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

UN ALGORITMO HÍBRIDO AC3-TABU SEARCH PARA RESOLVER PUZZLES SUDOKU

CRISTIAN ALEJANDRO GALLEGUILLOS MICCONO

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

DICIEMBRE, 2012

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

UN ALGORITMO HÍBRIDO AC3-TABU SEARCH PARA RESOLVER PUZZLES SUDOKU

CRISTIAN ALEJANDRO GALLEGUILLOS MICCONO

Dr. Ricardo Soto de Giorgis
Profesor Guía

Dr. Broderick Crawford Labrín
Profesor Co-referente

Carrera
Ingeniería Civil Informática

Diciembre, 2012

Dedicatoria

A mi familia, polola y amigos por todo el apoyo durante todas las etapas de aprendizaje y vida, en especial a Raquel Huerta (QEPD). Todos los logros han sido posible gracias a ellos.

Agradecimientos

A la nane, Rossana Miccono, mi Mamá, por quién estoy inmensamente agradecido, por todo su apoyo durante todos estos años.

Agradezco también a mi Papá, Oscar Galleguillos, por ser un gran apoyo durante toda mi vida y logros, que aunque estando lejos, siempre estuvo conmigo.

A mi hermano Oscar, hermana Rocío y sobrino Pepe Nacho, con quién hemos pasado momentos inolvidables.

A mi mani y lel, Raquel y María Eugenia Huerta, quienes me criaron y malcriaron, enseñaron y dieron todo lo que tuvieron a su alcance, ayudándome a ser quién soy, agradezco totalmente toda su ayuda y dedicación durante tantos años.

A mi polola, Francisca Venegas, que durante estos años de carrera ha sabido apoyarme para continuar y nunca renunciar, gracias por su amor incondicional y por toda su ayuda.

A los profesores Guía y Co-Referente, Ricardo Soto y Broderick Crawford, quienes brindaron su ayuda en este proyecto.

Cristian Galleguillos Miccono.

"No intentes ser un hombre de éxito.
Intenta ser un hombre de valor."
Albert Einstein.

Resumen

El presente trabajo propone un preproceso para la metaheurística *Tabu Search*, la cual está orientada a encontrar soluciones de Problemas de *Optimización Combinatorial* o *Satisfacción de Restricciones*. El objetivo del preproceso, es disminuir el tiempo computacional para encontrar el óptimo global, en su defecto el óptimo local. Este proyecto ilustra la integración de la técnica de consistencia local AC3 y Tabu Search para facilitar la solución de diferentes instancias del SuDoku Problem.

Palabras Clave: Optimización, Metaheurísticas, Tabu Search, Programación con Restricciones, Problemas de Satisfacción de Restricciones, Consistencia Local, SuDoku.

Abstract

This paper proposes a preprocessing phase to the *Tabu Search* metaheuristic, which is aimed at finding solutions to *Combinatorial Optimization* or *Constraint Satisfaction* Problems. The purpose of the preprocessing is to reduce the computational time to find the global optimum, if not the local optimum. This project illustrates the integration of the local consistency technique AC3 and Tabu Search in order to facilitate the resolution of different instances of the Sudoku Problem .

Palabras Clave: Optimization, Metaheuristics, Tabu Search, Constraint Programming, Constraint Satisfaction Problems, Local Consistency, SuDoku.

Índice

1. Introducción **1**

2. Definición de Objetivos **2**
2.1. Objetivo General 2
2.2. Objetivos Específicos 2

3. Estado del Arte **3**

4. Programación con Restricciones **4**
4.1. Problema de satisfacción de restricciones 4
4.1.1. Definiciones y conceptos básicos 4
4.1.2. Modelado del problema 5
4.1.3. Definición y Tipología de las Restricciones 5
4.1.4. Ejemplos de CSP 6
4.2. Algoritmos de búsqueda 8
4.2.1. Generate And Test 8
4.2.2. Backtraking Cronológico (BT) 9
4.3. Técnicas de consistencia 11
4.3.1. Consistencia completa 11
4.3.2. Consistencia incompleta 12
4.4. Estrategias de Enumeración 15
4.5. Solvers 17
4.5.1. Programación lógica con restricciones 17
4.5.2. Librerías 17
4.5.3. Lenguajes de Modelado 18

5. Metaheurísticas **20**
5.1. Optimización Combinatorial 20
5.1.1. Definición formal 21
5.2. Clasificación de Técnicas Metaheurísticas 22
5.2.1. Metaheurísticas basadas en trayectoria 22
5.2.2. Metaheurísticas basadas en población 23
5.3. Algoritmo Tabu Search 24
5.3.1. La estructura de la Búsqueda Tabú 24
5.3.2. Búsqueda por entorno 25
5.3.3. Características de la Búsqueda Tabú 26
5.3.4. Criterios de Aspiración 27

6. Problema de Satisfaccion de Restricciones: Sudoku **29**
6.1. Definición del Problema 29
6.2. El número mínimo de números dados 32

7. Integración de Arc-Consistency con Tabu Search **33**
7.1. Fase 1: Pre-Proceso 33
7.2. Fase 2: Aplicación de Tabu Search 35

8. Experimentos **38**
8.1. Equipo utilizado 38
8.2. Instancias de Sudoku utilizadas 38
8.3. Resultados y discusión 38

9. Conclusiones **40**

10. Referencias **41**

Lista de Figuras

1. Solución al Problema de 4-Reinas mediante Generate And Test. 9
2. Solución al Problema de 4-Reinas mediante BackTracking. 10
3. Ejemplo de Arc Consistency. 14
4. Ejemplo de función de costos. 22
5. Estructura de Tabu Search. 25
6. Puzzle Sudoku Resuelto 29
7. Ejemplo de Puzzle Sudoku 30

Lista de Tablas

1. Tipo de restricciones según número de variables involucradas. 6
2. Definición de restricciones. 6
3. Relación entre restricciones. 6
4. Representación 4-Reinas para Tabu Search. 24
5. Explicación de la representación 4-Reinas para Tabu Search. 24
6. Comparativa de la cantidad de soluciones, según la cantidad de sus bloques. 30
7. Términos de un problema de Sudoku 31
8. Solución parcial inicial de un problema(puzzle) Sudoku 32
9. Solución inicial de un problema Sudoku de $2^2 \times 2^2$ 33
10. Pre-Proceso en Fase 1 34
11. Generación de Vecinos en Fase 2 36

Lista de Algoritmos

1. Algoritmo de BackTracking 10
2. Algoritmo de Adaptive Consistency 12
3. Algoritmo NC-1 13
4. Algoritmo Revise 14
5. Algoritmo AC-3 14
6. Algoritmo de Resolución 16
7. Método de Búsqueda en el Entorno 26
8. Condition Stop 36
9. Generar Candidatos 36
10. Sudoku Solver 37

1. Introducción

Éste trabajo se centra en la integración de *Programación con Restricciones* con la *Metaheurística Tabu Search* para la solución de instancias de Puzzles de Sudoku. Más específicamente, se analizará la integración de técnicas específicas de filtrado como lo es *Arc Consistency*, de la *Programación con Restricciones*, y la *lista tabú* de *Tabu Search*.

Debido al gran tamaño y complejidad de ciertos *Problemas de Optimización Combinatorial* y de *Satisfacción de Restricciones*, se han introducido las Metaheurísticas para resolver problemas que se escapan en tiempos de respuesta o soluciones, a las ya tradicionales técnicas de optimización.

Si bien las Metaheurísticas obtienen soluciones en tiempos reducidos, en comparación a las técnicas más clásicas como la *Investigación Operativa*, existen tiempos "muertos" en donde el algoritmo busca soluciones siendo que estas no son factibles, según las restricciones del problema, es por esto que se ha ideado realizar un preproceso al dominio del problema acotando la búsqueda de soluciones. Para realizar el preproceso, se ha propuesto inicialmente utilizar la técnica de *Arc Consistency* que permite actualizar el dominio del problema, dejando que todos las posibles soluciones sean consistentes.

Se ha seleccionado la Metaheurística *Tabu Search*, esto se debe a que ésta Metaheurística hace uso de *estructura de memoria* al momento de realizar las búsquedas de soluciones, para restringir la búsqueda a soluciones ya visitadas. Es por esto que la actualización realizada por las técnicas de consistencia local será importante para la estructura de memoria o "lista tabú", ya que es ahí en donde se encuentran las restricciones para la búsqueda de soluciones. Se definiran, en un momento previo a la ejecución de la Búsqueda Tabú, las restricciones de la solucion inicial encontradas por el algoritmo de Consistencia Local.

El problema escogido para aplicar esta integracion de técnicas es el *SuDoku Problem*, en adelante simplemente sudoku, el cual es un popular juego de tipo puzzle de tipo NP-completo [29]. Como se mencionó anteriormente el problema de Sudoku es ampliamente estudiado en la Programación con Restricciones [13] [16] [28] y Metaheurísticas [20] [28]. Éstos métodos son eficientes para resolver Sudokus, pero no entregan soluciones para todos las complejidades de problemas Sudoku. Sin embargo, en éste estudio se centrará en resolver Sudoku con un híbrido conformado con la Metaeurística Tabu Search y la integracion de Arc Consistency para el filtrado y actualización del dominio de las soluciones parciales.

La razón principal de por qué resolver Sudokus con la técnica anteriormente mencionada, es aprender más acerca de las capacidades de la integración de distintas técnicas orientadas a la resolución de problemas combinatorios, de forma de poder conocer nuevas técnicas que permitan resolver y/o satisfacer de forma más eficiente este tipo de problemas.

Es por lo anterior, que en este trabajo, se ha decidido conocer los contenidos fundamentales tanto de Programación con Restricciones como de Metaherísticas, para que así en las siguientes secciones del trabajo se declare cómo aplicar los objetivos del proyecto, el cual es resolver el *Sudoku Problem*.

2. Definición de Objetivos

2.1. Objetivo General

Resolver el Sudoku problem utilizando Tabu Search y pre-proceso basado en Arc-Consistency

2.2. Objetivos Específicos

- Comprender las técnicas de filtrado pertenecientes a la programación con restricciones en particular, la denominada arc-consistency.
- Implementar un algoritmo de arc-consistency para filtrar soluciones infactibles en el Sudoku problem.
- Combinar el algoritmo de filtrado basado en arc-consistency con un algoritmo Tabu Search para resolver el Sudoku problem.

3. Estado del Arte

Se han presentado distintos enfoques generales para resolver problemas de optimización o satisfacción de restricciones. Relacionado a ésta investigación, existen publicaciones en que se ha hecho uso de metaheurísticas apoyándose en técnicas de Programación con Restricciones.

Ejemplo de lo anterior, es la llamada metaheurística híbrida CN-Tabu o Consistent NeighborHood Tabu Search [7], la cual es un nuevo enfoque general para resolver problemas de optimización de restricciones. TabuSearch se realiza en una vecindad consistente bien diseñada, o sea, es construida utilizando Programación con Restricciones, para verificar la consistencia de las soluciones. Después de cada asignación de variable, las variables de conflicto se eliminan para mantener la coherencia de las restricciones de instancias. Por lo tanto, en lugar de permitir que se hagan movimientos no factibles en configuraciones completas, estas trabajan con configuraciones parciales, pero consistentes, hasta que se encuentre una solución.

En [3] se realizó una examinación de los efectos de la aplicación del pre-procesamiento a través de Arc Consistency, con el fin de verificar el desempeño de TabuSearch. Se utilizó un set de datos del Centre d'Electronique de l'Armement - CELAR. En el cual para ciertos problemas no se encuentran soluciones, debido al gran espacio de búsqueda y a que cuenta con solo un pequeño número de soluciones en este espacio. Es por esto, que para encontrar una solución más fácilmente se redujo el tamaño del espacio de búsqueda, mediante el uso de arco consistencia. En [8], hace referencia a un problema de redes de telecomunicaciones hertzianas, que consiste en la asignación de recursos de frecuencias con el fin de minimizar la interferencia electromagnética por la proximidad de las antenas, la utilización se hizo de Arc Consistency y Path-Inverse-Consistency con el fin de reducir el espacio de búsqueda, y generar soluciones parcialmente consistentes, para apoyar a TabuSearch en la búsqueda de soluciones.

En relación al problema escogido, a pesar de ser un juego de rompecabezas muy popular, es un problema NP-Hard, que puede ser muy difícil de resolver dependiendo de las condiciones iniciales del problema. En relación a como abordar este problema, existen investigaciones que siguen el enfoque anteriormente mencionado, como lo es en la combinación de metaheurísticas con las técnicas de la Problemas de satisfacción de restricciones (CSP) en donde hacen uso Simulated Annealing y distintas heruísticas que hacen utilizan técnicas de consistencia. Algunas investigaciones que se resumen en [20], demuestrando de forma empírica, que las Metaheurísticas basadas en población superan enormemente a las basadas en trayectoria o búsqueda local, por lo que será interesante realizar la comparación con un híbrido de Tabu Search y Arc Consistency.

