PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA INFORMÁTICA

CHOICE FUNCTION BASADA EN SKYLINE PARA AUTONOMOUS SEARCH

KARIN ELIZABETH GALLEGUILLOS TORRES

INFORME FINAL DE PROYECTO PARA OPTAR AL TÍTULO

PROFESIONAL DE INGENIERO DE EJECUCIÓN INFORMÁTICA

Mayo 2013

PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA INFORMÁTICA

CHOICE FUNCTION BASADA EN SKYLINE PARA AUTONOMOUS SEARCH

KARIN ELIZABETH GALLEGUILLOS TORRES

Profesor Guía: **Ricardo Soto de Giorgis**

Profesor Co-referente: **Wenceslao Palma Muñoz**

Carrera: **Ingeniería de Ejecución en Informática**

Mayo 2013

Dedicatoria

Para todos los que me apoyaron,
especialmente a mi madre y Martín.

Índice

Índice de Figuras

Índice de Tablas

Resumen

Autonomous Search (AS) dentro de la programación con restricciones provee la habilidad de reemplazar en forma dinámica estrategias de bajo desempeño por otras más prometedoras. La idea es agilizar los tiempos de computación en la resolución de problemas de satisfacción de restricciones. El reemplazo de estrategias es llevado a cabo en base a un ranking de calidad, el cual es calculado por medio de una función de selección (choice function). Esta función determina el rendimiento de una estrategia en un tiempo determinado, en base a un conjunto de indicadores y parámetros de control. En el presente proyecto se propone el diseño e implementación de una nueva choice function. Esta nueva choice function emplea una técnica de rankeo llamada Skyline, la cual es ampliamente utilizada en el área de las bases de datos. Se ilustran resultados experimentales que demuestran la factibilidad de uso de esta propuesta utilizando como benchmark los problemas n-reinas, sudoku, magic squares y knight tour.

Palabras clave: Autonomous Search, Programación con restricciones, Choice Function, Skyline.

Abstract

Autonomous Search within constraint programming provides the ability of dynamically replacing low-performance strategies by more promising ones. The idea is to speed-up solving times in the resolution of constraint satisfaction problems. The replacement of strategies is carried out depending on a quality rank, which is calculated by means of a choice function. This function determines the performance of a strategy in a given amount of time via a set of indicators and control parameters. In the present project we propose to design and implement a new choice function. This new choice function employs a rank technique named Skyline, which is widely used in the database area. We illustrate experimental results that demonstrates the feasibility of the proposed approach by using as benchmark the n-queens, sudoku, magic squares and knight tour problems.

Keywords: Autonomous Search, Constraint Programming, Choice Function, Skyline.

1. Introducción

Existe una rama de la investigación informática que estudia la resolución de problemas con la programación con restricciones que se define como un paradigma donde un problema, llamado problema de satisfacción de restricciones se describe y resuelve.

Para obtener la solución de un problema de satisfacción de restricciones es necesario describir del problema sus variables dominios y restricciones. El desarrollo de la solución es complejo ya que son necesarios: algoritmos de búsqueda, que generan un árbol de búsqueda de acuerdo a la asignación entre valores y variables y que cumpliendo las restricciones generan una solución al problema. El proceso de asignación o enumeración es aquel que se lleva gran parte del trabajo, ya que existen diversas heurísticas de valor y variable cuya combinación formará las estrategias de enumeración que son de gran importancia en la solución del problema y que según su comportamiento obtendrán mejor calidad de resultados (rapidez, menos vueltas atrás, etc.), si éstas son elegidas correctamente. Con el objetivo de que el desarrollo de la solución ocurra con mejor rendimiento existe Autonomous Search que posee una arquitectura donde cada uno de sus componentes juega un papel importante para cumplir ese objetivo. Cada componente posee información actualizada cada vez, uno de estos componentes posee una choice function. La choice function es la que da una prioridad a cada estrategia, dada esta misma se elegirá la siguiente para continuar con la solución.

En este trabajo se aborda el desarrollo de una choice function basada en la metodología Skyline. Ésta metodología basa la toma de decisiones en un criterio claro y definido. Además posee una acotada cantidad de algoritmos para realizar su cálculo, los que proveen la información suficiente para elegir una opción del universo del Skyline que se está calculando, entre algunos de estos algoritmos se encuentran Block Nested Loops y Divide and Conquer.

La nueva choice function implementa Skyline para elegir una estrategia a utilizar en cada paso del desarrollo del problema y a través de las pruebas realizadas se presentan las características de su rendimiento, lo que se puede analizar a través del valor de los indicadores utilizados, así como también del tiempo de ejecución. Las pruebas fueron realizadas sobre problemas como: sudoku, n-queens, magic squares entre otros, estas pruebas dan un primer acercamiento sobre sus rendimientos, teniendo en cuenta algunos indicadores importantes como lo son el número de backtracks y la cantidad de nodos visitados. Una vez desarrolladas las pruebas y analizados los indicadores, anteriormente nombrados junto con los tiempos de ejecución, es posible comparar los resultados con otras técnicas utilizadas, en consecuencia se pueden sacar conclusiones sobre el rendimiento y comportamiento de Skyline como choice function en la resolución de un problema de satisfacción de restricciones.

Los contenidos presentes en este informe son: los objetivos del proyecto y una pequeña descripción del estado del arte. En primera instancia de investigación (capítulo 4) se introduce al estudio de todos los ámbitos que forman la programación con restricciones, describiendo el problema de satisfacción de restricciones en su formalidad, algunos algoritmos de búsqueda y técnicas de consistencia, al final de este capítulo se encuentran las estrategias de enumeración y la descripción del solver ECL^{i}PS^e. En el siguiente capítulo (capítulo 5) se define Autonomous Search y su arquitectura, la descripción de cada uno de sus componentes e

interacción entre ellos. En el capítulo 7 se describe el proceso de creación desarrollo e implementación de la choice function basada en Skyline. En el capítulo 8 se presentan los experimentos realizados, su análisis y comparaciones con otras metodologías. En el último capítulo se manifiestan las conclusiones sobre la investigación realizada.

2. Objetivos

2.1 Objetivo General

Diseñar e implementar una choice function basada en Skyline para Autonomous Search.

2.2 Objetivos Específicos

- Investigar y comprender el contexto de la programación con restricciones y su metodología para la resolución de problemas.
- Analizar la arquitectura de Autonomous Search.
- Implementar una nueva choice function integrando la metodología de apoyo a la toma de decisiones Skyline.
- Probar la nueva choice function y analizar los resultados obtenidos.

3. Estado del Arte

La programación con restricciones es utilizada en la descripción y posterior resolución de un problema de diversos tipos, incluyendo entre ellos la inteligencia artificial, investigación operativa, bases de datos, etc. Los primeros trabajos relacionados con la programación con restricciones son justamente en el área de la inteligencia artificial. Un ejemplo de la implementación de satisfacción de restricciones fue el desarrollo de Sketchpad [13] en 1963, sistema utilizado para manipular directamente objetos gráficos, fue el primer programa de dibujo para computador.

Hoy en día existe una cantidad no menor de trabajos de investigación relacionados con la programación con restricciones como por ejemplo [1, 2, 12] entre otros. También este paradigma ha sido utilizado recientemente para el modelado y resolución del Nurse Rostering Problem (NRP) [14].

El estudio de Autonomous Search [11, 15], es relativamente nuevo por lo cual existe poca literatura asociada a él, pero no menos importante, ya que su investigación ha ido aumentando y su arquitectura cambiando, por lo tanto ha mejorado la calidad de los resultados de un problema de satisfacción de restricciones. A inicios del año 2012 se completó una Arquitectura modular para Autonomous Search [3], herramienta completa para la solución de problemas de satisfacción de restricciones como por ejemplo n-queens

El operador Skyline [16, 17] ha sido investigado para su aplicación en bases de datos, tratando de extender SQL para este propósito. Skyline ayuda a la toma de decisiones, aspectos del día a día, por lo cual su aplicación puede extenderse en muchos ámbitos, como también al estudio de sus algoritmos.

4. Programación con Restricciones

La programación con restricciones, en inglés Constraint Programming (CP) es un paradigma que se utiliza para describir y resolver problemas complejos que se pueden presentar en muchas áreas de la vida. Sus comienzos se sitúan en trabajos que datan de los años 1960 y 1970 en el campo de la inteligencia artificial. Presenta un enfoque alternativo a la programación, se basa en una combinación de razonamiento y la informática. Algunos ejemplos de aplicaciones de CP son:

- Biología molecular: secuencia de ADN, construcción de modelos 3D de las proteínas.
- Aplicaciones de negocios (comercio).
- Ingeniería eléctrica: localización de fallas en los circuitos, diseño de circuitos y pruebas y verificación del diseño.
- Álgebra computacional: resolución y/o simplificación de ecuaciones sobre varias estructuras algebraicas.
- Problemas de investigación de operaciones (problemas de optimización).

Los problemas de satisfacción de restricciones se resuelven a través de algoritmos de búsqueda, que buscan en un espacio definido las posibles asignaciones de valores a las variables considerando todas las restricciones, encontrando la solución o demostrando que ésta no existe. Entre algunos de estos algoritmos se encuentran generate and test, backtracking, forward checking y maintaining arc consystency, cada uno con diferentes métodos de trabajo que demuestran su rendimiento.

Para mejorar la eficiencia de los algoritmos de búsqueda existen las técnicas de consistencia [10] que se realizarán antes de comenzar o bien durante el proceso de búsqueda. Existen varios tipos, por ejemplo nodo-consistencia, arco-consistencia y consistencia de camino.

Para la selección de las variables y los valores se puede establecer el orden en que realiza, esto mediante ciertas heurísticas de valor y de variable que según su disposición generarán la cantidad de estrategias de enumeración.

La programación con restricciones puede ser implementada con diversos lenguajes de programación, entre ellos algunos que tienen su base en la programación lógica.

La programación con restricciones posee una ventaja poco común en la informática, ésta es que no es necesario especificar los pasos o la secuencia de instrucciones que se deben ejecutar para encontrar la solución, por ejemplo podemos nombrar la flexibilidad de las restricciones que se pueden añadir o cambiar, y que el orden en que están definidas no influye en el problema, por lo tanto no tendrán un impacto negativo en la interacción con otras restricciones ya existentes.

4.1 Problema de Satisfacción de Restricciones

Diversos tipos de problemas de distintos ámbitos pueden ser resueltos como un problema de satisfacción de restricciones (en inglés Constraint Satisfaction Problem, CSP) por ejemplo, éstos son ampliamente realizados en el área de la investigación de operaciones.

Un CSP se compone de las variables que el problema requiere definir y el dominio de cada una de éstas (ambos deben ser un conjunto finito) y la o las restricciones que limitarán el conjunto de asignaciones para las variables implicadas, necesarias para resolver el problema. Esta solución, si es que existe, será la asignación de un valor del dominio a cada variable de forma que se cumpla cada una de las restricciones.

La resolución de un problema de satisfacción de restricciones se compone de dos partes la primera es el modelado (que corresponde a la definición de variables, dominios y restricciones) y la segunda parte es la resolución donde se utilizarán los algoritmos de búsqueda que según su propio método de funcionamiento encontrarán la solución o de lo contrario probará que no existe.

De lo anterior podemos definir cuáles son los objetivos de un CSP:

- Determinar si tiene una solución que sea consistente, es decir que no viole ninguna restricción.
- Obtener una o todas las soluciones del problema.
- Obtener los dominios mínimos.
- Encontrar una o todas las soluciones óptimas.

4.1.1 Definición Formal de un CSP

Un problema de satisfacción de restricciones *P* es definido como una tripleta *P=(X, D, C)* donde:

- *X* es un conjunto de variables $x_1, x_2,\ldots,x_n$.
- *D* es un conjunto de dominios $d_1, d_2,\ldots, d_n$. Donde d_i es el dominio de X_i, e i=1,…,*n.*
- *C* es el conjunto de restricciones $c_1, c_2,\ldots, c_m$.

  La solución para el CSP será la asignación $\{x_1 \rightarrow a_1, \ldots, x_n \rightarrow a_n\}$ tal que $a_i \in D_i$ para $i = 1, \ldots, n$ y $(a_{j_1}, \ldots, a_{j_{n_j}}) \in C_j$ para $j = 1, \ldots, m$

A continuación se presentan tres ejemplos de problemas con su correspondiente modelado.

Ejemplo 1: N-queens

Se requiere ubicar *n* queens (reinas) sobre un tablero de ajedrez de tamaño *n* x *n,* de tal forma que ninguna de ellas se pueda atacar. Considerando *n* = 4.

Modelo:

- Definición de las variables, en este caso serán las 4 reinas: R_1, R_2, R_3, R_4.
- Se define el dominio [1…4], que corresponde a la cantidad de filas disponibles.
- Y el conjunto de restricciones para $i \in [1,3]$ y $j \in [i+1,4]$

Colocar las reinas en distintas filas: $R_i \neq R_j$

No colocar las reinas en la diagonal noroeste, sureste: $R_i - R_j \neq j - i$

No colocar las reinas en la diagonal suroeste, noreste: $R_i - R_j \neq i - j$

La figura 1 muestra la disposición sobre el tablero de 4 reinas sin que se puedan atacar, satisfaciendo cada una de las restricciones, por lo tanto es una solución, ya que es consistente con las restricciones.

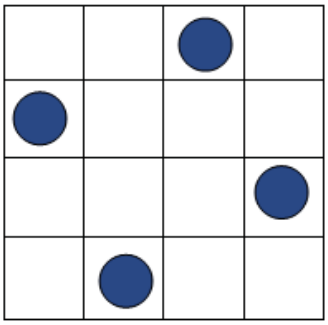

Figura 1: Solución al problema 4-reinas.

Ejemplo 2: Coloreado de mapa

Se requiere colorear un mapa como el ejemplo que muestra la figura 2, con c cantidad de colores y $c+1$ regiones, teniendo en cuenta que los colores de cada región vecina deben ser diferentes. Considerando $c = 3$.

Modelo:

- Las variables comprometidas son cada una de las regiones que se definirán como X, Y, Z, W.
- El dominio corresponde a los posibles colores que puede tomar cada región en este caso pueden ser rojo, verde o amarillo: $\{r, v, a\}$.
- Las restricciones representan que cada región vecina debe colorearse con distinto color: $X \neq Y$, $Y \neq W$, $W \neq Z$, $Z \neq X$, $X \neq W$.

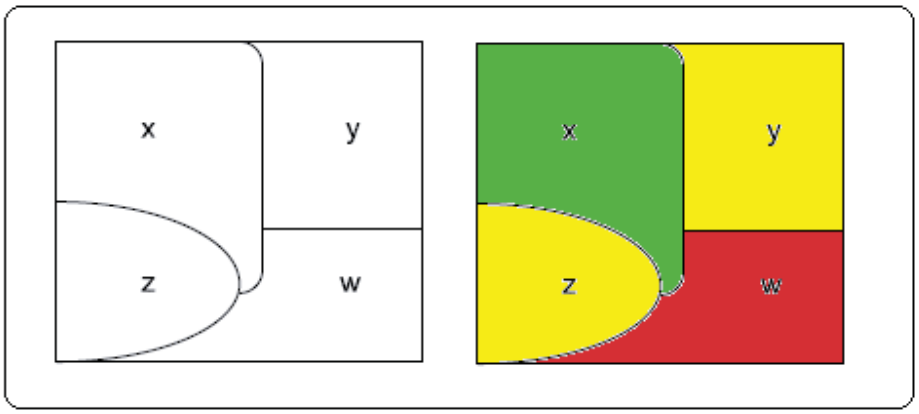

Figura 2: Problema de coloración de mapa.

Ejemplo 3: Criptoaritmética

Se requiere asignar distintos valores entre el 1 y 8 a cada letra para que la ecuación indicada sea cierta.

$$(C_1\ C_2\ C_3)$$

$$\begin{array}{r} \mathbf{I\ D\ E\ A} \\ +\ \mathbf{I\ D\ E\ A} \\ \hline \mathbf{M\ E\ N\ T\ E} \end{array}$$

Modelo:

- Variables, cada letra de la ecuación: I, D, E, A, M, N, T, C_1, C_2, C_3.
- Dominio para I, D, E, A, M, N, T = [1…8] y para C_1, C_2, C_3= [0,1] ya que son la posible reserva de la suma.
- Restricciones: Corresponden a las igualdades que deben ocurrir para satisfacer la ecuación inicial:

$$2 \times A = (10 \times C_3) + E$$
$$(2 \times E) + C_3 = (10 \times C_2) + T$$
$$(2 \times D) + C_2 = (10 \times C_1) + N$$
$$(2 \times I) + C_1 = (10 \times M) + E$$

4.2 Algoritmos de Búsqueda

Los algoritmos de búsqueda, recorren el espacio y buscan todas las posibles asignaciones de valores a las variables, de tal manera de poder garantizar una solución o de lo contrario demostrar por qué ésta no existe. Estos algoritmos se nombran como algoritmos completos si recorren todo el espacio en la búsqueda y garantizando una solución, o incompletos si exploran sólo una parte del espacio y no se garantiza encontrar una solución, por lo que su rapidez es mayor que los algoritmos completos, dada la incorporación de heurísticas en el proceso de resolución, son mayormente usados en problemas de satisfacción de restricciones como los ejemplos anteriormente nombrados y en problemas de optimización.

Existen diversos algoritmos de búsqueda cada uno utiliza un método diferente en busca de una solución, algunos pueden resultar mejores que otros en cuanto a su efectividad y/o rendimiento. Se explicarán brevemente los más comunes resaltando sus características, funcionamiento y principales desventajas.

Como fue explicado anteriormente los algoritmos de búsqueda asignan valores a variables, es por esto que se genera un árbol de búsqueda como se muestra en la figura 3, donde la raíz es una tripleta (X_1 , X_2, X_3) vacía lo que muestra que aún a ninguna variable se le ha asignado algún valor. Así en el primer nivel se le ha asignado valor a X_1, siguiendo esta misma lógica los próximos niveles tendrán las correspondientes asignaciones de valores a las variables representadas por *X*. Cabe destacar que para un árbol donde existe *m* cantidad de variables los nodos del nivel *m*, que representan también las hojas del árbol, serán la asignación de todos los valores a variables contenidos en el problema y si alguna de estas asignaciones (de los nodos hoja) no transgrede ninguna restricción entonces es solución.

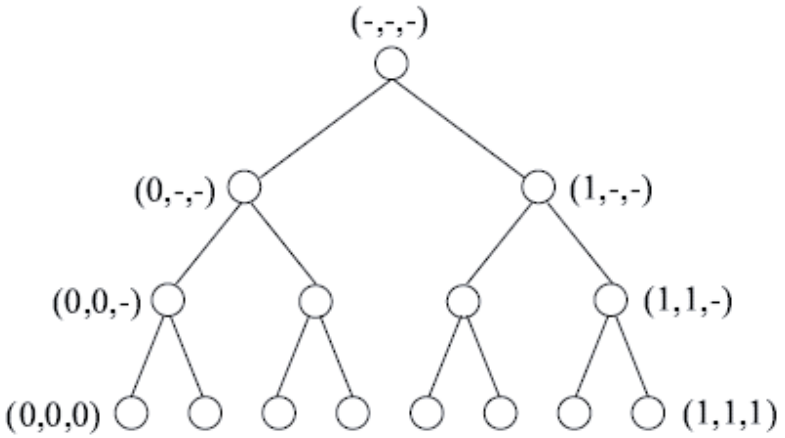

Figura 3: Árbol de búsqueda.

4.2.1 Generate and Test

Es una de las técnicas de búsqueda más sencillas, permite encontrar la solución de un problema de satisfacción de restricciones. Su funcionamiento se basa en que cada combinación posible de la asignación de variables y dominio es generada [5]. Luego se prueba si la combinación cumple con el conjunto de restricciones, la primera combinación que lo haga será la solución al problema.

