Investigación sobre optimizar la mejor solución de un juego en la programación

Research on optimizing the best game solution in programming

Autor 1: Alejandra Correa Galvis Autor 2: Katherine Marín Garay

*Universidad Tecnológica de Pereira, Colombia, Pereira-Risaralda*

Correo-e: [Jenny.correa@utp.edu.co](mailto:Jenny.correa@utp.edu.co)

[k.marin@utp.edu.co](mailto:k.marin@utp.edu.co)

***Resumen*— El objetivo de este trabajo es implementar y resolver técnicas de profundidad y búsqueda de una solución mas optima, en sistema de computación masiva que nos disponen de recursividad, en la exploración de árboles de juegos, elegido para dicho estudio el sudoku; donde se analizará como va variando sus respectivas combinaciones para llegar a un buen resultado posible.**

***Palabras clave— búsqueda, combinaciones, profundidad, resultado.***

***Abstract*—**   
**The objective of this work is to implement and solve depth techniques and search for a more optimal solution, in a massive computing system that we have recursively, in the exploration of game trees, chosen for said study the sudoku; where it will be analyzed how its respective combinations will vary to reach a good possible result.**

***Key Word* —** s**earch, combinations, depth, result.**

1. INTRODUCCIÓN

Tenemos el juego del sudoku que consideramos un problema de espacio de estados con factor de ramificación constante y con una única solución que se encuentra a profundidad. Calcular, tanto en el mejor como en el peor de los casos, el número de nodos que se necesitan analizar para encontrar la solución, al aplicar un algoritmo de búsqueda en anchura. Análogamente para la búsqueda profunda.

Especificación, razonando la respuesta, un problema de espacio de estados en el que la lista de cerrados en los algoritmos de búsqueda en anchura y en profundidad sea un camino factible.

Tenemos este problema de búsqueda en este caso tenemos una matriz de 9X9, lo que nos da un total de 81 variables, cada variable, puede contener un número del 1 al 9. las restricción,es aparecen primero entre las variables según las filas y columnas, de manera que una fila o columna en un numero puede aparecer solamente una vez y además hay una restricción adicional entre las variables de las nueve submatrices que forman el problema, dentro de cada submatriz no poder haber dos variables con el mismo número.

En el caso de este problema, el entretenimiento no viene de encontrar una solución, sino de obtener una solución dado que hemos dado ya ciertos valores a algunas de las variables. La dificultad de encontrar la solución se puede variar estableciendo el número de variables a las que damos el valor.

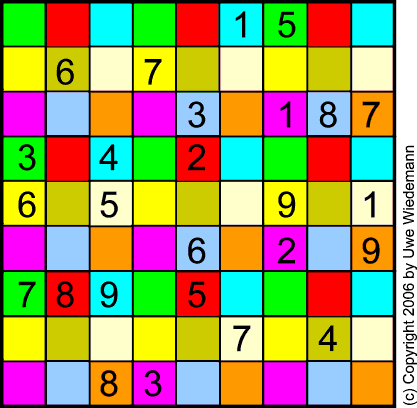


Figura 1

1. CONTENIDO

En la programación con restricciones, toda vez que la reducción por propagación no es suficiente por si sola, para encontrar una solución, recurrimos a algoritmo de búsqueda sistemática.

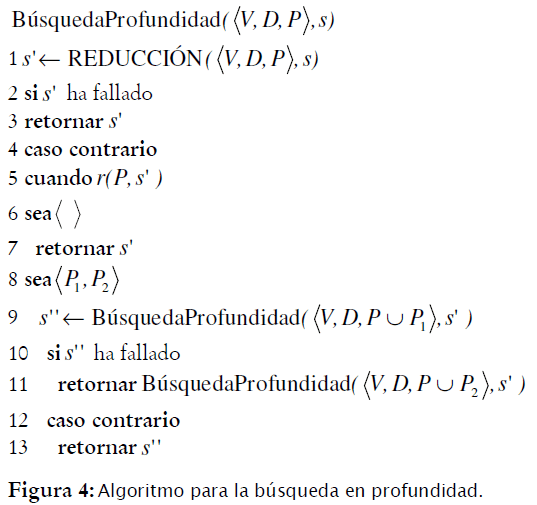
Una vez que se determina que la reducción por propagación no puede encontrar una solución por sí misma, se empieza a buscar la solución dividiendo el problema en otros subproblemas más pequeños o simples.