4. Programación con Restricciones

La *Programación con Restricciones* es un paradigma de programación en donde las variables son expresadas en restricciones. La idea de la programación de restricciones es resolver problemas mediante la declaración de restricciones sobre el área del problema y consecuentemente encontrar soluciones que satisfagan todas las restricciones, y en su caso optimicen unos criterios determinados.

4.1. Problema de satisfacción de restricciones

Preparar un plato de comida o definir el diseño aerodinámico de un transbordador, son problemas de la vida real que suceden día a día ¿Qué tienen estos problemas en común? Estos tipos de problemas poseen un conjunto de variables que modelan la problemática, y asociadas a éstas se encuentran las restricciones. Estas restricciones restringen los valores que pueden ser asignados a las variables.

Comúnmente los problemas del tipo anterior hacen referencia al modelo de *Problemas de Satisfacción de Restricciones-Constraint Satisfaction Problems (CSP)*, y se resuelven utilizando técnicas de satisfacción de restricciones. En general, estos problemas se tienen las características de ser grandes y complejos, típicamente de complejidad NP (Nondeterministic Polynomial time).

Las etapas básicas para solucionar un CSP, consiste en establecer el modelo y posteriormente resolverlo aplicando técnicas específicas, tales cómo búsquedas apoyadas en métodos heurísticos y procesos inferenciales.

4.1.1. Definiciones y conceptos básicos

Existen conceptos claves de este paradigma, que corresponden a los aspectos de:

- *Modelamiento del problema*, que permite representar un problema mediante un conjunto finito de variables, un dominio de valores finito para cada variable y un conjunto de restricciones que acotan las combinaciones válidas de valores que las variables pueden tomar. En el modelo del CSP, es fundamental la capacidad expresiva, con el fin de poder captar todos los aspectos significativos del problema.

- *Técnicas de consistencia o inferenciales* que permiten deducir nueva información sobre el problema a partir de la explícitamente representada. Estas técnicas también permiten acotar y hacer más eficiente el proceso de búsqueda de soluciones.

- *Técnicas de búsqueda de solución*, apoyadas generalmente por criterios heurísticos, bien dependientes o independientes del dominio. El objetivo es encontrar un valor para cada variable del problema de manera que se satisfagan todas las restricciones del problema.

4.1.2. Modelado del problema

Un problema de satisfacción de restricciones puede ser representado mediante una terna (X, D, C), en donde:

- $X = \{x_1, ..., x_n\}$ es un conjunto de n variables.
- $D =< D_1, ..., D_n >$ es una tupla de dominios finitos donde se interpretan las variables X, tal que la i-ésima componente D_i es el dominio que contiene los posibles valores que pueden asignarse a la variable x_i. La cardinalidad de cada dominio es $d_i = \|D_i\|$.
- $C = \{c_1, c_2, ..., c_p\}$ es un conjunto finito de restricciones. Cada restricción k-aria c_i está definida sobre un conjunto de k variables $var(c_i) \in X$, denominado su ámbito, y restringe los valores que dichas variables pueden simultáneamente tomar. Particularmente, una restricción es binaria cuando relaciona únicamente a dos variables $x_i \wedge x_j$, y se suele denotar como c_{ij}. Todas las restricciones definidas en un CSP son conjuntivas, de manera que una solución debe de satisfacer a todas ellas.

La **instanciación** de una variable es un par variable-valor (x, a) que representa la asignación del valor a a la variable x $(x = a)$. La instanciación de un conjunto de variables es una tupla de pares ordenados, donde cada par ordenado (x_i, a_i) asigna el valor a, siendo $a_i \in D_i$, a la variable x_i. Una tupla $< (x_1, a_1), ..., (x_i, a_i) >$ es localmente consistente si satisface todas las restricciones formadas por variables $x_1, ..., x_i$ de la tupla. Para simplificar la notación, se sustituirá la tupla $< (x_1, a_1), ..., (x_i, a_i) >$ por $(a_1, ..., a_i)$.

Un valor $a_i \in D_i$ es un **valor consistente** para x_i si existe al menos una solución del CSP en la cual $x_i = a_i$. El dominio mínimo de una variable x_i es el conjunto de todos los valores consistentes para la variable, es decir, quedan excluidos aquellos valores que no forman parte de ninguna solución ($\forall x_i \in X, \forall a \in D_i, x_i = a$ forma parte de una solución del CSP).

Una **solución** a un CSP es una asignación $(a_1, a_2, \ldots, a_n)$ de valores a todas sus variables, de tal manera que se satisfagan todas las restricciones del CSP. Es decir, una *solución* es una tupla consistente que contiene todas las variables del problema. Una *solución parcial* es una tupla consistente que contiene algunas de las variables del problema. Un CSP es *consistente*, si tiene al menos una solución, es decir una tupla consistente. Dos CSPs son *equivalentes* si ambos representan el mismo conjunto de soluciones.

4.1.3. Definición y Tipología de las Restricciones

Las restricciones se caracterizan fundamentalmente por su *aridad*, que es el número de variables involucradas en dicha restricción. Una *restricción unaria* es una restricción sobre una sola variable. Una *restricción binaria* es una restricción que consta de dos variables. Una *restricción no binaria (o n-aria)* es una restricción que involucra a un número arbitrario de 3 o más variables.

A continuación se presentan ejemplos de los tipos de restricciones recién expuestos:

Una restricción sobre un conjunto de variables puede definirse **extensionalmente** mediante un conjunto de tuplas válidas o no válidas y también **intensionalmente** mediante una función aritmética. La representación extensional de una restricción k-aria está formada por un conjunto de tuplas, cada una con k elementos, y expresa el conjunto de

Restricción según N° Variables	Ejemplo
Unaria	$x \leq 5$
Binaria	$x_4 - x_3 \neq 3$
Ternaria	$2x_1 - x_2 + 4x_3 \geq 4$
N-Aria	$x_1 + 2x_2 - x_3 + 5x_4 \leq 9$

Tabla 1: Tipo de restricciones según número de variables involucradas.

valores que las k variables pueden tomar simultáneamente. Claramente, en el caso de CSP continuos es imposible representar las restricciones extensionalmente ya que generalmente hay un número infinito de tuplas válidas.

Tipo de definición de restricciones	Ejemplo
Intensionalmente	$x_1 + x_2 \leq x_3 + x_4$
Extensionalmente	$\{(1,1),(1,2),(2,1),(2,2)\}$

Tabla 2: Definición de restricciones.

En el caso de una definición intensional, se puede tener restricciones *no-disyuntivas* o *disyuntivas*, según expresen una única relación o más de una relación disyuntiva entre las variables, además de

Tipo de relación entre variables	Ejemplo
No-Disyuntiva	$x_1 \leq x_2$
Disyuntiva	$x_1 < x_2 \vee x_1 > x_2 \Leftrightarrow x_1 \neq x_2$

Tabla 3: Relación entre restricciones.

Por otra parte, las restricciones también pueden ser *cualitativas*, cuando expresan una relación de orden entre las variables (por ejemplo, $x_1 < x_2$), o *métricas* cuando expresan una distancia métrica entre las mismas (por ejemplo, $x_1 < x_2 + 7$). Las restricciones métricas requieren de una métrica en el dominio de interpretación de las variables.

Un tipo especial de restricciones son las restricciones *lineales*. Una relación lineal sobre $X = \{x_1, ..., x_k\}$ es una expresión de la forma:

$$\sum_{i=1}^{k} p_i x_i \{<, \leq, \neq, \geq, >\} b \tag{1}$$

Donde p_i son los coeficientes y $b \in \mathbb{R}$. En base a combinaciones lógicas de desigualdad ($\neq$) e inigualdad ($\leq$), se pueden expresar todas las relaciones en $\{<, \leq, \neq, \geq, >\}$. Las Restricciones Lineales Disyuntivas (DLR) son disyunciones de restricciones lineales.

4.1.4. Ejemplos de CSP

El primer paso en la resolución de un CSP es su modelamiento, es decir, su representación en términos de variables, dominios y restricciones. Al igual que ocurre con el lenguaje

natural, el modelamiento de un problema se puede realizar de muchas maneras diferentes. Respecto al modelamiento de un problema CSP hay dos aspectos básicos:

- La potencia expresiva de las restricciones, es decir, la capacidad de modelar las restricciones realmente existentes en el problema real.
- La eficiencia de la representación, ya que dependiendo del modelamiento del CSP, el problema se resolverá con más o menos eficiencia.

4.1.4.1. El problema de las N-Reinas

Este problema consiste en colocar N reinas en un tablero de ajedrez de dimensión NxN, de forma que ninguna reina esté amenazada. De esta forma no puede haber dos reinas en la misma fila, misma columna, o misma diagonal. Si asociamos cada columna a una variable y su valor representa la fila donde se coloca una reina, el problema puede ser formulado como:

- Variables: $\{x_i\}, i = 1...N$.
- Dominio $= \forall x \in \mathbb{N}$, para todas las variables.
- Restricciones: $(\forall x_i, x_j i \neq j)$:
  - $x_i \neq x_j$, No en la misma fila
  - $x_i - x_j \neq -j$, No en la misma diagonal SE
  - $x_j - x_i \neq -j$, No en la misma diagonal SO

Resultando para el caso particular de $N = 4$ (Tablero $4x4$):

- $\|x_1 - x_2\| \neq 1$;
- $\|x_1 - x_3\| \neq 2$;
- $\|x_1 - x_4\| \neq 3$;
- $\|x_2 - x_3\| \neq 1$;
- $\|x_2 - x_4\| \neq 2$;
- $\|x_3 - x_4\| \neq 1$;
- $\neq (x_1, x_2, x_3, x_4)$;

Más adelante (pag. 10) se mostrará gráficamente cómo es resuelto este problema, utilizando un algoritmo de búsqueda llamado BackTracking Cronológico, y se detallará en **Algoritmo 1** .

$$\begin{array}{cccccc} & & s & e & n & d \\ & + & m & o & r & e \\ \hline & m & o & n & e & y \end{array}$$

4.1.4.2. Criptografía

El típico problema criptográfico "$send + more = money$" consiste en asignar a cada letra $\{s, e, n, d, m, o, r, y\}$ un dígito diferente del conjunto $\{0, ..., 9\}$ de forma que se satisfaga: $send + more = money$.

La manera más fácil de modelar este problema es asignando una variable a cada una de las letras, todas ellas con un dominio $\{0, ..., 9\}$ y con las restricciones que obliguen a que todas las variables tomen valores distintos y con la correspondiente restricción para que se satisfaga "$send + more = money$". De esta forma las restricciones son:

- $10^3(s + m) + 10^2(e + o) + 10(n + r) + d + e = 10^4m + 10^3o + 10^2n + 10e + y$;
- $\forall x_i, x_j$ con $(x_i \neq x_j)$; donde: $x_i \wedge x_j \in (s, e, n, d, m, o, r, y)$;

4.2. Algoritmos de búsqueda

Los métodos de búsqueda se centran en explorar el espacio de estados del problema. Estos métodos pueden ser completos, explorando todo el espacio de estados en busca de una solución, o incompletos si solamente exploran una parte del espacio de estados. Los métodos que exploran todo el espacio de búsqueda garantizan encontrar una solución, si existe, o demuestran que el problema no es resoluble. La desventaja de estos algoritmos es que son muy costosos. A continuación se explicarán brevemente los dos métodos completos más usuales.

4.2.1. Generate And Test

Este método genera las posibles tuplas de instanciación de todas las variables de forma sistemática y después *testea sucesivamente sobre cada instanciación* si se satisfacen todas las restricciones del problema. La primera combinación que satisfaga todas las restricciones, será la solución al problema. Mediante este procedimiento, el número de combinaciones generado por este método es el Producto Cartesiano de la cardinalidad de los dominios de las variables:

$$\prod_{k=1}^{k=n} d_i \tag{2}$$

Esto es un inconveniente, ya que se realizan muchas instanciaciones erróneas de valores a variables que después son rechazadas en la fase de testeo. Por ejemplo, veáse **Figura 1**, para el caso del problema de las 4-Reinas, se generarían $4^4 = 256$ tuplas a testear. El proceso de generación requeriría $256 \cdot 4 = 1024$ asignaciones de variable.