Este algoritmo de búsqueda tiene como desventaja que instancia todas las variables a cada elemento del dominio asociado, por cual si existe una gran cantidad tanto de variables como de elementos en el dominio la cantidad de combinaciones será igual al producto cartesiano de ambos conjuntos, gran cantidad de asignaciones que podrían no llevar a la solución, por lo tanto es un algoritmo ineficiente para problemas con gran cantidad de variables y de amplios dominios.

La figura 4 muestra gráficamente el algoritmo de Generate and Test para el problema de las 4 reinas descrito anteriormente, se puede ver claramente el alto costo de usar GT para este problema, ya que la gran mayoría de las primeras asignaciones completas (todas las variables han sido asignadas a un elemento del dominio) no satisfacen algunas de las restricciones y por lo tanto no son soluciones.

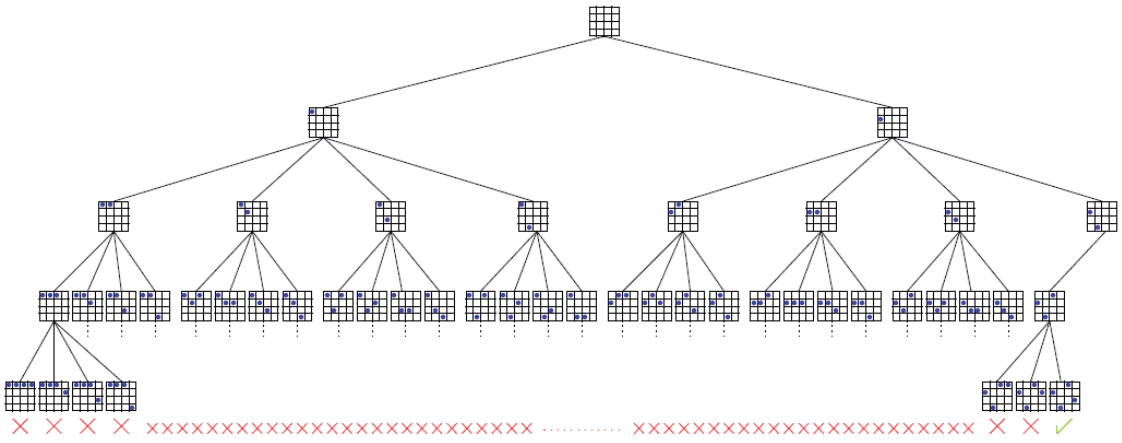

Figura 4: Algoritmo generate and test aplicado al problema 4-reinas.

4.2.2 Backtracking

También llamado Backtracking Cronológico, es una mejora al algoritmo Generate and Test, la mejora radica en que cada vez que se asigna un nuevo valor a una variable actual (R_i), se comprueba si este es consistente con los valores que hemos asignado a las variables pasadas. Si la consistencia no se cumple, se abandona esta asignación parcial y se le asigna un nuevo valor a R_i repitiendo el chequeo de consistencia, y si ya se agotaron todos los valores de su dominio el algoritmo Backtracking retrocede para probar otro valor para la variable R_{i-1}, si tras asignarle otro valor y comprobando consistencia, nuevamente se agota el

dominio (D_{i-1}), se repite el proceso para el nivel *i-2* y para los niveles necesarios que posean esas mismas características.

En resumen la mejora que realiza es que si se detecta inconsistencia en la asignación parcial ésta se descarta, ya que en los siguientes subárboles no podrá existir asignación completa que sea solución al problema. Ahorrándose recorrer y verificar los subárboles que cuelgan de la asignación parcial como lo haría el algoritmo Generate and Test.

La desventaja de este algoritmo es que detecta inconsistencias que se producen por la misma razón en distintas partes de la búsqueda. Otra desventaja es que sólo comprueba inconsistencia entre la variable actual respecto de la(s) anterior(es), pudiéndose así generar inconsistencias de la variable actual y las futuras, como lo muestra la figura 5 en el recuadro rojo donde la ubicación de las dos reinas es consistente, pero ninguna posición de la tercera reina será consistente con la posición de sus predecesoras.

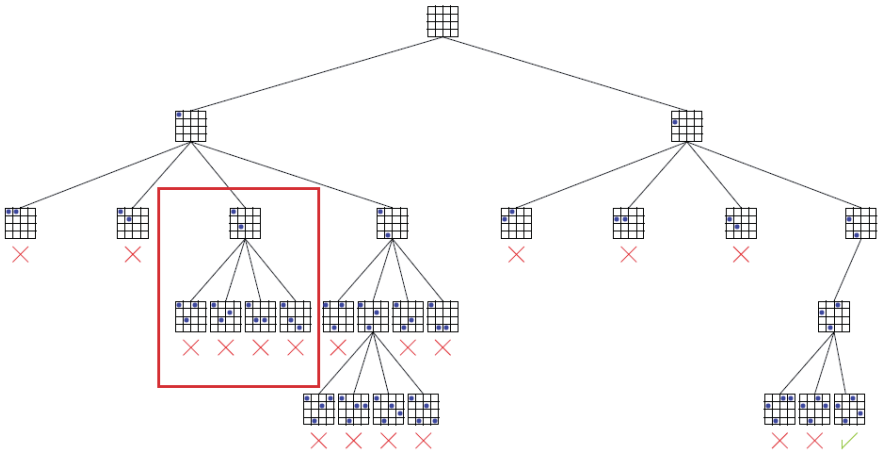

Figura 5: Algoritmo Backtracking aplicado al problema 4-reinas.

4.2.3 Forward Checking

Es un algoritmo tipo Look-Ahead (comprobación inferencial hacia adelante) cuya principal característica es que revisa los valores futuros de la asignación actual, es decir que si existen inconsistencias de la variable actual y las futuras entonces esos valores no serán instanciados, además realiza el chequeo de la variable actual con las pasadas.

Forward Checking revisa la asignación actual con todos los valores del dominio de las futuras variables y que dada las restricciones sobre la asignación actual no podrán ser instanciadas. Así estos valores son borrados del dominio temporalmente, en caso de que el dominio quede sin valores la variable actual se debe instanciar con otro valor y volver a realizar el proceso de chequeo de las variables futuras. Si ningún valor instanciado es solución entonces se realiza Backtracking.

Este algoritmo limita el espacio de búsqueda, ya que al revisar las variables futuras se reduce el dominio posible para las variables que serán instanciadas posteriormente. Pese a que es un algoritmo bastante más eficiente que los descritos anteriormente posee la desventaja que no evalúa la consistencia de las variables futuras con otras futuras, esto podría ser posible ya que este algoritmo conoce los valores del dominio que pueden ser instanciados y posiblemente verificar la consistencia de éstos con los futuros.

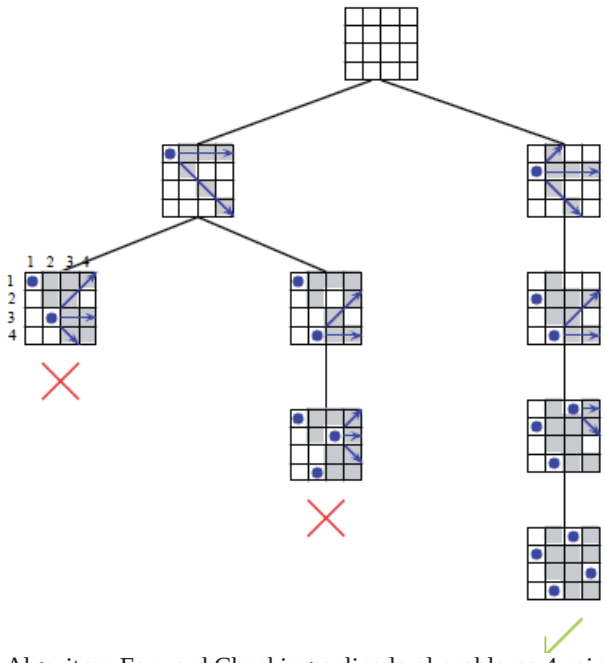

Figura 6: Algoritmo Forward Checking aplicado al problema 4-reinas.

4.2.4 Maintaining Arc Consistency

MAC elimina inmediatamente de la primera asignación aquello que genere un conflicto posterior, luego con los elementos del dominio que estén aún disponibles realiza el mismo proceso verificando así inmediatamente si existe una solución dada la primera asignación o devolviéndose y, eliminando esa asignación si no existiera una solución dada esas características.

La figura 7 muestra el funcionamiento de este algoritmo y en las matrices *a*, *b*, *c* y *d* muestra más detalladamente lo que ocurre. Si la primera reina (R_1) es colocada en la posición (1,1), en matriz *a* se ve que el dominio que genera conflicto con esa posición y las futuras es eliminado (recuadros azules), se puede observar que aún queda dominio para las siguientes reinas, por lo tanto el algoritmo comienza a revisar esos casos y lo hace con la celda (3,2) se descubre que esa posición para la segunda reina (R_2) no es consistente (recuadros naranjos) porque no deja lugar para las siguientes reinas (R_3 y R_4) así la asignación de la segunda reina en la celda (3,2) es eliminada procedimiento que se observa en la matriz *b*. El algoritmo continúa ahora con la posición (4,2) para colocar la siguiente reina (R_2), elimina los posibles conflictos las futuras posiciones (recuadros rojos en matriz *c*), entonces se establece que la posición para la tercera reina sería la celda (2,3) que se muestra en la matriz *d*, pero se provocaría inconsistencia ya que no queda una posición disponible para la cuarta reina (R_4),

por lo tanto para la posición de la primera reina en la celda (1,1) no existe solución. El algoritmo para otra primera reina en otra posición se repite de forma equivalente hasta llegar a la solución como se muestra en la figura.

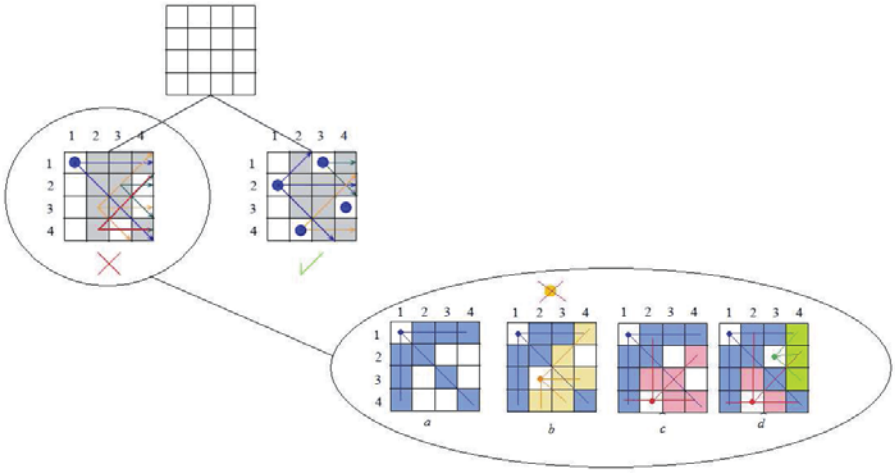

Figura 7: Algoritmo MAC aplicado al problema 4-reinas.

4.3 Técnicas de Consistencia

Las técnicas de consistencia, también llamadas técnicas de inferencia, se utilizan para mejorar la eficiencia de los algoritmos de búsqueda, éstas se pueden utilizar antes o durante el proceso de búsqueda, reduciendo la cantidad de nodos a instanciar en el árbol de búsqueda, eliminado los dominios que debido a una restricción no corresponderán a una solución.

4.3.1 Nodo-Consistencia

La consistencia de nodo asegura que todos los valores del dominio de una variable satisfacen todas las restricciones unarias sobre esa variable, es decir se eliminan los valores inconsistentes con las restricciones unarias donde la variable participe.
Definición formal: Una variable (X_i) es nodo-consistente si y sólo si los valores que están en su dominio satisfacen la restricción unaria donde participa la variable:

$$\forall X_i \in X, \forall R_i \in R, \exists a \in D_i : a \text{ satisface } R_i.$$

Definición formal: Un problema de satisfacción de restricciones en nodo-consistente si todas sus variables son nodo consistentes:

$$\forall X_i \in X, \forall R_i \in R : X_i \text{ satisface } R_i.$$

Ejemplo: Un problema cuya variable es X_1, dominio es el conjunto {-1, 0, 1, 2, 3, 4} y la restricción unaria es $R: X_1 \geq 2$.

Al aplicar nodo consistencia se eliminarán del dominio los valores que no satisfacen la restricción unaria, por lo tanto el dominio quedaría dado por el conjunto {2, 3, 4}, llamado dominio nodo-consistente.

4.3.2 Arco-Consistencia

La técnica es muy efectiva y se aplica a las restricciones que involucran dos variables, el arco-consistencia asegura que cada valor en el dominio de cada variable está soportado por un valor de la otra variable que participa en la restricción.
Definición formal: Un valor $a \in D_i$ es arco-consistente relativo a X_j si y sólo si existe un valor $b \in D_j$ tal que (X_i, a) y (X_j, b) satisfacen la restricción R_{ij}.
Definición formal: Una variable X_i es arco-consistente relativa a X_j si y sólo si todos los valores de D_i son arco-consistentes relativos a X_j. Es decir una variable es arco-consistente si es consistente relativa a todas aquellas con las que interviene en alguna restricción.

Por lo tanto un problema de satisfacción de restricciones es arco-consistente si todas las variables son arco-consistente.

Ejemplo: Dada una restricción R_{ij}: $x_i < x_j$ de la figura 8 se puede ver que el arco R_{ij} es consistente, ya que para cada valor de $a \in [3,6]$ hay al menos un valor en $b \in [8,10]$ de manera que satisface la restricción R_{ij}.

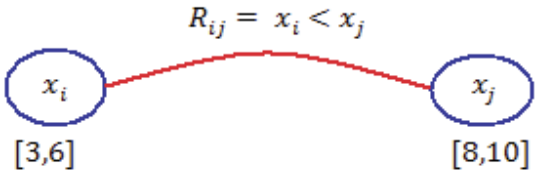

Figura 8: Arco-Consistencia.

4.4 Estrategias de Enumeración

4.4.1 Heurísticas

Para el funcionamiento de los algoritmos de búsqueda es necesario contar con un orden específico en que cada una de las variables será estudiada, el orden en que los valores del dominio serán instanciados para cada variable y por último también se puede especificar el orden de las restricciones presentes en el problema. La correcta selección del orden para la variable y valor, puede mejorar la eficiencia de la solución generada. Para establecer estos órdenes es que existen las heurísticas de selección de variable y las heurísticas de selección de valor, las distintas combinaciones de éstas heurísticas forman las estrategias de enumeración. Cada una de las distintas estrategias de enumeración tendrá un comportamiento distinto por lo tanto pueden existir algunas con un comportamiento más eficiente que otras. A continuación se describirán algunas heurísticas de selección de variable y valor.

4.4.1.1 Heurísticas de Selección de Variable

El orden en que las variables son asignadas durante la búsqueda puede tener un efecto significativo en cuanto al espacio de búsqueda. Generalmente estas heurísticas tratan de seleccionar antes las variables que más restringen a las demás, con el fin de reducir el número de vueltas atrás. Las heurísticas de selección de variables pueden ser estáticas o dinámicas.

- **Selección estática de variable**: Asigna un orden fijo de las variables antes del inicio de la búsqueda, cada nivel de árbol de búsqueda se asocia con una variable, es necesario que cada nodo se asocie con una variable para la generación de sus sucesores.
- **Selección dinámica de variable**: Puede cambiar el orden de las variables dinámicamente basándose en información local que se genera durante el proceso de búsqueda. En el árbol de búsqueda se ve representada porque existen diferentes variables en un mismo nivel.

Generalmente estas heurísticas seleccionan antes las variables que más restringen a las demás, para así reconocer las situaciones sin salida tempranamente.

- **Heurísticas de selección estática de variables**

1) **Minimum Width (MW)**: Al comienzo impone un orden total sobre todas las variables, de forma que el orden tiene la mínima anchura (ancho de la red). La anchura de la variable *x* es el número de variables que están antes que *x*, de acuerdo a un orden dado y que son adyacentes a *x*. Así mismo la anchura de un orden es la máxima anchura de todas las variables bajo ese orden. En resumen las variables que se encuentran al principio de la ordenación son las más restringidas y las menos restringidas son las variables que se encuentran al final. Asignando antes las variables más restringidas, se pueden identificar antes las situaciones sin salida reduciendo el backtracking.

2) **Maximun Degree (MD)**: Las variables son ordenadas de forma decreciente según su grado en el grafo de restricciones original. El grado de un nodo hace referencia al número de nodos adyacentes a él, es decir el número de variables con la que está conectada. Debido al orden decreciente las variables de mayor grado (las más conectadas) son seleccionadas primero.

3) **Minimum Domain Variable (MDV)**: Selecciona las variables de acuerdo a la menor cardinalidad de su dominio. Por lo tanto aquellas variables cuyo dominio es pequeño son elegidas antes.

- **Heurísticas de selección dinámica de variables**

1) **First-fail**: Primer fallo, sugiere que *para tener éxito deberíamos intentar primero donde sea más probable que falle,* entonces en cada paso se selecciona la variable más restringida. La heurística first-fail también conocida como minimum remaining values (MRV), trata de hacer lo mismo seleccionando la variable con el

dominio más pequeño, esto se sostiene en que si una variable posee un dominio pequeño es más difícil encontrar un valor consistente.

2) **Round Robin**: Se seleccionan las variables en algún orden racional e equitativo de esta manera se logran instanciar todas las variables presentes.

4.4.1.2 Heurísticas de Selección de Valor

Existe poco trabajo asociado a las heurísticas de selección de valor a diferencia de las heurísticas de selección de variable. La idea para éstas heurísticas es que seleccionen un valor para la variable actual, que sea más probable que lleve a una solución, en otras palabras identificar qué rama del árbol de búsqueda nos puede llevar a la solución. En su mayoría éstas heurísticas tratan de instanciar un valor que esté poco restringido en relación a la variable actual, es decir aquel valor que reduce mínimamente el número de valores útiles para las futuras variables.

1) **Min-conflicts**: Esta heurística ordena los valores de acuerdo a los conflictos que puedan suceder con las variables no instanciadas. Se asocia a cada valor a de la variable actual número total de valores en los dominios de las futuras variables adyacentes que son incompatibles con a. El valor seleccionado es el que tenga la suma más baja.

2) **Max-domain-size**: Selecciona el valor de la variable actual que deja el máximo dominio en las variables futuras. En otras palabras deja los máximos tamaños de dominio para las variables futuras.

3) **Weighth-max-domain-size**: Es una mejora del anterior y radica en que presenta una manera de elegir si hay empates, está basado en el número de futuras variables que tiene una talla de dominio dado. Por ejemplo, si un valor a_i deja cinco variables futuras con dominios de dos elementos, y otro valor a_j deja siete variables futuras con dominios de dos elementos también, entonces en este caso se selecciona el valor a_j.

4) **Point-domain-size**: Esta heurística asigna un peso (unidades) a cada valor de la variable actual dependiendo del número de variables futuras que se quedan con ciertas tallas de dominio.

También existen heurísticas de selección de valor más simples llamadas Smallest, Median y Maximal donde sólo se elige el valor del dominio más pequeño, medio y máximo respectivamente.

Según las diferentes combinaciones entre heurísticas de variable y valor, se generarán las estrategias de enumeración, éstas pueden tener rendimientos significativamente diferentes, por lo tanto es de mucha importancia seleccionar una buena estrategia de enumeración [1].

4.5 Resolución de un CSP

El algoritmo de la figura 9 presenta el procedimiento para la solución de un CSP, incorporando las características revisadas anteriormente (heurísticas de valor, variable, algoritmo de búsqueda, etc). El algoritmo muestra en la línea 1 la acción de cargar el problema que se quiere resolver, luego dentro del ciclo (`while not`) se encuentra el conjunto de acciones que se ejecutarán hasta que todas las variables sean instanciadas o se produzca una falla, por ejemplo que no se encuentre una solución. Los dos primeros pasos dentro del ciclo corresponden a la selección de variable y valor. La tercera acción (línea 5) es un llamado al procedimiento `propagate()` encargado de "podar" el árbol y finalmente dos condiciones para realizar backtrack, el primero para probar el siguiente valor disponible del dominio de la variable actual y el segundo backtrack que retorna la variable instanciada recientemente que aún posee valores que pueden llegar a ser solución.