Típicamente un algoritmo de búsqueda requiere tanto de un árbol (de búsqueda) como de un algoritmo de exploración. En la programación con restricciones están a cargo de definir el árbol de búsqueda mientras que los algoritmos de exploración construyen, incrementalmente, el árbol hasta que una solución

O soluciones optimas que se puedan encontrar. Los algoritmos de exploración no siempre toman decisiones correctas, algunas veces comenten errores de los cuales necesitan aprender para retroceder y tomar otra decisión.

El propósito de tomar decisiones informadas, de acuerdo a alguna heurística, para sugerir nuevas restricciones que permitan dividir el problema en subproblemas de manera inteligente. (por ejemplo, para determinar cuál es la variable con el dominio mas pequeño) y el conjunto de propagadores (por ejemplo, para determinar cuál es la variable sobre la cual esta definida la mayor cantidad de restricciones).

Dos típicas estrategias de exploración son la búsqueda en profundidad y la búsqueda por ramificación y acotación



Estrategias de exploración son la búsqueda en profundidad y la búsqueda por ramificación y acotación.

Miremos los ramificadores que retornan o una tupla vacía {} o un par {Q1Q2} (en lugar de una tupla de N elementos).

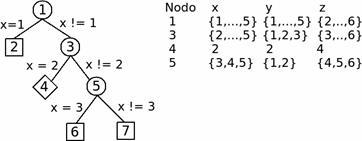
****

Figura 2

Ejemplo, consideramos un problema de restricciones con V={x, y}, D ={1,…6} , y C={X+Y=z,x.y=z} . lo podemos ver en la figura 2 el árbol de búsqueda para este problema con la heurística simple descrita al inicio de esta sección. En el árbol,

Un círculo representa un nodo con dominio de cardinalidad mayor a uno, un cuadrado representa un nodo que ha fallado y un diamante representa un nodo resuelto.