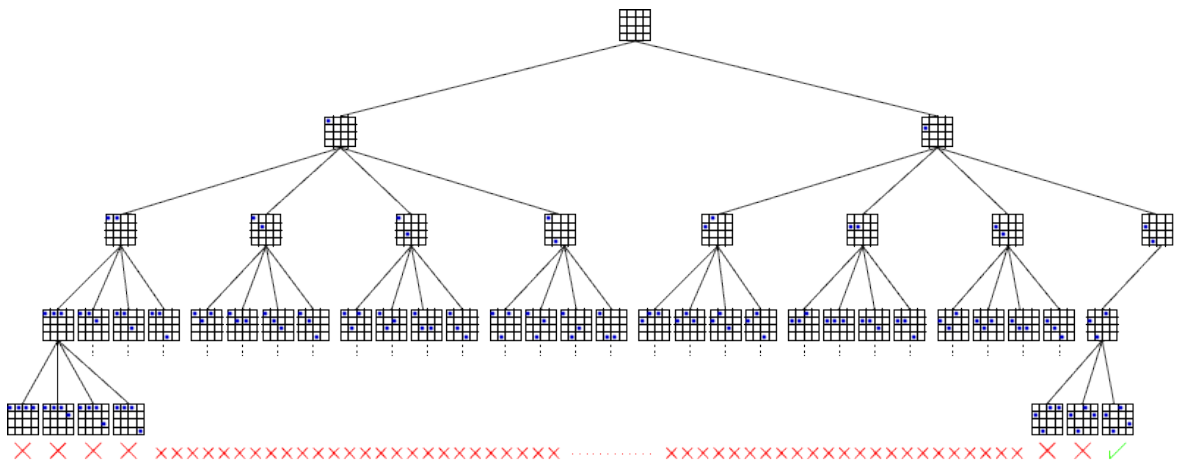

Figura 1: Solución al Problema de 4-Reinas mediante Generate And Test.

4.2.2. Backtraking Cronológico (BT)

Este método realiza una exploración en profundidad del espacio de búsqueda, instanciando sucesivamente las variables y comprobando ante cada nueva instanciación si las instanciaciones parciales ya realizadas son localmente consistentes. Si es así, sigue con la instanciación de una nueva variable. En caso de conflicto, intenta asignar un nuevo valor a la última variable instanciada, si es posible, y en caso contrario retrocede a la variable asignada inmediatamente anterior.

El **Algoritmo 1** describe el típico proceso de BT aplicado a la resolución de CSP, donde $V[n]$ representa el vector de asignaciones a las variables $(x_1, x_2, ..., x_n)$ del problema, ordenadas según un determinado criterio. La función $Comprobar(K;\ V[n])$, descrita en (3) comprueba la validez de las k instanciaciones ya realizadas $\{V[1], V[2], ..., V[k]\}$, es decir, si se satisfacen las restricciones locales existentes entre las variables $(x_1, x_2, ..., x_k)$.

$$Compobar(k, V[n]) = Tsii : \forall i(1 \leq k) : (V[k], V[i]) \in c_{ki} \tag{3}$$

Si el algoritmo finaliza con éxito, el vector $V[n]$ representa la tupla de valores solución. Los algoritmos tipo BT realizan solo una comprobación hacia atrás en cada instanciación, por lo que pertenece al tipo de algoritmos *Look-Backward*. Los algoritmos Look-Backward tratan de explotar la información del problema para comportarse más eficientemente en las situaciones sin salida, llevando a cabo la comprobación de la *consistencia hacia atrás*, es decir, entre la variable actualmente instanciada y las pasadas ya instanciadas. El backtracking cronológico es un algoritmo muy simple pero también es muy ineficiente. El problema es que tiene una visión local del problema, de forma que solo comprueba las restricciones entre las variables ya instanciadas, e ignora la relación entre dichas variables $(x_1, x_2, ..., x_k)$ y las que quedan por instanciar $(x_{k+1}, ..., x_n)$.

En la **Figura 2**, se detalla gráficamente cómo es resuelto el problema de las N-Reinas, siendo para este ejemplo $N = 4$. En este ejemplo se indican los puntos de backtracking y la solucion encontrada. Los nodos marcados con X representan que la asignación realizada *no es consistente* con las asignaciones previas, y por tanto son puntos de backtracking.

Algoritmo 1 Algoritmo de BackTracking

```
function BACKTRACKING(k, V[n])
V[k] = Seleccion(d_k)        ; Selecciona un valor de d_k para asignar a x_k
    if Comprobar(k, V[n]) then
        if k = n then
            devolver V[n]        ; Es una solución
        else
            Backtracking(k + 1, V[n])
        end if
    else
        if quedan_valores(d_k) then
            Backtracking(k, V[n])
        else
            if k = 1 then
                devolver 0        ; Falló
            else
                Backtracking(k − 1, V[n])
            end if
        end if
    end if
end function
```

Como cabía esperar, BT no realiza tanta búsqueda como Generate And Test, al detectar cada inconsistencia en cuanto aparece. Pero para este simple problema, aún requiere generar 26 asignaciones.

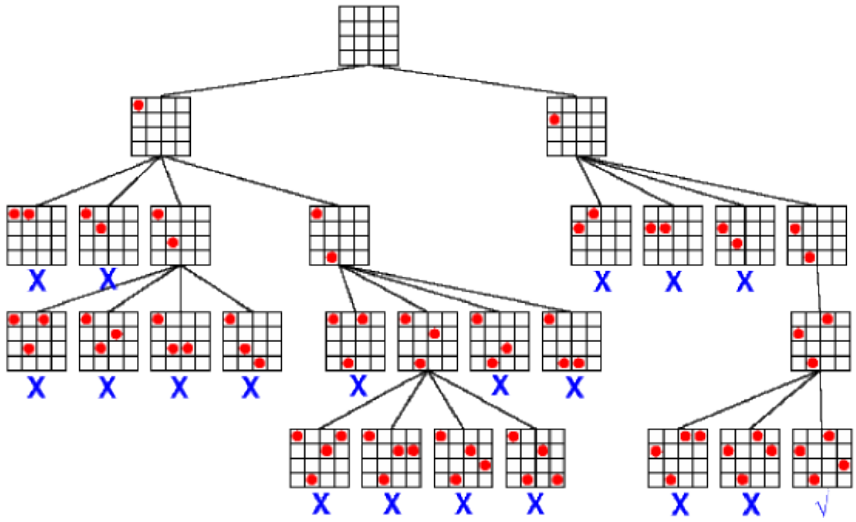

Figura 2: Solución al Problema de 4-Reinas mediante BackTracking.

4.3. Técnicas de consistencia

Las *Técnicas de Consistencia* o *Procesos Inferenciales* en un CSP tienen como objetivo deducir nuevas restricciones, derivadas de las explícitamente conocidas sobre el problema. Concretamente, estas técnicas borran valores inconsistentes de los dominios de las variables o inducen restricciones implícitas entre las variables, obteniendo un nuevo CSP, equivalente al inicial, donde se han hecho explícitas las nuevas restricciones implícitamente contenidas en el primero. La obtención de estas nuevas restricciones permite:

- Obtener respuestas a preguntas sobre el problema, relativas a restricciones implícitamente existentes entre las variables o sobre sus dominios.

- Acotar el espacio de soluciones, al haberse eliminado valores inconsistentes, haciendo más eficientes los procesos de búsqueda.

4.3.1. Consistencia completa

Proceso que permite la síntesis de todas las restricciones de un problema en una única restricción global, las soluciones se obtienen directamente. El problema de la obtención de esa restricción global es que se trata de un problema exponencial.

Una restricción C puede verse como una relación R (denotada por R_A) sobre un conjunto de variables (o ámbito) A. Esta relación, R_A, se compone de un conjunto de tuplas (t). una tupla $t \in R_A$ es una secuencia finita de valores, donde cada valor (o componente) corresponde a una variable de A. Operaciones sobre relaciones fundamentales:

- **Proyección**: dada una relación R_A, su proyección sobre un conjunto de restricciones $B(B \subset A)$, es una nueva relación $\Pi_B(R_A)$ formada por las tuplas de R_A en donde se han eliminado las componentes de las variables en $A - B$.

- **Join**: Dadas dos relaciones $R_A \wedge R_B$, su join $R_A \times R_B$ es una nueva relación con ámbito $A \cup B$. Una tupla t pertenece a la nueva relación, si y sólo si, sus componentes corresponden a los de dos tuplas $r \in R_A \wedge s \in R_B$, tales que $r \wedge s$ tienen los mismos componentes en las variables comunes $A \cap B$.

4.3.1.1. Adaptive Consistency

Sintetizar una única restriccion global que capture el conocimiento contenido en todas las restricciones del problema es conceptualmente simple. Solo basta con calcular la restricción n-aria, de la **Ecuación (2)**, en donde se obtienen todas las tuplas que la satisfacen son las soluciones del problema, pero como se vió calcular esa restricción es costosa en tiempo y espacio.

Dechter y Pearl [6], han propuesto el algoritmo ADC (*Adaptive Consistency*). Este algoritmo requiere un orden estático de variables. Para cada variable, al algoritmo infiere una nueva restriccion que resume el efecto de la variable en el restro del problema. Dicha variable se sustituye por esa nueva restriccion, obteniendo un problema equivalente con una variable menos. Este proceso, denominado eliminacion de variables, se repite eliminando una variable a la vez, hasta que el problema se resuelve trivialmente. Si no se ha

encontrado la restriccion vacía, el problema tiene solución que se contruye de forma directa, sin búsqueda.

Todas las restricciones que tienen a x_i en su ámbito, forman el conjunto B_i. El join de todas esas restricciones en una nueva restriccion denominada R_{A_i}, esta restricción se proyecta sobre el conjunto A ($A := A_i - \{x_i\}$), obteniedo la restricción R_A. Si R_A es la restriccion vacía, el problema no tiene solucion y retorna falso, en el otro caso, se elimina x_i del conjunto de variables X, elimina B_i del conjunto de restricciones C y se añade R_A a C. Cuando se han eliminado todas las variables, la solución se obtiene de forma directa, procesando las variables de $1, ..., n$

Algoritmo 2 Algoritmo de Adaptive Consistency

```
function ADC(X,D,C)
    for i = n → 1 do
        B_i := {R_{S_j} ∈ C || x_i ∈ S_j};
        A_i := R_{S_j} ∈ B ∪ S_j;
        R_{A_i} := (R_{S_j ∈ B_i} ⋈ R_{S_j});
        A := A_i − {x_i};
        R_A := Π_A(R_{A_i});
        if R_A = 0 then
            return false;
        else
            X := X − {x_i};
            C := C ∪ {R_A} − B_i;
        end if
    end for
    for i = 1 → n do
        x_i := Asignación consistente con previas asignaciones y R_{A_i};
    end for
return true;
end function
```

4.3.2. Consistencia incompleta

Sintetizar la restriccion global de una red de restricciones puede paracer complicado. Una forma de abordar ese problema de manera mas asequible es el de resolver subredes de un tamaño prefijado. Formalmente, una subred (X', D', C'), está definida por un subconjunto de variables $X' = \{x_1, ..., x_k\}$, $X' \subset X$, sobre los dominios originales $D' = (D_1, ..., D_k)$, bajo el conjunto de restricciones $C' = \{c \| c \in C \wedge var(c) \subseteq X'\}$.

Encontrar la solucion a una subred no garantiza que el problema tenga solución. Si resolviendo una subred se encuentra ciertos valores o combinaciones de valores no aparecen en niguna solucion de la subred, estos tampoco aparecerán en las soluciones de la red.

Con este proceso, que se denomina de *consistencia local*, se añaden nuevas restricciones al problema y permiten acelerar la búsqueda de la solucion global. Si se deduce la restricción vacía, el problema no tiene solución.

4.3.2.1. Consistencia de nodos

Una variable x_i es *nodo consistente* si y solo si todo valor de su dominio D_i está permitido por las restricciones unitarias sobre x_i. Una red de restricciones es nodo consistente si y solo si todas las variables son nodo consistente. Una red se convierte en nodo consistente con el Algoritmo 3, y es equivalente a la operación:

$$D_i \leftarrow D_i \cap R_i \qquad i = 1, ..., n \tag{4}$$

Algoritmo 3 Algoritmo NC-1

```
function NC-1(X,D,C)
    for x_i ∈ X do
        for a ∈ D_i do
            if a ∉ R_i then
                D_i := D_i − {a};
            end if
        end for
    end for
end function
```

4.3.2.2. Consistencia de arcos

Un **arco** es una restricción cualquiera del problema, asociada con una de las variables implicadas en la misma. Se dice que un *arco es consistente* con respecto a un estado del problema, si para cada valor (en el estado) de la variable asociada al arco, existen valores (en el estado) para las restantes variables implicadas en la restricción, que hacen esta última válida. Si no es así, dicho valor de la variable se puede eliminar de su dominio. El algoritmo de consistencia de arcos parte de un estado (variables + dominios) y modifica los dominios hasta que todos los arcos sean consistentes. En general, los algoritmos de consistencia de arco no permiten decidir si un CSP tiene solución, excepto en los siguientes casos:

- Si algún dominio queda vacío, entonces no hay ninguna solución.
- Si todos los dominios contienen un único elemento, entonces tiene una solucion que es la que se obtiene al asignar a cada variable el valor de su dominio.
- Si una variable tiene un dominio con m elementos y el resto de las variables tienen un dominio con un único elemento, entonces hay m soluciones.

