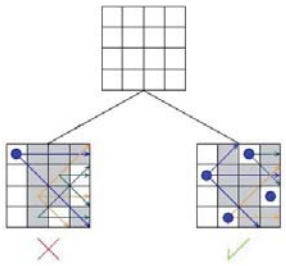

Figura 4.6 MAC en el problema de 4-reinas

4.6 Estrategias de enumeración

La manera y el orden en que los algoritmos de búsqueda seleccionan las variables para examinarlas e instanciarlas puede cambiar notablemente la eficiencia en la resolución. Seleccionar el orden correcto puede mejorar intensamente el tiempo en que se soluciona el CSP. También puede ser importante un orden adecuado en las restricciones del problema, aunque este punto no tiene mucho estudio. Ahora se pasará a revisar las heurísticas más importantes en la ordenación de variables y valores.

4.6.1 Heurísticas de Ordenación de Variables

La ordenación de variables puede ser estática o dinámica. Estas heurísticas se basan en la percepción de que es más conveniente asignar primero las variables más restringidas, de esta forma se identifican mucho antes las situaciones sin salida y se reduce la cantidad de backtracks (vueltas atrás).

4.6.1.1 Ordenación de Variables Estáticas

Basado en información global derivada de la estructura de la red de restricciones inicial, esta heurística genera un orden fijo de las variables antes de iniciar la búsqueda. Entre las ordenaciones de variables estáticas más utilizadas se encuentran las siguientes:

- **Minimum Width (MW):** esta heurística impone en primer lugar un orden total sobre las variables, de forma que el orden tiene la mínima anchura, y entonces selecciona las variables en base a ese orden [6].

- **Maximum Degree (MD):** esta heurística ordena las variables en un orden decreciente con respecto a su grado en el grafo de restricciones. Esta heurística también tiene como objetivo encontrar un orden de anchura mínima, aunque no lo garantiza [7].

- **Maximum Cardinality (MC):** selecciona la primera variable arbitrariamente y después en cada paso, selecciona la variable que es adyacente al conjunto más grande de las variables ya seleccionadas [8].

4.6.1.2 Ordenación de Variables Dinámicas

Las heurísticas de ordenación de variables estáticas no tienen en cuenta los cambios en los dominios o relaciones de las variables causados por la propagación de las restricciones durante la búsqueda. La ordenación de variables dinámicas es capaz de ir cambiando el orden en que las variables son seleccionadas de forma dinámica cada vez que se instancia una variable en el proceso de búsqueda. Las heurísticas más comunes se basan en el principio de primer fallo (First Fail, FF) que sugiere que para tener éxito se debería intentar primero donde sea más probable el fallo. Así, las situaciones sin salida se

identifican antes y se ahorra espacio de búsqueda. Entre las heurísticas más utilizadas que usan este principio se pueden identificar:

- **Minumum Remaining Values (MRV):** esta heurística en cada paso selecciona la variable con el dominio más pequeño. Si una variable tiene pocos valores, entonces se intuye que es más difícil encontrar un valor consistente.

- **Maxium Cardinality (MC):** esta heurística selecciona la primera variable aleatoriamente, luego en cada paso selecciona la variable relacionada con la mayor cantidad de variables ya instanciadas. Esta variable resulta la más restringida al estar relacionada con el mayor número de variables ya instanciadas.

4.6.2 Heurísticas de selección de valor

Hay menos trabajo en el área de ordenación de valores comparado con la ordenación de variables. Básicamente, la idea de estas heurísticas es la de seleccionar el valor de la variable actual que tenga más probabilidad de llegar a una solución, es decir la rama del árbol de búsqueda donde sea más probable encontrar una solución. Estas heurísticas seleccionan, en su mayoría, el valor que restringe menos la variable actual, o sea, el valor que menos reduce los valores útiles para las variables que aún no son instanciadas.

- **Min-conflicts:** selecciona el valor que provoca la menor cantidad de conflictos en el futuro. Esto lo hace asociando cada valor a de la variable actual, el numero total de valores en los dominios de las futuras variables adyacentes con la actual que son incompatibles con a, selecciona la suma más baja.

- **Max-domain-size:** es alternativa a la heurística anterior, selecciona el valor de la variable actual que deja el máximo dominio en las variables futuras.

- **Point-domain-size:** esta heurística asigna un peso (unidad) a cada valor de la variable actual dependiendo del número de variables futuras que se quedan con ciertas tallas de dominios. Al final se elige el valor con el menor peso.

- **Superviviencia:** es una variación de *min-conflicts*. El número de valores incompatibles dentro del dominio es dividido por el tamaño de este. Esto da el porcentaje de los valores útiles del dominio de esa variable. Al final se elige el valor más bajo de las sumas.

- **Max-promise:** en esta heurística, para cada valor a de la variable actual se cuenta el número de valores que hay compatibles con a en cada variable futura adyacente, y se toma el producto de las cantidades contadas, este producto se llama *promesa de valor*. De esta manera se selecciona el valor con la máxima promesa.

4.7 Solvers

Los solvers fueron creados para solucionar CSP's, los cuales se utilizan para resolver problemas que involucren variables, dominios y restricciones. Empleando estos elementos el Solver podrá encontrar una solución al problema. Existen varios tipos de solvers debido a que no todos pueden soportar las mismas restricciones y paradigmas, es por esto que no siempre ofrecen las mismas respuestas.

4.7.1 Programación Lógica con Restricciones

La Programación Lógica con Restricciones (*Constraint Logic Programming*, CLP) une dos paradigmas: la programación lógica con CP. La unión de estos dos paradigmas ayuda a generar programas más flexibles que otros, además mucho más eficientes ya que disminuye dramáticamente el tiempo de ejecución mientras logra una eficiencia similar a los lenguajes procedimentales [9].

El lenguaje que se utiliza se define por contar con una estructura algebraica, las funciones especiales y los predicados simbólicos se interpretan sobre un dominio fijo, formando los símbolos interpretados, las relaciones que establecen sobre el dominio son las restricciones.

CLP puede utilizarse como un lenguaje o como una biblioteca, la cual será usada en un programa. A continuación se explican algunos de los lenguajes más importantes de esta:

- **Eclipse:** Es un software basado en Prolog, que consiste en una gran cantidad de aplicaciones para solucionar problemas de programación de restricciones como listas, arreglos y registros.

- **GNU Prolog:** Es un compilador que utiliza Prolog, que incluye extensiones de programación con restricciones que actúa sobre dominios finitos.

Y por último las bibliotecas, ya que son necesarias para expresar problemas bajo restricciones al haber limitaciones en un lenguaje de programación, como: Choco e Ilog Solver.

4.7.2 ECLiPSe

Es un sistema que se basa en CLP, se utiliza para el desarrollo y despliegue de aplicaciones de CP. ECLiPSe se compone de una colección de bibliotecas para la resolución de restricciones, modelado de alto nivel, mediante un lenguaje basado en Prolog.

Este sistema es utilizado mayormente en áreas de planificación, programación, asignación de recursos y de horarios, ya que se ha transformado en una herramienta

importante para la programación con restricciones. ECLiPSe fue diseñado especialmente para:

- Generar prototipos de una forma más rápida, en el área de la programación.
- Utilizar las bibliotecas y los paradigmas de la CLP para la resolución de problemas.
- El desarrollo de nuevos solvers, basados en otros existentes.

Como ECLiPSe está basado en un programa lógico se interpreta como una base de hechos y reglas sobre las que se realizan consultas. El programa nos proporciona las bibliotecas o librerías que se caracterizan por: la implementación de dominios y restricciones y restricciones definidas por el usuario.

Y por último también nos permite utilizar heurísticas (son parte de las estrategias de búsqueda) para la selección de variables y valores, de modo que nos permite alterar el orden de elección y resolver los problemas de manera más eficiente [10], por ejemplo primero las variables con un dominio más reducido o primero se enfoca en los valores centrales y le aplica las restricciones.

4.7.2.1 N-Reinas en ECLiPSe

Los ejemplos de CP que se explicaron anteriormente pueden ser implementados en ECLiPSe en la Tabla 4.1 se puede ver el ejemplo de N-reinas.

```
:-lib(ic).

queens(N,Board):-

    dim(Board, [N]),
    Board[1..N] :: 1..N,

    ( for(I,1,N), param(Board,N) do
        ( for(J,I+1,N), param(Board,I) do
        Board[I] #\= Board[J],
        Board[I] #\= Board[J]+J-I,
        Board[I] #\= Board[J]+I-J
        )
    ),
```

Tabla 4.1 N-reinas en eclipse

En este código se reciben las dimensiones de tablero y la cantidad de reinas, ingresadas por el usuario. Luego están expresados las variables y sus dominios. A continuación, se enuncian las restricciones, dentro de ambos *for* se expresan estas para cada reina, es decir que no pueda haber 2 o más reinas en la misma fila, columna o diagonales.

5 Autonomous Search

El objetivo de *Autonomous Search* es mejorar el rendimiento en la resolución de un problema en un sistema adaptativo, para esto modifica sus componentes internos cuando se encuentre con fuerzas externas.