Algoritmo: Resolución CSP

```
 1:  load_CSP()
 2:  while not all_variables_fixed or fairule do
 3:        heuristic_variable_selection()
 4:        heuristic_value_selection()
 5:        propagate()
 6:        if empty_domain_in_future_variable() then
 7:           shallow_backtrack()
 8:        end if
 9:        if empty_domain_in_current_variable() then
10:           backtrack()
11:        end if
12:  end while
```

Figura 9: Algoritmo para la resolución de un CSP [15]

4.6 Solvers

Los solvers que se utilizan para la programación con restricciones se basan en la programación declarativa, así es necesario declarar las variables, los dominios pertinentes y las restricciones. Sólo con esos tres elementos el solver será capaz de encontrar una solución. Distintos solvers ofrecerán distintos resultados, por ejemplo un solver puede obtener más rápido los resultados, o bien mostrar más robustez en comparación con otro.

4.6.1 Programación Lógica con Restricciones

La programación lógica con restricciones une la programación lógica y la programación con restricciones en un nuevo paradigma que puede ser utilizado con éxito en múltiples áreas,

como por ejemplo la planificación de producción, programación de transporte, el análisis numérico y la bioinformática entre otros.

La programación lógica con restricciones puede ser implementada con un lenguaje propio, como por ejemplo: ECL^{i}PS^e, Oz, SICtus Prolog [8], Chip. O como bibliotecas incluidas en un programa, ésta última provee de dos características, primero los objetos especiales que corresponden a las variables, dominios y restricciones y segundo las funciones especiales para encontrar una solución y la siguiente solución. Algunos ejemplos de bibliotecas son Ilog Solver y Choco [9]. Los lenguajes y librerías son descritos brevemente a continuación.

Lenguajes:

- **Oz**: Está basado en la programación funcional y la programación lógica con restricciones. Combina funciones con relaciones. Proporciona algoritmos para decidir la satisfactibilidad de las restricciones básicas. Proporciona objetos y concurrencia.

- **SICtus Prolog**: Basado en Prolog, posee una estructura del control específica para las iteraciones sencillas llamada do-loop. Posee corrutinas para la resolución de restricciones (block). Sintaxis de programación orientada a objetos, los que pueden ser modificados. Posee constraint solvers de dominios finitos, booleanos y reales entre otros.

- **ECL^{i}PS^e**: Software de código abierto, basado en Prolog. Incluye las restricciones de dominio finito tradicional. Las características más importantes y detalladas de este lenguaje se presentan más adelante.

Bibliotecas:

- **Ilog Solver**: Es una biblioteca de C++ para la programación con restricciones, por esto es necesario que tanto los datos como las estructuras de control deben ser definas en C++. Así una restricción puede ser un objeto o una expresión booleana falso (IlcFalse) o verdadero (IlcTrue) dependiendo de la satisfactibilidad de la restricción. Las expresiones pueden combinarse con operadores lógicos como por ejemplo and, or y not para poder crear restricciones más complejas. Es posible postergar una restricción con la función IlcPost. El mayor énfasis de esta biblioteca es la eficiencia.

Para un mismo problema se pueden utilizar distintos lenguajes y obtener rendimientos más eficientes en un lenguaje o en otro y así poder comparar y analizar los resultados.

4.6.2 ECL^{i}PS^{e}

El solver utilizado en la implementación de este trabajo es ECLiPSe por lo cual a continuación se describen algunas de sus principales características.

Es un sistema software libre, basado en el paradigma de la programación lógica con restricciones para el despliegue y desarrollo de aplicaciones de programación con restricciones. Es ideal para la enseñanza de la mayoría de los aspectos de un problema combinatorio, por ejemplo modelado de problemas, programación con restricciones, programación matemática y técnicas de búsqueda. Su amplio alcance lo hace ser una buena herramienta para la investigación de métodos de resolución de problemas híbridos. Los principales libros de CLP (Constraint Logic Programming) hacen referencia al uso de ECLiPSe [7] por lo cual se ha convertido en una herramienta importante en la programación con restricciones.

Está constituido por un núcleo en tiempo de ejecución, una colección de bibliotecas, modelado y control de lenguaje, un entorno de desarrollo, interfaces para integrarse en un entorno host e interfaces para solvers anexos.

ECL^{i}PS^{e} está diseñado para:

- Tareas generales de programación, especialmente para la creación rápida de prototipos.
- La resolución de problemas utilizando las librerías de resolución disponibles y el paradigma de la programación lógica con restricciones.
- El desarrollo de nuevos constraint solvers, basados en otros existentes.

ECLiPSe es en gran medida compatible con el lenguaje de programación lógico e interpretado Prolog (basado en tres tipos de clausulas: hechos, reglas y consultas) y algunas extensiones que permitirán manejar bases de datos. El programa proporciona varias bibliotecas de constraint solvers las cuales pueden ser utilizadas en programas de aplicación:

- Además de enteros, la librería de dominios finito permite elementos atómicos, por ejemplo string y floats y también de elementos compuestos básicos por ejemplo f(a,b).
- Conjunto finito de restricciones.
- Restricciones lineales.
- Reglas de manejo de restricciones CHR (Constraint Handling Rules) que contiene bibliotecas de más de 20 constraint solvers adicionales.

La figura 10 muestra el problema N-queens usando ECLiPSe, se modeló el problema con una variable por reina, asumiendo que cada reina ocupará una columna. Las variables van desde los rangos de 1 hasta N que indican la fila en la cual la reina será colocada. Las restricciones se aseguran de que dos reinas no ocupen la misma fila o la misma diagonal. El tablero se maneja con una lista (`Board`) para la ubicación de las reinas.

Algoritmo N-Reinas

```
 1: :- lib(ic).
 2: queens_lists(N, Board) :-
 3:    length(Board, N),
 4:    Board :: 1..N,
 5:    ( fromto(Board, [Q1|Cols], Cols, []) do
 6:          ( foreach(Q2, Cols), param(Q1), count(Dist,1,_) do
 7:                Q2 #\= Q1,
 8:                Q2 - Q1 #\= Dist,
 9:                Q1 - Q2 #\= Dist
10:          )
11:    ),
12:    labeling(Board).
```

Figura 10: Algoritmo problema N-reinas en ECLiPSe

Sentencias utilizadas:

- **lib(library):**Para cargar una biblioteca, en el ejemplo ic (Interval Constraint).
- **fromto (First, In, Out, Last):** Itera comenzando por In=First hasta que Out=Last.
- **foreach(X, List):** Iterar objetivos con X que abarcan todos los elementos de las lista (List). X es variable local.
- **count(I, Min, Max):** Itera sobre enteros desde Min hasta Max, la variable local I puede ser utilizada para controlar la iteración o para contar.
- **param(Var1, Var2,…):** Para la declaración de variables.
- **length(List, Var):** Puede ser utilizado para calcular la longitud de una lista, o como muestra el ejemplo para construir una lista de una longitud dada.
- **labeling(X,Y):** Es la parte de búsqueda del programa que trata de encontrar las soluciones, tratando todas las instanciaciones de las variables.

4.6.3 Problemas

A continuación se describen los problemas presentes en la herramienta y de los cuales se toman las pruebas de la choice function utilizando skyline, presentando su modelo matemático y el código ECL^iPS^e que ejecuta la herramienta.

4.6.3.1 Problema: N-queens

El objetivo es disponer n reinas en un tablero de dimensiones $n \times n$, de tal forma que no se puedan atacar entre ellas. Modelo para este problema está descrito en la sección 4.1.1.

Algoritmo: n-reinas

```
:-lib(ic).
:-lib(lists).
:-lib(random).
:-lib(lib_autonomous_search).

queen2(N) :-
      queens(N, Board),
      lib_autonomous_search(Board).

queens(N, Board) :-
    length(Board, N),
    Board :: 1..N,
    ( fromto(Board, [Q1|Cols], Cols, []) do
       ( foreach(Q2, Cols), param(Q1), count(Dist,1,_) do
            noattack(Q1, Q2, Dist)
        )
    ).

noattack(Q1,Q2,Dist) :-
    Q2 #\= Q1,
    Q2 - Q1 #\= Dist,
    Q1 - Q2 #\= Dist.
```

Figura 11: Algoritmo n-reinas en ECLiPSe.

4.6.3.2 Problema: Sudoku

El objetivo del sudoku es rellenar una cuadrícula de 9×9 celdas divididas a su vez en cuadrículas de 3×3 con números del 1 al 9 (algunos ya dispuestos en ciertas celdas), éstos números no se deben repetir tanto en las filas y las columnas de ambas cuadrículas.

Modelo

Variables: $X_i = x_1, \dots, x_n$
Dominio: $X_i = 1, \dots, 9$
Restricciones: $x_{i,1} \neq x_{i,2} \neq \cdots \neq x_{i,9}$
$x_{1,i} \neq x_{2,i} \neq \cdots \neq x_{9,i}$

Algoritmo: Sudoku

```
:-lib(ic).
:-lib(lists).
:-lib(random).
:-lib(lib_autonomous_search).

sudoku(ProblemName) :-
      problem(ProblemName, Board),
      sudoku2(3, Board).

sudoku2(N, Board) :-
      N2 is N*N,
      dim(Board, [N2,N2]),
      Board[1..N2,1..N2] :: 1..N2,
      ( for(I,1,N2), param(Board,N2) do
          Row is Board[I,1..N2],
          alldifferent(Row),
          Col is Board[1..N2,I],
          alldifferent(Col)

      ),
      ( multifor([I,J],1,N2,N), param(Board,N) do
          ( multifor([K,L],0,N-1), param(Board,I,J), foreach(X,SubSquare) do
            X is Board[I+K,J+L]
          ),
          alldifferent(SubSquare)
      ),
      term_variables(Board, Vars),
      lib_autonomous_search(Vars).
```

Figura 12:Algoritmo sudoku en ECLiPSe.

4.6.3.3 Problema: Magic Squares

Magic Squares, es un juego matemático donde dada una variable n se tendrán n^2 variables en el cuadrado, la idea principal es que cada una de las casillas posea distintos números, es decir ningún número puede repetirse, además cada una de las filas, columnas y diagonales debe sumar la misma cantidad como lo muestra la figura 13.

Ejemplo: $n = 3$

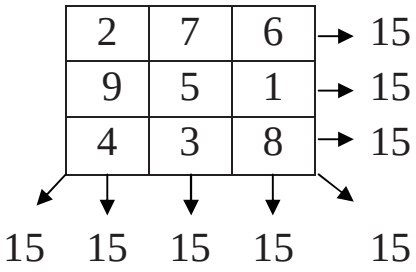

Figura 13: Ejemplo magic squares n=3.

Modelo

Variables: $X_i = \{x_{1,1}, \dots, x_{n,n}\}$

Dominio: $Dom\ (X_i) = 1, \dots, n^2$

Restricciones: $Sum = \frac{n(n^2+1)}{2}$

$$Sum = \sum_{i=1}^{n} x_{i,1}\,;\ \ Sum = \sum_{i=1}^{n} x_{1,i}\,;\ \ Sum = \sum_{i=1}^{n} x_{i,i}\ \ ;\ \ Sum = \sum_{i=1}^{n} x_{i,(n-i+1)}$$

Algoritmo: Magic Squares

```
:-lib(ic).
:-lib(lists).
:-lib(random).
:-lib(lib_autonomous_search).

magic(N):-
      magic2(N,Board,Vars),
        lib_autonomous_search(Board).

magic2(N,Vars,Square):-
      NN is N*N,
      Sum is N*(NN+1)//2,
      dim(Square, [N,N]),
      Square[1..N,1..N] :: 1..NN,
      Rows is Square[1..N,1..N],
      flatten(Rows, Vars),
      alldifferent(Vars),

      (
          for(I,1,N),
          foreach(U,UpDiag),
          foreach(D,DownDiag),
          param(N,Square,Sum)
      do
          Sum #= sum(Square[I,1..N]),
          Sum #= sum(Square[1..N,I]),
          U is Square[I,I],
          D is Square[I,N+1-I]
      ),
      Sum #= sum(UpDiag),
      Sum #= sum(DownDiag),

      Square[1,1] #< Square[1,N],
      Square[1,1] #< Square[N,N],
      Square[1,1] #< Square[N,1],
      Square[1,N] #< Square[N,1].
```

Figura 14: Algoritmo magic squares en ECliPSe.

4.6.3.4 Problema: Knight Tour

Es un antiguo problema matemático en cual se tiene una cuadrícula de tamaño $n \times n$ y un caballo (de ajedrez) colocado en una posición cualquiera (x, y), el objetivo principal es que el caballo pase por todas las casillas sólo una vez, esto resulta en $n^2 - 1$ movimientos. La figura 15 muestra el recorrido de un caballo en un tablero de 8×8, partiendo en la casilla [1,5].

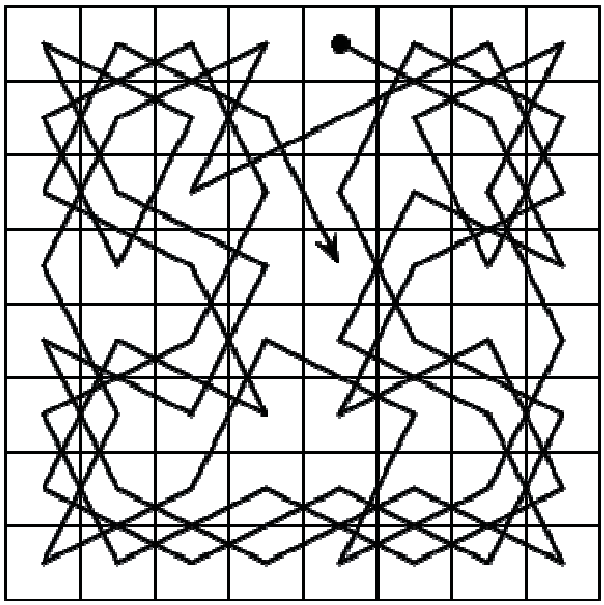

Figura 15: Knight tour n=8.

Modelo

Variables: $X_i = \{x_1, \dots, x_n\}$
$C_i = \{a, b\}$
$D_i = \{I, J\}$
Dominio: $Dom\ (X_i) = 1, \dots, n^2$
$Dom\ (C_i) = [-2, -1, 1, 2]$
$Dom\ (D_i) = 1, \dots, n$
Restricciones: $|a| + |b| = 3$
$IA = I + a, JB = J + b, IA > 1, JB > 1, IA < N, JB < N$
$K_1 = (I - 1) \times N + J$
$K_2 = (I - 1 + a) \times N + (J + b)$

Algoritmo: Knight Tour

```
:-lib(ic).
:-lib(ic_global).
:-lib(listut).
:-lib(lib_autonomous_search).

knightTour(N):-
      knight_tour(N, Board),
      lib_autonomous_search(Board).

knight_tour(N, Board):-
      N2 is N*N,
      length(Board, N2),
      C = [-2,-1,1,2],
      ic_global:alldifferent(Board),
        ( fromto(Board, [Q1|Cols], Cols, []),count(Dist,1,_), param(N,C) do
            tomar_Q2(Cols,Q2),
            I :: 1..N,
            J :: 1..N,
            A :: C,
            B :: C,
            IA #= I+A,
            JB #= J+B,
            IA #>= 1,
            JB #>= 1,
            IA #=< N,
            JB #=< N,
            abs(A)+abs(B) #= 3,
            (I-1)*N + J #= Q1,
            Q2 #= (I-1+A)*N + J+B
      ).
tomar_Q2([Q2|Cols],Q2).
tomar_Q2([],Q2).
```

Figura 16: Algoritmo knight tour en ECLiPSe.

5. Autonomous Search

Autonomous Search (AS) [11] es un caso particular de sistemas adaptativos, tiene como objetivo mejorar el rendimiento de la solución de un problema de satisfacción de restricciones, adaptándose al problema en cuestión. Posee la habilidad de modificar sus componentes internos cuando es expuesto a cambios externos y a oportunidades.

Su objetivo es mejorar el rendimiento de la solución, este se puede medir a través del valor de los indicadores (ver Anexo A) o del tiempo en el cual se obtiene la solución al problema, para lograr este objetivo adapta la estrategia de búsqueda del problema. Los componentes internos, que se pueden modificar, corresponden a varios algoritmos involucrados en el proceso de búsqueda: heurísticas, inferencias, etc. Los componentes externos corresponden a la evolución de la información recogida durante el proceso de búsqueda. Esta información también puede ser recogida directamente del problema (por ejemplo: tamaño del espacio de búsqueda exacto o estimado, número de sub-problemas) o indirectamente a través de la eficiencia percibida de los componentes individuales, por ejemplo la capacidad de poda de las técnicas de inferencia. La información también puede referirse al ambiente computacional.

Autonomous Search, permite a los sistemas mejorar su rendimiento, ya que guía la toma de decisiones en la fase de enumeración donde el orden de las variables y los valores son seleccionados, además posee una arquitectura donde sus componentes interactúan entre sí entregándose información, lo que sirve para el análisis de la siguiente estrategia a utilizar.

5.1 Arquitectura [12]

Decidir qué heurísticas de variable y valor utilizar es un trabajo difícil, aún cuando estas sean elegidas, no se asegura encontrar una solución para un problema de satisfacción de restricciones, o que si se encuentra la solución ésta sea la óptima. En consecuencia es necesario llevar a cabo un proceso de resolución, donde será indispensable evaluar las estrategias, para encontrar la solución.

La arquitectura de AS está basada en 4 componentes (ver Figura 17): solver, observación, análisis y actualización [15] los cuales son descritos particularmente a continuación.

- **Solver:** Componente que corre un algoritmo CSP genérico, alternando la propagación de restricciones y las fases de enumeración. Posee un conjunto de estrategias de enumeración básicas cada una caracterizada por una prioridad que evoluciona durante el cálculo, el componente actualización evalúa las estrategias y actualiza sus prioridades. Para cada enumeración (asignación un valor a una variable), la estrategia de enumeración dinámica selecciona la estrategia básica para ser usada en base a las prioridades fijadas.
- **Observación:** Este componente tiene como objetivo registrar alguna información acerca del actual árbol de búsqueda, por ejemplo observar el proceso de resolución en el componente solver. Estas observaciones, llamadas snapshots, no son realizados continuamente y consisten en extraer y registrar (desde el árbol de búsqueda) cierta información del estado de la solución.

- **Análisis:** Este componente es el encargado de estudiar los snapshots tomados por el componente información. También evalúa las diferentes estrategias y provee de indicadores para el componente actualización. Los indicadores pueden ser extraídos, calculados o deducidos desde uno o varios snapshots de la base de datos de snapshots.

- **Actualización:** Componente que toma las decisiones usando una Choice Function, ésta determina el desempeño (o rendimiento) de una estrategia dada en una determinada cantidad de tiempo. Ésta puede ser calculada basada en los indicadores entregados por el componente análisis.

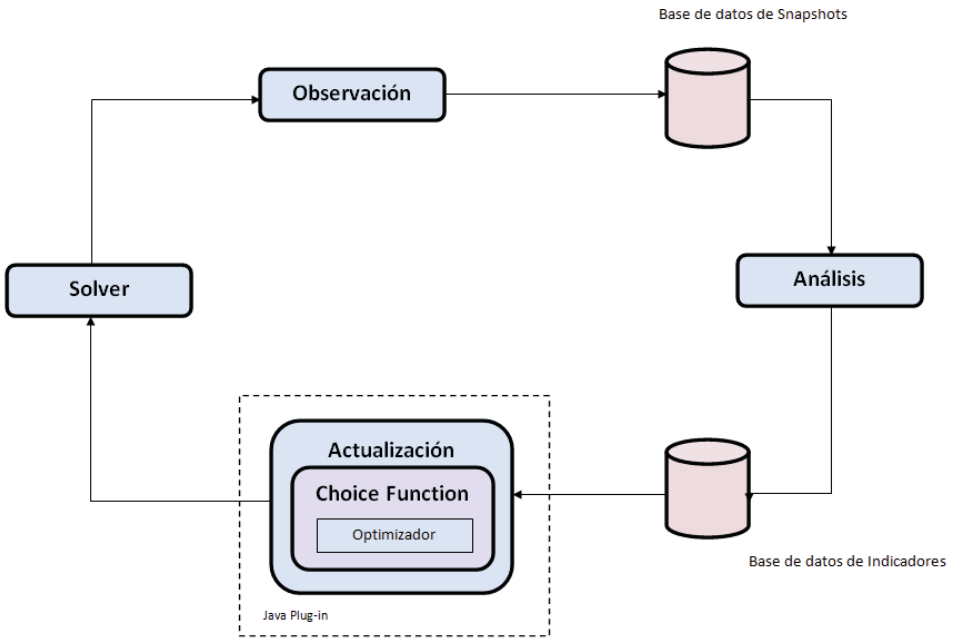

Figura 17: Diagrama del Framework actual [12].