Una variable x_i se hace arco consistente con respecto a la variable x_j por medio del Algoritmo 4, borrando de D_i aquellos valores que no son compatibles con ningún valor de x_j. El efecto de la función $revise(i, j)$ es equivalente a la operacion:

$$D_i \leftarrow D_i \cap \Pi_i(R_{ij} \bowtie D_j) \tag{5}$$

Una red de restricciones binarias se convierte en arco consistente mediante el Algoritmo 5.

Algoritmo 4 Algoritmo Revise

```
function REVISE(i,j)
    cambio := false;
    for a ∈ D_i do
        if ∄b ∈ D_j / (a,b) ∈ R_ij then
            D_i := D_i − {a};
            cambio := true;
        end if
    end for
return cambio;
end function
```

Algoritmo 5 Algoritmo AC-3

```
function AC-3(X,D,C)
    Q := {(i,j)(j,i)‖c_{i,j} inC};
    while Q ≠ ∅ do
        extrae(i,j) ∈ Q;
        if revise(i,j) then
            Q := Q ∪ {(k,i)‖c_ki ∈ C, k ≠ j, k ≠ i};
        end if
    end while
end function
```

Para ilustrar, se puede ver el ejemplo en la Figura 3 el funcionamiento de esta técnica, se supondrá que existe una restricción binaria C_{ij} entre las variables $x_i \wedge x_j$, entonces $arc(x_i, x_j)$ es un arco con consistencia *ssi* $\forall a \in D_i$ / $\exists b \in D_j$, tal que la asignación $x_i := a \wedge x_j := b$ satisface la restricción C_{ij}.

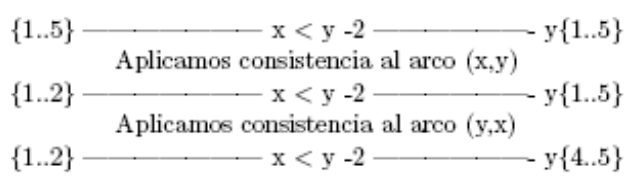

Figura 3: Ejemplo de Arc Consistency.

Quitando del dominio todos los valores que hacen que el arco no sea consistente, permite reducir los dominios, provocando que el problema se resuelva "más fácilmente".

4.3.2.3. k-consistencia

La consistencia de nodos elimina aquellos valores que no son posibles para una variable; la consistencia de arcos elimina aquellos valores de una variable que no se pueden extender de forma consistente a una segunda variable. De esta forma, se puede considerar conjuntos de $k-1$ variables y ver si se pueden extender de forma consistente a cualquier k-ésima variable. Esta idea es la base de la k-consistencia. Una red de restricciones es k-consistente

si y solo si para todo conjunto de $k-1$ varaibles asignadas de forma consistente y cualquier k-ésima variable, existe un valor de esta variable consistente con los valores de las $k-1$ variables anteriores. Los algoritmos que alcanzan la k-consistencia descartan combinaciones de $k-1$ valores que no se pueden extender de forma consistente a una k-ésima variable.

4.4. Estrategias de Enumeración

Resolver un CSP consiste en encontrar una asignación de valores a las variables, de manera tal que no se viole ninguna restricción [24]. El proceso consiste en la exploración sistemática de todas las posibles asignaciones de valores a las variables, formando un árbol de búsqueda [5]. Con técnicas de inferencia se procede a podar el árbol de búsqueda, en donde se eliminan los valores que no son parte de alguna solución. La enumeración crea una rama instanciando una variable y otra rama cuando la primera no se ha podido satisfacer. Entonces, al ejecutar una fase de enumeración deben resolverse dos situaciones: seleccionar la variable a ser enumerada y seleccionar el valor a asignar a dicha variable. Para la primera decisión se utiliza una *Heurística de Selección de Variable*, encargada de determinar el orden en que se enumerarán las variables, y para la segunda elección se utiliza una *Heurística de Selección de Valor*.

En conjunto, una Heurística de Selección de Variable y una Heurística de Selección de Valor, constituyen lo que se conoce como una *Estrategia de Enumeración* [30], que se encarga de guiar la fase de enumeración. En el ámbito de la Programación con Restricciones, estas estrategias son cruciales en el rendimiento del proceso de resolución, donde una elección adecuada puede reducir considerablemente el costo computacional de buscar una solución a un CSP.

A continuación se explican las heurísticas:

- Selección de Variables: Luego de investigaciones sobre la selección de variables durante la búsqueda puede tener un impacto significativo en el tamaño del espacio de búsqueda explorado. Generalmente las heurísticas de selección de variables tratan de escoger lo antes posible las variables que más restringen a las demás. La intuición es tratar de asignar lo antes posible las variables más restringidas y de esa manera identificar las situaciones sin salida lo antes posible y así reducir el número de vueltas atrás. La ordenación de variables puede ser estática y dinámica.

  - Las heurísticas de selección de variables estáticas generan un orden fijo de las variables antes de iniciar la búsqueda, basado en información global derivada del grafo de restricciones inicial.
  - Las heurísticas de selección de variables dinámicas pueden cambiar el orden de las variables dinámicamente basándose en información local que se genera durante la búsqueda.

  El problema de los algoritmos de selección estáticos es que ellos no tienen en cuenta los cambios en los dominios de las variables causados por la propagación de las restricciones durante la búsqueda, o por la densidad de las restricciones. Esto es porque estas heurísticas generalmente se utilizan en algoritmos de comprobación hacia atrás donde no se lleva a cabo la propagación de restricciones.

- Selección de Valores: En comparación, se ha realizado poco trabajo sobre heurísticas para la selección de valores. La idea básica que hay detrás de las heurísticas de ordenación de valores es seleccionar el valor de la variable actual que más probabilidad tenga de llevarnos a una solución, es decir identificar la rama del árbol de búsqueda que sea más probable que obtenga una solución. La mayoría de las heurísticas propuestas tratan de seleccionar el valor menos restringido de la variable actual, es decir, el valor que menos reduce el número de valores útiles.

Con una estrategia de enumeración no se busca mejorar la resolución de un solo problema, sino que se está interesado en encontrar eficientemente soluciones para un espectro amplio de diferentes tipos de problemas, a través de la observación de indicadores que permitan reaccionar frente a la elección de una mala estrategia. Los algoritmos a desarrollar podrán readecuar (auto- ajustar) su proceso de búsqueda, aun si la estrategia inicial no es la más adecuada para el problema que se está resolviendo.

El marco general de las técnicas de resolución empleada es la Programación con Restricciones, que consiste fundamentalmente en alternar fases de consistencia y enumeración. Las técnicas de consistencia reducen el dominio de las variables, removiendo aquellos valores que no pueden participar en la solución; rigurosamente hablando, se fuerza algún nivel de consistencia local, tal como arco consistencia o camino consistencia [15][4][24]. Sus instancias más conocidas son Forward Checking, que asegura la consistencia de las variables relacionadas (que participan en una misma restricción) con la variable recientemente enumerada, y Look-Ahead, que fuerza la consistencia de un CSP completo [15][25].

Por su parte la Enumeración consiste en tomar un CSP y dividirlo en dos CSPs más pequeños, hasta alcanzar alguno de ellos el estado solucionado o fallido; de esta forma, si se tiene un CSP P, la etapa de enumeración sobre una variable crea un CSP P' donde un valor es asignado a esta variable y otro CSP P" con el resto de valores del dominio de la variable. En esta fase es donde se utiliza la estrategia de enumeración, constituida por las heurísticas de selección de variable y valor, las cuales ayudan a determinar qué variable elegir en un determinado momento y qué valor asignarle a dicha variable. De esta forma, la función **Enumeración** del Algoritmo 6 es la aplicación de la estrategia seleccionada y **Proceder por casos** es la función que administra la elección de los puntos creados por la enumeración, lo cual permite la exploración del árbol de búsqueda. En general, la estrategia de enumeración se especifica al comienzo del proceso de resolución y permanece constante durante toda la ejecución de dicho proceso.

Algoritmo 6 Algoritmo de Resolución

```
while no solucionado OR fallido do
    Propagacin de restricciones;
    if no fallido AND no solucionado then
        Enumeracin;
        Proceder por casos;
    end if
end while
```

4.5. Solvers

Fueron creados para desarrollar y resolver CSP, el primero surgió alrededor del año 1960, siendo utilizados hasta la actualidad para desarrollar problemas que involucran variables, dominios y restricciones. Para soportar restricciones y diversos paradigmas, han sido creados variados tipos de *solvers* de diversa naturaleza. Si bien existen distintas clasificaciones, se dará énfasis en explicar y ejemplificar tres categorías: los que utilicen la programación lógica; aquellos que utilizan librerías; y por último los lenguajes de modelado.

4.5.1. Programación lógica con restricciones

La programación lógica con restricciones (CLP, Constraint Logic Programming) es el paradigma que une la programación lógica y la satisfacción de restricciones. El lenguaje utilizado se caracteriza por tener una estructura algebraica. Las funciones especiales y predicados simbólicos se interpretan sobre un dominio fijo, toda relación que se establezca con el dominio es denominada restricción.

Esta idea fue impulsada por Colmerauer en el desarrollo de Prolog II [1], luego fue generalizado en el esquema CLP realizado por Jaffar and Lassez [14]. Algunos ejemplos de *solvers* que soportan CLP se nombran a continuación.

4.5.1.1. ECL^{i}PS^e

ECL^{i}PS^e [19] es el más reciente sistema CLP, proporciona una amplia gama de características para solucionar problemas con restricciones tales como manejo de listas, arreglos y registros, además de un conjunto de instrucciones de control como las condicionales y las repetitivas. ECL^{i}PS^e también provee librerías con la finalidad de tratar cierto tipo de problemas como los que involucran dominios continuos o programación matemática, que además se pueden combinar para resolver problemas en forma híbrida.

4.5.1.2. GNU Prolog

Es otro sistema que pertenece al grupo de CLP [11], ha sido diseñado para tratar problemas con dominios finitos, sin embargo, también se puede utilizar para trabajar con números reales. Tiene entre sus herramientas una larga lista de predicados predefinidos de Prolog y restricciones para el manejo de listas y estructuras condicionales. Se ha incluido soporte para problemas de optimización y heurísticas de ordenamiento, además de una interfaz para llamar a rutinas externas escritas en el lenguaje C.

4.5.2. Librerías

Las librerías proporcionan un lenguaje necesario para enunciar problemas bajo restricciones al haber limitaciones con algún lenguaje de programación. Por lo general son implementadas por medio de clases y métodos específicos. Por ejemplo, una determinada clase puede ser usada para definir el estado de las variables y los métodos, junto con las relaciones entre ellos. Esta es una forma de implementar un sistema de restricciones sin la necesidad de desarrollar un nuevo lenguaje de programación. No obstante, el usuario

está obligado a conocer el lenguaje nativo para la determinada librería que desee utilizar. Algunos solvers que pertenecen a esta categoría se exponen a continuación.

4.5.2.1. Generic Constraint Development Enviroment (Gecode)

Gecode es una librería escrita sobre C++. Fue diseñada para soportar variables de dominio finito. El conjunto de restricciones que abarca, permite trabajar sobre tipos de datos enteros, booleanos y conjuntos de variables, además soporta la definición de heurísticas de ordenamiento de variable y selección de valor. Entre algunos de los lenguajes en los cuales se han desarrollado interfaces para trabajar con librerías Gecode se encuentran Prolog (YAP Prolog), Java (Gecode/J), Ruby (Gecode/R), Lisp (GeLisp) y Phyton.

4.5.2.2. Koalog

Koalog es una librería perteneciente al lenguaje Java que permite trabajar con problemas relacionados con satisfacción y optimización de restricciones. Admite definir conjuntos de restricciones y dominios, ambos finitos. Entre sus cualidades, soporta especificación de heurísticas de selección de variables y valor. Además posee mecanismos de búsqueda que pueden ser declarados mediante objetos específicos.