Los componentes internos corresponden a los algoritmos que están involucrados en el proceso de búsqueda-heurísticas, inferencias, etc. Cuando se habla de fuerzas externas se refiere a los cambios en la información recogida durante el proceso de búsqueda. Esta información puede ser directamente recogida bajo el problema o computarizada a través de la percepción de componentes individuales. La información recolectada puede ser el espacio de búsqueda, el número de sub-problemas, etc. La información computarizada incluye el grado de discriminación de heurísticas, la capacidad de poda en cada una, etc.

5.1 Arquitectura de Autonomous Search [1]

Cuando se escoge una heurística de selección de variable o valor, no significa que se encontrara una solución a un CSP, ni mucho menos que dicha solución sea la óptima, por esta razón es necesario realizar un proceso de resolución, para así poder evaluar las diferentes estrategias que se poseen, con el fin de encontrar la solución.

Para decidir que alternativa se va a elegir, se requiere de un proceso previo para comenzar la resolución, el que consiste de un *sampling* que se encarga de probar cada estrategia de enumeración y guardar una estadística de su comportamiento que es realizado por el optimizador en base a los indicadores, la cual nos permitirá evaluar en base a una determinada prioridad, que estrategia debemos comenzar a usar para la resolución del problema.

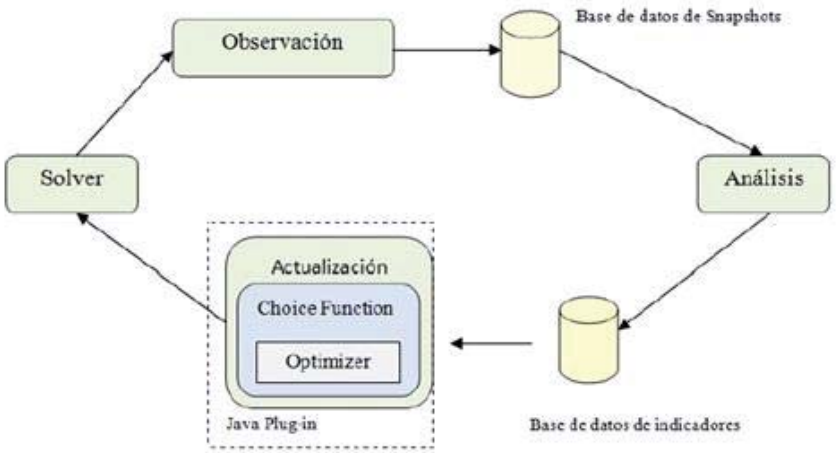

Figura 5.1 Esquema del Framework de Autonomous Search

En la figura 5.1 se ve el diagrama del *framework* de AS que está compuesta por: *solver*, observación, análisis y actualización. Los tres primeros son procesos que inciden directamente en la resolución, y la "actualización" participa solamente en la búsqueda de la mejor estrategia de enumeración. Este proceso está íntimamente ligado al proceso de análisis, lo que significa que es el encargado de entregar las prioridades de uso de cada estrategia.

Además, se puede ver en el diagrama que hay una base de datos de *snapshots* y una base de datos de indicadores. La información del estado de avance es capturada a través de dos métodos, siendo uno de ellos los *snapshots* (capturas de pantalla) e indicadores. Los primeros son los encargados de hacer observaciones sobre el actual árbol de búsqueda, como cuál es la profundidad del nodo actual y el tamaño actual del espacio de búsqueda. Los indicadores son la evidencia de la solución como por ejemplo la variación de la profundidad máxima.

5.1.1 Solver

Es el componente que se encarga de ejecutar el algoritmo CSP genérico. El cual básicamente realiza una búsqueda mediante el algoritmo. También va alternando las fases de propagación de restricciones y las fases de enumeración.

5.1.2 Observación

Este componente registra el proceso de resolución y guardar la información actual del estado del árbol de búsqueda. La información guardada no es continua, ya que son capturas o fotografías (se encuentran en la base de datos de los snapshot) de lo que está sucediendo en la búsqueda. Este proceso es crucial debido a que es cuando se guarda y extrae información de un estado de resolución.

5.1.3 Análisis

Componente que se encarga de analizar las capturas obtenidas del proceso de observación que se registraron en la base de datos snapshot, la cual evalúa además las estrategias y provee de indicadores al proceso de actualización. Los indicadores pueden ser calculados o deducidos desde una o varias observaciones del proceso.

5.1.4 Actualización

Es el componente más importante porque es el que toma las decisiones usando la *Choice Function*, con la cual va calculando los rendimientos de las estrategias en una cantidad de tiempo. Esta se obtiene a través de los indicadores proporcionados por el snapshot.

5.2 *Choice Function*

La tarea de la *Choice Function* es hacer una comparación entre el rendimiento que tiene cada estrategia de enumeración históricamente y el punto de decisión que se está analizando en ese momento. Cada punto de decisión es cuando se llama al solver para fijar una variable de enumeración.

Tabla 5.1 Ejemplos de indicadores de la Choice Function

Nombre	Descripción
VFP	Número de variables reparadas por propagación
$T_n(S_j)$	Número de pasos desde la ultima vez que una estrategia de enumeración S_j fue usada hasta el enésimo paso
n	Número de pasos o puntos de decisión (n incrementa cada vez que una variable es reparada)
SB	Número de vueltas atrás superficiales (ShallowBacktracks)
B	Número de vueltas atrás (Backtracks)
In1	Representa una variación de la profundidad máxima. Se calcula como: profundidad máxima actual = profundidad máxima anterior.
In2	Se calcula como: profundidad actual – profundidad anterior. Un valor positivo significa que el nodo actual es más profundo que el estudio en el paso
B-real	Número de marchas atrás considerando además el número de vueltas atrás superficiales
d	Profundidad actual en el árbol de búsqueda
Thrash	El proceso de resolución alterna enumeraciones y las vueltas atrás en un pequeño número de variables sin terminar sin llegar a tener una fuerte orientación. Se calcula como: $d_{t-1} - VFP_{t-1}$

En cada llamado, la *Choice Function* rankea, califica y luego elige entre las estrategias de enumeración. Para cualquier estrategia de enumeración S_j, la *Choice Function* f en el paso n por cada estrategia S_j está definida en la ecuación 1, donde l es el número de indicadores considerados (Tabla 5.1) y α es un parámetro para controlar la relevancia del indicador dentro de la *Choice Function*.

$$f_n(S_j) = \sum_{i=1}^{l} \alpha_i f_{i_n}(S_j)$$

A esto, hay que agregar que para controlar la relevancia de un indicador i para una estrategia S_j en un intervalo de tiempo, se utiliza una técnica estadística para producir una serie de tiempo suavizado llamado suavizado exponencial. La idea de esto es darle mayor importancia a rendimientos más recientes, asociándoles "pesos" exponencialmente decrecientes para mayores observaciones, o sea, las observaciones más recientes entregan mayor peso a los rendimientos que las más antiguas. Esta técnica es aplicada al cálculo de $f_{i_n}(S_j)$, el cual es definido en las siguientes ecuaciones, donde v_0 es el valor del indicador i para la estrategia S_j en la ecuación anterior, n es un paso dado del proceso, β es el factor de suavizado, y $0 < \beta < 1$.

$$f_{i_n}(S_j) = v_0$$

$$f_{i_n}(S_j) = v_n + \beta_i f_{i_{n-1}}(S_j)$$

Hay que tomar en consideración, que la velocidad a la cual la observación antigua es suavizada depende de β. Cuando β es cercana a 0, el suavizado es rápido y cuando es cercana a 1, el suavizado es lento.

.

6 Algoritmo de Rankeo: *Top-k* [11]

Los diferentes tipos de sistemas de información usan varias tecnologías para rankear respuestas de las consultas. En muchos dominios de aplicaciones, los usuarios finales están más interesados en las respuestas más importantes (Top-k respuestas) del potencial gran espacio de ellas.

Una manera común de identificar los *Top-k* objetos es calificando todos los objetos basados en alguna *ranking function*. Una puntuación de un objeto actúa como una tasación para ese objeto conforme a sus características. Los objetos de datos son evaluados usualmente mediante varios predicados de puntuación que contribuyen a la puntuación total de los objetos.

6.1 Ejemplo de una Consulta de *Top-k*

El siguiente ejemplo ilustra un escenario real donde el proceso eficiente de *Top-k* es crucial. El ejemplo destaca la importancia de adoptar técnicas de procesos eficientes de *Top-k* en ambientes tradicionales de base de datos.