5.2 Choice Function

La choice function tiene como objetivo hacer un ranking con las múltiples estrategias de enumeración, dependiendo del rendimiento, según indicadores, que cada una de éstas tiene. Así se podrá elegir entre las diferentes estrategias de enumeración en cada paso.

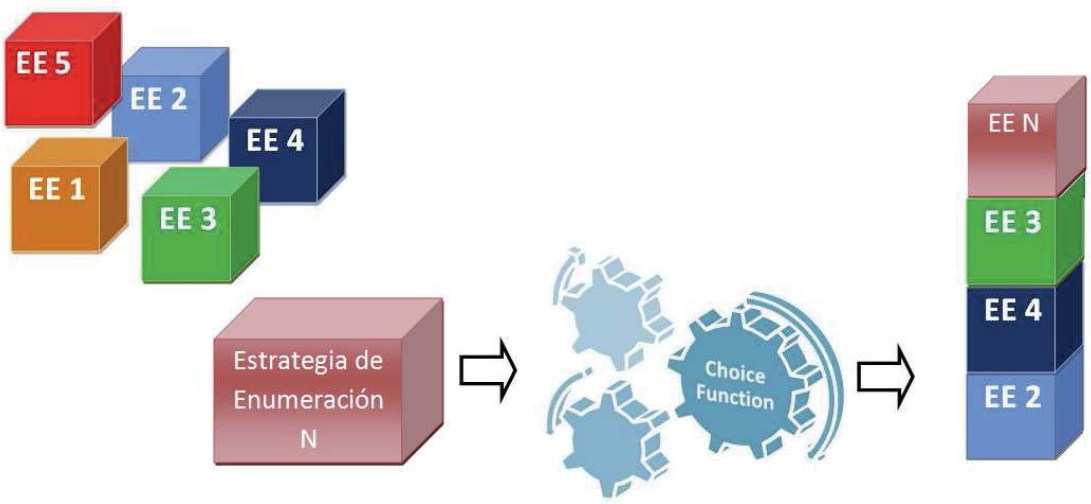

Figura 18: Esquema estrategias de enumeración y choice function

La figura 18 muestra el comportamiento general de la interacción entre las estrategias de enumeración y la choice function, ésta intenta capturar la correspondencia entre el comportamiento histórico de cada estrategia y el punto de decisión actualmente investigado, punto de decisión o paso, es cuando el solver se invoca para fijar una variable por enumeración. El esquema de la figura 18 muestra que la estrategia N será analizada por la choice function que tras el cálculo, deja a la estrategia N con el mayor valor de choice function (o mayor prioridad).

6. Skyline

La metodología Skyline ayuda a tomar decisiones teniendo múltiples alternativas a elegir, es por esto que su integración al componente actualización de la arquitectura de Autonomous Search, especialmente como Choice Function, ayudará en el proceso de selección de la siguiente estrategia a elegir para continuar con el proceso de resolución del problema.

Suponiendo que se necesita hospedaje en la ciudad donde se encuentra la universidad a la que se asistirá, se necesita que la pensión cumpla con dos características primero de bajo costo y segundo que esté cerca de la universidad. Después de haber visto las algunas pensiones que se encuentran en la ciudad, nos interesaremos por aquellas que más se ajusten de algún modo a las dos características que nos interesan. A este conjunto de pensiones que nos interesa se le llama Skyline. Del Skyline se puede tomar la decisión final.

Formalmente Skyline es definido como aquellos puntos que no son dominados por otros puntos.

Definición: Espacio numérico D definido mediante un conjunto de d dimensiones $\{d_1, d_2, d_3, \dots, d_d\}$ y un conjunto de datos $S \in D$.

Un dato $p \in S$ puede ser representado como un punto $p = \{p[1], p[2], \dots, p[d]\}$ donde $p[i]$ es un valor de la dimensión d_i.

Dominancia: Un punto $p = \{p[1], p[2], \dots, p[d]\}$ domina a otro punto $q = \{q[1], q[2], \dots, q[d]\}$ ssi $p[i] \leq q[i]$ $1 \leq i \leq d$ y existe al menos una dimensión j tal que $p[j] < q[j]$.

Skyline (S) es el conjunto de puntos no dominados por otros.

Ejemplo: Considerando la tabla 1. Se establece a continuación el Skyline para el problema.

Pensión	Precio	Distancia (km)
A	$350	0.5
B	$250	1
C	$150	2.5
D	$100	4
E	$400	1.5
F	$350	3
G	$250	3.5

Tabla 1: Datos problema para Skyline

Dada la tabla se puede establecer el gráfico (ver Figura 19) asociado con los puntos en el plano y el Skyline para este ejemplo.

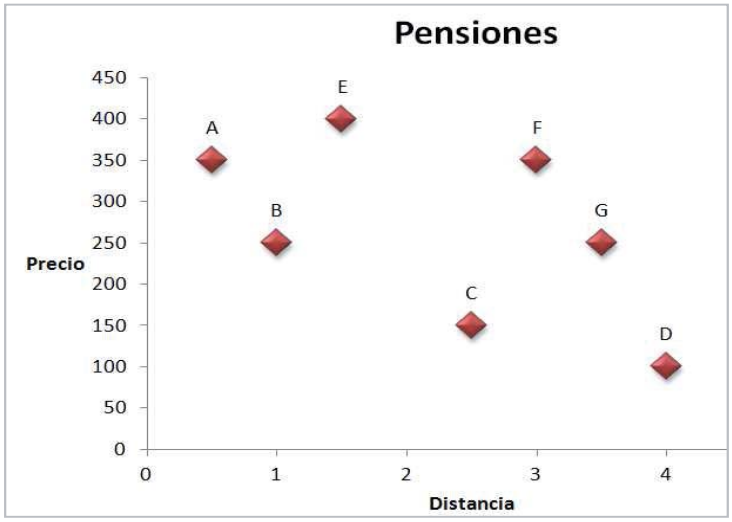

Figura 19: Gráfico con puntos Skyline [16].

El gráfico muestra cuales son los puntos que no son dominados por otros, estos son los puntos A, B, C y D que claramente en la relación distancia/precio cumplen con la definición propia de dominancia. En consecuencia estos puntos de dominancia son los que se debe tener en cuenta para determinar la solución al problema de elegir una pensión para un estudiante. Uno de los puntos (o pensiones) A, B, C o D según algún criterio determinado será el punto que dé la solución al problema (dado que todos los puntos de dominancia son posibles soluciones).

6.1 Algoritmos Skyline

Existe una cantidad muy acotada del algoritmos para encontrar el Skyline, entre algunos de estos se encuentran los algoritmos Block-Nested-Loops y Divide and Conquer ambos algoritmos poseen características diferentes en cuanto a la metodología que usan para encontrar el Skyline, el primero se basa en una ventana donde son ingresadas las tuplas dominantes, en cambio el segundo divide el espacio en que se disponen los datos para acotar el espacio de trabajo en la detección del Skyline. Ambos de describen con mayor detalle a continuación.

6.1.1 Algoritmo Block-Nested-Loops

Este algoritmo llamado BNL por sus siglas, tiene un funcionamiento sencillo, ya que aplica el algoritmo de ciclos anidados y compara cada tupla con las demás. El algoritmo de ciclos anidados se basa en aplicar a cada tupla recursivamente la definición formal que define la dominancia (descrita en el capítulo 6), este algoritmo puede ser aplicado no solamente en Skyline bi-dimensionales, pero obviamente será más ineficiente. El algoritmo BNL es significativamente más rápido, ya que produce un bloque de tuplas Skyline en cada iteración, en lugar de considerar una tupla a la vez. Block-Nested-Loops lee repetidamente un conjunto

de tuplas. La idea principal de este algoritmo es mantener una *ventana* de tuplas incomparables en la memoria principal. Cuando una tupla *p* es leída desde la entrada, *p* es comparada (buscando dominancia) con todas las tuplas de la *ventana* y, basada en esta comparación, *p* es eliminada, colocada en la ventana o en un archivo temporal (usada principalmente para Skyline con bases de datos). Tres casos pueden ocurrir:

- *p* es dominado por una tupla que está dentro de la ventana. En este caso *p* es eliminada y no será considerada en futuras iteraciones. En consecuencia *p* no necesita ser comparada con otras tuplas de la *ventana*.

- *p* domina una o más tuplas de la *ventana*. En este caso esas tuplas (dominadas) son eliminadas de la *ventana* y no serán consideradas en futuras iteraciones. La tupla *p* es insertada en la ventana.

- *p* es insertada en la ventana si ésta no posee ninguna tupla con la cual establecer dominancia. Por ejemplo en el caso que en sólo una dimensión sea menor.

Algoritmo: Block Nested Loops

```
 1: load_tuples()
 2: while tuple do
 3:    if window_is_empty() then
 4:       insert_on_window(tuple)
 5:    else
 6:       if dominance(tuple) then
 7:          delete()
 8:          insert_on_window(tuple)
 9:       else delete_tuple()
10:       end if
11:       if tuple_is_incomparable() then
12:          insert_on_window(tuple)
13:       end if
15:    end if
14: end while
```

Figura 20: Algoritmo BNL

6.1.1.1 Variantes para el algoritmo BNL

El objetivo de las variantes es obtener mayor eficiencia para el algoritmo, éstas son hechas en base a distintos comportamientos que pueden presentar las tuplas que serán analizadas y que en base a ello puede lograr por ejemplo minimizar el tiempo para cada iteración.

- **Mantener la ventana como una lista auto-organizada:** Una gran cantidad de tiempo en el algoritmo anterior es gastado en comparar una tupla con las tuplas que están en la ventana. Para poder acelerar este procedimiento se puede organizar la ventana como una lista que se auto-organice. Por ejemplo cuando una tupla *w* de la ventana se encuentra a dominar otra tupla, entonces *w* es trasladada al inicio de la ventana. Como resultado la siguiente tupla será comparada con *w* primero. Esta variante es particularmente efectiva si existen tuplas en la ventana que dado su valor en las dimensiones presentes serán capaces de dominar la mayor cantidad de tuplas dadas esas características.

- **Reemplazar tuplas en la ventana:** Esta variante trata de mantener el conjunto de tuplas más dominante en la ventana. Resulta ser más efectiva cuando la ventana está llena y si la siguiente tupla a analizar es incomparable con las que están presentes en la ventana.

6.1.2 Algoritmo Divide and Conquer

Este algoritmo trabaja de la siguiente forma:

1. Calcular la media m_p (o una aproximación a la media) de la entrada para una dimensión d_p. Dividir la entrada en 2 particiones donde p_1 contiene todas las tuplas cuyo valor del atributo d_p es mejor (por ejemplo menor) que m_p. $p_2$contiene todas las otras tuplas.

2. Calcular los Skylines S_1 de p_1 y S_2 de p_2 esto es hecho recursivamente aplicando todo el algoritmo a p_1 y p_2 (por ejemplo p_1 y p_2 son nuevamente particionados). La partición recursiva parará si una partición contiene una o muy pocas tuplas, en este último caso el cálculo del Skyline es trivial.

3. Calcular el Skyline total como resultado de la fusión de S_1 y S_2 eliminando aquellas tuplas de S_2 que son dominadas por una tupla de S_1. Cabe destacar que ninguna de las tuplas presentes en S_1 puede ser dominada por una tupla en S_2 porque una tupla en S_1 es mejor en la dimensión d_p que cualquier tupla de S_2.

6.1.2.1 Extensiones para Divide and Conquer

- **M-way partitioning:** Se utiliza principalmente cuando se tiene un espacio limitado y se particiona *m* veces hasta que los puntos encajen en el espacio asignado. Ésta extensión puede ser usada en el primer paso o en el tercer paso del algoritmo Divide and Conquer. En el primer paso m-way partitioning es usado para producir *m* particiones $p_1 \ldots p_m$ de manera que cada p_i encaja en la memoria y S_i el Skyline de p_i, puede ser calculado en la memoria utilizando el algoritmo (Divide and Conquer). La respuesta final es consecuencia del tercer paso mediante la fusión de pares S_i. Dentro de la función fusión se aplica esta extensión, de manera que todas las sub-particiones puedan ser puedan ser combinadas en la memoria principal.

- **Early Skyline:** Otra extensión de Divide and Conquer para situaciones en las cuales la memoria principal disponible es limitada. Se llevan a cabo los siguientes pasos:

1. Cargar un block largo de tuplas en la entrada, es decir cargar tantas tuplas de tal manera que encajen en los buffers de memoria principal disponible.
2. Aplicar Divide and Conquer a este block de tuplas en orden para eliminar inmediatamente las tuplas que son dominadas por otras a esto se refiere “early skyline”.
3. Particionar las tuplas restantes en *m* particiones.

Esta extensión posee ventajas y desventajas, claramente aplicando esta extensión se incurre en costo adicional de CPU, pero por otro lado ahora I/O porque menos tuplas necesitan ser escritas y releídas en los pasos de partición. Early Skyline es atractivo cuando el Skyline es selectivo, por ejemplo si el resultado de todo el Skyline es pequeño.

7. Implementación de Choice Function Utilizando Skyline

Las siguientes secciones describen el proceso de creación, desarrollo e implementación de la nueva choice function utilizando la metodología Skyline para la arquitectura de Autonomous Search.

7.1 Choice Function Utilizando Skyline: Ideas Preliminares

Para crear la choice function se necesitan los indicadores, utilizando la metodología Skyline no es necesario tener un parámetro de control de la relevancia para estos indicadores, esto debido a que Skyline puede minimizar las estrategias ineficientes que aparecerán en el espacio de n dimensiones. Las dimensiones serán la o las estrategia que integre la nueva choice function. Todos los puntos generados en el plano serán analizados por algún algoritmo especial para Skyline, para así determinar los puntos de dominancia, es decir aquellos que cumplen con el objetivo.

Entonces como una choice function es la encargada de rankear las estrategias según sus indicadores, algún punto del Skyline generará dicha prioridad.

Dada esa información se puede definir en primera instancia algunas de los datos que la choice function tendría:

S_j : La estrategia de enumeración j .
$f(S_j)$: podría minimizarse por ejemplo: la menor cantidad de nodos visitados.

Ejemplo 1: $p_1 = f(S_1) + f(S_2)$ y $p_2 = f(S_1) + f(S_2)$

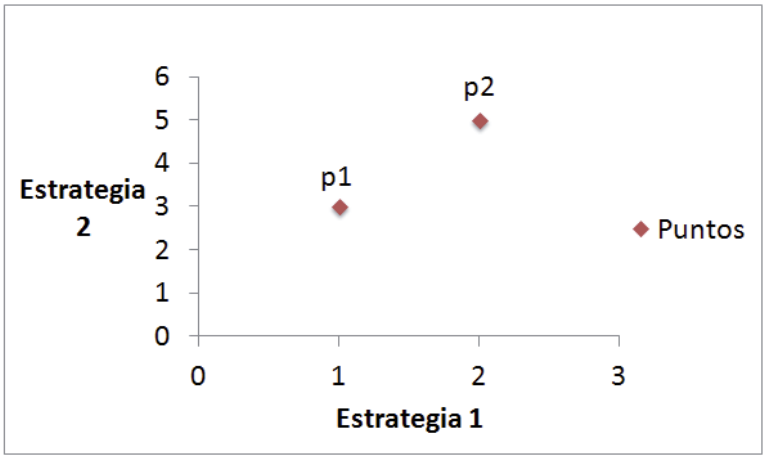

Figura 21: Gráfico representando Skyline entre dos puntos (ejemplo 1).

Entonces utilizando Skyline, los puntos p_1 y p_2 (ver figura 21) del plano de 2 dimensiones, dadas dos estrategias S_1 y S_2, representarán puntos en plano que determinando dominancia, serán los puntos cuyo rendimiento es mejor para la aplicación de las estrategias. En este caso el punto que domina al otro es p_1.

Ejemplo 2: Las dimensiones corresponden a la cantidad de indicadores presentes en el análisis del problema. Cada estrategia posee distinto comportamiento para cada indicador, por lo cual es posible calcular el Skyline de las estrategias utilizando el valor de sus indicadores. Esto se puede ver representado en la figura 22, donde dadas dos dimensiones correspondientes al indicador 1 y 2, se disponen los puntos en el plano (estrategias) con las cuales es posible determinar dominancia.

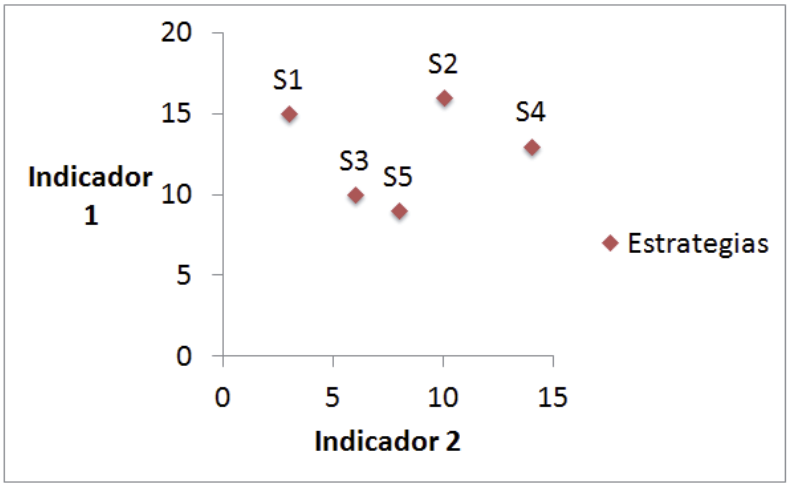

Figura 22 : Gráfico representando el Skyline de las estrategias (ejemplo 2).

La idea principal es ocupar la ventaja de Skyline de obtener un conjunto de puntos que sirven para el propósito (elegir las mejores estrategias) y de ellos tomar decisiones sobre cuál tendrá una determinada prioridad.

7.2 Prototipo de Choice Function Utilizando Skyline en Arquitectura de AS

El prototipo desarrollado posee las características que se presentan en la figura 23, donde se muestra de manera general la secuencia de acciones que se llevan a cabo en el prototipo de la choice function basada en Skyline. En primera instancia se cargan las estrategias con los valores de los respectivos indicadores, los que serán elegidos por el usuario a través del checkbox de la interfaz, a continuación en la línea 2 la condición muestra que se realiza el análisis de correlación, si existe correlación entre los indicadores entonces se deshabilita algún indicador que presente correlación (por ejemplo; si el indicador a y el indicador b presentan correlación, entonces uno de estos dos se deshabilitará), por lo tanto no formará parte del algoritmo Skyline que se realiza a continuación. En la línea 5 se realiza el método block nested loops el cual es el encargado de encontrar el Skyline de las estrategias, usando los indicadores como dimensiones, este método devuelve una ventana (`window`) con las estrategias que forman el Skyline. Después (línea 6) se verifica si la ventana contiene más de una estrategia, si es así es necesario elegir la estrategia a utilizar (`choose_strategy()`) para seguir el desarrollo del problema. En caso contrario, es decir la ventana posee sólo un elemento (estrategia), ésta misma será la elegida. Finalmente para cualquiera de las condiciones anteriores el método SkylineChoice() retorna la estrategia a usar en ese paso del problema.