4.5.2.3. Choco

Es un solver de programación con restricciones desarrollado como una librería de Java. Provee un variado conjunto de restricciones para ser aplicadas sobre números enteros, reales, y conjunto de variables. Soporta problemas de optimización y el proceso de búsqueda puede ser realizado por estrategia de selección de variables y valor, predefinida por el solver o definidas por el usuario.

4.5.3. Lenguajes de Modelado

Apunta a simplificar la definición de problemas basado en restricciones. Intenta alejar al usuario de la complicada codificación, aún presente en típicas librerías o lenguajes de programación. El centro del lenguaje es generalmente más comprensible porque proporcionan una semántica y sintaxis más simples. En algunos enfoques, la especificación de procedimientos de búsqueda es permitida, pero no obligatoria.

4.5.3.1. OPL

Uno de los principales lenguaje de modelado. Su sintaxis y semántica ha sido usada como base de los lenguajes de modelado modernos. El lenguaje OPL está compuesto por muchos constructores de alto nivel, por ejemplo, estructuras de datos como arreglos, variables de dominio finito, ciclos y estados condicionales y un conjunto de órdenes internas para la asignación de recursos. Heurísticas para definir variables y ordenar valores son también permitidas. Una característica interesante de OPL y tal vez su principal novedad, es que sus estrategias de búsqueda, pueden ser especificadas usando el mismo estilo utilizado para plantear el problema[23].

4.5.3.2. Zinc

Se encuentra entre los más recientes lenguajes de modelado[26]. Posee una sintaxis que soporta los predicados definidos por el usuario y las funciones. Está provisto de estructuras de datos, conjuntos, abstracciones de control y dominios finitos y continuos. La plataforma se apoya en una arquitectura solver-independiente, donde los modelos de Zinc pueden ser mapeados por tres modelos de Eclipse: modelo de programación con restricciones, modelo de búsqueda local y modelo de programación matemática. Un modelo intermedio llamado FlatZinc está envuelto para facilitar la traducción a otros modelos.

5. Metaheurísticas

Una *metaheurística* es un *método heurístico*, es un algoritmo que normalmente encuentra buenas soluciones o se ejecuta razonablemente rápido, para resolver un tipo de problema computacional general, usando los parámetros dados por el usuario sobre unos procedimientos genéricos y abstractos de una manera que se espera eficiente. Las metaheurísticas generalmente se aplican a problemas que no tienen un algoritmo o heurística específica que dé una solución satisfactoria; o bien cuando no es posible implementar ese método óptimo. La mayoría de las metaheurísticas tienen como objetivo los *problemas de optimización combinatorial*, pero por supuesto, se pueden aplicar a cualquier problema que se pueda reformular en términos heurísticos, por ejemplo en resolución de ecuaciones booleanas.

Glover y Col. introducen por primera vez el término de *metaheurística* en [18], definen las metaheurísticas como métodos que integran de diversas maneras, procedimientos de mejora local y estrategias de alto nivel para crear un proceso capaz de escapar de óptimos locales y realizar una búsqueda robusta en el espacio de búsqueda. En su evolución, estos métodos han incorporado diferentes estrategias para evitar la convergencia a óptimos locales, especialmente en espacios de búsqueda complejos. Este tipo de técnicas se caracterizan por las siguientes propiedades:

- Las metaheurísticas son estrategias generales que guían el proceso de búsqueda.
- El objetivo es una búsqueda eficiente que encuentre soluciones casi óptimas.
- Pueden incorporar mecanismos para evitar la exploración en regiones del espacio de búsqueda no óptimas.
- El procedimiento de cualquier metaheurística es genérico, no depende del problema.
- Las metaheurísticas utilizan métodos heurísticos específicos que son controlados por una estrategia de más alto nivel.
- Las metaheurísticas utilizan funciones de bondad para cuantificar el grado de adecuación de una determinada solución.

En resumen, una metaheurística es una estrategia genérica de alto nivel que usa diferentes métodos heurísticos para explorar en busca de una solución óptima o casi óptima en espacios de búsqueda de gran tamaño. La metaheurística debe identificar rápidamente las regiones prometedoras del espacio de búsqueda global, y no malgastar tiempo en regiones que hayan sido exploradas y/o no contienen soluciones de alta calidad.

5.1. Optimización Combinatorial

La *optimización combinatorial* es una rama de la optimización en matemáticas aplicadas y en ciencias de la computación, relacionada a la investigación de operaciones, teoría de algoritmos y teoría de la complejidad computacional. Los algoritmos de optimización combinatorial resuelven instancias de problemas que se creen ser difíciles en general, explorando el espacio de soluciones, usualmente de gran tamaño para estas instancias. Los

algoritmos de optimización combinatorial logran esto reduciendo el tamaño efectivo del espacio, y explorando el espacio de búsqueda eficientemente.

Como se especificó anteriormente, el objetivo principal de la optimización es encontrar la solucion a un problema, como se muestra en la **Ecuación 6**,en donde c es la funcion objetivo y F es la región factible.

$$Min\ c(f)\ sujeto\ af \in F \tag{6}$$

Las restricciones de un problema de optimización definen el conjunto de soluciones (o "área" factible) del problema. Si el conjunto de soluciones esta vacío, diremos que el problema es no factible (infeasible), sino, se dice que es factible (feasible).

5.1.1. Definición formal

Una Instancia de un Problema de Optimización corresponderá a un par (F, c), donde F es cualquier conjunto dentro del dominio de puntos factibles y c es la función de costos, tal que:

$$c : F \to \mathbb{R} \tag{7}$$

El problema es encontrar un $f \in F$, tal que

$$c(f) \leq c(y)\ \forall y \in F \tag{8}$$

En donde:

- F será llamado el *óptimo global* para esa instancia.
- Dado un punto factible $f \in F$ en una instancia, se define como el *vecindario* al conjunto de puntos $N(f)$ que están "cerca" del punto f.
- Un problema de optimizacion corresponderá a un conjunto I de instancias de un Problema de Optimización.
    - Instancia de un problema $\neq$ problema.
    - La instancia posee suficientes datos de entrada que permiten encontrar una solución.

Además se define el término de óptimo local si:

$$c(f) \leq c(g) \forall g \in N(f) \tag{9}$$

Si f es óptimo local y a la vez es óptimo global, se dice que el vecindario es *exacto*.

5.1.1.1. Ejemplo de Óptimo Local

Sea una instancia de un problema de optimización F, c definido por:

$$F = [0, 1] \subseteq \mathbb{R} \tag{10}$$

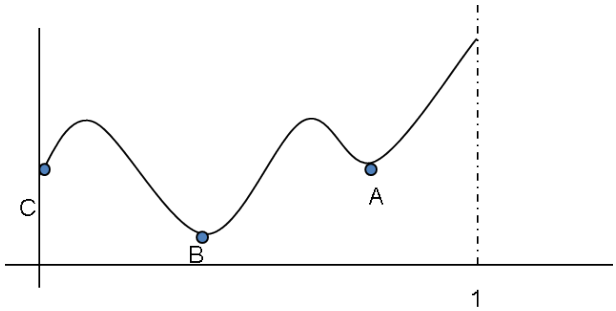

Figura 4: Ejemplo de función de costos.

y la función de costo c, que se muestra en la Figura 4. Se define el vecindario en función de la distancia Euclidiana, según la Ecuación 11, para algún $\epsilon > 0$

$$N_\epsilon(f) = \{x : x \in F \wedge \|x - f\| \leq \epsilon\} \tag{11}$$

Luego, si ϵ es lo suficientemente pequeño, se tiene que:

- A, B, C serán Óptimos Locales
- Sólo B es Óptimo Global

5.2. Clasificación de Técnicas Metaheurísticas

Hay diferentes formas de clasificar las técnicas metaheurísticas: basadas en la naturaleza (algoritmos bioinspirados) o no basadas en la naturaleza, basadas en memoria o sin memoria, con función objetivo estática o dinámica, etc.

La clasificación más empleada es la que se basa en si la técnica utiliza un único punto del espacio de búsqueda o trabaja sobre un conjunto o población. Según esta clasificación las metaheurísticas se dividen en las basadas en trayectoria y las basadas en población.

5.2.1. Metaheurísticas basadas en trayectoria

Estas técnicas parten de un punto inicial y van actualizando la solución presente mediante la exploración del vecindario, formando una trayectoria. La búsqueda finaliza cuando se alcanza un número máximo de iteraciones, se encuentra una solución con una calidad aceptable, o se detecta un estancamiento del proceso.

- **Simulated Annealing (SA)**, es una de las metaheurísticas más antigua [18]. SA simula el proceso de recocido de los metales y del cristal. En cada iteración se elige una solución S_1, a partir de la solución actual S_0. Si S_1 es mejor que S_0, S_1 sustituye a S_0 como solución actual. Si S_1 es peor que S_0, se sigue aceptando pero asignándole una determinada probabilidad. El algoritmo permite elegir soluciones peores a la actual para evitar caer en un óptimo local.

- **Tabu Search (TB)**, es una de las metaheurísticas más utilizadas en problemas de optimización [11]. TB se basa fundamentalmente en la utilización de un historial de búsqueda, que permite ejecutar su estrategia de análisis y exploración de diferentes regiones del espacio de búsqueda. Este *historial o memoria* se implementa como una lista tabú. En cada iteración se elige la mejor solución entre las permitidas y se añade a la lista tabú, donde se mantienen las soluciones recientes que se excluyen de las siguientes iteraciones.

5.2.2. Metaheurísticas basadas en población

Las técnicas metaheurísticas basadas en población trabajan con un conjunto de individuos que representan otras tantas soluciones. Su eficiencia y resultado depende fundamentalmente de la forma con la que se manipula la población en cada iteración.

- **Evolutionary Algorithm (EA)** Este grupo de técnicas se inspiran en la capacidad de la evolución de seres o individuos para adaptarlos a los cambios de su entorno. Cada individuo representa una posible solución. El funcionamiento básico de estos algoritmos es el siguiente: La población se genera de forma aleatoria. Cada individuo de la población tiene asignado un valor de su bondad con respecto al problema considerado, por medio de una función de *aptitud*, capacidad, adaptabilidad o estado, también denominada con bastante frecuencia por la palabra inglesa *fitness*. El valor de la aptitud de un individuo es la información que el algoritmo utilizar para realizar la búsqueda. La modificación de la población se efectúa mediante la aplicación de tres operadores: selección, recombinación y mutación. En estos algoritmos se pueden distinguir la fase de selección, explotación de buenas soluciones, y la fase de reproducción, búsqueda de nuevas regiones. Se debe de mantener un equilibro entre estas dos fases. La política de reemplazo permite la aceptación de nuevas soluciones que no necesariamente mejoran las existentes. Algunas técnicas son Evolutionary Programming (EP) [10], Evolution Strategy (ES) [12], Genetic Algorithm (GA) [22], entre otros.

- **Ant Colony Systems (ACO)** [19], se inspiran en el comportamiento de las hormigas cuando buscan comida: inicialmente, las hormigas exploran el área cercana al hormiguero de forma aleatoria. Cuando una hormiga encuentra comida, la lleva al hormiguero. En el camino, la hormiga va depositando una sustancia química denominada feromona que guía al resto de hormigas a encontrar la comida. El rastro de feromona sirve a las hormigas para encontrar el camino más corto entre el hormiguero y la comida. Este rastro es simulado mediante un modelo probabilístico.

- **Particle Swarm Optimization (PSO)** [17] son técnicas metaheurísticas inspiradas en el comportamiento del vuelo de las bandadas de aves o el movimiento de los bancos de peces. La toma de decisión por parte de cada individuo o partícula se realiza teniendo en cuenta una componente social y una componente individual, mediante las que se determina el movimiento de esta partícula para alcanzar una nueva posición.

5.3. Algoritmo Tabu Search

El nombre y la terminología de búsqueda tabú vienen de [9]. Una característica distintiva de este procedimiento es el uso de memoria adaptativa y de estrategias especiales de resolución de problemas. *Tabu Search* es el origen del enfoque basado en memoria y estrategia intensiva en la literatura de las metaheurísticas, en contraposición con los métodos que no tienen memoria o que solo usan una débil memoria basada en herencia.

Los principios fundamentales de la búsqueda tabú fueron elaborados en una serie de artículos y han sido unificados en el libro "Tabu Search" [11].

La filosofía de la búsqueda tabú es derivar y explotar una colección de estrategias inteligentes para la resolución de problemas, basadas en procedimientos implícitos y explícitos de aprendizaje. El marco de memoria adaptativa de TS no sólo explota la historia del proceso de resolución del problema, sino que también exige la creación de estructuras para hacer posible tal explotación.

5.3.1. La estructura de la Búsqueda Tabú

Para aportar bases para la comprensión de algunos de los elementos fundamentales de la búsqueda tabú, se ilustrará sus operaciones básicas con el ejemplo de N-Reinas ya visto.