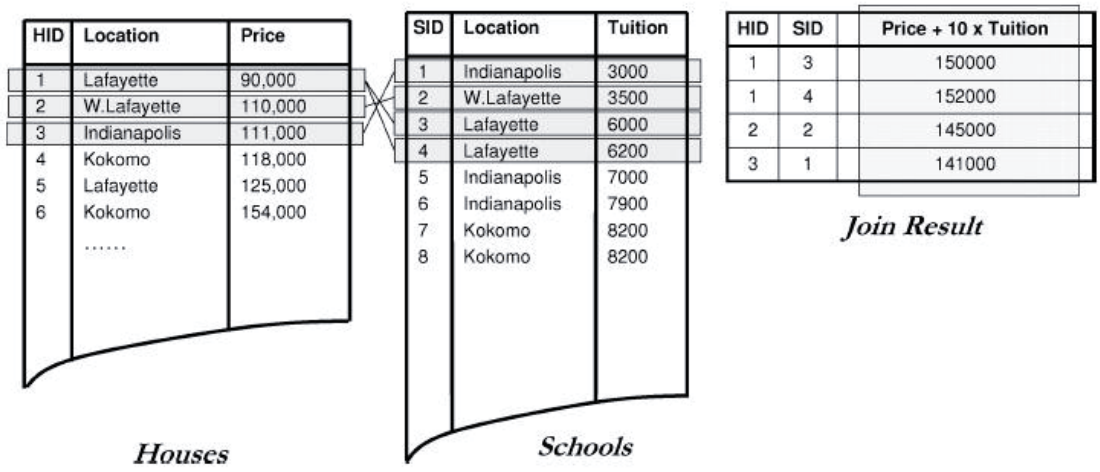

Figura 6.1 Ejemplo de Consulta de Top-K

En este ejemplo se considera la posibilidad de un usuario interesado en encontrar una ubicación (por ej.: ciudades) donde el costo combinado de comprar una casa y pagar la matrícula escolar de 10 años en ese lugar sea el mínimo. El usuario está interesado en los cinco lugares más baratos. Suponer que hay dos fuentes externas (bases de datos), una de casas y otra de escuelas, que pueden proporcionar información. La base de datos proporciona una lista de casas clasificadas desde las más baratas con sus ubicaciones. Del mismo modo, la base de datos de las escuelas proporciona un listado de las menos costosas

y sus ubicaciones. La Figura 6.1 muestra un ejemplo de las base de datos de casas y escuelas.

Estas listas se unen basándose en la ubicación de tal manera que el resultado esté compuesto de una casa y una escuela que se ubiquen en el mismo lugar. Los resultados de la unión de las listas es el costo total de cada par de casa-escuela, se calcula añadiendo el precio de la vivienda y el de matrícula escolar durante 10 años. Las cinco uniones más baratas constituyen la respuesta final a esta consulta. La Figura 8 también muestra una ilustración para el proceso de unión entre las casas y las escuelas listas, y los resultados parciales al combinarse. Hay que tener en cuenta que los primeros cinco resultados no pueden ser devueltos a los usuarios hasta que todos los resultados de la combinación se generen.

6.2 Ranking Function

La *Ranking Function* tiene propiedades que influyen de gran manera en el diseño de técnicas de procesamiento *Top-k*. Una propiedad importante es la capacidad de acotar por arriba las calificaciones de los objetos. Esta propiedad permite la poda temprana de ciertos objetos sin saber exactamente sus puntuaciones.

Una *ranking function* "monótona" puede facilitar en gran medida el cálculo del límite superior. Una función F, definido en predicados $P_1, \ldots, P_n$, es monótona si $\mathrm{F}(\mathrm{p}_1, \ldots, \mathrm{p}_\mathrm{n}) \leq \mathrm{F}(\mathrm{p}'_1, \ldots, \mathrm{p}_\mathrm{n})$ siempre que $\mathrm{p}_\mathrm{i} \leq \mathrm{p}'_\mathrm{i}$ para cada *i*.

En aplicaciones más complejas, una *ranking function* tiene que ser expresada como una expresión numérica para ser optimizada. En este contexto, la monotonía de la restricción de una *ranking function* se relaja para permitir más funciones genéricas. Las herramientas de optimización numérica, así como los índices (indicadores) se utilizan para superar los desafíos que impone el procesamiento de dicha *ranking function.*

Otro grupo de aplicaciones direcciona las categorías de objetos sin especificar una *ranking function*. En algunos entornos, como la exploración de los datos o la toma de decisiones, podría no ser importante rankear objetos basados en función de una *ranking function* específica. En su lugar, los objetos con alta calidad basados en diferentes atributos de datos deben ser reportados para futuros análisis. Estos podrían estar entre los *Top-k* objetos de algunas *ranking function* no especificadas. Este conjunto de objetos que no son "dominados" por otros, basados en algunos atributos dados, son generalmente denominados como *Skyline*.

6.2.1 Ranking Function Monótona

La mayoría de las técnicas de procesamiento actuales *Top-k* utilizan la *ranking function* monótona, ya que sirve en muchos escenarios prácticos, especialmente en

aplicaciones web. Por ejemplo, muchos escenarios de procesamiento en *Top-k* involucran combinaciones lineales de múltiples predicados de puntuación o funciones máximos/mínimos, las cuales son monótonas.

Las *Ranking Function* monótonas tienen propiedades especiales que pueden ser explotadas para un procesamiento eficiente de consultas *Top-k*. Al agregar puntuaciones de objetos desde múltiples listas rankeadas usando una función puntuadora de agregación monótona, una cota superior de las puntuaciones de los objetos invisibles se deriva fácilmente. Para ilustrar, supongamos que se exploran secuencialmente *m* listas ordenadas $L_1, \ldots, L_m$, para el mismo conjunto de objetos basados en diferentes tipos de clasificación. Si las puntuaciones de los últimos objetos recuperados de estas listas son $\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_m$, entonces la cota superior $\bar{F}$, sobre las puntuaciones de todos los objetos invisibles se calcula como $\bar{F} = F(\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_m)$.

Además, es fácil verificar que los objetos invisibles pueden posiblemente tener una puntuación superior a la anterior cota superior $\bar{F}$, por contradicción, supongamos que un objeto invisible o_u tiene una puntuación mayor que $\bar{F}$. Entonces, basados en la monotonicidad de *F*, o_u debe tener al menos un predicado p_i con puntuación superior al ultimo $\bar{p}_i$ visto. Esto implica que la recuperación de la lista L_i no está ordenada por puntuación, lo que contradice la hipótesis original.

Las funciones de puntuación monótonas permiten calcular una cota superior para cada objeto visto "o" substituyendo los valores de las predicados de puntuación desconocidos de "o" con las últimas puntuaciones vistas en las listas correspondientes. Un algoritmo de un proceso *Top-k* se detiene cuando hay k objetos cuyas cotas inferiores no están por debajo de la puntuación de los límites superiores de los demás objetos, incluyendo los objetos invisibles.

Una propiedad importante que se desprende de monotonicidad es que estos algoritmos son instancias óptimas, dentro de ciertos límites, en el sentido de que para cualquier instancia de base de datos, no hay un algoritmo que puede recuperar un número menor de objetos y devolver una respuesta correcta.

Las funciones lineales de puntuación constituyen un subconjunto de funciones monótonas. También definen la puntuación conjunta de los predicados como una suma ponderada. Varias técnicas de *Top-k* usan las propiedades geométricas de las funciones lineales para recuperar eficientemente las *Top-k* respuestas. Las funciones lineales de puntuación pueden ser representadas como vectores basados en los pesos asociados con los predicados de clasificación.

Esta representación puede ser utilizada para calcular geométricamente puntuaciones de objetos como proyecciones en el vector de *ranking function*.

6.3 Implementación del prototipo *Top-k* en la arquitectura de AS

En este capítulo se explicará cómo se desarrolló el prototipo de la *Choice Function Top-k*, que características posee y como fue el proceso para realizar el funcionamiento de la nueva *Choice Function* basada en *Top-k* en AS. En primer lugar, el prototipo debe contar con las estrategias de enumeración a evaluar y sus respectivos indicadores, los cuales estarán preestablecidos por el sistema.

Por defecto el sistema tendrá incorporado 4 *Choice Function* con sus respectivos indicadores, es decir el usuario tendrá la opción de elegir entre una de estas cuatro *Choice Function* que se dispondrán en el CheckBox que se presenta en la interfaz. Luego de elegir una de estas se realizara un análisis de correlación a los indicadores para evitar que se usen indicadores no útiles. Si existe correlación uno de los indicadores se desactivara y no se utilizara en el algoritmo para encontrar el *Top-k* de las estrategias.

Luego se realizará el método para rankear las estrategias llamado TopKChoiceFunction que será el encargado de obtener los puntajes de las estrategias utilizadas. Los indicadores se ponderarán con un peso, un valor alfa que estará establecido antes de la ejecución del programa, dichos valores son definidos más adelante (ver Tabla 7.2 y Tabla 7.3). Después la TopkChoiceFunction torna el ranking con las estrategias y realiza la elección de la mejor de estas que se usará en cada paso del problema.

A continuación se propone un pseudocódigo que representa lo anteriormente explicado:

Algoritmo Top_k_Choice

```
1. leer_estrategia_y_indicadores();
2. if AnalisisDeCorrelacion(indicadores) then
3.       ExecuteCorrelacion();
4. end if
5. ranking_estretegias[]=Top_K_Choice();
6.estrategiaElegida = seleccionarMejor(estrategiasEvaluadas);
7.return estrategia_elegida;
```

6.3.1 Análisis de Correlación [12]

El uso de múltiples indicadores en la *Choice Function* tiene un costo muy alto, debido a que es un problema combinatorio en sí mismo, es por esto que se utiliza el análisis de correlación para detectar que indicadores no son útiles porque se comportan como otro indicador, encontrar este problema ayuda a mejorar el rendimiento de la *Choice Function*.