Algoritmo: SkylineChoice

```
 1:  load_strategies_and_indicators();
 2:  if correlation_analysis(indicators)then
 3:     indicator_disable()
 4:  end if
 5:  window ← block_nested_loops()
 6:  if window > 1 then
 7:    choose_strategy(window)
 8:  return strategy_to_use
 9:  end if
10:  if window == 1 then
11:  strategy_to_use ← window
12:  return strategy_to_use
13:  end if
```

Figura 23: Algoritmo Procedimiento Skyline Choice.

7.3 Análisis de Correlación a los Indicadores

En la choice function se utilizan múltiples indicadores lo que representa un alto costo. Por lo tanto realizar un análisis de correlación para detectar aquellos indicadores que no son útiles, ya que se comportan como otro indicador es muy beneficioso para el desarrollo de la choice function como tal.

La principal tarea de un análisis de correlación es detectar pares o conjuntos de indicadores, altamente relacionados y por lo tanto equivalentes.

Para hacer el análisis es necesario utilizar el Pearson Correlation Coefficient, que es sensible a una relación lineal entre dos variables, ésta es obtenida dividiendo la covarianza de dos variables por el producto de sus desviaciones estándar. El Pearson Correlation es definido sólo si ambas desviaciones estándar son finitas y distintas de cero. El coeficiente de correlación es simétrico, es decir correlación(x,y)=correlación(y,x). En la tabla 2 se muestra el grado de correlación.

Correlation Coefficient	**Grado de correlación**
1	Perfecta
0	Menor
-1	Anticorrelación

Tabla 2: Grados de Correlación.

Se establece el criterio de pares de indicadores altamente relacionados con un Correlation Coefficient mayor o igual a 0.9. Luego, de los indicadores altamente relacionados se elige sólo uno de estos.

La figura que se muestra a continuación (figura 24) posee el algoritmo del análisis de correlación utilizado en el prototipo, este algoritmo consta de tres métodos el primero `analisisDeCorrelacion()` donde se llama a los otros dos métodos covarianza() y `desvEstandar()` estos últimos son los necesarios para hacer el análisis del coeficiente de correlación. En los últimos dos métodos referenciados se realiza el cálculo de la covarianza de dos indicadores representados por la variables enteras *i* e *ii* éstas representan las columnas donde se encuentran los indicadores que serán analizados y en el siguiente método se realiza el cálculo de las desviaciones estándar de los dos indicadores, el retorno de este es el producto de las desviaciones estándar de ambos indicadores. Donde se aplica el análisis de correlación como su definición lo presenta, es en el método `analisisDeCorrelacion()` donde si el producto de las desviaciones estándar calculadas es distinto de cero entonces se calcula el coeficiente `PearsonCorrelation` que muestra el grado de correlación de los indicadores analizados, que para el objetivo de este proyecto sólo es necesario si entrega la información de que la correlación es positiva, es decir si la correlación existe entre ambos indicadores el retorno corresponderá a true. De lo contrario para el caso que la desviación estándar sea cero o haya anticorrelación el retorno del método es false. Resultado que será ocupado en el método *SkylineChoice()* descrito más adelante en este proyecto.

Algoritmo: Análisis de Correlación

```
public boolean analisisDeCorrelacion(int i, int ii){
        double PearsonCorrelation=0;
        if(desvEstandar(i,ii)!=0){//no se puede realizar análisis
        PearsonCorrelation=covarianza(i,ii)/desvEstandar(i,ii);
        }else{
            return false;
        }
        if(PearsonCorrelation>=0.9){
           System.out.println("correlacion perfecta positiva");
           return true;//altamente correlacionados
        }else {
            return false; //anticorrelación
        }
    }
    public double covarianza(int i, int ii){
        double covarianza=0;
        double promedio1=0;
        double promedio2=0;
        double parcial=1;
        double sumatoria=0;
        int n=cantEstrategias;// cantidad de indicadores
            for (int k=0; k<cantEstrategias;k++){
             promedio1=promedio1 + dimensionIndicadores[k][i];
             promedio2=promedio2 + dimensionIndicadores[k][ii];
             }
        promedio1=promedio1/n;
        promedio2=promedio2/n;
         for(int k=0;k<cantEstrategias;k++){
            parcial=parcial*dimensionIndicadores[k][i];
            parcial=parcial*dimensionIndicadores[k][ii];
            sumatoria= sumatoria+parcial;
            parcial=1;
        }
     covarianza=sumatoria/cantEstrategias-promedio1*promedio2 ;
        return covarianza;
    }
```

```
public double desvEstandar(int i, int ii){
    double desvEstandar1=0;
    double desvEstandar2=0;
    double promedio1=0;
    double promedio2=0;
    double parcial1=0;
    double parcial2=0;
    int n=cantEstrategias;// cantidad de indicadores
    for (int k=0; k<cantEstrategias;k++){
        promedio1= promedio1 + dimensionIndicadores[k][i];
        promedio2= promedio2 + dimensionIndicadores[k][ii];
    }
    promedio1=promedio1/n;
    promedio2=promedio2/n;
    for (int k=0; k<cantEstrategias;k++){
   parcial1=parcial1+Math.pow(dimensionIndicadores[k][i]-promedio1,2);
   parcial2=parcial2+Math.pow(dimensionIndicadores[k][ii]-promedio2,2);
        }
    desvEstandar1=parcial1/n-1;
    desvEstandar1=Math.sqrt(desvEstandar1);
    desvEstandar2=parcial2/n-1;
    desvEstandar2=Math.sqrt(desvEstandar2);
    return desvEstandar1*desvEstandar2;
}
```

Figura 24: Código Java para Análisis de Correlación.

7.4 Modificaciones a la Herramienta

Se implementaron modificaciones en el código de la herramienta para así poder insertar la nueva choice function. Principalmente modificaciones para poder implementar el método SkylineChoice. Aquí se describen las modificaciones más importantes.

- Modificaciones en la navegación de las interfaces (ver sección 7.9.1).
- Se insertó al combobox el nuevo tipo de solución de nombre SkylineChoiceFunction.

```
jComboBox2.addItem("SkylineChoiceFunction");
datos_problema.tipo_solucion=jComboBox2.getSelectedItem().toString();
```

Figura 25: Nuevo tipo de solución.

- En la clase resolver se creó un método, que es análogo a los otros tipos de soluciones (one strategy, random, etc.) llamado `Skyline()` que es el encargado, entre otras funciones, de llamar a la choice function que utiliza Skyline.

```
public void Skyline(){
    try{
jegapv20.tabla_estado.modelo.setValueAt(calendario.get(Calendar.HOUR)+":"+c
alendario.get(Calendar.MINUTE)+":"+calendario.get(Calendar.SECOND)+":"+cale
ndario.get(Calendar.MILLISECOND),jegapv20.datos_problema.numero_prueba_actu
al, 2);
            Indicadores.ind_tiempoIniciomili=System.currentTimeMillis();
            Indicadores.IndicadoresInicio();
            Indicadores.rnd_seed_contructor();
            Indicadores.rnd_seed_set(Indicadores.semilla2);
            Indicadores.genetico = false;
            Indicadores.choicefunction = 5;
            Indicadores.InicioEstrategias();
                if(Indicadores.outputType==1){
                    Indicadores.crearTxt();
                    eclipse.ejecute(Indicadores.problema_a_resolver);
                }else{
                    if(Indicadores.outputType==0){
                        Indicadores.crearPlanilla();
                        Indicadores.genetico = false;
                        eclipse.ejecute(Indicadores.problema_a_resolver);
                        Indicadores.addporcentajes();
                        Indicadores.graficos2();
                    }
                }
            }catch(Exception ex){
              System.out.println(ex);
           }
}
```

Figura 26: Método Skyline en Clase Resolver.java.

7.5 Método SkylineChoice()

Este método se encuentra en la clase indicadores.java, ya que es ahí donde se disponen las choice functions y también donde se actualiza cada uno de los indicadores, tanto los base como los calculados. Los indicadores base se calculan en ECLiPSe, luego esta información es enviada a la herramienta, que la traduce y calcula los demás indicadores.

El método SkylineChoice será ejecutado las veces que sea necesario obtener una estrategia a utilizar para seguir el desarrollo de la solución del problema. Las características se describen a continuación:

- Las estrategias y sus respectivos indicadores están en una matriz donde la columna 0 de cada fila representa la estrategia, y las siguientes columnas son cada uno de los indicadores. Análogamente para la ventana del algoritmo BNL.

- Se realiza un análisis de correlación sobre los indicadores y si es el caso se desactiva alguno de estos, para ello se utiliza un arreglo con cada uno de los indicadores que permite saber con cuales indicadores se está trabajando, el valor del indicador será true si se marcó en el checkbox, de lo contrario, false.

- Empieza la ejecución del algoritmo BNL como tal, para esto se compara cada estrategia con sus indicadores y la ventana.

- El resultado del algoritmo BNL será una ventana con las estrategias dominantes, de entre las cuales se elegirá una para continuar el desarrollo de la solución.

7.6 Prototipo de Interfaz de la Nueva Choice Function

Al agregar la nueva choice function basada en Skyline, fue necesario hacer pequeñas modificaciones en la interfaz para así poder agregar la nueva opción de choice function. Se agregó al combobox (ver figura 27) la opción de *SkylineChoiceFunction,* la modificación se efectuó debido a que dadas sus características no era necesaria la cantidad de ventanas que se requiere para cualquier otro tipo de solución.

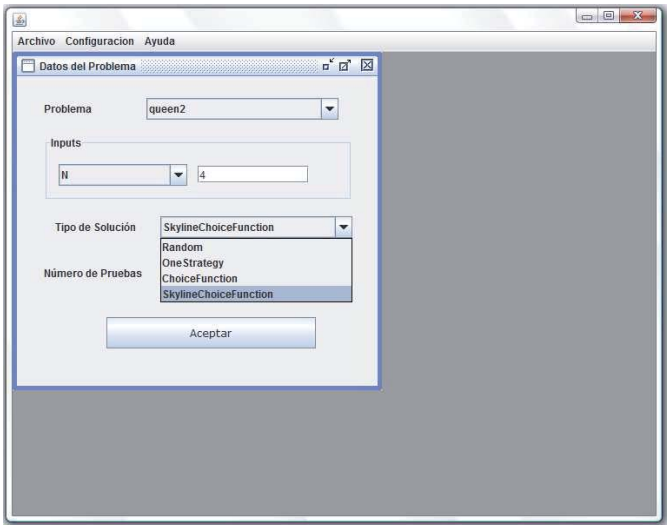

Figura 27: Interfaz Datos del Problema

Luego de haber completado los datos del problema aparece un checkbox (ver figura 28) con las estrategias que se quieren utilizar para el desarrollo del problema. Para el tipo de solución *SkylineChoiceFunction* es necesario marcar un mínimo de dos estrategias, de lo contrario no es posible realizar la metodología Skyline.

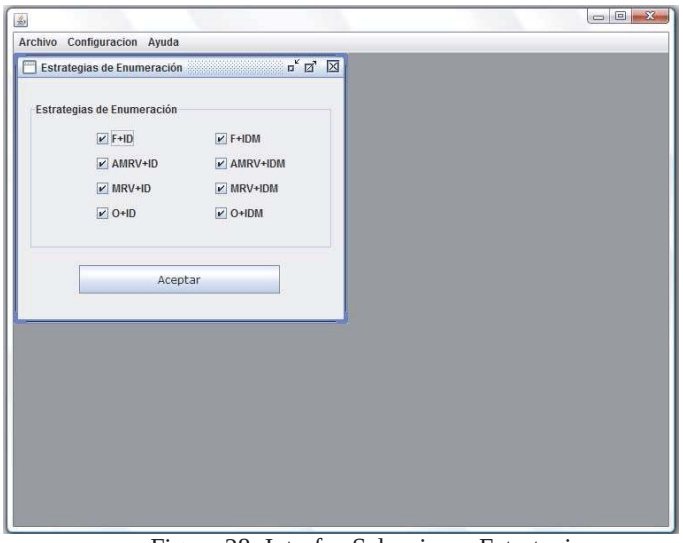

Figura 28: Interfaz Seleccionar Estrategias.

Una vez seleccionadas las estrategias de enumeración aparecen los *Indicadores de ChoiceFunction* donde se ve un checkbox con la cantidad de indicadores existentes de los cuales es necesario seleccionar preferentemente un mínimo de dos, sin embargo es posible seleccionar sólo un indicador (por ejemplo cualquier indicador base). El asistente se encuentra inactivo ya que éste sólo posee choice functions para resolver con optimizador o bien ingresando los valores que éste provee manualmente.

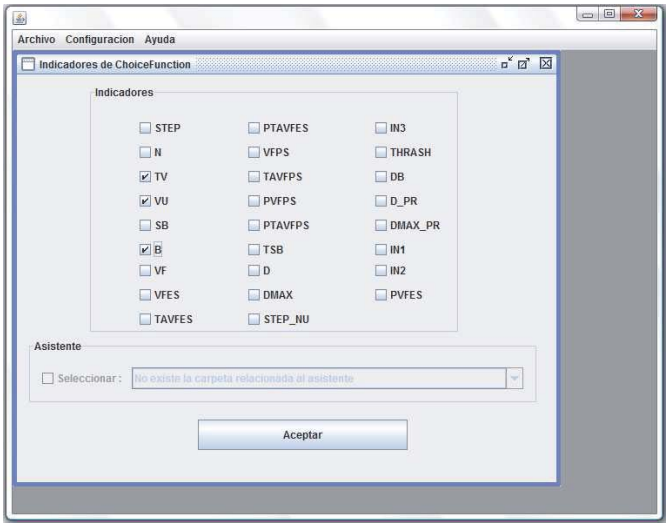

Figura 29: Interfaz Indicadores de ChoiceFunction.

Finalmente se procede a la resolución del problema planteado con el tipo de solución *SkylineChoiceFunction*, con la interfaz común para cualquier tipo de solución. Como lo muestra la figura 30.

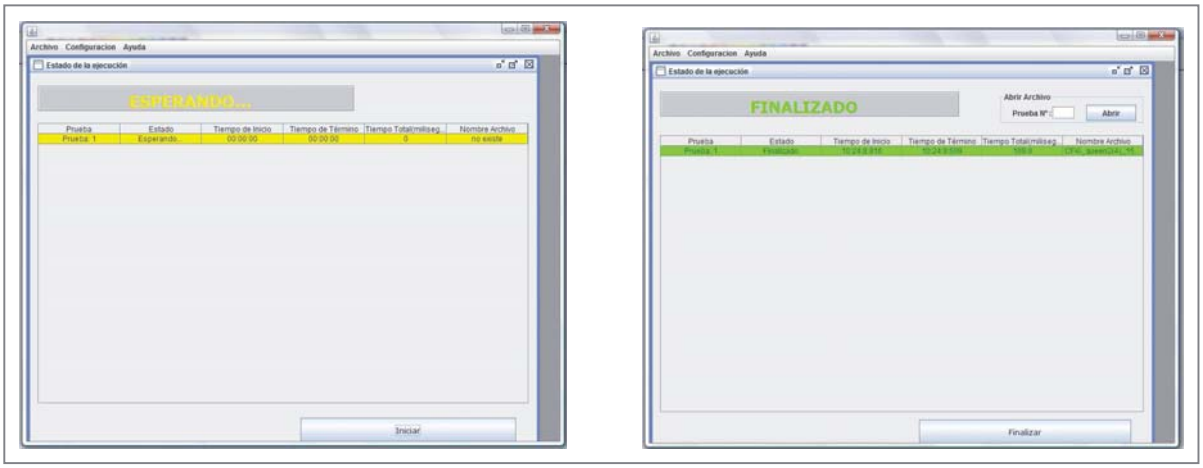

Figura 30: Interfaz Estado de la ejecución.

7.6.1 Restricciones del Prototipo

Para el prototipo de la choice function es necesario tener en cuenta las restricciones que posee para su óptimo funcionamiento.

- Sólo se pueden seleccionar los indicadores base (ver Anexo A).
- El análisis de correlación se realiza sólo sobre un par de indicadores.
- De la ventana Skyline que resulta del algoritmo BNL se selecciona una estrategia al azar.

7.7 Smoothing Factors

Los smoothing factors se utilizan para controlar la relevancia de los indicadores en cada una de las estrategias, es decir controlar la relevancia de un indicador i para una estrategia S_j en un periodo de tiempo se utiliza la técnica estadística llamada suavizado exponencial.

La idea principal es asociar mayor importancia a los rendimientos más recientes, decreciendo exponencialmente los pesos de las pasadas observaciones.

El smoothing factor es aplicado al cálculo de $f_{in}(S_j)$ el cual es definido por las ecuaciones (2) y (3), donde V_{i1} es el valor del indicador i para la estrategia S_j en el tiempo 1, n es el step del proceso y β es el factor de suavizado y está en el rango $0 < \beta < 1$.

$$f_{i_1}(S_j) = V_{i_1} \tag{1}$$

$$f_{i_n}(S_j) = V_{i_1} + \beta_i f_{i_{n-1}}(S_j) \tag{2}$$

$$f_{i_n}(S_j) = V_{i_n} + \beta_i V_{i_{n-1}} + \beta_i V_{i_{n-2}} \tag{3}$$

7.7.1 Smoothing Aplicado a Skyline

Dada la metodología de Skyline realizar el smooth de la manera convencional no es apropiado para esta misma, por lo cual se realizó de manera tal que sobre cada indicador seleccionado se aplica el smooth, como lo muestra la figura 31.

Algoritmo Smoothing sobre los indicadores para Skyline

```
1:if(activarSmooth == true){
2: if(indicador[actual][step]<=estrategiasIn){
3:  for(int i=1;i<cantIndicadores;i++){
4:   if(indicadorUtilizado[i])smoothingFactor[a][i-
5:                          1]=estrategiasEindicadores[a][i]*smoothBeta[i-1];
6:  }
7: }
8:   if(indicador[actual][step]>estrategiasIn){
9:     for(int i=1;i<cantIndicadores;i++){
10:      double aux=estrategiasEindicadores[a][i];
11:estrategiasEindicadores[a][i]=estrategiasEindicadores[a][i] +
12:                                          smoothingFactor[a][i-1];
13: smoothingFactor[a][i-1]=smoothingFactor[a][i-1]+ (aux*smoothBeta[i-1]);
14:                      smoothBeta[i-1]=smoothBeta[i-1]*smoothBeta[i-1];
15:     }
16:  }
17:}
```

Figura 31: Smoothing para Skyline.

7.8 Trade-Off Skylines [18]

Se presenta como un método eficiente para el cálculo de skylines permitiendo una compensación cualitativa entre los atributos. Aquí los usuarios proveen información sobre cuánto están dispuestos a sacrificar para obtener mejoras en algún otro atributo(s).

7.8.1 Pareto Skylines

Asumiendo una relación de la base de datos: $R \subseteq D_1 \times ... \times D_n$ con n atributos.

- Una preferencia P_i sobre un atributo A_i con dominio D_i es un orden parcial estricto sobre D_i. Si algún valor del atributo $a \in D_i$ es preferente sobre algún otro valor $b \in D_i$ entonces $(a, b) \in P_i$. Esto es escrito como $a >_i b$, que significa a domina b con respecto a P_i.
- Análogamente, una equivalencia Q_i es una equivalencia en relación a D_i compatible con P_i . Si dos valores de los atributos $a, b \in D_i$ son equivalentes, por ejemplo $(a, b) \in Q_i$ escribimos $a \approx_i b$.
- Finalmente, si un valor de un atributo $a \in D_i$ se prefiere sobre o equivalente a otro valor $b \in D_i$, escribimos $a \gtrsim_i b$.

Como ejemplo, asumiendo alguna preferencia transitiva para la compra de un automóvil y considerando los atributos precio, color, caballos de fuerza y aire acondicionado. Para

calcular el Skyline, el atributo de preferencia puede ser agregado con respecto a la semántica Pareto resultando el orden completo del producto P. Este orden puede entonces ser usado para testear cualquier objeto de la base de datos domina a cualquier otro objeto (en este caso si el primer objeto se muestra mejor o al menos igual rendimiento con respecto a todos los atributos que el segundo objeto).