**2 OTRA OPCION ES LA TÉCNICA DEL BACTRACKING**

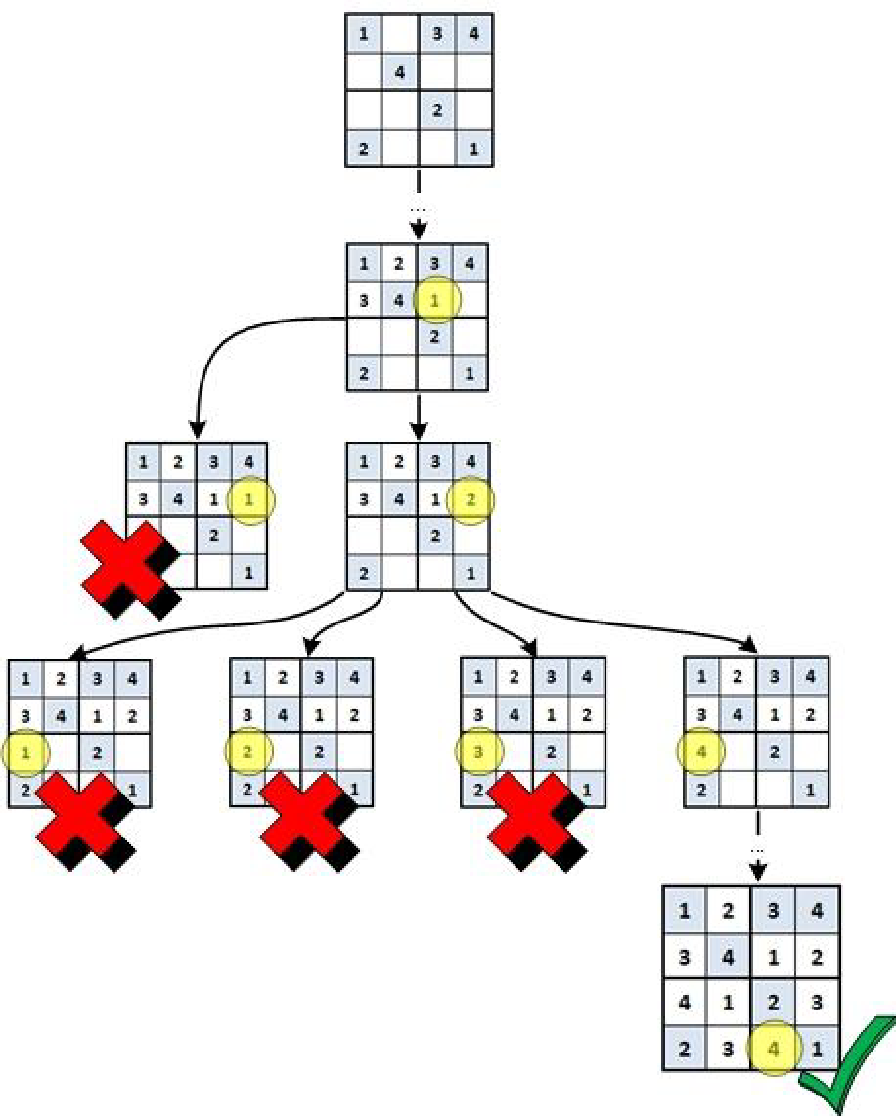
El algoritmo que se implementó con esta técnica se denomina en realidad o ramificación y poda, es uno de los más elegidos por los programadores debido a la velocidad con la que encuentra la respuesta. Sin embargo, para los investigadores se trata del menos útil a causa de su ineficiencia.

Dicha técnica se basa en un algoritmo de búsqueda en el cual se explora un árbol de soluciones en anchura o profundidad. Una de las alternativas de esta técnica es la de emplearlo en su forma más primitiva, que es la de un algoritmo de fuerza bruta.

En este caso el algoritmo se encargará de probar en cada casilla, ya partir de una casilla inicial, cada uno de los valores disponibles, analizando inmediatamente la validez de la decisión tomada. Si la solución parcial no satisface las condiciones de la Regla Única, el

algoritmo retrocederá (backtrack) al nivel de búsqueda anterior y generará otro tablero, con el siguiente valor disponible ubicado en la posición última de análisis. Este proceso se repetirá hasta concretar el análisis de la última casilla, con lo cual el costo computacional será de 𝑛(𝑅×𝐶)−𝑝, con “n” igual al número de valores posibles, “R” el número de filas, “C” el número de columnas y “p” el número de casillas con pista. A este costo se debe agregar, también, el de cada consulta de validez con lo cual el número es extremamente grande.

Para determinados tableros, entonces, el algoritmo tardará mucho tiempo, aún con el hardware actual. La ventaja, sin embargo, radica en que el algoritmo devolverá, tarde o temprano, la solución esperada. En la siguiente figura mostramos el espacio de soluciones para un pequeño Sudoku de ejemplo de 22 × 22 casillas, con 𝑁 = {1,2,3,4} .