Este análisis se encarga de encontrar pares o conjuntos de indicadores muy relacionados entre si, y por lo tanto equivalentes, es por esto que se usa el coeficiente de

correlación de Pearson, el cual es sensible a la relación lineal entre dos indicadores, que se obtiene dividiendo las dos variables (indicadores en el caso de nuestro proyecto) por el producto de su desviación estándar.

$$X,Y = correlacion(X,Y) = cov(X,Y)/\sigma x \sigma y.$$

La correlación de Pearson es 1 en el caso de que sea una perfecta correlación lineal positiva, en el caso contrario de una perfecta correlación negativa se demuestra con el -1, y los que están entre el 1y -1 indican el grado de dependencia lineal entre ellos.

En el caso del prototipo para evaluar si dos indicadores tienen un alto grado de correlación se mide desde al menos 0.9, entonces se escoge uno de los indicadores para descartarlo de la función en nuestro caso de la TopKChoiceFunction, que es cuando se analizan las estrategias.

6.4 Modificaciones a la herramienta

En la herramienta de AS se hicieron ciertas modificaciones, todo esto para integrar la nueva *Choice Function*. Las modificaciones más importantes que se están implementando son:

- Creación del método TopKChoiceFunction, en donde se manejará todo lo referente al manejo de la ranking function y todo lo referente a *Top-k*.
- Las modificaciones a la interfaz.

6.4.1 Método TopKChoiceFunction

El método TopKChoiceFunction será implementado en la clase indicadores.java que se encuentra en el *package* EclipseJava, en esta clase es donde están implementadas las demás *Choice Function* (*Skyline*, OneStrategy, etc). En esta clase también se modifican los indicadores, tanto los base como los calculados, que se obtienen a partir de los primeros en ECLiPSe.

Este método será ejecutado tantas veces como sea necesario para obtener la estrategia que se usará en cada paso (step) para encontrar una solución al problema.

- En una matriz se encuentran las estrategias y los indicadores de estas, en la primera columna de cada fila es representada la estrategia y las demás representan los indicadores.
- Para detectar los indicadores que no son útiles, es decir, que se comportan como otro indicador, se utiliza un análisis de correlación sobre estos. Si ocurre que algún indicador no es útil se desactiva, esto se hace mediante un arreglo en donde se indica mediante un booleano si el indicador está activo o no.

- Se ejecuta el algoritmo rankeador de las estrategias, se comparan estas mediante los valores de sus indicadores. Al final del rankeo, se elegirá la mejor estrategia para cada paso.

6.4.2 Prototipo de la *Choice Function* basada en *Top-k*

Para poder añadir la nueva *Choice Function* basada en *Top-k* que será implementada en la herramienta de AS fue necesario modificar un poco la interfaz para que el programa entregue la opción de utilizar esta nueva *Choice Function*.

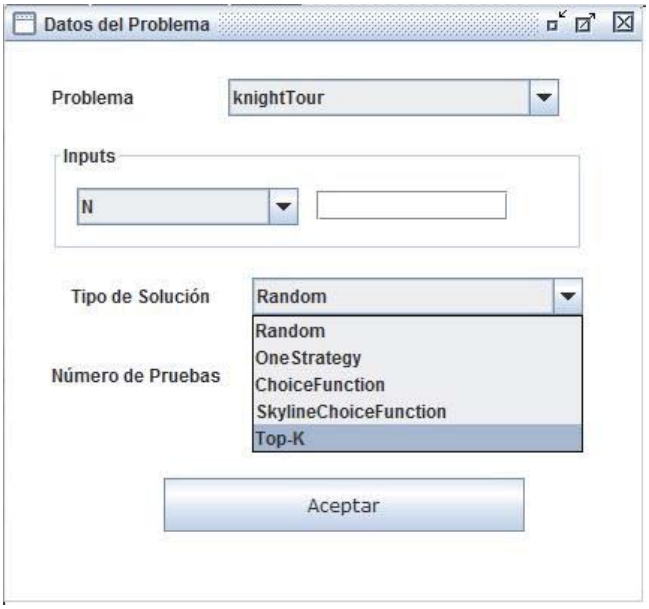

Figura 6.2 Ventana Datos del Problema

Primero que todo, en la primera ventana donde se especifican los Datos del Problema se agrega la opción de la nueva *Choice Function* en el *ComboBox* de "Tipo de Solución" (Figura 6.2).

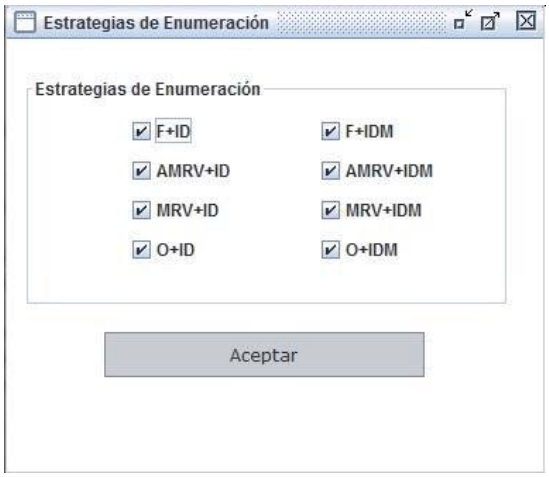

Figura 6.3 Ventana Estrategias de Enumeración

Después de haber llenado correctamente esta ventana aparecerá otra, la cual mostrará un conjunto de *CheckBox* en donde se listarán las 8 estrategias para resolver el problema (Figura 6.3). En el caso de la *Choice Function* nueva basada en *Top-k* el usuario debe marcar un mínimo de 2, de lo contrario no habrían las estrategias suficientes para rankear.

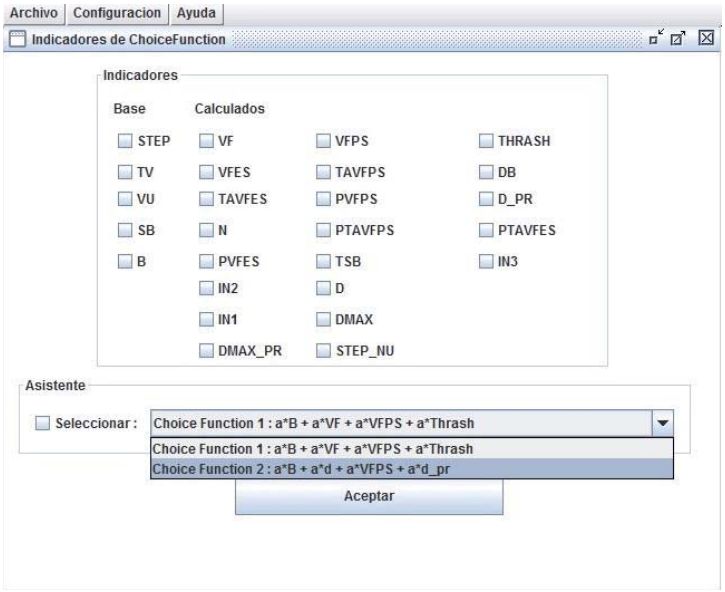

Figura 6.4 Ventana Indicadores de Choice Function

Después de elegir al menos 2 estrategias en la ventana de Estrategias de Enumeración, emergerá una nueva ventana donde se escogerán los indicadores que evaluará la *Choice Function*, por el momento se podrán elegir algunas *Choice Function* con sus indicadores predeterminados mediante un *ComboBox* (Figura 6.4).

Figura 6.5 Ventana Estado de la Ejecución

Por último, luego de elegir los indicadores con los que se evaluarán las estrategias saldrá una ventana en donde se podrá empezar la resolución del problema planteado con las especificaciones hechas en las ventanas anteriores (Figura 6.5).

6.4.2.1 Restricciones del Prototipo

En la implementación del prototipo, es necesario aplicar las siguientes restricciones para que funcione correctamente:

- En la ventana 2 de Estrategias de Enumeración, se deben seleccionar al menos 2 estrategias para que la función pueda hacer un rankeo.
- En la ventana 3 de Indicadores de la *Choice Function*, se deben seleccionar las *Choice Function* que aparecen en el asistente.

6.5 Ejemplo de Top-k en AS

Ya se realizó un ejemplo de consulta en Top-k utilizando una base de datos de casas y colegios, en este punto se explicará mediante un ejemplo cómo funciona Top-k en la herramienta de Autonomous Search.

Una vez abierta la herramienta se debe escoger un problema, además de las estrategias y la Choice Function que se usará para resolver este. Supongamos que se utiliza la Choice Function con los siguientes indicadores: *N – TV – VU* y se seleccionaron las 8 estrategias (detalladas en la Tabla 7.1). Para este ejemplo, el problema a resolver no es relevante.

AS resuelve los problemas paso a paso, es decir, en cada paso se evalúa y se usa la estrategia de mejor rendimiento. Para resolver los problemas, lo primero que hace el

algoritmo de Top-k es probar cada una de las estrategias y puntuarlas. O sea, en el primer utiliza y puntea la estrategia 1, en el segundo paso hace lo mismo con la estrategia 2, y así sucesivamente en los primeros 8 pasos. A medida que se van obteniendo los puntajes de las estrategias, Top-k va creando un ranking a partir de estos. Desde el paso 9 en adelante (con el ranking ya creado) el algoritmo elige la estrategia que está en la primera posición del ranking. Luego de ser usada la estrategia es evaluada nuevamente, se le asigna un nuevo puntaje y es reemplazada en el ranking a partir de la nueva puntuación.