7.8.2 Trade-Off Skylines

Trade-offs pueden ser considerados como la decisión de un usuario entre dos objetos de ejemplo enfocándose en un subconjunto de atributos disponibles. Por ejemplo, considerando el dominio de autos, un usuario podría centrarse en los atributos color y precio. Un trade-off entonces describe de manera cualitativa cuánto un usuario está dispuesto sacrificar en alguna dimensión (es), ejemplo: Un posible trade-off t_1 podría ser "Prefiero un auto por 18000 UM con pintura azul metálico que un auto por 16000 UM y azul claro", entonces escribimos $t_1 := ((18000,\ azul\ metálico) \rhd (16000, azul))$.

Formalmente un trade-off t es siempre definido sobre un conjunto de atributos dados por los respectivos atributos denotados como $\mu \subseteq \{1, \dots, n\}$.

Un trade-off es la relación entre dos tuplas $x, y \in D_\mu$ denotada como $t := (x \rhd y)$. Obviamente los dos componentes del trade-off x e y son incomparables, por ejemplo no existe relación de dominancia $x \geq_p y$, ni $x \geq_p y$ existentes entre ellas. Cada trade-off es definido sobre un atributo individual del conjunto μ.

7.8.3 Algoritmo Trade-Off Skyline

Existe un algoritmo básico para el cálculo de trade-off skyline, éste representa un subconjunto del Pareto Skyline. El algoritmo básico (ver figura 32) primero calcula el Pareto Skyline y luego filtra todos los objetos adicionales que son dominados por otros objetos vía cualquier secuencia de trade-off, los parámetros de este algoritmo corresponden a las preferencias sobre los atributos nombradas como $P_1, \dots, P_n$, otro parámetro corresponde al conjunto de trade-offs T.

Algoritmo: Cálculo Trade-Off Skyline

```
1: Compute_Pareto_Skyline()
2: Generate_Set_of_Trade_Off()
3: for_all o1 in Sky
4:     for_all o2 in Sky and o1≠o2
5:         if o1>_T test()
6:            remove_from_sky(o2)
7:         end_if
8:     end_for
9:end_for
```

Figura 32: Algoritmo básico trade-off skyline.

El algoritmo de la figura 33 corresponde a un método utilizado en el anterior (test ()) posee como parámetros la información asociada a los objetos o1 y o2, y todos los posibles trade-offs T'. Este método retorna true en casos como por ejemplo si o1 domina a o2 vía t_i, de lo contrario retorna false.

Algortimo: Basic Object Domination Test $>_T$

```
1: for_all trade-offs t_i ≔ (x ⊳ y) ∈ T'
2:      if((o1 ≳_p x ↶ o1)∧(y ↶ o1 ≳_p o2)) return true
3:end_for
4:return false
```

Figura 33: Algoritmo basic object domination test.

7.8.4 Secuencias Trade-Off

La idea es crear incrementalmente un árbol de estructuras (llamado trading tree, TTree), enumerando todas las actuales posibles secuencias trade-off. Cada nodo del árbol representa un trade-off el cual puede ser proveído por el usuario o por algún trade-off integrado. Inicialmente el árbol con una raíz vacía que corresponde al trade-off $t_\emptyset \coloneqq (() \rhd ())$ sobre un conjunto de atributos vacío representado por $\mu_\emptyset \coloneqq \emptyset$. Luego los trade-offs son insertados uno por uno desde el conjunto de los trade-offs T del usuario.

7.8.5 Trade-Off Skyline Aplicado a SkylineChoice()

Se utilizó la metodología trade-off skyline en el método de selección de una estrategia, es decir sobre aquellas estrategias presentes en la ventana del método BNL.

- Pareto Skyline: En primera instancia se debe hacer el Pareto Skyline, es decir se requiere tener ciertos valores de preferencia para algunos atributos, en este caso los atributos para los cuales se tendrán valores de preferencia son: backtracks (B), nodos visitados (N), shallow backtrack (SB), variables instanciadas por enumeración en cada paso (VF). La tabla 3 muestra el Pareto Skyline con su respectiva notación, en la primera columna se encuentran los cuatro atributos sobre los cuales se presentan preferencias y/o equivalencias.

Atributo	Preferencia
$A_1: b$	$P_1: b >_1 vf$ $P_2: b >_2 n$
$A_2: sb$	$P_3: sb >_3 vf$ $P_4: sb >_4 n$
$A_3: vf$	$P_5: vf \approx_5 n$
$A_4: n$	$P_6: n \approx_6 vf$

Tabla 3: Pareto Skyline.

- Trade-Offs: Los atributos que forman el trade-off son b y sb. El trade-off $t_1$corresponde a la afirmación "prefiero que la cantidad de backtracks y shallow backtrack sea menor, que la cantidad de backtracks sea mayor y los shallow bactrack menor"
  $\mu: b, sb$
  $t_1 \coloneqq \big((b\ menor, sb\ ) \rhd (b\ mayor, sb\ menor)\big)$

7.9 Ajuste e Integración a la Interfaz de la Herramienta

Dados los nuevos requerimientos para Skyline fue necesario crear nuevas interfaces que muestren las opciones para el desarrollo de un problema.

7.9.1 Nueva Interfaz Para Skyline Choice Function

- **Interfaz Datos del problema:** La única modificación es que se agregó el tipo de solución SkylineChoiceFunction al combobox de tipo de solución.

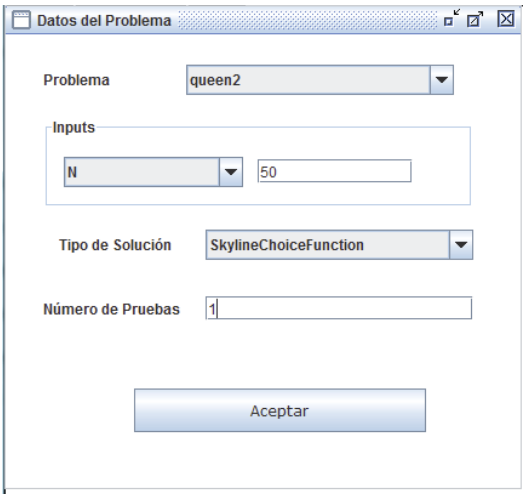

Figura 34: Interfaz datos del problema.

- **Interfaz Indicadores de Choice Function**: Se agregaron 5 nuevas choice functions para ser utilizadas con Skyline sin el indicador alfa, ya que en este caso no es necesario.

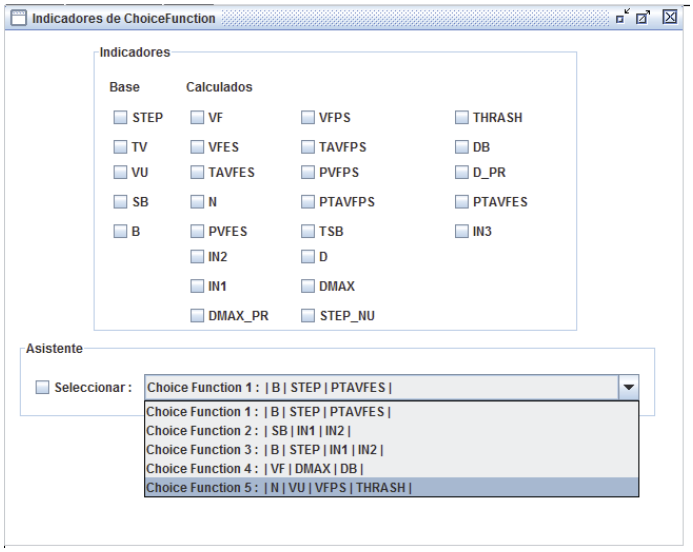

Figura 35: Interfaz indicadores de choice function.

- **Interfaz Smoothing Factors:** Se agregó la opción de realizar la actualización de los indicadores sin smoothing, sin embargo también es posible realizarlo si se desea.

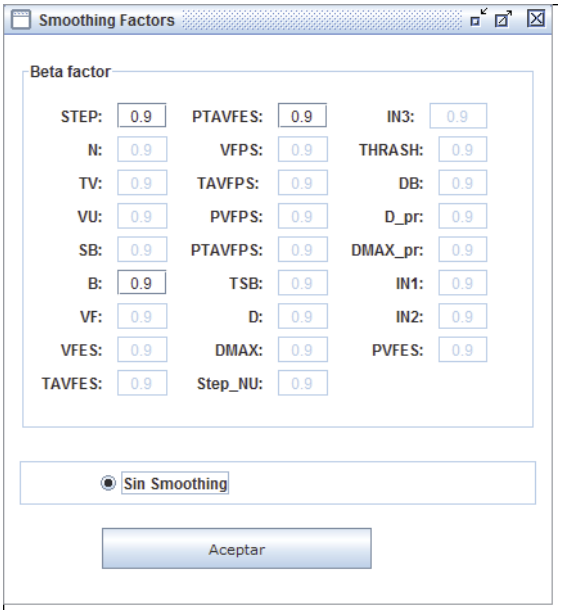

Figura 36: Interfaz smoothing factors.

- **Interfaz Análisis de Correlación:** Los indicadores que no se utilizan están desmarcados, sobre aquellos que sí están presentes en la choice function puede realizarse análisis de correlación, seleccionando el checkbox correspondiente.

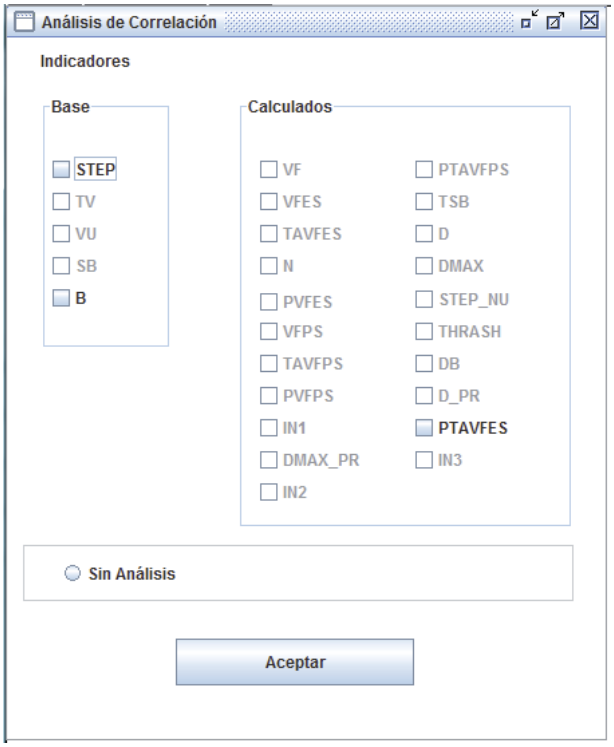

Figura 37: Interfaz análisis de correlación.

- **Interfaz Trade-Offs:** Es posible seleccionar uno o dos trade offs para un problema, éstos son seleccionados por el usuario en los combobox que muestran los indicadores utilizados en la choice function.

Figura 38:Interfaz trade-offs.

- **Interfaz Archivo de Salida:** Es posible seleccionar el tipo de archivo de salida, archivo Excel que además incluye gráficos ó archivo de texto que posee la misma información que el archivo Excel excepto por los gráficos.

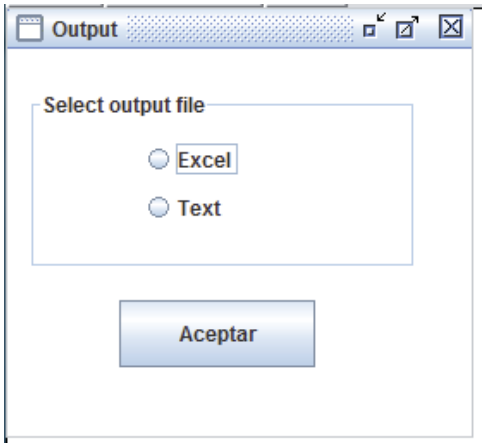

Figura 39:Interfaz archivo de salida.

7.11 Nuevo Archivo de Salida .txt

Se realizó un nuevo tipo de archivo se salida de texto plano, éste puede ser utilizado cuando un problema ejecuta una gran cantidad de steps, por ejemplo si el archivo Excel llega a su límite no se podrá saber si se llega o no a la solución, ya que producirá errores como una alternativa se utiliza el tipo de salida .txt ya que no tiene límite claro, sólo se crea el archivo y lo completa mientras exista memoria disponible.

La figura 40 muestra el comienzo del archivo donde se escriben los indicadores en cada step, esta información se agrega línea por línea mientras se desarrolla el proceso de resolución por lo tanto este tiempo es parte del tiempo de ejecución final del problema. La figura 41 muestra el resumen final del problema ejecutado, el tiempo de ejecución ya fue calculado, por lo tanto el tiempo de escritura de esta parte no se toma en cuenta en el runtime final.

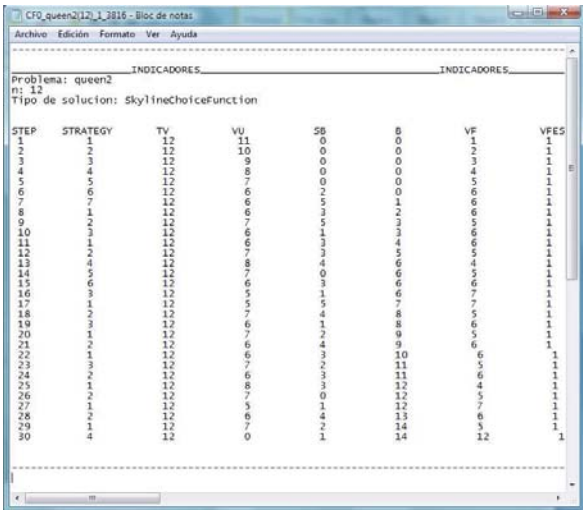

Figura 40: Archivo de resumen indicadores .txt.

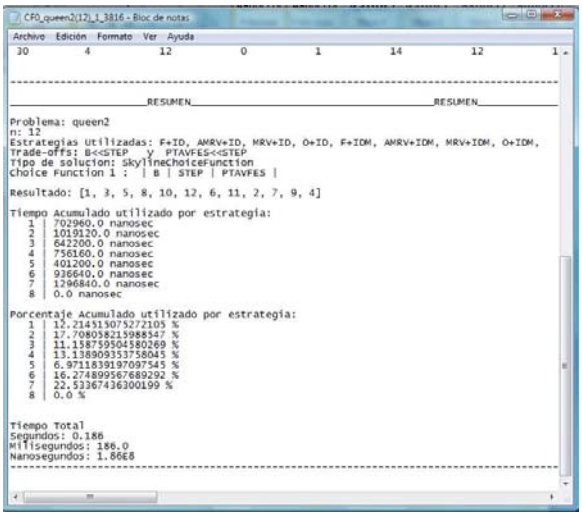

Figura 41: Archivo .txt resumen del problema.

8. Experimentos

Los experimentos se realizaron en un equipo bajo las siguientes características:

- Sin conexión a internet.
- Sin otros programas abiertos.
- Se realizan 10 pruebas por problema y se escoge la mejor.

Características del equipo:

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

Las pruebas fueron realizadas bajo las características nombradas anteriormente, con la versión ECLiPSe 6.0, se seleccionaron todas las estrategias en la mayoría de los problemas, excepto donde se indica posteriormente, no se realizó smoothing.

Cada uno de los problemas se ejecutó 10 veces con cada una de las choice functions, con el objetivo de encontrar la mejor entre éstas. Dado el tipo de solución es necesario que para cada choice function se ejecuten distintos trade offs los que se componen de los distintos indicadores presentes en la choice function, cada uno de estos se visualizan en la tabla 4. Para CF_1 los trade offs que se utilizaron para la elección de la mejor estrategia fueron aquellos donde se marca la preferencia de menor cantidad de backtrack en vez del número de step, esto debido a que se busca minimizar la cantidad de backtracks totales durante la ejecución del problema. La choice function 2 en sus trade offs marca una preferencia sobre la menor cantidad de shallow backtracks, que la variación de la máxima profundidad (*In1*), se marca esta preferencia debido a que los shallow backtrack no representan variación de profundidad porque sólo se instancian diferentes valores para la variable. En la choice function 4 ambos trade offs marcan preferencia sobre la mayor cantidad de variables fijadas. A diferencia de la última choice function donde se da prioridad a la menor cantidad de variables no fijadas, es decir aquellas estrategias que han fijado mayor cantidad de variables.

Choice Functions		**Trade Offs**
CF_1	B-Step-PTAVFES	$t_1 \coloneqq B \rhd Step$
		$t_2 \coloneqq PTAVFES \rhd B$
CF_2	SB-In1-In2	$t_1 \coloneqq SB \rhd In1$
		$t_2 \coloneqq In1 \rhd In2$
CF_3	B-Step-IN1-IN2	$t_1 \coloneqq B \rhd Step$
		$t_2 \coloneqq In1 \rhd In2$
CF_4	VF-Dmax-DB	$t_1 \coloneqq VF \rhd Dmax$
		$t_2 \coloneqq VF \rhd Db$
CF_5	N-VU-VFPS-THRASH	$t_1 \coloneqq VU \rhd VFPS$
		$t_2 \coloneqq VU \rhd THRASH$

Tabla 4: Choice functions y trade offs utilizados.

Las tablas 5 y 6 muestran los tiempos de ejecución para los problemas n-queens, magic square, knight tour y sudoku para cada una de las choice functions presentes en la herramienta, destacando cuál choice function obtuvo el menor tiempo de ejecución. La sigla t.o. indica time out, es decir que dada cierta cantidad de tiempo (24 horas) no logró llegar a la solución. En general para el problema n-queens la mejor choice function fue la número 2, ya que logró llegar a resultados más rápido que las demás, también fue la única choice function que llegó al resultado de 50 y 75 queens con un tiempo de 23.04 y 9.28 segundos respectivamente, cabe destacar que para lograr estos resultados se trabajó con las estrategias 3, 4, 5, 6, 7 y 8 (ver Anexo B). Para las variables menores como por ejemplo n=8 y n=10 todas las choice functions lograron obtener el mismo menor tiempo de ejecución. Finalmente 12-queens tuvo menores tiempos de ejecución en CF_1 y CF_3.

El problema magic square n=4 no marca grandes diferencias de tiempo en comparación a las 5 choice functions, los rangos de tiempo van desde 0.124 a 0.141 segundos, al contrario de n=5 donde sí existen grandes diferencias entre sus tiempos de ejecución, esto se ve representado por la diferencia entre el mayor y menor tiempo de ejecución que es de 7.472 segundos. En el problema knight tour, que es de mayor complejidad, la única choice function que logró llegar al resultado fue CF_5, utilizando las estrategias 3, 5, 6, 7 y 8. Finalmente para la resolución del problema sudoku los resultados son homogéneos dado que no existe gran diferencia entre el menor y el mayor tiempo de ejecución.

N-Queens							
	n=8	**n=10**	**n=12**	**n=15**	**n=20**	**n=50**	**n=75**
CF_1	**0.109**	**0.109**	**0.124**	0.375	0.671	t.o	t.o.
CF_2	**0.109**	**0.109**	0.125	**0.312**	**0.280**	**23.041**	**9.286**
CF_3	**0.109**	**0.109**	0.125	0.359	0.670	t.o.	t.o.
CF_4	**0.109**	**0.109**	0.140	0.436	0.452	t.o.	t.o.
CF_5	**0.109**	**0.109**	**0.124**	**0.312**	0.312	t.o.	t.o.

Tabla 5: Runtimes en segundos problema n-queens.

	Magic Squares		**Knight Tour**		**Sudoku**
	n=4	**n=5**	**n=5**	**n=6**	**Problem1**
CF_1	**0.124**	8.065	t.o.	t.o	**0.109**
CF_2	0.140	**0.593**	t.o.	t.o.	0.156
CF_3	0.125	8.221	t.o.	t.o.	0.125
CF_4	0.125	3.090	t.o.	t.o.	0.124
CF_5	0.141	2.465	**5.039**	**1.763**	**0.109**

Tabla 6: Runtimes en segundos, problemas magic square, knight tour y sudoku.