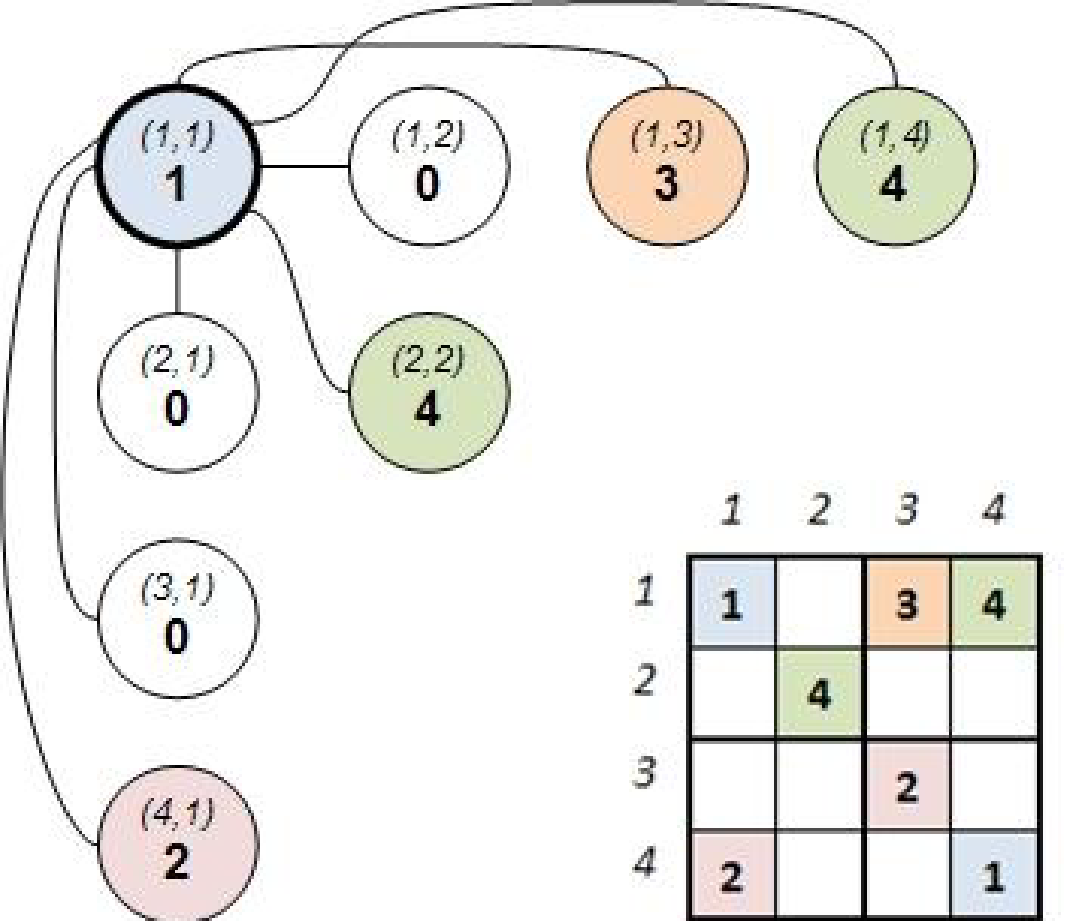
para un tablero sudoku de 4 x 4 de ejemplo. Obsérvese que el algoritmo de backtracking no expanda los nodos que no son válidos, y corta la búsqueda al encontrar la solución.

Otro de los algoritmos basados en backtracking consiste en

relacionar la búsqueda de una solución a un tablero Sudoku con el clásico problema de coloreo de grafos. Un grafo G = 〈V, E〉 es una estructura formada por un conjunto de vértices “V” y un conjunto de aristas o arcos “E” que los unen. El problema de coloreo de grafos es un problema NP, y consiste en hallar la cantidad mínima de colores necesarios para colorear cada vértice del grafo, teniendo en cuenta que dos vértices adyacentes (unidos entre sí por un arco) no pueden ser coloreados de la misma forma.