A continuación se usará un ejemplo usando notas de 1 a 7 para que quede más claro. Al finalizar el paso 8 las puntuaciones de las estrategias fueron las siguientes (tabla de la izquierda, Figura 6.6), después de rankear las estrategias quedan ordenadas en la tabla de la derecha:

S1	5
S2	4
S3	2
S4	1
S5	3
S6	7
S7	5
S8	3

S4	1
S3	2
S5	3
S8	3
S2	4
S1	5
S7	5
S6	7

Figura 6.6 Ejemplo de puntuaciones de las estrategias al finalizar el paso 8.

Según el algoritmo, la estrategia que se utiliza en el paso numero 9 sería la numero 4, ya que se toma la estrategia con la menor nota. Una vez utilizada, es evaluada nuevamente y obtiene una nota 6, con esta nueva nota se reemplaza en el ranking, el cual quedaría de la siguiente manera:

S3	2
S5	3
S8	3
S2	4
S1	5
S7	5
S4	6
S6	7

Figura 6.7 Ejemplo de puntuaciones de las estrategias al finalizar el paso 9

Siguiendo el algoritmo, la estrategia utilizada en el siguiente paso sería la numero 3. En el caso de que al evaluar la estrategia, esta obtenga un puntaje menor al actual se mantendrá al tope del ranking y será utilizada en el siguiente paso. O sea, siguiendo la Figura 6.1, si la estrategia 3 obtiene una nota 1 esta se reemplaza por la anterior, pero de igual manera se mantiene en la primera posición del ranking. Esta forma de escoger las estrategias se seguirá utilizando hasta solucionar el problema.

7 Pruebas

En este proyecto se realizaron pruebas para evaluar la eficacia de la *Choice Function* basada en *Top-k* para AS. Las pruebas fueron realizadas en el entorno de desarrollo *NetBeans* con la herramienta AS que contiene la versión del solver *ECLiPSe 6.0*, en un computador con las siguientes características:

- Procesador Intel Core i3-2120 de 3.30 GHz.
- 4,00 GB de RAM.
- Disco Duro de 250 Gb.
- Sistema operativo: Windows 7 Professional de 32 bits.

7.1 Problemas

En las pruebas se utilizaron diferentes instancias de problemas los cuales son detallados a continuación:

- **N–Reinas:** Este problema consiste en situar *n* reinas sobre un tablero de ajedrez de *n x n*, sin que se amenacen entre ellas. Una reina amenaza a otra si se encuentran en la misma fila, columna o diagonal. Se hicieron pruebas con 8, 10, 12, 15, 20, 50 y 75 reinas.

- **Magic Square:** Son ordenaciones de números en celdas formando un cuadrado, de tal modo que la suma de cada una de sus filas, de cada una de sus columnas y de cada una de sus diagonales entregue el mismo resultado. Se hacen pruebas con *n* igual a 4 y 5.

- **Knight´s Tour:** Es un problema que consiste en posicionar un caballo en un tablero de ajedrez de *n x n*, el caballo es ubicado en el tablero y, moviéndolo de acuerdo a las reglas del ajedrez debe visitar cada casilla solo una vez. Se prueba con *n* igual a 5 y 6.

- **Sudoku:** Este problema consiste en rellenar una cuadrícula de 9x9 celdas (divididas en subcuadrículas de 3x3) con las cifras del 1 al 9, partiendo de algunos números ya dispuestos en las celdas, estos 9 elementos diferenciadores no se pueden repetir ni en la misma fila, columna o subcuadrícula. Se usó el *problema1* para las pruebas.

7.2 Estrategias

A continuación se mostrarán las estrategias utilizadas en las pruebas realizadas. Cada estrategia consta de una Heurística de Selección de Variable y de una Heurística de Selección de Valor. Las Heurísticas de Selección de Variable son:

- **First:** Escoge la primera variable de la lista.

- **AMRV:** Escoge la variable con el dominio más grande.
- **MRV:** Escoge la variable con el dominio más pequeño.
- **Occurence:** Escoge la variable con mayor cantidad de restricciones.

Las Heurísticas de Selección de Valor son:

- **Indomain:** Escoge el elemento más pequeño del dominio.
- **Indomain Max:** Escoge el elemento más grande del dominio.

Al hacer una combinación de las Heurísticas de Selección de valor con las de Selección de Variables nacen las estrategias que se muestran en la siguiente tabla:

Tabla 7.1 Estrategias utilizadas en la herramienta.

Estrategia	Heurísticas de Selección de Variable	Heurística de Selección de Valor
S1	First	Indomain
S2	AMRV	Indomain
S3	MRV	Indomain
S4	Occurence	Indomain
S5	First	Indomain Max
S6	AMRV	Indomain Max
S7	MRV	Indomain Max
S8	Occurence	Indomain Max

7.3 Resultados: Primeras Pruebas con *Top-k*

Los primeros resultados, utilizando la *Choice Function* basada en *Top-k* se obtuvieron realizando pruebas con las siguientes características:

- Sin conexión a internet, ni otros programas abiertos.
- Se realizaron 10 pruebas por cada problema, de los cuales se escoge el mejor tiempo, al cual se le realiza un análisis.
- En las pruebas se incluyeron todas las estrategias, y se utilizó la siguiente *Choice Function*:

$$N - TV - VU.$$

N representa los Nodos visitados, *TV* el número Total de Variables y *VU* las Variables No Instanciadas. Se usó esta *Choice Function* ya que se necesitaba una que tuviera los alfas de sus indicadores bien definidos, es decir, que la importancia de cada indicador estuviera precisada para cada problema a resolver. Los valores de los alfas de cada indicador fueron definidos en [13] y se muestran en las tablas a continuación.

Tabla 7.2 Mejores alfas de la *Choice Function* N-TV-VU para los problemas de N-reinas

	N-reinas						
	n=8	*n=10*	*n=12*	*n=15*	*n=20*	*n=50*	*n=75*
NODE	85,50416947	-28,5615149	-5,77211062	-94,75945846	-34,15023321	-6,17226325	-45,55026055
TV	-73,35176235	9,66339676	21,3137597	-4,67885834	92,48428876	66,9109334	71,13952890
VU	38,91520529	79,5551703	48,7837311	23,85410183	4,41454714	27,3191610	-64,61747428

Tabla 7.3 Mejores alfas de la *Choice Function* N-TV-VU para los problemas de Knight Tour, Magic Square y Sudoku

Magic Square		**Knight Tour**		**Sudoku**
n=4	*n=5*	*n=5*	*n=6*	*n=6*
17,63425316	44,98891819	95,54912708	-100,00000000	60,37186430
53,06369955	-79,28445160	-95,00557410	39,78389064	17,65122812
-100,00000000	-33,09528095	16,28990084	18,45969630	89,70424506

Con las pruebas realizadas se obtuvieron los siguientes resultados detallados en la Tabla 7.4 se detallan los tiempos (en segundos) que demoró cada una de las 10 pruebas en las distintas instancias de cada problema, se encuentran resaltados con negrita los tiempos más bajos en cada una de estas.

Tabla 7.4 Tiempos (en segundos) en las primeras utilizando la *Choice Function* Basada en *Top-k*

	N-reinas							**Knight Tour**		**Magic Square**		**Sudoku**
Prueba/N	***8***	***10***	***12***	***15***	***20***	***50***	***75***	***5***	***6***	***4***	***5***	***6***
1	0,047	0,125	**0,047**	1,529	**3,728**	OOM	OOM	131,52	140,86	0,216	93,549	55,659
2	0,124	0,125	0,187	**0,39**	4,087	OOM	OOM	135,94	**137,89**	0,23	93,716	61,165
3	0,125	0,14	0,171	1,638	4,04	OOM	OOM	137,18	140,57	0,136	95,367	55,849
4	**0,031**	0,047	0,188	0,468	4,196	OOM	OOM	137,64	150,78	**0,063**	106,24	60,441
5	0,109	0,063	0,187	1,435	4,586	OOM	OOM	138,2	153,03	0,224	96,395	57,897
6	0,125	0,032	0,187	1,497	3,9	OOM	OOM	138,03	154,26	0,246	93,627	55,875
7	0,109	**0,031**	0,078	1,466	3,838	OOM	OOM	132,37	152,02	0,229	98,624	53,904
8	0,109	0,032	0,047	0,561	4,103	OOM	OOM	133,05	155,23	0,235	95,435	54,666
9	0,125	0,031	0,203	0,562	4,134	OOM	OOM	135,1	152,91	0,261	**88,783**	53,574
10	0,109	0,125	0,171	0,39	4,337	OOM	OOM	**131,19**	156,59	0,232	95,583	**53,284**
Media	0,1013	0,0751	0,1466	0,9936	4,0949	-	-	135,02	149,41	0,2072	95,732	56,231

Al final de la tabla están los promedios de los tiempos en cada ensayo. En la Tabla 7.5 se muestran los indicadores arrojados en la prueba que tomó el menor tiempo de ejecución.

En los resultados de los tiempos de las pruebas realizadas en las *n-reinas* ocurre algo extraño, ya que con n=8 tarda más en resolverse que con n=10, pero desde n=10 y a medida que se aumenta la cantidad de reinas en la prueba el tiempo de resolución del problema aumenta. Algo similar ocurre en los problemas de *Magic Square* y *Knigth Tour*; aunque en el primero el tiempo promedio aumenta considerablemente cuando *n* es igual a 5. En esta primera fase de pruebas no se lograron obtener resultados para los problemas de N-reinas con n = 50 y n = 75.