Las tablas que se muestran a continuación (tablas 7 y 8) contienen la cantidad total de backtracks y nodos visitados para la mejor choice function de cada problema.

Para n-queens en cuanto a la menor cantidad de backtracks fueron para n=10 con 6 backtracks totales, en cambio la mayor cantidad de ellos fue para n=50 con 5025 backtracks. La cantidad de nodos visitados está entre los valores 46 y 25368 para n=8 y n=50 respectivamente. La menor cantidad de nodos visitados es para 10-queens que posee 41 nodos visitados.

El mejor comportamiento, es decir la menor cantidad de backtracks lo tuvo el problema sudoku donde no hubo ninguno, también en este problema la cantidad de nodos visitados fue menor ya que sólo hubo 10 nodos visitados, cabe destacar que para este problema se utilizó una disposición de números en 24 casillas, por lo tanto 57 casillas vacías. El problema knight tour n=5 posee la mayor cantidad de nodos visitados (26436) en comparación a todos los demás problemas.

N-Queens							
	n=8	**n=10**	**n=12**	**n=15**	**n=20**	**n=50**	**n=75**
Backtracks	9	6	14	147	89	5025	1255
Nodos visitados	46	41	95	743	470	25368	6601
Choice Function	CF_1	CF_1	CF_1	CF_2	CF_2	CF_2	CF_2

Tabla 7: Cantidad total de backtracks y nodos visitados problema n-queens.

	Magic Squares		**Knight Tour**		**Sudoku**
	n=4	**n=5**	**n=5**	**n=6**	**Problem1**
Backtracks	10	171	1470	370	0
Nodos visitados	85	1325	26436	7068	10
Choice Function	CF_1	CF_2	CF_5	CF_5	CF_5

Tabla 8: Cantidad total de backtracks y nodos visitados.

8.1 Análisis de Resultados

Una vez terminadas las pruebas y conociendo el mejor rendimiento sobre el tiempo de ejecución, backtracks y nodos visitados, se puede analizar cómo se comportó Skyline en la resolución de cada uno de los problemas teniendo en cuenta factores como por ejemplo la choice function utilizada, los indicadores involucrados y la cantidad total de variables que requiere instanciar el problema.

Para aquellos problemas donde *n* total es un número menor (por ejemplo 8 y 10 queens), el tipo de solución SkylineChoice tuvo un muy buen rendimiento, ya que obtuvo resultados rápidos, si bien estos resultados poseen mayor cantidad de backtracks y nodos visitados, se ahorra mucho tiempo al no tener un optimizador y como Skyline es un método que se ejecuta varias veces durante el desarrollo de la solución el tiempo que éste demora por sí solo, hace encontrar la solución rápidamente. Por el contrario cuando los problemas tienen mayor cantidad de variables que instanciar, la cantidad de backtracks y nodos visitados es bastante mayor que otros tipos de solución como por ejemplo utilizando sólo una estrategia, sin embargo posee la ventaja de que el tiempo de ejecución es menor.

Las choice functions analizadas tuvieron distintos rendimientos según el problema seleccionado, es así como para aquellos problemas de mayor complejidad las choice functions 2 y 5 lograron llegar al resultado en menor tiempo que las demás. Para aquellos problemas con menor cantidad total de variables que instanciar todas las choice functions llegaron al resultado en tiempos similares. Para algunas choice functions el método SkylineChoice no fue capaz de seleccionar las mejores estrategias para encontrar la solución, un caso de esto es por ejemplo el problema knight tour donde sólo una choice function logró llegar al resultado, sin embargo se utilizó una disposición distinta de estrategias de enumeración nombradas anteriormente.

La cantidad de nodos visitados por problema varía considerablemente si el problema a resolver posee un *n* mayor, la cantidad de nodos visitados es bastante alta como ejemplo de esto se encuentra el problema knight tour (n=5) donde hubo un total de 26436, en base a este número podemos concluir que el proceso de resolución no fue muy eficiente si lo que se busca es minimizar aquellos indicadores que producen un resultado más lento, lo anterior también tiene que ver en gran medida con la gran cantidad de backtracks que se producen en el proceso de resolución ya que si una variable instanciada lleva a una ruta sin salida es decir la variable instanciada no es parte de la solución es necesario volver atrás y realizar otros procesos de enumeración, por lo tanto otros nodos que visitar.

Los problemas 50 y 75 queens y knight tour n=5 y n=6 presentaron time out en las choice functions 1, 3 y 4, esto ocurrió debido a que Skyline no elegía las mejores estrategias por lo cual entraba en un proceso de fijar variables con valores que no llegaban en ningún caso a la solución, es por esto que se desmarcó de la interfaz aquella (s) estrategia (s) que Skyline elegía y no eran las adecuadas para continuar la resolución del problema. Una vez solucionado esto la solución se ejecutó correctamente y en un tiempo considerable para la complejidad de los problemas.

En cuanto a las estrategias seleccionadas por Skyline para continuar el desarrollo de la solución, podemos tomar el ejemplo de la figura 42 que corresponde al problema 20-queens, para este problema las estrategias que fueron seleccionadas para solucionar la mayor cantidad del problema fueron las estrategias 1 y 3 con más del 39% de uso para cada una, relegando a las otras estrategias con menos de un 4% de uso durante el proceso de resolución, lo que demuestra una clara preferencia del Skyline para las estrategias 1 y 3 (F+ID y MRV+ID). En general este fue el comportamiento que tuvo Skyline para la selección de estrategias, en primer lugar elegir una vez cada una de las estrategias, a continuación teniendo información sobre su comportamiento según el valor de sus indicadores realizar el Skyline y de acuerdo a su criterio elegir la próxima estrategia. Se muestran ciertas preferencias sobre algunas estrategias en todos los problemas esto se ve reflejado porque existe cierta cantidad de estrategias que se utilizan en mayor cantidad que otras, sin embargo Skyline da la oportunidad a todas las estrategias de demostrar su rendimiento, lo que las hace a todas posibles candidatas.

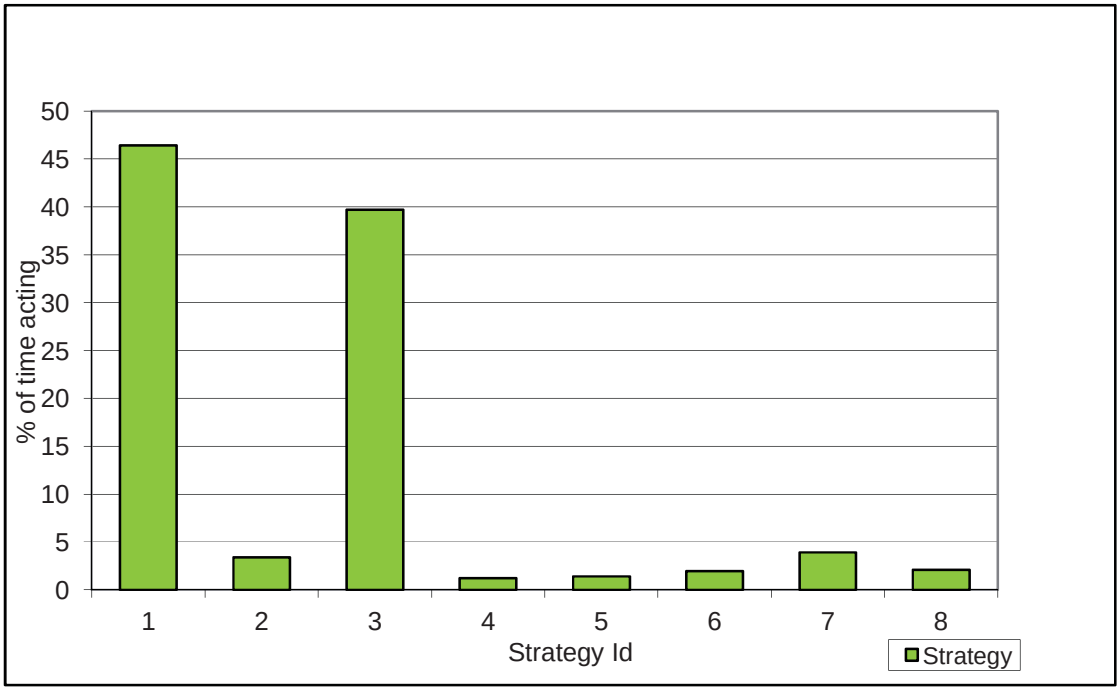

Figura 42: Porcentaje de tiempo utilizado por estrategia.

En general los indicadores analizados (nodos visitados y backtracks) mostraron números altos, sin embargo el tiempo que se ahorra en realizar un método más acotado en comparación a otros da como resultado tiempos menores, en consecuencia al utilizar el método Skyline se producen resultados cuya principal característica es la eficacia con que se llega al resultado del problema.

8.2 Comparación de Resultados

Utilizando distintas estrategias se pueden obtener diferentes resultados en cuanto a la cantidad de backtracks, nodos visitados y tiempos de ejecución, esto sirve para establecer comparaciones y analizar cada una de las estrategias lo que análogamente se puede realizar con Skyline. Las siguientes tablas muestran los resultados de los problemas nombrados anteriormente, un mayor análisis sobre parte de estas tablas se encuentra en [19]. En este caso se produce time out (t.o.) cuando se produce el step 65535 por experimento.

Las tablas 9 y 10 muestran la información sobre el problema n-queens, magic square, knight tour y sudoku resueltos con cada una de las estrategias presentes en la herramienta, también con la opción random y Skyline.

Para el problema n-queens el resultado que presenta Skyline no es el con mayor ni con la menor cantidad de backtracks, esta cantidad se mantiene entre estos márgenes para la mayoría de los valores de *n*, sin embargo la cantidad total de vueltas atrás es bastante menor que la estrategia con mayor cantidad, por ejemplo en 50-queens el resultado de Skyline no está cerca de las estrategias que marcan más de 25 mil backtracks, por lo cual utilizar Skyline es más efectivo que utilizar sólo una estrategia, si se busca minimizar la cantidad de backtracks.

El mejor resultado que obtuvo Skyline fue para el problema sudoku, donde no hubo ningún backtrack, también obtuvo el mejor resultado en el problema knight tour n=6 donde la cantidad de vueltas atrás es considerablemente menor que los otros métodos utilizados. Cabe destacar que para ningún problema Skyline fue el peor comportamiento, la cantidad total siempre se mantiene entre la mayor y la menor cantidad, con una tendencia al valor menor para la mayoría de los casos.

En cuanto a la comparación que se puede realizar con el algoritmo genético los resultados de Skyline son mejores para 8, 10 y 20 queens, en los demás valores de *n* los resultados que muestra el algoritmo genético son mucho mejores que Skyline, por ejemplo la cantidad de backtracks del genético en 50 queens es de tan sólo 5, en cambio Skyline obtuvo 5025 backtracks. Sin embargo para los problemas presentes en la tabla 10 Skyline obtuvo resultados considerablemente mejores que el algoritmo genético, el cual obtuvo gran cantidad de backtracks sobre todo en los problemas magic square 5 y knight tour 5 y 6, donde éste último el algoritmo genético no fue capaz de resolverlo por lo que se produce time out. Cabe destacar que el problema knight tour 6 es considerado de alta complejidad, por lo tanto el hecho de que Skyline muestre menor cantidad de backtracks es un resultado significativo para la investigación de este algoritmo en la arquitectura de Autonomous Search.

N-Queens							
Strategy	**n=8**	**n=10**	**n=12**	**n=15**	**n=20**	**n=50**	**n=75**
F + ID	10	6	15	73	10026	>27406	>26979
AMRV + ID	11	12	11	808	2539	>39232	>36672
MRV + ID	10	4	16	1	11	177	818
O+ID	10	6	15	73	10026	>26405	>26323
F+IDM	10	6	15	73	10026	>27406	>26979
AMRV + IDM	11	12	11	808	2539	>39232	>36672
MRV+ IDM	10	4	16	1	11	177	818
O+IDM	10	6	15	73	10026	>26405	>26323
Random	5	8	18	98	32	>32718	>32973
Genetic	10	12	10	9	415	5	506
Skyline	**9**	**6**	**14**	**147**	**89**	**5025**	**1255**

Tabla 9: Cantidad de backtracks problema n-queens resuelto con distintas estrategias.

	Magic Square		**Knight Tour**		**Sudoku**
Strategy	**n=4**	**n=5**	**n=5**	**n=6**	
F + ID	12	910	767	>19818	18
AMRV + ID	1191	>46675	>42889	>43098	10439
MRV + ID	3	185	767	>19818	4
O+ID	10	5231	>18838	>19716	18
F+IDM	51	>46299	767	>19818	2
AMRV +IDM	42	>44157	>42889	>43098	6541
MRV+ IDM	97	>29416	767	>19818	9
O + IDM	29	>21847	>18840	>19716	2
Random	17	>39742	>40022	>35336	250
Genetic	180	2366864	249489	t.o.	4
Skyline	**10**	**171**	**1470**	**370**	**0**

Tabla 10: Cantidad de backtracks para los problemas magic, knight y sudoku, resuelto con distintas estrategias.

Las tablas 11 y 12 muestran la cantidad de nodos visitados durante la ejecución de cada problema. El mejor resultado con Skyline lo obtiene el problema sudoku que tiene un total de 15 nodos visitados, también el problema knight tour n=6 obtiene la menor cantidad de nodos visitados. Otra característica importante es que Skyline obtiene la mayor cantidad de nodos visitados para los problemas 8, 10 y 12 queens con valores que doblan a la estrategia que obtuvo la menor cantidad de nodos visitados.

Para el problema knight tour n=6 la cantidad de nodos visitados fue la menor en cuanto a las otras estrategias las cuales presentan más de 65535, en cambio Skyline obtuvo el total de 7608. En la cantidad de nodos visitados Skyline también obtuvo la menor cantidad de nodos visitados en comparación al algoritmo genético para los problemas presentes en la tabla 12.

N-Queens							
Strategy	**n=8**	**n=10**	**n=12**	**n=15**	**n=20**	**n=50**	**n=75**
F + ID	24	19	43	166	23893	>65535	>65535
AMRV + ID	21	25	30	1395	4331	>65535	>65535
MRV + ID	25	16	45	17	51	591	2345
O+ID	25	19	46	169	24308	>65535	>65535
F+IDM	24	19	43	166	23893	>65535	>65535
AMRV +IDM	21	25	30	1395	4331	>65535	>65535
MRV+IDM	25	16	45	17	51	591	2345
O+IDM	25	19	46	169	24308	>65535	>65535
Random	15	23	41	205	78	>65535	>65535
Genetic	45	72	24	46	2437	65	2952
Skyline	**46**	**44**	**95**	**743**	**470**	**25368**	**6601**

Tabla 11: Cantidad de nodos visitados, problema n-queens resuelto con distintas estrategias.

	Magic Square		**Knight Tour**		**Sudoku**
Strategy	**n=4**	**n=5**	**n=5**	**n=6**	
F + ID	37	1901	3113	>65535	158
AMRV + ID	1826	>65535	>65535	>65535	30139
MRV + ID	22	546	3113	>65535	76
O+ID	31	13364	>65535	>65535	196
F+IDM	110	>65535	3113	>65535	58
AMRV +IDM	69	>65535	>65535	>65535	19550
MRV+IDM	230	>65535	3113	>65535	153
O+IDM	61	>65535	>65535	>65535	62
Random	47	>65535	>65535	>65535	1019
Genetic	1659	3.126516e7	2072805	t.o.	29
Skyline	**85**	**1325**	**26436**	**7608**	**15**

Tabla 12: Cantidad de nodos visitados para los problemas magic, knight y sudoku, resuelto con distintas estrategias.

Los tiempos de ejecución en segundos se presentan en las tablas 13 y 14. Skyline posee los menores tiempos en los problemas 8, 10 y 12 queens, a diferencia de 50 y 75 queens donde se llega al resultado en un tiempo mayor que el mejor rendimiento, con una diferencia de 21.9 y 2.8 segundos respectivamente.

Para el problema knight tour n=6 se produce t.o. (time out) para todas las estrategias excepto para Skyline donde se obtiene un tiempo de 1.763 segundos.

Los tiempos de ejecución de Skyline en general son buenos, en su mayoría son los menores o muy cerca de éstos. El resultado más importante es knight tour ya que fue el único donde no se produce time out y por lo tanto llega a su resultado satisfactoriamente con un tiempo menor.

También existe otro tiempo de ejecución que se destaca en el problema 75-queens donde Skyline tiene un tiempo cercano al que poseen las estrategias MRV+ID y MRV+IDM.

En la comparación de Skyline con el algoritmo genético, este último sólo obtiene un menor tiempo que Skyline para 50 queens, para todos los demás problemas Skyline obtiene menores tiempos de ejecución que usar el algoritmo genético como optimizador. Esto puede deberse a que al algoritmo genético es mucho más complejo que el algoritmo Skyline por lo cual este último es más eficaz.

N-Queens							
Strategy	**n=8**	**n=10**	**n=12**	**n=15**	**n=20**	**n=50**	**n=75**
F + ID	0.125	0.125	0.156	0.296	35.354	t.o.	t.o.
AMRV + ID	0.125	0.124	0.156	2.106	7.785	t.o.	t.o.
MRV + ID	0.124	0.125	0.156	0.124	0.156	1.138	6.645
O+ID	0.125	0.125	0.156	0.312	35.179	t.o.	t.o.
F+IDM	0.125	0.124	0.141	0.296	34.246	t.o.	t.o.
AMRV +IDM	0.125	0.125	0.140	2.137	7.785	t.o.	t.o.
MRV+IDM	0.141	0.125	0.156	0.125	0.171	1.154	6.505
O+IDM	0.124	0.125	0.141	0.312	34.570	t.o.	t.o.
Random	0.156	0.141	0.125	0.296	2.293	t.o.	t.o.
Genetic	1.397	1.443	1.468	1.658	3.288	12.848	37.95
Skyline	**0.109**	**0.109**	**0.124**	**0.312**	**0.280**	**23.041**	**9.486**

Tabla 13: Runtimes en segundos problema n-queens, resuelto con distintas estrategias.

	Magic Square		**Knight Tour**		**Sudoku**
Strategy	**n=4**	**n=5**	**n=5**	**n=6**	
F + ID	0.140	2.919	4.306	t.o.	0.156
AMRV + ID	0.109	t.o.	t.o.	t.o.	6.271
MRV + ID	0.125	0.795	4.290	t.o.	0.141
O+ID	0.156	17.534	t.o.	t.o.	0.140
F+IDM	0.296	t.o.	4.305	t.o.	0.156
AMRV+IDM	0.187	t.o.	t.o.	t.o.	3.338
MRV+IDM	0.405	t.o.	4.352	t.o.	0.171
O+IDM	0.218	t.o.	t.o.	t.o.	0.187
Random	0.265	t.o.	t.o.	t.o.	0.280
Genetic	5.08	9849.363	607.599	t.o.	1.625
Skyline	**0.124**	**0.593**	**5.039**	**1.763**	**0.109**

Tabla 14: Runtimes en segundos problemas magic, knight y sudoku, resuelto con distintas estrategias.

9. Conclusiones

La investigación sobre las características de la programación con restricciones ayudó a la completa comprensión de este paradigma, conocimiento importante para la continuación y desarrollo de todas las etapas del proyecto.

Al conocer la arquitectura de Autonomous Search, especialmente el funcionamiento del componente actualización, se definió que no era necesario asignar una importancia (o peso) a las estrategias en la fórmula choice function, ya que al incorporar el operador Skyline cada función de la estrategia donde se evalúan los indicadores se puede por ejemplo minimizar. Por lo tanto no se utiliza un optimizador en la resolución de un problema con el tipo de solución (SkylineChoiceFunction) desarrollada en este proyecto.

Se investigó sobre los algoritmos Skyline y entre los dos algoritmos se eligió uno (Block Nested Loops), para desarrollar el trabajo de buscar las tuplas dominantes. También se integró un análisis de correlación para que entre aquellos indicadores que entreguen la misma información, elegir sólo uno de estos y luego desarrollar el análisis del Skyline.