Si tenemos en cuenta que el conjunto de colores será el de los

valores que se utilizan para completar el Sudoku, el quid de esta relación radica en hallar una forma para representar el tablero Sudoku con un grafo tal que cumpla las condiciones del problema de coloreo. Sea “V” el conjunto de vértices de nuestro grafo. Si consideramos 𝑉 = {(𝑖, 𝑗), 0 ≤ 𝑗, 𝑖 ≤ 9} (es decir, tomamos cada casilla del tablero como un vértice del grafo), podemos construir “E” (el conjunto de aristas o arcos) de acuerdo al siguiente criterio: “dos vértices u y v están unidos por un arco no dirigido (u, v) sí y solo sí u y v comparten la misma fila, columna o bloque”. En la siguiente figura se observa, para el Sudoku del ejemplo anterior, cómo queda conformado el subgrafo de vértices adyacentes para la casilla siguiendo el criterio enunciado



Subgrafo creado a partir de las casillas relacionadas

con la (1,1). Se omitieron los vínculos entre todos los demás

vértices por cuestiones de prolijidad.

Luego, dicho algoritmo debe solo recorrer el grafo obtenido

coloreando de forma válida cada uno de los vértices, a partir del conjunto “N” de colores o de cifras del sudoku. En nuestro caso no tendremos que encontrar el conjunto mínimo de colores, sino buscar un coloreo válido para el grafo. Como todo tablero Sudoku debe admitir una y sólo una posible solución, queda asegurado que si el tablero es válido, el algoritmo devolverá un resultado satisfactorio en un tiempo exponencial, cuya expresión es una función de la cantidad de vértices que restan por colorear (cantidad de casillas vacías) y de la cantidad de colores disponibles (cantidad de valores con los que puede completarse una casilla).

**3. ALGORITMO DE BACKTRACKING**

**PARA EL SUDOKU EN JAVA.**

Con el algoritmo backtracking implementamos una técnica de

resolución de problemas mediante una búsqueda sistemática de soluciones.

Para optimizar el backtracking se descompone cada tarea en

tareas parciales y se prueba de manera sistemática cada una de

estas subtareas. Cuando al elegir una tarea comprobamos que no lleva a una solución, volvemos hacia atrás y probamos con una nueva. El siguiente fragmento de código Java comprueba utilizando backtracking si a través de una serie de pistas podemos obtener una solución de un Sudoku, o si no existe solución para esas pistas.

**public class** Sudoku {

**public static final int** *DIMENSION* =9;

**public static void** main(String[] args){

**int**[][] tablero= **new int**[][] {

{0,7,0, 0,0,0, 0,8,0},

{0,5,0, 6,0,0, 0,0,1},

{0,0,3, 1,4,0, 0,0,0},

{9,0,6, 0,5,0, 3,0,0},

{0,0,0, 0,0,0, 0,0,0},

{0,0,5, 0,2,0, 1,0,7},

{0,0,0, 0,6,5, 7,0,0},

{3,0,0, 0,0,1, 9,2,0},

{0,4,0, 0,0,0, 0,1,0},

};

*imprimir*(tablero);

**if**(!*resolver*(tablero)){

System.*out*.println("El Sudoku no

tiene solución");

}

}

**public static void** imprimir(**int**[][] tablero){

**for**(**int** i=0;i<*DIMENSION*;i++){

**if**(i%3==0){

System.*out*.println();

}

**for**(**int** j=0; j<*DIMENSION*;j++){

**if**(j%3==0){

System.*out*.print(" ");

}

System.*out*.print(tablero[i][j]);

}

System.*out*.println();

}

}

**public static boolean** resolver(**int**[][] tablero){

**for**(**int** i=0; i<*DIMENSION*; i++){

**for**(**int** j=0; j<*DIMENSION*; j++){

**if**(tablero[i][j]!=0){

**continue**;

}

**for**(**int** k=1;k<=9;k++){

**if**(*esPosibleInsertar*(tablero,i,j,k)){

tablero[i][j]=k;

**boolean** b=*resolver*(tablero);

**if**(b){

**return true**;

}

tablero[i][j]=0;