Tabla 7.5 Indicadores arrojados en el mejor tiempo de cada problema utilizando la *Choice Function* Basada en *Top-k*.

	N-reinas							**Knight Tour**		**Magic Square**		**Sudoku**
Indicador/N	8	10	12	15	20	50	75	5	6	4	5	6
Step	14	15	27	143	924	-	-	22295	23363	25	18940	6823
B	7	6	13	73	476	-	-	8156	7099	15	11991	2789
Node	30	39	86	383	2901	-	-	142993	138130	212	152310	16327

Viendo la tabla de indicadores se puede ver algo similar a lo que sucede con los tiempos. En los problemas de *N-reinas* y de *Magic Square* aumenta la cantidad de nodos visitados a medida que se aumenta el valor *n*, al contrario de lo que sucede con *Knight Tour* en donde n=5 se visitan casi 143.000 nodos y con n=6 se visitan poco más de 138.000.

7.4 Resultados: Segundas Pruebas con *Top-k*

Con las primeras pruebas no se lograron obtener resultados en los problemas de N-reinas con n=50 y n=75. Es por esto que se decidió testear nuevamente *Top-k* con el fin de poder obtener datos para estos problemas, de esta manera poder hacer una comparación completa con otras *Choice Function*.

Analizando los resultados de las primeras pruebas, y viendo cuales son las estrategias que tienen peor rendimiento, se decidió sacar las 3 primeras de la Tabla 7.1. Con las pruebas realizadas se obtuvieron los tiempos que se muestran en la Tabla 7.6.

En las segundas pruebas en las distintas instancias de cada problema, se encuentran resaltados con negrita los tiempos más bajos en cada uno. En la última fila de la tabla están los promedios de los tiempos. A continuación, se muestran los indicadores arrojados en la prueba que tomó el menor tiempo de ejecución.

Tabla 7.6 Tiempos (en segundos) en las segundas pruebas utilizando la *Choice Function* Basada en *Top-k*

	N-reinas							**Knight Tour**		**Magic Square**		**Sudoku**
Prueba/N	***8***	***10***	***12***	***15***	***20***	***50***	***75***	***5***	***6***	***4***	***5***	***6***
1	**0,02**	0,042	0,091	3,529	13,593	OOM	144,956	OOM	OOM	0,438	59,486	**113,017**
2	0,021	0,031	0,083	3,829	0,098	2,605	OOM	OOM	OOM	0,067	62,749	113,225
3	0,044	0,044	0,084	3,552	0,128	2,797	OOM	OOM	OOM	0,405	58,583	113,955
4	0,035	0,036	0,087	0,037	0,098	2,895	118,161	OOM	OOM	**0,048**	**57,415**	114,49
5	0,022	0,037	0,059	0,034	0,098	2,816	OOM	OOM	OOM	0,292	75,377	113,541
6	0,043	**0,03**	0,062	0,034	0,1	2,752	116,913	OOM	OOM	0,474	67,247	112,556
7	0,047	0,033	0,06	0,035	0,112	2,922	**104,936**	OOM	OOM	0,06	59,085	111,897
8	0,046	0,034	0,057	0,03	0,098	**2,477**	OOM	OOM	OOM	0,421	64,445	114,157
9	0,042	0,028	**0,05**	**0,028**	0,098	2,555	110,887	OOM	OOM	0,056	62,32	**111,906**
10	0,045	0,031	0,057	0,032	**0,096**	2,494	117,479	OOM	OOM	0,536	66,648	113,628
Media	0,0365	0,0346	0,069	1,114	0,1029	2,7014	142,6660	-	-	0,2797	63,3355	113,237

En estas pruebas ocurre algo similar que en las anteriores con el problema de N-Reinas, con n=8 se tarda un poco más en resolverse que con n=10. Desde ahí en adelante el tiempo aumenta hasta llegar a n=75, a excepción del problema resuelto con n=20, donde el tiempo promedio de resolución es menor a n=15. Entre las pruebas de n=50 y n=75 hay una gran diferencia entre los promedios de tiempo, mientras la primera se demora en promedio casi 3 segundos, el mismo resuelto con el *n* mayor tarda casi 2 minutos.

Tabla 7.7 Indicadores arrojados en el mejor tiempo de cada problema utilizando la *Choice Function* basada en *Top-k*.

	N-reinas							**Knight Tour**		**Magic Square**		**Sudoku**
Indicador	**8**	**10**	**12**	**15**	**20**	**50**	**75**	**5**	**6**	**4**	**5**	**6**
Step	12	15	24	11	39	505	16688	-	-	14	7674	10354
B	5	6	11	1	11	177	7914	-	-	8	37241	4236
Node	27	39	82	15	72	1017	47714	-	-	77	5231	27197

En esta instancia de las pruebas no se pudieron obtener resultados para ninguno de los *n* en Knight Tour. En el problema de Magic Square se redujo el tiempo considerablemente con n=5 en comparación a las primeras pruebas, con n=4 el tiempo promedio de resolución es similar. En el problema de sudoku, el tiempo que tarda en resolver los problemas es casi el doble.

Analizando los indicadores, la gran diferencia entre ambas pruebas se nota en los problemas más grandes en resolver: N-reinas con n=20, 50, 75, Magic Square n=5, por

ejemplo. En general, las segundas pruebas (sin usar todas las estrategias) el algoritmo recorre menos nodos y hace menos Backtracks que en las primeras, lo que conlleva tener un menor tiempo promedio de resolución.

Además, en estas pruebas se lograron obtener resultados a los problemas que no se pudieron conseguir en las primeras. De esta manera se puede realizar una comparación de los tiempos e indicadores arrojados en todos los problemas con los de otras *Choice Function*, en este caso la comparación se hará con *Skyline* [2].

7.5 Terceras Pruebas y Comparación con *Skyline*

Con el fin de poder hacer comparaciones más concretas, se resolvieron con la *Choice Function* basada en *Skyline* los mismos problemas resueltos con el algoritmo basado en *Top-k*. Para la resolución de los problemas, *Skyline* utiliza 5 variantes de la *Choice Function* con distintos indicadores que se encuentran detallados en su informe [2].

En la siguiente tabla se detallan las *Choice Function* utilizadas en *Skyline* y sus indicadores.

Tabla 7.8 *Choice Function*s utilizadas en *Skyline*

Sigla	**Indicadores**
CF1	B – Step – PTAVFES
CF2	SB – In1 – In2
CF3	B – Step – In1 – In2
CF4	VF – Dmax – DB
CF5	N – VU – VFPS - Thrash

Para esto se utilizó en todas las pruebas el ambiente detallado al comienzo de este capítulo, excepto para el N-reinas con n=50 y n=75, para estas instancias del problema se usaron los resultados obtenidos en [2] al no conseguirlos con la maquina usada en las demás pruebas. Las características del computador utilizado en esas pruebas se detallan a continuación:

- Procesador Intel (R) Core (TM) 2 Duo CPU P8700 2.53 GHz.
- 4,00 GB de RAM.
- Disco Duro de 320 Gb.
- Sistema operativo: Windows Vista 64 bits.

A continuación, se encuentra la tabla con los mejores tiempos de resolución (en segundos) de los problemas resueltos con *Skyline*.

Tabla 7.9 Tiempo de resolución en segundos de los problemas usando *Skyline*

	N-Reinas							**Knight Tour**		**Magic**		**Sudoku**
	8	**10**	**12**	**15**	**20**	**50**	**75**	**5**	**6**	**4**	**5**	**6**
CF1	**0,004**	0,013	**0,003**	**0,018**	**0,265**	-	-	-	-	**0,006**	15,201	-
CF2	0,09	0,11	0,1	0,12	0,3	**23,041**	**9,286**	-	-	0,093	**0,405**	13,369
CF3	0,007	**0,007**	0,02	0,062	0,378	-	-	-	-	0,021	8,774	-
CF4	0,008	0,013	0,018	0,088	0,316	-	-	**21,575**	91,832	0,083	0,572	**6,349**
CF5	0,009	0,008	0,011	0,052	0,286	-	-	146,07	**86,997**	0,023	10,023	14,911

7.5.1 Resolución de Problemas en *Top-k* con las *Choice Function* utilizadas en *Skyline*

Al realizar las comparaciones entre *Skyline* y *Top-k* se puede notar que no se utilizaron las mismas *Choice Function* en estas. Como se dijo anteriormente, *Skyline* utiliza las que se detallan en la Tabla 7.8 al comienzo de esta sección; y *Top-k* usa la *Choice Function*: *N-TV-VU*. Es por esto que para poder obtener resultados y un análisis más consistente, se decidió volver a realizar las pruebas con *Top-k*, pero utilizando las *Choice Function* usadas en *Skyline*.

En *Skyline* se resolvieron los mismos problemas que con *Top-k*, en cada problema hubo una Choice Function que resolvió las pruebas en un tiempo menor a las otras. En la última fila de la Tabla 7.10 se indica el número de la Choice Function que tuvo el mejor rendimiento, y que finalmente fue usada en *Top-k* para resolver el problema. Se realizaron 5 pruebas por cada problema.