Una de las principales complicaciones de la Búsqueda Tabú, es establecer la estructura que representará el problema, o *representación del problema*. Para el caso de las N-Reinas, según el ejemplo antes visto, 4-Reinas la mejor representación sería un vector de tamaño 4 (X) (Tabla 4):

3	4	1	2

Tabla 4: Representación 4-Reinas para Tabu Search.

La representación anterior, incluye una restricción la cual restringe una reina por columna, indicando en que fila se encuentra la reina:

		R_3	
			R_4
R_1			
	R_2		

Tabla 5: Explicación de la representación 4-Reinas para Tabu Search.

Como se ve en la tabla anterior, en la Tabla 5, se pueden extraer los siguientes valores del vector:

- $i = 1, ..., 4$ Representa al número de Reinas, donde 4 puede ser cambiado por N para generalizar el problema.
- $x_i \in X$ Representa la fila en donde se encuentra la Reina i.

- i Representa el numero de la columna.

El valor inicial de X es escogido según la estratégia que se considere, puede ser un orden aleatorio del vector, correr un algoritmo *Greedy* con alguna solución factible. Los métodos TS operan bajo el supuesto de que se puede construir un entorno para identificar *soluciones adyacentes* que puedan ser alcanzadas desde la solución actual. Los intercambios por pares son frecuentemente usados para definir entornos en problemas de permutaciones, identificando movimientos que conducen una solución a la siguiente. En el problema anterior, un intercambio cambia la posición de dos reinas como se ilustra en la Figura 5.

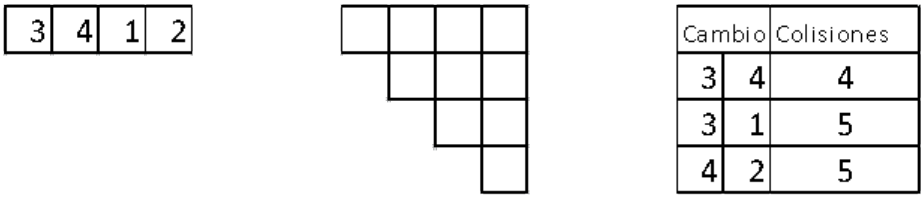

Figura 5: Estructura de Tabu Search.

En la figura, se hace el cambio de posicion de las reinas que se indican, por ejemplo Cambio (3,4), La R_3 se traslada a la fila 2 y la R_4 a la fila 1. Los valores de los movimientos generalmente proporcionan una base fundamental para evaluar la calidad de los mismos, aunque también pueden ser importantes otros criterios. Un mecanismo principal para explotar la memoria en la búsqueda tabú es clasificar un subconjunto de movimientos en un entorno como prohibidos (o tabú). La clasificación depende de la historia de la búsqueda, determinada mediante lo reciente o frecuente que ciertos movimientos o componentes de soluciones, llamados atributos, han participado en la generación de soluciones pasadas.

Como base para evitar la búsqueda desde combinaciones de intercambio repetidas usadas en el pasado reciente, invirtiendo potencialmente los efectos de movimientos anteriores por intercambios que podrían devolver a posiciones previas, clasificaremos como tabú todos los intercambios compuestos por cualquiera de los pares de capas más recientes, estableciento una frecuencia de repeticiones.

5.3.2. Búsqueda por entorno

TS puede ser convenientemente caracterizado mediante referencia a la búsqueda en el entorno, aunque se insiste en que la búsqueda en el entorno tiene un significado más amplio en TS que en algunas otras partes de la literatura de las metaheurísticas. Por ejemplo, TS incluye procedimientos constructivos y destructivos entre los procesos que dirige por memoria adaptativa, mientras que tales procedimientos y sus combinaciones son a menudo excluidos de la definición de búsqueda en el entorno en otros enfoques. Una representación conveniente de búsqueda en el entorno identifica, para cada solución $x \in X$, un conjunto asociado de vecinos, $N(x) \subset X$, llamado entorno de x. En búsqueda tabú, los entornos normalmente se asumen simétricos, es decir, x' es un vecino de x si y solo si

Algoritmo 7 Método de Búsqueda en el Entorno

function PASO 1: INICIALIZACIÓN
(A) Seleccionar una solución de arranque $xActual \in X$
(B) Almacenar la mejor solucion actal a la conocida haciendo $xMejor = xActual$ y definiendo $MejorCoste = c(xMejor)$
end function

function PASO 2: ELECCIÓN Y FINALIZACION
Elegir una solucion $xSiguiente \in N(xActual)$. Si lo criterios de eleccion empleados no pueden ser satisfechos por ningún miembro de $N(xActual)$, o si se aplican otros criterios de parada, entonces el método para.
end function

function PASO 3: ACTUALIZACIÓN
Rehacer $xActual = xSiguiente$, y si $c(xActual) < MejorCoste$, ejecutar el paso 1(B). Volver al paso 2.
end function

x es un vecino de x'. Los pasos en la búsqueda en el entorno se muestran en el Algoritmo 7.

5.3.3. Características de la Búsqueda Tabú

La búsqueda tabú, en contraste con los métodos precedentes, emplea una filosofía diferente para ir más allá del criterio de finalizar en un óptimo local. Se reduce énfasis a la aleatorización, y generalmente se usa en un modo altamente restringido, con el supuesto de que la búsqueda inteligente debería estar basada en formas más sistemáticas de dirección.

Por consiguiente, muchas implementaciones de búsqueda tabú son en gran parte deterministas. Una excepción ocurre con la variante llamada búsqueda tabú determinística, la cual selecciona movimientos según probabilidades basadas en el estado y en las evaluaciones asignadas a estos movimientos por los principios básicos de la búsqueda tabú.

5.3.3.1. Usos de Memoria TS especiales: Modificaciones de Estructuras de Entorno

La idea de explotar ciertas formas de memoria adaptativa para controlar el proceso de la búsqueda es el tema central subyacente en la búsqueda tabú. El efecto de tal memoria puede ser previsto estipulando que TS mantiene una historia selectiva H de los estados encontrados durante la búsqueda, y reemplaza $N(xActual)$ por un entorno modificado que puede ser denotado $N(H, xActual)$. La historia determina, por tanto, qué soluciones pueden ser alcanzadas por un movimiento desde la solución actual, seleccionando $xSiguiente \in N(H, xActual)$.

En las estrategias de Tabu Search basadas en consideraciones de período corto, $N(H, xActual) \subset N(xActual)$, y la clasificación tabú sirve para identificar elementos de $N(xActual)$ excluídos de $N(H, xActual)$. En las estrategias de período intermedio y largo, $N(H, xActual)$

puede contener soluciones que no estén en $N(xActual)$, generalmente soluciones elite seleccionadas (óptimos locales de alta calidad) encontradas en varios momentos en el proceso de solución. Estas soluciones elite se identifican típicamente como elementos de un grupo regional en estrategias de intensificación de período intermedio, y como elementos de diferentes grupos en estrategias de diversificación de período largo. Además, las componentes de las soluciones elite, en contraste con las soluciones en sí mismas, se incluyen entre los elementos que pueden ser conservados e integrados para proporcionar entradas al proceso de búsqueda.

5.3.4. Criterios de Aspiración

Los criterios de aspiración se introducen en la búsqueda tabú para determinar cuándo pueden ser reemplazadas las restricciones tabú, eliminando así una clasificación tabú aplicada a un movimiento en otro caso. El uso apropiado de estos criterios puede ser muy importante para permitir a un método TS proporcionar sus mejores niveles de ejecución.

Las aspiraciones son de dos tipos: *aspiraciones de movimiento* y *aspiraciones de atributo*. Una aspiración de movimiento, cuando se satisface, revoca la clasificación tabú del movimiento. Una aspiración de atributo, cuando se satisface, revoca el estado tabú-activo del atributo. En el último caso el movimiento principal puede no cambiar su clasificación tabú, dependiendo de si la restricción tabú puede ser activada por más de un atributo.

Los siguientes criterios determinan la admisibilidad de una solución ensayo, $xEnsayo$, como un candidato a ser considerado, donde $xEnsayo$ es generado por un movimiento que ordinariamente sería clasificado tabú.

5.3.4.1. Aspiracion por Defecto

Si todos los movimientos disponibles están clasificados tabú, y no se han hecho admisibles mediante algunos otros criterios de aspiración, entonces se selecciona el movimiento "menos tabú".

5.3.4.2. Aspiracion por Objetivo

- **Forma Global**: Se satisface una aspiracion de movimiento, permitiendo que $xEnsayo$ sea un candidato para la seleccion, si $c(xEnsayo) < MejorCost$.

- **Forma Regional**: Subdividimos el espacio de búsqueda en regiones $R \subseteq X$, identificada mediante cotas sobre los valores de funciones $g(x)$. Se denota por $MejorCoste(R)$ el mínimo $c(x)$ encontrado en R. Entonces para $xEnsayo \in R$, se satisface una aspiracion de movimiento si $c(xEnsayo) < MejorCoste(R)$.

5.3.4.3. Aspiracion por Direccion de Búsqueda

Sea $direccion(e) = mejora$ si el movimiento más reciente conteniendo a e fue un movimiento de mejora, y $direccion(e) = nomejora$, en otro caso. Se satisface una aspiracion de atributo para e si $direccion(e) = mejora$ y el movimiento ensayo actual es un movimiento de mejora, es decir, si $c(xEnsayo) < c(xActual)$

5.3.4.4. Aspiracion por Influencia

Sea $influencia(e) = (0 \vee 1)$ según si el movimiento que establece elvalor de $ComienzoTabu(e)$ es un movimiento de baja influencia o un movimiento de alta influencia. Además, sea $Ultima(L)$, para $L = (0 \vee 1)$, igual a la iteracion más reciente en la que fue realizado un movimiento de nivel de influencia L.

Entonces una aspiración de atributo para e se satisface si $influencia(e) = 0$ y $ComienzoTabu(e) < Ultima(1)$. Para múltiples niveles de influencia $L = 0, 1, 2, ...$, la aspiracion para e se satisface si hay un $L > influencia(e)$ tal que $ComienzoTabu(e) < Ultima(L)$.

6. Problema de Satisfaccion de Restricciones: Sudoku

El SuDoku es creado a partir de los trabajos del matemático suizo Leonhard Euler (1707-1783) cuando se encontraba trabajando en la demostración de teoremas sobre el cálculo de probabilidades.

Este juego fue exportado a Japón publicándolo en el periódico Monthly Nikolist en abril de 1984 bajo el título "Suji wa dokushin ni kagiru", que se traduce como "los números deben estar solos", y que posteriormente se abrevió como SuDoku (su: número, doku: solo).

En el ámbito de las matemáticas es visto como un problema de satisfacción de restricciones. Los juegos de puzzle como el SuDoku son muy populares debido a que se caracterizan por la simplicidad de sus reglas, simplicidad que necesariamente no significa no complejo, pues por el contrario se necesita de un razonamiento metódico para llegar a una solución. Cada instancia de puzzle tiene una solución única.

El análisis de la complejidad del SuDoku ya ha sido revisada por muchos autores, entre ellos [29] quién postula que el SuDoku es un problema puzzle de tipo ASP-completo (Another Solution Problem) a partir del cual, mediante inferencias formales, demuestra que un SuDoku de NxN casillas, es también *NP-completo* [1] . El autor de [27] presenta el problema del SuDoku como un Problema de Satisfacción Restricciones, donde se comparan los diferentes regimenes de propagación para resolver el SuDoku y ademas sugiere nuevas técnicas para resolver el problema sin la necesidad de realizar búsquedas.

	7	4	6		8	3	5	
1		3	4	5		6		9
2	5			3	1		7	
	3	7	2	4	9	1	8	
4		2	8		3	7	9	
	9		1	7			4	3
8		5	3	1	4	9		7
3		9		8		5	1	2
7	6	1		9	2	8		

9	7	4	6	2	8	3	5	1
1	8	3	4	5	7	6	2	9
2	5	6	9	3	1	4	7	8
5	3	7	2	4	9	1	8	6
4	1	2	8	6	3	7	9	5
6	9	8	1	7	5	2	4	3
8	2	5	3	1	4	9	6	7
3	4	9	7	8	6	5	1	2
7	6	1	5	9	2	8	3	4

Figura 6: Puzzle Sudoku Resuelto

6.1. Definición del Problema

El análisis del problema del Sudoku se divide en dos áreas principales, el análisis de las propiedades

- Cuadrículas Completas: Análisis de enumeracion de posibles soluciones para distintas variantes.
- Puzles: Análisis centrado en los valores dados.

[1] Subconjunto de NP. Los problemas NP-Completo pueden ser resueltos en tiempo polinomial en una Máquina de Turing No Determinista.