Sobre la investigación de los algoritmos Skyline y el análisis de correlación se pudo establecer un método que se desarrolla sobre cada estrategia y sus indicadores. La idea principal es seleccionar la mejor estrategia para cada paso del desarrollo de la solución.

Se desarrolló la choice function basada en Skyline de tal manera que cada uno de los indicadores tanto los indicadores base como los calculados, pueden ser utilizados en ésta. Además se implementó en la choice function el análisis de correlación de manera que el usuario sea el que decida sobre qué indicadores hacer el análisis de correlación. Para esto se implementó una pantalla en la que se puede marcar si se quiere realizar un análisis de correlación y sobre qué indicadores hacerlo.

Al implementar el prototipo de la nueva choice function sobre los indicadores base se pudo establecer que su integración en la herramienta era correcto, es decir el nuevo tipo de solución llamado SkylineChoiceFunction se ejecuta correctamente tanto internamente como en las opciones de interfaz que ingresa el usuario, información que es necesaria para el desarrollo de la solución.

Una vez implementado el prototipo se integraron los indicadores calculados para dar función completa a la choice function. Además se incorporó el concepto de smoothing factor a la actualización de los indicadores de manera que se puede elegir si utilizarlo o no, una vez que la estrategia fue utilizada en un paso en el desarrollo de la solución por el solver ECLiPSe.

Se implementó el trade-off Skyline de manera general, ya que para implementarlo de mejor forma es necesario tener los valores que representen las preferencias de cada indicador, ya que una vez realizadas las pruebas se conoce entre qué valores se encuentran los indicadores para cada problema y para cada valor de la variable.

Se integraron a la herramienta múltiples interfaces que dan la opción al usuario que ejecuta las pruebas dar una configuración personalizada y conveniente para el desarrollo de Skyline choice function. Además se integró otro tipo de archivo de salida (texto plano) el cual se puede elegir como preferencia en aquellos casos de pruebas iniciales sobre una nueva metodología donde no se tiene claro la cantidad de tiempo que la prueba se va a ejecutar, por lo tanto la cantidad de información que se guarda sobre los indicadores es de gran cantidad, por lo que un archivo Excel que posee límites claros de escritura, no es la mejor opción en este caso.

Se realizaron experimentos para cada uno de los problemas y choice functions presentes en la herramienta. Las pruebas realizadas mostraron, que en general Skyline obtiene buenos resultados, esto se concluye porque cada uno de los problemas pudo ser solucionado lo que representa el objetivo principal de la metodología. En cuanto a los indicadores analizados (backtracks y nodos visitados) los resultados fueron también positivos ya que en ningún caso Skyline presentó una mayor cantidad en comparación con otras estrategias utilizadas.

Analizando los problemas se pudo ver buenos resultados, por ejemplo en el problema n-queens donde los tiempos de ejecución se acercan a las estrategias que obtienen los menores valores, cabe destacar que para los problemas con mayor cantidad de variables que instanciar el tiempo de ejecución fue mayor que la mejor estrategia con mejor rendimiento, sin embargo se pudo solucionar el problemas en menos de 30 segundos lo que es un tiempo bueno para la complejidad de este mismo. Otro problema difícil de solucionar para la programación con restricciones es el problema knight tour en la solución de este problema utilizando Skyline también se obtuvieron buenos resultados ya que para ambos valores de n (5 y 6) se obtuvieron resultados correspondientes a tiempos menores a los 5 segundos, lo que muestra la eficiencia de Skyline eligiendo aquellas estrategias que resuelven el problema satisfactoriamente en cada paso.

También se realizó la comparación de las pruebas entre Skyline y el algoritmo genético las mayores diferencias entre esta choice function y optimizador se encontraron para los indicadores backtracks y runtimes, donde para el primero el algoritmo genético obtuvo mejores resultados, es decir la cantidad de backtracks fue significativamente menor en comparación a Skyline, por el contrario los tiempos de ejecución de Skyline fueron mucho menores en la mayoría de los experimentos realizados, lo que se produce ya que el algoritmo genético como optimizador es mucho más complejo que una choice function basada en Skyline en cuanto a sus metodologías.

El valor de los indicadores analizados aumentaba notablemente en cuanto a la complejidad del problema lo que es un comportamiento normal dada la gran cantidad de variables que fijar y las restricciones propias de cada problema. Cada problema obtuvo con Skyline valores de nodos visitados y backtracks menores que las estrategias que obtuvieron el mayor valor para estos indicadores, por lo cual se concluye que es más efectivo utilizar Skyline que una estrategia al azar para resolver un problema.

Como trabajo futuro se puede seguir investigando sobre la integración de la metodología Skyline en la programación con restricciones y específicamente en Autonomous Search, proveer de otros algoritmos que lo resuelvan y también mejorar su proceso de selección de estrategias con otras metodologías.

A través de este trabajo se aprendió el paradigma de la programación con restricciones, la metodología de apoyo a la toma de decisiones Skyline y se logró unir esta última a Autonomous Search, integrándola como una nueva choice function. Se cumplió el objetivo de realizar experimentos y analizar sus resultados con otras estrategias, algoritmos y metodologías.

10. Referencias

[1] B. Crawford, C. Castro, E. Monfroy. **Programación con restricciones dinámicas**. 2009.

[2] F. Rossi, P. van Beek, T. Walsh, **Handbook of Constraint Programming**. 2006.

[3] V. Bustos. **Una arquitectura modular para autonomous search**. Informe Final para optar al título de Ingeniero Civil en Informática, Pontificia Universidad Católica de Valparaíso, 2012

[4] F. Barber, M. A. Salido. **Introducción a la programación de restricciones**. Disponible vía web en http://users.dsic.upv.es/~msalido/papers/aepia-introduccion.pdf. Revisado por última vez el 14 marzo de 2012.

[5] R. Barták. **Guide to Constraint Programming.** Disponible vía web en http://ktiml.mff.cuni.cz/~bartak/constraints/backtrack.html. Revisado por última vez el 18 marzo de 2012.

[6] http://eclipseclp.org/. Revisado por última vez el 18 de abril de 2012.

[7] R. Krzysztof, G. Wallace. **Constraint Logic Programming using Eclipse**. 2006.

[8] http://www.sics.se/isl/sicstuswww/site/highlights.html. Revisado por última vez el 18 de abril de 2012.

[9] http://www.emn.fr/z-info/choco-solver/. Revisado por última vez el 18 de abril de 2012.

[10] M. Arangú. **Modelos y técnicas de consistencia en problemas de satisfacción de restricciones**. 2011.

[11] Y. Hamadi, E. Monfroy, F. Saubion. **What is Autonomous Search**. 2008.

[12] B. Crawford, R. Soto, M. Montecinos, C. Castro, and E. Monfroy. **A Framework for Autonomous Search in the Eclipse Solver.** Proceedings of the 24th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE), páginas 79-84, LNCS 6703, Springer, 2011.

[13] I.E. Sutherland. **Sketchpad: A man-machine graphical communication system**. Technical Report, Computer Laboratory, University of Cambridge. 2003.

[14] R. Pizarro, G.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.

[15] R. Soto, B. Crawford, E. Monfroy, V. Bustos. **Using Autonomous Search for Generating Good Enumeration Strategy Blends in Constraint Programming.** Proceedings

of the 12th International Conference on Computational Science and Its Applications (ICCSA), páginas 607-617, LNCS 7335, Springer, 2012.

[16] S. Börzsönyi, D. Kossmann, K. Stocker. **The Skyline Operator**. 17th International Conference on Data Engineering, páginas 421-430, 2002.

[17] M. Berlinger. **Selector de Estrategias de Enumeración en Constraint Programming**. Informe Final para optar al título de Ingeniero Civil en Informática, Pontificia Universidad Católica de Valparaíso. 2010.

[18] C. Lofi, U. Güntzer, W. Balke. **Efficient Computation of Trade-Off Skylines**. 13th International Conference on Extending Database Technology, páginas 597-608, 2010.

[19] B. Crawford, R. Soto, C. Castro, E. Monfroy, and F. Paredes. **An Extensible Autonomous Search Framework for Constraint Programming.** Int. J. Phys. Sci. Vol.6(14), páginas 3369-3376, 2011.

[20] B. Crawford, R. Soto, C. Castro, E. Monfroy, W. Palma, F. Paredes. **Parameter Tuning of a Choice Function based Hyperheuristic using Particle Swarm Optimization**. Expert Systems with Applications, 40(5), pp. 1690-1695. 2012.

Anexo

A: Indicadores

Código	Nombre	Descripción	Cálculo
Step	Número de Mediciones	Cada vez que se usa la choice function.	Step++
TV	Número Total de Variables	Número total de variables en el problema. Indicador en relación a las características del problema, este valor es constante durante el proceso.	length(AllVars,N)
VU	Variables No Instancias	Número de variables no instanciadas.	var_count(Rest,VU)
SB	Shallow Backtrack	Cuando se trata de asignar un valor a la variable actual sin éxito entonces se recorre el próximo valor.	
SS	Espacio de Búsqueda	Actual espacio de búsqueda.	$\prod(Dom_i)_t$
SS_pr	Anterior Espacio de Búsqueda	Espacio de búsqueda anterior.	$\prod(Dom_i)_{t-1}$
B	Backtracks	Cuando la variable actual lleva a una ruta sin salida, entonces de retrocede a la variable anterior.	Cada vez que suceda el contador aumenta en uno.
N	Nodos	Cantidad de nodos visitados.	El contador se incrementa en uno cada vez que se visita un nodo.

Tabla A1: Indicadores Base.

Código	Nombre	Descripción	Cálculo
VF	Variables Instanciadas	Número de variables instanciadas por enumeración y propagación.	$(TV - VU)$
VFE	Variables Fijadas por Enumeración en cada Paso	Funciona cuando el número de pasos es mayor que 1.	l
TVFE	Número Total de VFE	Número total de variables instanciadas.	$\sum VFE$
VFP	Variables Fijadas por Propagación en cada Paso.	Número de variables instanciadas por propagación.	$length(Rest, N) - VU$
TVFP	Número Total de VFP	Número total de variables instanciadas por propagación.	$\sum VFP$
TSB	Total Shallow Backtracks	Número total de Shallow Backtracks.	$\sum SB$
d_pr	Profundidad previa	Profundidad previa en el árbol de búsqueda.	$(TV - VU)_{t-1}$
D	Profundidad	Profundidad actual en el árbol de búsqueda.	$(TV - VU)_t$
dmax_pr	Profundidad máxima previa	La profundidad máxima previa en el árbol de búsqueda.	$MAX(dmax)_{n-1}$
In1	Profundidad máxima actual – Profundidad máxima previa	Representa la variación de la profundidad.	$(dmax - dmax\_pr)$
In2	Profundidad actual – profundidad previa.	Si es positivo significa que el nodo actual se encuentra a más profundidad que el del paso previo.	$(d_t - d_{t-1})$
In3	Reducción del espacio de búsqueda.	Si es positivo, el actual espacio de búsqueda es más pequeño que el del Snapshot anterior.	$((SS_{pr} - SS)/_{SS\_pr}) \times 100$
PVFE	Porcentaje de VFE		$(VFE/_{TV}) \times 100$
PVFET	Porcentaje de las variables totales instanciadas por enumeración en cada paso		$(TVFE/_{TV}) \times 100$
PVFP	Porcentaje de variables instanciadas por propagación en cada paso		$(VFP/_{TV}) \times 100$

PVFPT	Porcentaje de las variables totales instanciadas por propagación en cada paso		$({TVFP}/{TV}) \times 100$
Thrash	Thrashing		$(d_{t-1} - VFPS_{t-1})$

B_real		Si la variable actual lleva a una ruta sin salida, entonces se va a la variable previa y cuenta ese retroceso.	`Si D-d_pr==0 entonces`
			`  Incrementar`
			`Si D < d_pr entonces`
			`  d_pr – D + 1`

Tabla A2: Indicadores Calculados.

B: Estrategias de Enumeración

	Heurística De Selección Variable	**Heurística De Selección Valor**
Estrategia 1	First	Indomain
Estrategia 2	AMRV	Indomain
Estrategia 3	MRV	Indomain
Estrategia 4	Occurence	Indomain
Estrategia 5	First	Indomain Max
Estrategia 6	AMRV	Indomain Max
Estrategia 7	MRV	Indomain Max
Estrategia 8	Occurence	Indomain Max

Tabla B1: Estrategias de enumeración presentes en la herramienta.

Nombre	**Descripción**
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.

Tabla B2: Heurística de Selección de Variable.

Nombre	**Descripción**
Indomain	Escoge el elemento más pequeño del dominio.
Indomain Max	Escoge el elemento más grande del dominio.

Tabla B3: Heurística de Selección de valor.

C: Instalación de la Herramienta

Programas requeridos:

- Netbeans 6.8
- ECLiPSe 6.0

Sistema operativo:

- Windows Vista 64 o 32 bits ó Windows XP 32 bits.
- RAM mayor que 1 Gb.

Una vez instalados los programas requeridos es necesario abrir con Netbeans la carpeta donde se encuentra la herramienta, luego con el botón secundario ir a las propiedades del proyecto y cambiar la ruta donde se encuentra instalado ECLiPSe en el computador, como lo muestra la figura C1 en el recuadro rojo, por ejemplo la ruta donde se instaló ECLiPSe en el ejemplo es: "C:\Program Files (x86)\ECLIPSe 6.0". Es necesario realizar este paso, ya que esta ruta se establece por defecto y se puede dar el caso que no sea la ruta correcta donde se instaló ECLiPSe.

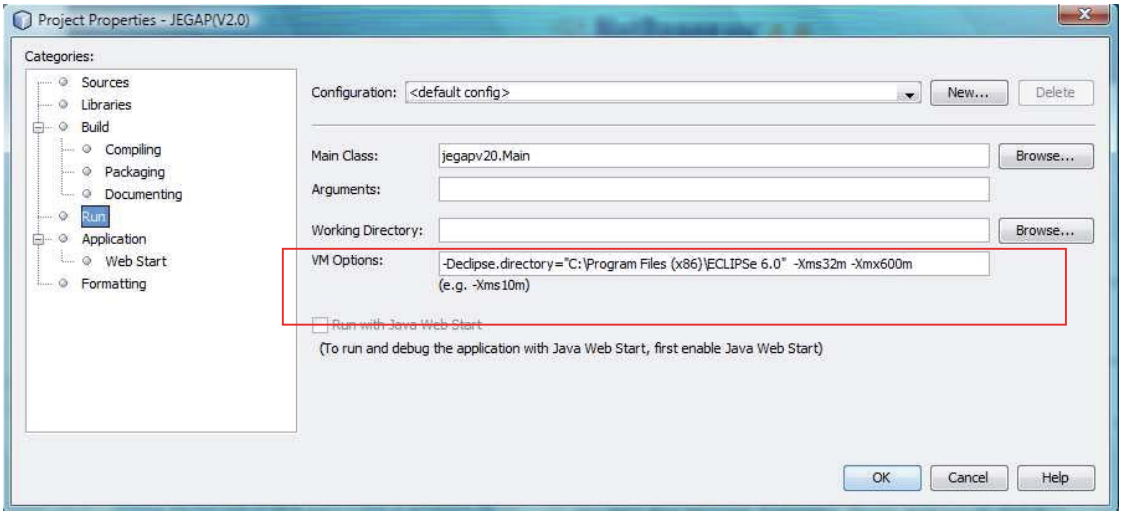

Figura C1: Propiedades del proyecto.

A continuación de haber modificado la ruta correctamente es necesario copiar los archivos que se muestran en la figura C2, éstos se encuentran en la carpeta librería_autonomous_search que se ubica en la carpeta de la herramienta (en este caso JEGAP(V2.4)). Es necesario realizar este procedimiento ya que, los problemas que la herramienta contiene utilizan la librería autonomous search.

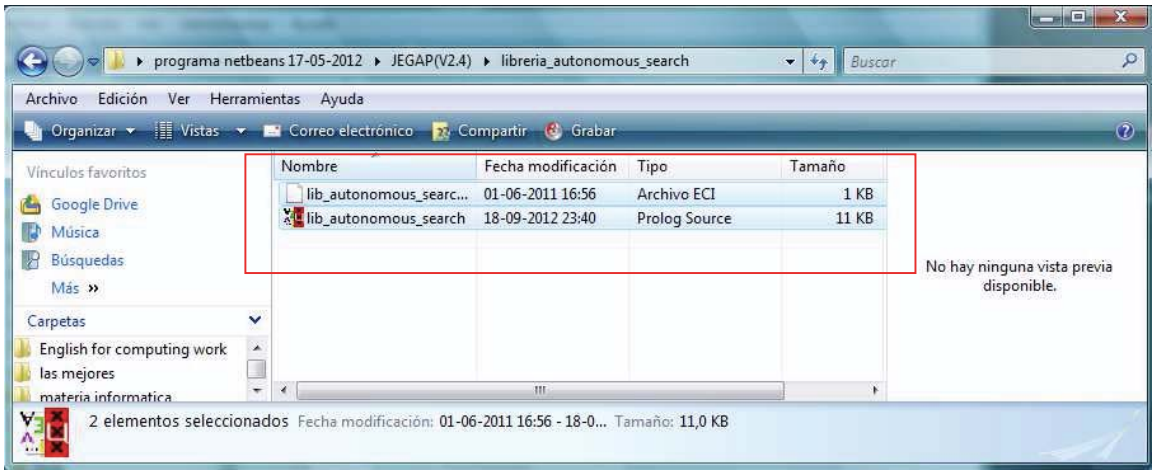

Figura C2: Archivos para copiar.

Los dos archivos se deben copiar en la carpeta de ECLiPSe llamada lib_public como lo muestra la figura C3. Una vez realizados estos pasos se puede utilizar la herramienta correctamente ejecutándola a través de Netbeans. No es necesario abrir ECLiPSe ya que esta instancia se ejecutará automáticamente, una vez que se ejecute el proyecto en Netbeans.

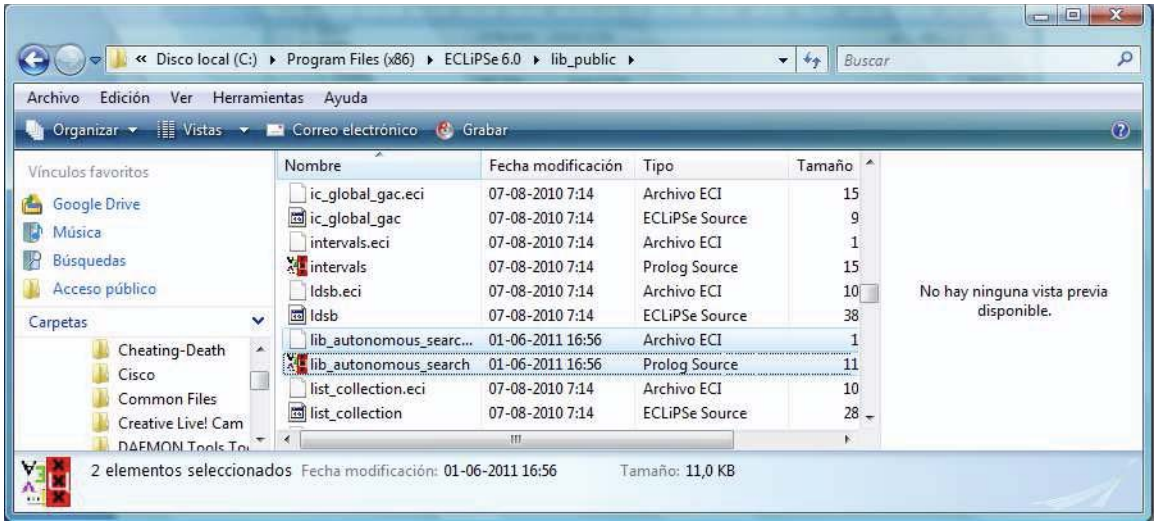

Figura C3: Carpeta lib_public de ECLiPSe.