Tabla 7.10 Tiempos de resolución en segundos de problemas en *Top-k* con las *Choice Function* utilizadas en *Skyline*

	N-reinas							**Knight Tour**		**Magic Square**		**Sudoku**
Prueba/N	**8**	**10**	**12**	**15**	**20**	**50**	**75**	**5**	**6**	**4**	**5**	**6**
1	0,015	0,015	0,125	**0,005**	0,03	1,53	-	-	-	0,125	**1,749**	-
2	0,01	0,01	0,025	0,15	0,025	**0,665**	-	-	-	0,135	5,974	-
3	0,01	0,01	0,02	0,005	0,025	0,685	-	-	-	**0,005**	2,602	-
4	**0,005**	**0,005**	**0,015**	0,005	**0,02**	0,705	-	-	-	0,005	2,677	-
5	0,005	0,005	0,02	0,005	0,03	0,695	-	-	-	0,01	2,55	-
CF	1	3	1	1	1	2	2	4	5	1	2	4

Se concluyó que se debían eliminar los pesos alfas de los indicadores. El detalle era que les restaban eficiencia y rapidez a la resolución. Y como *Skyline* no trabaja con pesos (alfas) se decidió seguir la misma línea.

7.5.2 Análisis

Habiendo recopilado los datos de los tiempos de resolución en cada problema y con distintas *Choice Function* se pueden hacer mejores comparaciones y de esta manera realizar un análisis más consistente. En la Tabla 7.11 se detallan los mejores tiempos de resolución en cada instancia de los problemas resueltos.

Nota: la organización de las siguientes tablas se explica a continuación. La primera columna muestra con que algoritmo se resolvió el problema: las primeras 5 filas (CF1, CF2, CF3, CF4 y CF5) son los problemas resueltos utilizando las distintas *Choice Function* de *Skyline*. Las siguientes filas detallan los problemas resueltos con *Top-k* y sus distintas instancias (Primeras Pruebas, Segundas Pruebas y la tercera utilizando la *Choice Function* en la que *Skyline* tuvo mejor tiempo). En las últimas pruebas de *Top-k* se usó una *Choice Function* distinta para cada problema, esta es detallada en la penúltima fila de la tabla.

Finalmente, en las siguietnes tablas hay un número para cada problema al final de esta, este número representa la diferencia entre el menor tiempo de resolución de *Skyline* y el menor tiempo en *Top-k* (ambos marcados con negrita). Cuando este número es negativo es cuando el tiempo de *Skyline* fue mejor que el de *Top-k*, lo contrario ocurre cuando ese número es positivo.

Tabla 7.11 Mejores tiempos de resolución de cada problema resuelto con las distintas *Choice Function*

	N-Reinas							**knight Tour**		**Magic**		**Sudoku**
	8	**10**	**12**	**15**	**20**	**50**	**75**	**5**	**6**	**4**	**5**	**6**
CF1	**0,004**	0,013	**0,003**	**0,018**	**0,265**	-	-	-	-	**0,006**	15,201	-
CF2	0,09	0,11	0,1	0,12	0,3	**23,041**	**9,286**	-	-	0,093	**0,405**	13,369
CF3	0,007	**0,007**	0,02	0,062	0,378	-	-	-	-	0,021	8,774	-
CF4	0,008	0,013	0,018	0,088	0,316	-	-	**21,575**	91,832	0,083	0,572	**6,349**
CF5	0,009	0,008	0,011	0,052	0,286	-	-	146,073	**86,997**	0,023	10,023	14,911
Top-k (1)	0,02	0,03	0,05	0,028	0,096	-	-	**131,92**	**156,589**	0,048	57,415	**38.017**
Top-k (2)	0,031	0,031	0,047	0.39	3.728	2,477	**104,936**	-	-	0,232	95,583	53,284
Top-k (3)	**0,005**	**0,005**	**0,015**	**0,005**	**0,02**	**0,665**	-	-	-	**0,005**	**1,749**	-
	CF1	CF3	CF1	CF1	CF1	CF2	CF2	CF4	CF5	CF1	CF2	CF4
Δ	-0,001	0,002	-0,012	0,013	0,245	22,376	-95,65	-110,345	-69,592	0,001	-1,344	-38.011

Para las siguientes tablas se tomó el menor tiempo de resolución de cada problema, a partir de esto se obtuvieron los datos de los indicadores arrojados. Estos datos se detallan en las tablas a continuación.

Tabla 7.12 Cantidad de Steps realizados en el problema con el mejor tiempo resuelto con las distintas *Choice Function*

	N-Reinas							**Knight Tour**		**Magic**		**sudoku**
	8	**10**	**12**	**15**	**20**	**50**	**75**	**5**	**6**	**4**	**5**	**6**
CF1	**10**	36	**6**	**33**	378	-	-	-	-	**6**	8331	-
CF2	**10**	33	**6**	37	289	NN	NN	-	-	**6**	**347**	10367
CF3	15	**15**	33	143	378	-	-	-	-	21	8662	-
CF4	17	20	35	157	302	-	-	28353	65535	69	599	**6798**
CF5	18	15	34	128	**290**	-	-	140341	99480	21	7674	10354
Top-k (1)	**12**	15	**24**	11	39	505	16688	-	-	14	7674	**3734**
Top-k (2)	14	15	27	143	924	-	-	22295	23363	25	18940	6823
Top-k (3)	18	**12**	34	**12**	**39**	**505**	-	-	-	**10**	**901**	-
Δ	-2	3	-18	21	251	-	-	6058	42172	-4	-554	3065

Tabla 7.13 Cantidad de Backtracks realizados en el problema con el mejor tiempo resuelto con las distintas *Choice Function*

	N-Reinas							**Knight Tour**		**Magic**		**sudoku**
	8	**10**	**12**	**15**	**20**	**50**	**75**	**5**	**6**	**4**	**5**	**6**
CF1	**5**	20	**0**	**13**	200	-	-	-	-	**1**	5881	-
CF2	5	20	0	15	123	**5025**	**1255**	-	-	**1**	**207**	4226
CF3	8	**6**	15	73	200	-	-	-	-	10	6047	-
CF4	10	12	15	88	152	-	-	**12337**	**23756**	46	427	**2744**
CF5	10	6	15	58	**122**	-	-	48193	35053	10	5231	4236
Top-k (1)	5	6	**11**	**1**	11	177	**7914**	-	-	8	37241	9361
Top-k (2)	7	6	13	73	476	-	-	**8156**	**7099**	15	11991	**2789**
Top-k (3)	**2**	**4**	15	2	**11**	**177**	-	-	-	**3**	**597**	-
Δ	**3**	**2**	-11	12	**111**	**4848**	-6659	4181	16	-2	-397	-45

Tabla 7.14 Cantidad de Nodos Visitados en el problema con el mejor tiempo resuelto con las distintas *Choice Function*

	N-Reinas							**Knight Tour**		**Magic**		**sudoku**
	8	**10**	**12**	**15**	**20**	**50**	**75**	**5**	**6**	**4**	**5**	**6**
CF1	**23**	103	**7**	91	1200	-	-	-	-	**22**	49342	-
CF2	26	91	**7**	**83**	**660**	**25368**	**6601**	-	-	**22**	**1464**	27052
CF3	34	**39**	97	383	1200	-	-	-	-	200	50196	-
CF4	42	72	101	505	925	-	-	**185067**	**428103**	618	3300	**17275**
CF5	42	39	88	335	704	-	-	824376	653184	200	37241	27197
Top-k (1)	27	39	**82**	**15**	72	1017	**47714**	-	-	**25**	**5231**	**1430**
Top-k (2)	30	39	86	383	2901	-	-	**142993**	**138130**	212	152310	16327
Top-k (3)	**10**	**21**	88	21	**72**	**1017**	-	-	-	30	5479	-
Δ	13	18	-75	68	588	24351	-41113	42074	289973	-3	-3767	15845

Se observa que los tiempos de resolución de Top–k, en esta tercera y última prueba disminuyeron con respecto a las anteriores de manera notable, esta mejora se debe a los

cambios que realizamos en el *Choice Function*, los cuales fueron nombrados anteriormente además se elimina el análisis de correlación, que se aplicaba a los indicadores antes de generar las comparaciones y posterior rankeo, porque claramente no agilizaba los tiempos eliminado lo indicadores no útiles, si no que generaba un aumento de los tiempos de resolución de los problemas.

Por otra parte en los problemas con los *"n"* más pequeños los tiempos de resolución fueron similares a los de *Skyline*, incluso se tuvieron mejores tiempo como en el caso de las n= 10, 15, 20 y 50 siendo los problema de las 20 y 50 reinas con el que se logró la mayor diferencia en la tiempos de resolución. Con respecto a los otros problemas se aprecia que cuando la complejidad crece, como es el caso de Knight Tour, Magic Square y Sudoku, *Skyline* tiene un mejor desempeño en los tiempos de resolución en comparación con *Top-k*.

Aunque se debe destacar que *Top-k* se comporta de mejor manera con los problemas de N-reinas que *Skyline*, pero sin desmerecer que en los otros problemas se destaca *Skyline*. Por este motivo no se puede decir que *Choice Function* es la mejor, ya que tiene un rendimiento relativamente similar.