}

}

**return false**;

}

}

System.*out*.println("Encontrada solución:");

*imprimir*(tablero);

**return true**;

}

**public static boolean** esPosibleInsertar(**int** [][] tablero,

**int** i, **int** j, **int** valor){

//Comprueba columna

**for**(**int** a=0; a<*DIMENSION*; a++){

**if**(a!=i &&tablero[a][j]==valor){

**return false**;

}

}

//Comprueba fila

**for**(**int** a=0; a<*DIMENSION*; a++){

**if**(a!=j &&tablero[i][a]==valor){

**return false**;

}

}

//Comprueba cuadardo

**int** y= (i/3)\*3;

**int** x= (j/3)\*3;

**for**(**int** a=0; a<*DIMENSION*/3;a++){

**for**(**int**

b=0;b<*DIMENSION*/3;b++){

**if**(a!=i

&&b!=j&&tablero[y+a][x+b]==valor){

**return false**;

}

}

}

**return true**;

}

}

-

**4. CONCLUSIONES**

Se planteó el Sudoku, como un problema de optimización que

presenta características de alta complejidad matemática, como la presencia de una solución única y se propone un modelo

matemático. La estrategia de búsqueda local que implementa “Búsqueda Tabú “permite salir de configuraciones que tienen pocas repeticiones, pero en las que pequeños cambios muestran un empeoramiento en la solución (óptimo locales). Al prohibir movimientos en celdas modificadas recientemente se consigue escapar de esos óptimos locales.

El proceso de cruzamiento empleado por el algoritmo genético, da prioridad a las mejores configuraciones y extrae sus mejores atributos para generar la nueva población. Este comportamiento hace que el método no se desvíe y siempre trate de mejorar la solución durante el proceso de búsqueda.

El sudoku tiene solución única, lo que significa que no se

consideran soluciones válidas los tableros con situaciones

cercanas a la solución. En este sentido, el Sudoku no parece un problema adecuado para resolver utilizando un algoritmo

genético; pero si es un problema muy útil para experimentar con diferentes operadores genéticos sobre permutaciones, dado su alto grado de dificultad.

Los algoritmos de backtracking se caracterizan por permitirnos obtener la solución a todo problema de satisfacción de restricciones o de optimización. El costo computacional, sin embargo, está acotado en O (𝑛𝑘), con n representando la cantidad de hijos de cada nodo y k la cantidad de niveles del espacio de soluciones.

Como se ha dicho antes, nuestros tableros sudokus están formados por 81 casillas, y hasta hoy se han descubierto sudokus que pueden ser resueltos por el ser humano si tienen un mínimo de 17 pistas. Si consideramos que n es la cantidad de casillas modificables de un tablero y que k representa la cantidad de valores posibles que pueden ocupar una celda, el costo final de un algoritmo de backtracking sin poda es O (981−17) = O (964).

Ahora bien, las podas implementadas en nuestro algoritmo de

backtracking nos permiten reducir la complejidad temporal

significativamente, de acuerdo a un criterio que depende de forma exclusiva de las características de los tableros. Esto es, cuanto mayor sea la densidad de pistas en las primeras celdas del sudoku, el número de iteraciones se disminuirá. De la misma forma, si los primeros valores de la solución son altos y no están dados como pistas, el número de iteraciones necesarias se incrementará.

**REFERENCIAS**

1. Algoritmos de Backtracking

http://ocw.uc3m.es/ingenieriainformatica/

programacion/transparencias/tema7.pdf

1. Conceptos matemáticos del Sudoku

<http://en.wikipedia.org/wiki/Mathematics_of_Sudoku>

1. Algoritmos para resolver Sudokus

<http://en.wikipedia.org/wiki/Algorithmics_of_Sudoku>

1. Los Cuadrados Mágicos

http://eloviparo.wordpress.com/2010/04/19/los-cuadradosmagicos/