Las técnicas utilizadas en ambos son básicamente los mismos: la combinatoria y la permutación (Teoría de Conteo), las cuales se refuerzan mediante la aplicación de herramientas de programación. Existe muchas variantes de Sudoku, (parcialmente) caracterizadas por el tamaño (N) y la forma de sus regiones.
Los calculos que existen sobre la cantidad de posibles soluciones de Sudoku rectangulares se puede resumir en la tabla Tabla 6.1.

Dimensiones	Nr Soluciones
$1x?$	ver Latin squares
$2x2$	288
$2x3$	$28200960 = 2{,}8x10^{7}$
$2x4$	$29136487207403520 = 2{,}9x10^{16}$
$2x5$	$1903816047972624930994913280000 = 1{,}9x10^{30}$
$2x6$	$3829627892073810786374632473201249248618741760000 0 = 3{,}8x10^{49}$
$3x3$	$6670903752021072936960 = 6{,}7x10^{21}$
$3x4$	$81171437193104932746936103027318645818654720000 = 8{,}1x10^{46}$
$3x5$	Desconocido, estimado $3{,}5086x10^{84}$
$4x4$	Desconocido, estimado $5{,}9584x10^{98}$
$4x5$	Desconocido, estimado $3{,}1764x10^{175}$
$5x5$	Desconocido, estimado $4{,}3648x10^{308}$

Tabla 6: Comparativa de la cantidad de soluciones, según la cantidad de sus bloques.

Para el Sudoku clásico, $N = 9$ y las regiones son cuadrados de $3x3$ (llamados bloques o cajas), como se muestra en Figura 7.

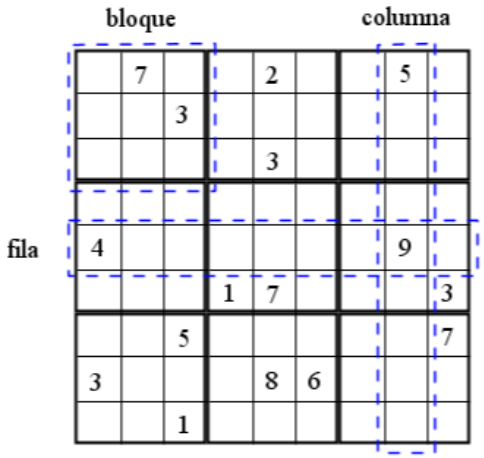

Figura 7: Ejemplo de Puzzle Sudoku

Un Sudoku rectangular, utiliza las regiones rectangulares de dimensión de las filas de la columna RxC. Para $Rx1$ (y $1xC$), es decir, cuando la región es una fila o columna,

Sudoku se convierte en un latin square. [2]

El problema general de resolver problema SuDoku de un tamaño n^2xn^2 y bloques de nxn, siendo el SuDoku clásico es de $n = 3$.
Sea:

- s una solucion de un problema de Sudoku con dimensiones específicas, satisfacion de la restriccion de la Regla Única [3].
- $S = \{s\}$, es el conjunto de todas las soluciones.
- $\|S\|$, es la cardinalidad de S, o sea el número de elementos en S.

El número de soluciones depende de las dimensiones del problema, las normas aplicadas y la definición de soluciones distintas. Para la el bloque de $3x3$, la convención para el etiquetado de las filas, columnas, bloques son como se muestran en Tabla 6.1.

Las filas se numeran de arriba a abajo, las columnas de izquierda a derecha. Por

1	2	3	4	5	6	7	8	9
2	B_1			B_3			B_3	
3								
4								
5	B_4			B_5			B_6	
6								
7								
8	B_7			B_8			B_9	
9								

Tabla 7: Términos de un problema de Sudoku

extensión, el sistema de etiquetado se aplica a cualquier bloque de un Sudoku rectangular.

Lss tripletas son combinaciones desordenadas de 3 valores que se utilizan en bloque-fila (o bloque-columna), por ejemplo, una *tripleta* $= \{3, 5, 7\}$ significa que los valores $3, 5, 7$ se presentan en un bloque-fila(columna), sin especificar el orden de ubicación. Una tripleta tiene 6(3!) permutaciones ordenadas. Por convención, los valores de la triplea están representados por sus dígitos ordenados. Los términos de triplea de *bloque-fila* y *bloque-columna* son:

- rBR, identifica una tripleta de fila de un bloque B y fila R, por ejemplo, r56 es la tripleta para el bloque 5, fila 6, con la etiqueta de fila de la cuadrícula.
- cBC, identifica de manera similar una tripleta en la columna del bloque B y de la columna C.

La notación $\{a, b, c\}$ también refleja el hecho de un triplete es un subconjunto de los dígitos deseados.

[2] Matriz de nxn elementos, en la que cada casilla está ocupada por uno de los n símbolos, de tal modo que cada uno de ellos aparece exactamente una vez en cada columna y en cada fila.

[3] La restricción es que cada dígito aparece en cada fila, columna y bloque

6.2. El número mínimo de números dados

En un Sudoku Clásico, según el trabajo [21] se demostró a través de una búsqueda exhaustiva que el número mínimo de números dados, la cual es la solución inicial parcial de un problema, es de 17 números.

Por lo tanto, para efectuar calculos de rendimiento de la integración, habrá que tener en consideracion el número mínimo de "pistas". A continuación se puede ver un ejemplo.

	7			2			5	
		3						
				3				
4							9	
			1	7				3
		5						7
3				8	6			
		1						

Tabla 8: Solución parcial inicial de un problema(puzzle) Sudoku

7. Integración de Arc-Consistency con Tabu Search

Desde la perspectiva de la *Programación con Restricciones*, y útil para esta integración, se ha trabajado con *Arc Consistency*, siendo utilizado para el pre-proceso y actualización del dominio, en cada generación de vecinos de la instancia del problema.

El desafío de este trabajo, es la integración de Arc Consistency y Tabu Search, por lo que para ilustrar de mejor forma como se realizará e esta integración se realizará mediante un ejemplo, el cual será una guía para abordar al menos el problema clásico de Sudoku ($9x9$).

La instancia inicial de un problema de Sudoku es de $n^2 \times n^2$ y las celdas pueden contener valores de $1, ..., n^2$. Las celdas sin valores se consideran vacías.

Una instancia de un problema de sudoku tiene $3N^2$ grupos de celdas n^2. Hay un grupo para cada una de las filas n^2, una para cada una de las columnas n^2 y una para cada una de los bloques n^2.

Una solución a un problema es colocar un valor de $1, ..., n^2$ en cada celda vacía de modo que cada grupo de células contenga exactamente un ocurrencia de cada valor de $1, ..., n^2$.

Las restricciones usan las mismas variables y dominios, pero tienen diferentes limitaciones. En todos las restricciones hay una variable X_c distinta para cada celda c en el puzle, siendo el dominio de cada variable $1, ..., n^2$.

Para la ilustración de esta integración se utilizará solamente un tablero de $4x4$, con la siguiente solución parcial:

	1		
		3	
			4
2			

Tabla 9: Solución inicial de un problema Sudoku de $2^2 \times 2^2$

7.1. Fase 1: Pre-Proceso

Hay que hacer énfasis, que si en la solución inicial la celda contiene un valor, v entonces el modelo inicial tiene la restricción $X_c = v$, por lo que sería útil aplicar la técnica de consistencia con el fin de actualizar el dominio de las variables, y además en caso de que sea posible, actualizar la solución inicial. Se destaca que en este proceso se itera hasta que ya no existan más cambios en la solución inicial o el dominio de las variables.

En la Tabla 7.1 ,se presenta el resultado de la primera fase sobre la solución inicial y el dominio de las variables.

Cabe notar que, *Solución* y la última *Solución'* son equivalentes, y por lo tanto el resultado esperado es el mismo. En caso de que se hubiese encontrado definido el valor 4, el cuál no se encuentra en el tablero, el Sudoku hubiese podido ser resuelto solamente en la fase de pre-proceso.

	1		
		3	
2			

Solución

3,4	1	2,4	2,4
4	2,4	3	1,2,4
1,3,4	3,4	1,2,4	1,2,3,4
2	3,4	1,4	1,3,4

Dominio

	1		
4		3	
2			

Solucion'

3	1	2,4	2,4
4	2	3	1,2
1,3	3,4	1,2,4	1,2,3,4
2	3,4	1,4	1,3,4

Dominio'

3	1		
4	2	3	
2			

Solucion'

3	1	2,4	2,4
4	2	3	1
1	3,4	1,2,4	1,2,3,4
2	3,4	1,4	1,3,4

Dominio'

3	1		
4	2	3	1
1			
2			

Solucion'

3	1	2,4	2,4
4	2	3	1
1	3,4	2,4	2,3,4
2	3,4	1,4	3,4

Dominio'

Tabla 10: Pre-Proceso en Fase 1

7.2. Fase 2: Aplicación de Tabu Search

Una vez realizado el pre-proceso, el dominio generado y la solución equivalentes se procede a encontrar una solución satisfactoria al problema Sudoku, en caso de que no haya sido resuelto durante el pre-proceso. Para comenzar, la búsqueda tabú (ver Algoritmo 10), se comienza con una solución inicial, la cuál puede haber sido actualizada o no en la Fase 1.

Para generar nuevas soluciones parciales, también llamados vecinos o candidatos (ver Algoritmo 9), se procede a verificar en donde falta la menor cantidad de valores por asignar para completar una fila o columna. Una vez seleccionada la fila o columna, se procede a calcular la probabilidad acumulada de los valores de una variable, entre las que falta por definir, permitiendo dar mayor probabilidad a los valores que menos se encuentran definidos en la solución.

Siguiendo con el ejemplo, puede escogerse la $fila1$ o la $columna2$, ya que ambos poseen la misma cantidad faltante de asignaciones en las variables, para este ejemplo se tomará la columna 2, las variables disponibles para realizar la asignación son C_23 y C_24, supondremos que aleatoriamente se seleccionó la variable C_23. El paso a seguir es calcular la probabilidad acumulada del dominio de C_23 , el cual es $\{3, 4\}$, siendo para $3 = [0;\ 0{,}4[$ y para $4 = [0{,}6;\ 1]$. Supiendo que aleatoriamente se generó un valor comprendido en el rango $[0{,}6;\ 1]$, se selecciona el valor 4, quedando $C_23 = 4$. Entonces, la solución parcial se actualiza, y se somete a verificar su consistencia y actualización de dominio.

3	1		
4	2	3	1
1	**4**		
2			

Solución actualizada.

Como se puede observar, al aplicar la técnica de consistencia el sudoku se ha resuelto por lo que el proceso se detiene.

Para prevenir ciclos y para salir de los óptimos locales, los nuevos atributos son agregados a la lista tabú si es que es aceptada en el vecindario de soluciones, y luego se procede a la ejecución del algoritmo AC3, con el fin de actualizar el dominio y la solución en caso de que existan nuevos valores que hayan sido "descubiertos". Se continuan creando nuevas soluciones hasta que se cumple el criterio de detención, ver Algoritmo 8, en donde todas las variables se encuentran definidas con un valor y se han satisfecho todas las restricciones. Una vez que la búsqueda tabú se detiene, la mejor solución es el Sudoku resuelto.

La elección y comparación de la mejor solución conocida hasta un cierto instante, viende dada por el cálculo del fitness. Para este CSP, se considerará la solución con mayor cantidad de variables definidas, debido a que se trabaja con soluciones parciales y en cada iteración se van descubriendo nuevos valores en las variables del problema.

3	1		
4	2	3	1
1	**4**		
2			

Solución

3	1	4	2
4	2	3	1
1	4	2	3
2	3	1	4

Dominio

3	1	4	2
4	2	3	1
1	4	2	3
2	3	1	4

Solución'

3	1	4	2
4	2	3	1
1	4	2	3
2	3	1	4

Dominio'

Tabla 11: Generación de Vecinos en Fase 2

Algoritmo 8 Condition Stop

if !ExistenCeros(sBest) **then**
  if $checkAC3(sBest)$ **then** ▷ Verifica la consistencia de la solución, es verdadera si cumple todas las restricciones, falso en caso contrario
    $Solved$
  else
    $NotSolved$
  end if
  $break$
end if

Algoritmo 9 Generar Candidatos

Require: $solution$, $varDomain$
$val \Leftarrow varDomain(findColRow())$;
$pos \Leftarrow randi(length(varDomain(val)))$; ▷ Valor aleatorio entre 1 y la cantidad de elementos.
$prob \Leftarrow calcProb(solution(pos))$; ▷ Calcular la probabilidad acumulada para la posición seleccionada de la solución parcial.
$solution(pos) \Leftarrow (prob > rand())(1)$; ▷ Seleccionar el primer valor que se encuentra en el rango.