Al analizar los resultados de los indicadores, tanto para el caso de los backtracks realizados y nodos visitados en cada problema no se observa un dominio claro de una *Choice Function* u otra si hablamos de eficiencia. En algunos problemas como en Knight Tour y en la mayoría de las instancias del N-reinas el algoritmo de Top- K plasma una menor cantidad de Backtracks y visita menos nodos, esto demuestra lo anteriormente descrito sobre el comportamiento de Top–k y sus resultados en los problemas de N-Reinas. En los demás problemas es mucho más eficiente *Skyline* si analizamos los datos arrojados por esto indicadores.

Para esta última prueba los resultados reflejados siguen mostrando una leve ventaja de *Skyline* sobre *Top-k*, aunque no son suficientes para decir que una es mejor que la otra porque tienen un comportamiento muy similar y hay resultados en que Top–k muestra tiempos mucho más eficientes que *Skyline*.

8 Conclusiones

En este proyecto se ha investigado y analizado cada uno de los componentes de *Autonomous Search* con el propósito de generar una *Choice Function* basada en el algoritmo de rankeo *Top-k*. También se tuvo que estudiar y comprender las características de la programación con restricciones, junto con sus metodologías de resolución de problemas, sobre como usan los árboles de búsqueda y las estrategias de enumeración, que corresponde a una parte fundamental del Solver para el posterior desarrollo del proyecto.

También se han estudiado los demás componentes, en especial la "actualización" que es en donde se aplica la *Choice Function*, se analizó su funcionamiento que es el que permite capturar el comportamiento de las estrategias para lograr luego el objetivo del algoritmo que es generar un ranking con estas.

Por otro lado se ha investigado sobre *Top-k*, que es una técnica de rankeo la cual cuenta con distintos métodos de procesado. Entre estas se encuentra una "ranking function", la que fue utilizada en este proyecto como base para formular e implementar una *Choice Function* que discriminará las estrategias según sus rendimientos. Se ha modificado la herramienta de *Autonomous Search* creada en trabajos pasados para poder integrar la función basada en *Top-k*.

Nos hemos apoyado en estudios anteriores para poder complementar el algoritmo de *Top-k*, usando datos como las mejores alfas de las estrategias de numeración, que se obtuvieron en el optimizador de QPSO.

Con el primer prototipo de *Choice Function* terminado, se realizaron las pruebas suficientes como para generar un análisis. De esta manera poder determinar si el algoritmo de *Top-k* mejora los tiempos de resolución de problemas comparando sus resultados con los de otras formas de resolución. Analizando aún más a fondo se pudo saber si esta función es eficaz a la hora de elegir las estrategias en cada paso de resolución del problema.

Para la parte final se hizo una comparación con el algoritmo *Skyline* en la cual se tomaron los mejores tiempos en cada uno de los problemas de satisfacción, además de registrar y evaluar cómo funciona los indicadores más importantes que en este caso son Steps, backtracks y Nodos.

Con el fin de tener una comparación realmente consistente, se realizaron las pruebas en *Top-k* con las mismas *Choice Function* utilizadas en *Skyline*. Así se pudo determinar que con la utilización de *Choice Function*s específicas para determinados problemas los tiempos de resolución mejoraban notablemente. Es decir, para poder obtener mejores resultados se debe determinar para cada problema la *Choice Function* que obtenga el mejor rendimiento.

Se puede concluir que el proyecto fue realizado con éxito, con los resultados obtenidos en las pruebas podemos decir que *Top-k* no mostró una mejora significativa en los tiempos resolución de problemas si lo comparamos con *Skyline*, los tiempos son relativamente parejos. En cuanto a eficacia el algoritmo de *Top-k* pudo resolver más problemas de los que solucionó *Skyline* con cada una de las 5 instancias de la *Choice Function*.

En resumen podemos decir que *Top-k* es más eficiente que *Skyline* ya que mostró mejores resultados al observar la cantidad de problemas resueltos, pero no es tan eficiente en los tiempos de resolución comparándolo con el otro algoritmo de rankeo. Es decir, *Top-k* resuelve más problemas, pero *Skyline* los resuelve en menos tiempo.

Finalmente, se puede decir que tanto el algoritmo de rankeo como la herramienta donde fue implementado pueden seguir siendo evaluados, se puede continuar realizando pruebas y a partir de esto mejorarse y pulirse. De esta manera se deja la puerta abierta para que personas capacitadas puedan seguir trabajando en esta área y así mejorar el rendimiento de *Autonomous Search* y de la *Choice Function* basada en *Top-k*, entre otras.

Referencias

[1] Víctor Bustos Allendes, *"Una Arquitectura Modular para Autonomous Search"*, Informe final de proyecto para optar al título de ingeniero de ejecución informática, Pontificia Universidad Católica de Valparaíso, 2012.

[2] Karin Elizabeth Galleguillos Torres, *"Choice Function basada en Skyline para Autonomous Search"*, Informe final de proyecto para optar al título de ingeniero de ejecución informática, Pontificia Universidad Católica de Valparaíso, 2013.

[3] Irene Barba Rodríguez, *"Algoritmos de planificación basados en restricciones para la sustitución de componentes defectuosos"*, Informe final para grado de Ph.D. en Ingeniería Informática, Universidad de Sevilla, 2008.

[4] Marlene Arangú Lobig, *"Modelos y técnicas de consistencia en Problemas de Satisfacción de Restricciones"*, Tesis para el grado de obtención del grado de Doctor en informática, Universidad Politécnica de Valencia, 2011.

[5] Roman Barták, Miguel A. Salido, Francesca Rossi, *"Constraint satisfaction techniques in planning and scheduling"*,publicación online, University of Padova , 2010.

[6] Eugene Freuder, *"A sufficient condition for backtrack-free search"*, Journal of the Association for Computing Machinery, Vol. 29, University of New Hampshire, 1982.

[7] Rina Dechter and Itay Meiri, *"Experimental evaluation of preprocessing algorithms for constraints satisfaction problems"*, publicado por Elsevier B.V., University of California, 1994.

[8] Paul Walton Purdom, *"Search rearrangement backtracking and polynomial average Time"*, investigación para el departamento de ciencias de la computación, Indiana University, 1983.

[9] Anatoli Koulinitch, Wendy García Martínez *"Análisis De Sistemas De Programación Lógica por Restricciones"*, Tesis profesional para tener el título de Ingeniero en Computación, Universidad Tecnológica de la Mixteca, 2006.

[10] Krzysztof Apt and Mark Wallace, *"ECLiPSe CLP-Constraint Logic Programming"*, Universidad de Granada, Open-sourced (MPL) by Cisco 9/2006.

[11] Ihab Ilyas, George Beskales, y Mohamed Soliman, *"A Survey of Top-k Query Processing Techniques in Relational Database Systems"*, Publicación de ACM Computing Surveys, Vol. 40, University of Waterloo, pp. 1-5 y 42-45, 2008.

[12] Broderick Crawford, Carlos Castro, Eric Mofroy, Ricardo Soto, Wenceslao Palma and Fernando Paredes, *"Dynamic Selection of Enumeration Strategies for Solving Constraint*

Satisfaction Problems", Romanian Journal of Information Science And Technology Vol. 15, pp. 106-128, 2012

[13] Diego Andrés Moraga Lippians, Roberto Ignacio Morales Aguirre, *"Optimizador de Parámetros basado en QPSO para Autonomous Search"*, Informe final para proyecto II, Pontificia Universidad Católica de Valparaíso, 2012.

[14] Ricardo Soto, Hakan Kjellerstrand, Juan Gutierrez, Alexis Lopez, Broderick Crawford, and Eric Monfroy. *"Solving Manufacturing Cell Design Problems Using Constraint Programming",* In proceedings of the 25th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE), pp. 400-406, Springer, 2012.

[15] Broderick Crawford, Carlos Castro, Eric Monfroy, Ricardo Soto, Wenceslao Palma, and Fernando Paredes. *"Hyperheuristic Approach for Guiding Enumeration in Constraint Solving",* EVOLVE - A Bridge between Probability, SetOriented Numerics, and Evolutionary Computation II, Advances in Intelligent Systems and Computing (AISC), Vol. 175, pp. 171-188, Springer, 2013.

[16] Ricardo Soto, Broderick Crawford, Eric Monfroy, and Victor Bustos. *"Using Autonomous Search for Generating Good Enumeration Strategy Blends in Constraint Programming"*. In proceedings of the 12th International Conference on Computational Science and Its Applications (ICCSA), pp. 607-617, Springer, 2012.

[17] Renzo Pizarro, Gianni Rivera. *"Modelado y resolución del Nurse Rostering Problem (NRP) utilizando programación con restricciones: un caso de estudio"*, Informe Final para optar al título de Ingeniero de Ejecución Informática. Pontificia Universidad Católica de Valparaíso, 2011.

[18] Ivan Edward Sutherland. *"Sketchpad: A man-machine graphical communication system"*. Technical Report, Computer Laboratory, University of Cambridge. 2003.

[19] Youssef Hamadi, Eric Monfroy, Frédéric Saubion. *"What is Autonomous Search?"* Springer, 2008.

[20] Kam Pui Chow, *"Airport counter allocation using constraint logic programming"*, In Proc. of Practical Application of Constraint Technology, 19