Algoritmo 10 Sudoku Solver

Require: $initSolution$, $varDomain$,$iterationStop$, $iterationRestart$

```
tabuList ⇐ zeros(n^2, n^2)
sBest ⇐ s
f_sBest ⇐ countZero(sBest)
j ⇐ 0
restart ⇐ 0
while TRUE do
    candidateList ⇐ null
    for i ⇐ 0 to cantidateLength do
        candidateList ⇐ candidateList + sCandidate   ▷ Genera vecinos de la solución actual según el dominio.
    end for
    tabuList = update(tabuList)
    sCandidate ⇐ bestCandidate(candidateList, tabuList) ▷ Busca el mejor candidato de la lista.
    if sCandidateFitness < sBestFitness then
        varDomain, sCandidate ⇐ AC3(s)
        tabuList ⇐ differences(tabuList, sBest, sCandidate)
        sBest ⇐ sCandidate
        restart ⇐ 0
    else if restart = iterationRestart then
        s, sBest ⇐ initSolution
        f_sBest ⇐ countZero(sBest)
        restart ⇐ 0
    else
        restart++
    end if
    conditionStop()
end while
```

8. Experimentos

8.1. Equipo utilizado

Todas las pruebas han sido ejecutadas en un Notebook Compaq 610 (Procesador Intel Core 2 Duo T5870 2.0Ghz y 1 GB de RAM), bajo Fedora 17 de 32bits y los algoritmos han sido desarrollado en Octave 3.6.3.

Los algoritmos no han sido desarrollados para procesamiento paralelo, por lo que solo fue utilizado un núcleo al ejecutar las pruebas.

8.2. Instancias de Sudoku utilizadas

Los problemas de Sudoku escogidos[4], pertenecen a una investigacion hecha en [20], en donde se explica que para determinar el nivel de dificultad se considera que si el problema es resuelto en los siguientes intervalos de generaciones (iteraciones, ya que fueron generados y resueltos con *Algoritmo Genético*).

- $iteraciones < 3{,}500$ es un problema fácil de resolver.
- $3{,}500 < iteraciones < 10{,}000$ es un problema de complejidad media de resolver.
- $iteraciones > 10{,}000$ es un problema difícil de resolver.

Además, cabe mencionar que todos los problemas considerados poseen solamente una solución.

8.3. Resultados y discusión

Para efectos de comparación se consideraron todos los problemas seleccionados en la tabla 5 de [20], además se incluyeron los demás problemas de complejidad media y difícil, con el fin de verificar que la metodología utilizada puede resolver cualquier tipo de Sudoku.

Algo importante a destacar, es que todos las pruebas han sido ejecutadas por lo menos 50 veces de Sudokus resueltos con un máximo de 1.000 iteraciones, por lo que si al momento de realizar una prueba las iteraciones superar el máximo, la prueba se detiene y se marca como fallida.

A continuación se especifican las pruebas y los tiempos asociados a la resolución de los problemas:

[4]Sudoku Easy, Medium, Hard se encuentran disponibles en http://lipas.uwasa.fi/~timan/sudoku/

Problema	Pruebas	Resueltos	Tiempo Min.	Tiempo $\bar{X}$	Tiempo Max.
Easy a	30	30	1.324	1.331	1.354
Easy b	30	30	1.026	1.032	1.049
Easy c	30	30	1.046	1.067	1.113
Medium a	30	30	2.296	5.170	13.389
Medium b	31	30	5.151	69.807	245.346
Medium c	33	30	3.687	78.042	252.876
Hard a	35	30	2.608	88.550	259.610
Hard b	49	30	3.875	112.667	309.799

Para comparar los resultados con la tabla 5 del estudio [20], se considerarán solo los resultados de Mantere *et al*, debido a que éste es superior en todas las pruebas a los de Moraglio *et al.* [2]

Debido a que en este estudio solo se consideran los test con máximo de 1.000 iteraciones, no es posible comparar los resultados en relación a la cantidad de iteraciones o los tiempos que este toma para encontrar solución a los problemas. La comparación se realizará por la cantidad de soluciones encontradas, con un máximo de 30, en ciertos intervalos de intentos.

	Propuesta		GA	
Tipos	Intentos Ilimitados	100.000 Intentos	Intentos Ilimitados	100.000 Intentos
Easy a	30	30	30	30
Easy b	30	30	30	30
Easy c	30	30	30	30
Medium a	30	30	30	22
Hard a	30	30	30	2

En la tabla anterior se puede apreciar que el rendimiento de la metodología propuesta en este trabajo es perfecto, en comparación a los resultados del Algoritmo Genético, y por transición a las otras técnicas que se mencionan en [20].

La sinergia que existe entre ambas técnicas es lo que permite resolver el problema, por una parte se tiene al algoritmo de consistencia local, que permite actualizar el dominio de las variables y verificar tempranamente cuando un valor asignado a una variable no es válido, lo cual restrige ampliamente a la Búsqueda Tabú al momento de generar los candidatos. Por el lado de Tabu Search no hay que olvidar la Lista Tabú, que complementa el trabajo de Arc Consistency potenciándolo durante el paso de las iteraciones que se definan en la configuracion de la metaheurística.

9. Conclusiones

En esta investigación se han presentado dos técnicas para la solución de problemas complejos, en los cuales existen características similares, como lo son las variables y restricciones.

En el estudio de la primera técnología, *Programación con Restricciones* (CP), se revisaron los modelos y las técnicas más relevantes de un *Problema de Satisfacción de Restricciones*, siendo esta una de las dos ramas de CP. Dentro de las técnicas analizadas, se trabajó con técnicas de consistencia con el fin reducir el espacio de búsqueda. En específico, se utilizo *Arc Consistency*, la cual permite la eliminación de valores del dominio de la variable, provocando la reducción del espacio de búsqueda, sin verse afectado el nuevo espacio, el cual es equivalente al original.

Por otra parte, como se introdujo, las *Metaheurísticas* permiten resolver problemas en tiempos reducidos en comparación a técnicas más clásicas. Para este estudio se presentó la metaheurística *Tabu Search*, que a diferencia de otras provee de una estructura de memoria llamada *Lista Tabú*, permitiendo llevar un registro de las "soluciones ya visitadas" para la aplicación de distintas estrategias. Una de las estrategias de la Lista Tabú, es la restricción a través de la estructura de memoria, la cuál permite limitar la selección de los valores al momento de realizar la búsqueda en el dominio de las variables.

Para la aplicación del híbrido se escogió el problema de Sudoku, el cual es un problema tipo puzzle matemático con restricciones simples pero de alto costo computacional. Para este problema se estableció un nuevo enfoque, el cual se divide en 2 etapas, para poder realizar la integracion entre las técnicas anteriormente mencionadas. La primera etapa consta de filtrar el dominio de las variables del Sudoku, y en caso de poderse mejorar la solución inicial. Luego las variables recién generadas se someten a la segunda etapa en donde la metaheurística pueda encontrar la solución que satisfaga las restricciones del problema.

A través de un gran número de experimentos con instancias utilizadas en trabajos relacionados a la resolución y generacion de problemas de Sudoku, en este estudio se ha podido verificar la eficacia para la resolución del problema en los distintos niveles de complejidad. Obviamente, luego de los experimentos se ha observado que al aumentar la complejidad, sólo en su nivel más alto, existe una tasa de fallos en los cuales el Sudoku no ha sido posible de solucionar en una cantidad menor de iteraciones a las 1.000, pero esto se debe a que como es un método heurístico, parte de éxito se encuentra asociado a la selección correcta de valores en procedimientos aleatorios. Cabe señalar que si no existiese el tope de iteraciones la tasa de éxito de resolución del problema sería del 100 %.

Para finalizar, aunque se ha demostrado en éste trabajo que el enfoque propuesto de un proceso netamente de búsqueda estocástica y un soporte en la verificación de la consistencia de las variables, es capaz de resolver todas las instancias en las que fue probado, por lo que es evidente que la sinergia que existe entre las técnicas utilizadas.

Para un trabajo futuro, cabe mencionar que durante el desarrollo y prueba del híbrido se identificó que el principal problema es la selección de los valores a asignar en la fase de generación de candidatos, por lo que una mejora al híbrido sería disminuir el uso de la aleatoridad de la asignación de los valores.

10. Referencias

Referencias

[1] Colmerauer A. *Prolog II Reference Manual and Theoretical Model.* ., 1982.

[2] J. Togelius A. Moraglio and S. Lucas. Product geometric crossover for the sudoku puzzle. *IEEE Congress on Evolutionary Computation*, pages pages 470–476, 2006.

[3] A. Bouju, J.F. Boyce, Dr. A. Bouju, Dr. J. F. Boyce, J.G. Taylor, C.H.D. Dimitropoulos, G. Vom Scheidt, and Prof J. G. Taylor. Tabu search for the radio links frequency assignment problem. In *Proceedings of the International Conference on Digital Signal Processing*, 1995.

[4] Figueroa C. Meneses R. Castro C., Monfroy E. *Constraint Solving Using Dynamic Variable and Value Selection.* XXXI Conferencia Latino-Americana de Informática, CLEI, 2005.

[5] Monfroy E. Crawford B., Castro C. *Programación con Restricciones Dinámica.* Ingeniería Industrial - Año 8 N 2: 63 - 74.

[6] Pearl J. Dechter R. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[7] A. Dupont, M. Vasquez, and D. Habet. Consistent neighbourhood in a tabu search, chapter 17. In *Metaheuristics: Progress as real Problem Solvers. MIC-Kluwer*, pages 367–386, 2005.

[8] Audrey Dupont, Eric Alvernhe, and Michel Vasquez. Efficient filtering and tabu search on a consistent neighbourhood for the frequency assignment problem with polarisation. *Annals of Operations Research*, 130:179–198, 2004. 10.1023/B:ANOR.0000032575.38969.ab.

[9] Glover F. *Future Paths for Integer Programming and Links to Artificial Intelligence.* Computers and Operations Research, 1986.

[10] Walsh M Fogel L., Owens A. *Artificial Intelligence through Simulated Evolution.* John Wiley, 1966.

[11] Laguna M. Glover F. *TabuSearch.* Kluwer, 1997.

[12] Rechenberg I. Evolutionsstrategie – optimierung technischer systeme nach prinzipien der biologischen evolution. 1971.

[13] J. Ouaknine I. Lynce. Sudoku as a sat problem. *In 9th International Symposium on Artificial Intelligence and Mathematics AIMATH'06*, 2006.

[14] Lassez J. Jafar J. *Constraint Logic Programming.* 14th Annual ACM Symposium on Principles of Programming Languages (POPL), 1987.

[15] Apt k. *Principles of Constraint Programming.* Cambridge University Press., 2003.

[16] J. Gunther K. Moon. Multiple constrain satisfaction by belief propagation: An example using sudoku. *In 2006 IEEE Mountain Workshop on Adaptive and Learning Systems*, pages 122 – 126, 2006.

[17] Eberhart R Kennedy J. *Particle Swarm Optimization.* Proceedings of IEEE International Conference on Neural Networks, 1995.

[18] Vecchi M. Kirkpatrick S., Gelatt C. *Optimization by Simulated Annealing.* Journal of Optimization Theory and Applications, 1985.

[19] Dorigo M. *Optimization, Learning and Natural Algorithms.* Politecnico di Milano, Italy., 1992.

[20] T. Mantere and J. Koljonen. Solving, rating and generating sudoku puzzles with ga. *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on Evolutionary Computation (CEC 2007)*, 2007.

[21] Gary McGuire, Bastian Tugemann, and Gilles Civario. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem. *CoRR*, abs/1201.0749, 2012.

[22] Barricelli N. *Esempi numerici di processi di evoluzione.* ., 1954.

[23] Van Hentenryck P. The opl optimization programming language. *The MIT Press*, 1999.

[24] Zoeteweij P. *Composing Constraint Solvers.* Enschede, 2005.

[25] Barták R. *On-Line Guide to Constraint Programming.* 1998.

[26] Marriot K. Wallace M. Rafeh R., García M. *From Zinc to Design Model.*

[27] H. Simonis. Sudoku as a constraint problem. *In CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, pages pages 13–27, 2005.

[28] E. Monfroy T. Lambert and F. Saubion. A generic framework for local search: Application to the sudoku problem. *ICCS 2006, Part I, LNCS 3991*, page pages 641–648, 2006.

[29] T. Tato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Trans Fundam Electron Commun Comput Sci (Inst Electron Inf Commun Eng)*, E86-A:1052–1060, 2003.

[30] S. Van Roy P., Haridi. *Concepts, techniques, and models of computer programming.* MIT Press, 2004.