Universidad de Santiago de Chile
Departamento ingeniería informática
Ingeniería civil informática
Algoritmos Avanzados
Laboratorio

1-2019

Como obtener la mínima cantidad de movimientos para resolver un puzle 8 con el método de Backtracking

Alberto Rodríguez Zamorano
Alberto.rodriguez.z@usach.cl
https://github.com/Albertorz31

Algorithm design: Explicit enumeration.

RESUMEN: El siguiente documento consiste en la resolución de un problema por medio de la implementación de Backtracking. El problema que trata el documento es el poder resolver un puzle 8 cualquiera, y que este entregue la menor cantidad de movimientos que necesita el puzle para resolverse. El documento trata el problema mediante la creación de un algoritmo que recibe de entrada un archivo, el cual contiene 3 líneas correspondientes a una matriz 3x3, esta representa el estado actual del puzle 8, en la cual la letra X representa el espacio. Por esto se debe entregar un archivo de texto en el que muestre el mínimo número de pasos a resolver el puzle anteriormente mencionado, junto con la lista de los movimientos de la X para resolverlo. Para su solución, cada posible movimiento que se puede hacer en el puzle se toma como un nuevo nodo de un árbol, el cual la raíz será el estado inicial del puzle. Se tomó como la idea la creación de una lista enlazada que guarde los movimientos que va realizando el puzle hasta llegar a la solución. Si encuentra después en otra hoja del árbol una mejor solución que la anterior guardará esa. El algoritmo es recursivo y se descompone en máximo 4 subproblemas y el tiempo para definir el subproblema es de O(1), el algoritmo tiene una complejidad de O(4^n). Al no ser la complejidad un polinomio se puede decir que el algoritmo no es eficiente, aunque para este caso, el puzle es pequeño, entonces el valor de n no será tan grande (solo para algunos casos) y no siempre se descompone en 4 subproblemas . Sin embargo, no se sabe si puede que haya otra forma de resolver el problema con tal que sea más efectiva.

PALABRAS CLAVE: Lista, algoritmo, puzle, movimientos, matriz.

ABSTRACT: The following document consists in the resolution of a problem through the implementation of Backtracking. The problem dealt with in the document is to be able to solve any puzzle 8, and that it delivers the least amount of movements that the puzzle needs to be solved. The document addresses the problem by creating an algorithm that receives a file input, which contains 3 lines corresponding to a 3x3 matrix, this represents the current state of puzzle 8, in which the letter X represents the space. Therefore, a text file must be submitted in which it shows the minimum number of steps to solve the puzzle, together with the list of movements of the X to solve it. For its solution, every possible move that can be made in the puzzle is taken as a new node of a tree, which root will be the initial state of the puzzle. It was taken as the idea to create a linked list that keeps the movements that the puzzle is making until reaching the solution. If you find later in another leaf of the tree a better solution than the previous one will keep that one. The algorithm is recursive and is broken down into a subproblem and the time to define the subproblem is O (1), the algorithm has a complexity of O (n). Being the complexity a polynomial we can say that the algorithm is efficient, therefore, the idea is viable and functional. However, it is not known if there may be another way to solve the problem as long as it is more effective.

KEYWORDS: List, algorithm, puzzle, movements, matrix.

1 INTRODUCCIÓN

En la asignatura de Algoritmos Avanzados, la temática que se trata es la creación de algoritmos de poca complejidad para resolver problemas. El documento presente busca la resolución de un problema por medio de Backtracking o algoritmo de retroceso, cuyo problema trata de buscar el mínimo de movimientos para resolver un puzle 8 (una matriz de 3x3). Lo que se espera de la aplicación de Fuerza Bruta es que sea poco compleja como solución, dejándola para posibles comparaciones en el futuro con otras formas de solución, de tal forma que se busque cual tiene un orden de complejidad menor.

2 DESCRIPCIÓN DEL PROBLEMA

El equipo JURASCH (Juega Usach) se inscribirá en la competencia mundial de speed run y planea llevarse el título de "*los jugadores más rápidos"* de este año. Pero su mayor problema es el puzle 8 ya que pierden mucho tiempo en resolverlo. Dado esto necesitan un programa que entregue la mínima cantidad de movimientos que necesita el puzle para resolverse y así no perder tiempo valioso en la competencia. Para esto la entrada debe ser un archivo de texto que contenga las 3 líneas correspondientes a un matriz, la cual simula el estado inicial del puzle.

Entrada1.in

```
4 1 3
7 X 6
5 2 8
```

Figura 1: ejemplo de entrada.

Y la salida debe ser un archivo de texto que contenga el mínimo número de pasos para resolver el puzle y después en cada línea los movimientos de X para resolverlo.

Salida1.out

```
8
Abajo
Izquierda
Arriba
Arriba
Derecha
Abajo
Abajo
```

Figura 2: ejemplo de salida.

Además, el algoritmo debe incluir las funcionalidades backtracking() y printCurrent(). La función backtracking() debe ser una función que genera todas las combinaciones y al mismo tiempo encuentre la solución. La función printCurrent() debe ser una función que imprima el estado actual de un nodo del árbol, es decir, muestre el estado del puzle según la iteración y los movimientos que se ha hecho (puede mostrar estados que no necesariamente llegan a la solución):

```
printCurrent(...) {

        #ifdef DEBUG
        printf("enter para continuar...\n")
        while(getchar() != '\n');
        /*
        en esta parte debe escribir su código para imprimir
        lo que sea necesario para mostrar el estado actual
        del nodo.
        */
        #endif
    }
```

Figura 3: Parte del algoritmo printCurrent()

3 MARCO TEÓRICO

Para comprender los conceptos utilizados en el presente documento, se entiende lo siguiente: *Backtracking*, en informática, consiste en un método por el cual se busca la o las mejores

soluciones que satisfagan ciertas restricciones. Es considerado un algoritmo de Búsqueda en profundidad, retrocediendo en el camino si este no cumple con las condiciones y tomando otro camino posible. Uno de los principales casos son los arboles de búsqueda y su aplicación de *backtracking* a mayor escala es la *Ramificación y Acotamiento,* donde va construyendo el camino óptimo mientras cumpla con las condiciones. Como se vio anteriormente en el laboratorio 1, *backtracking* es similar a Fuerza Bruta, pero este incluye el *Acotamiento*, lo cual lo haca acotar ramas del árbol, no, así como Fuerza Bruta que si recorre todas las ramas.

Una matriz, en informática, es una estructura de datos que almacena conjuntos de variables, donde cada conjunto es representado con un índice (filas) y cada valor tiene asociado un índice (columnas). En este caso, si se desea obtener el dato de la matriz, se debe indicar el índice de posición de la fila, junto con el índice del dato en la columna. Las matrices en programación tienen similitud con las matrices matemáticas. Un ejemplo de matriz es:

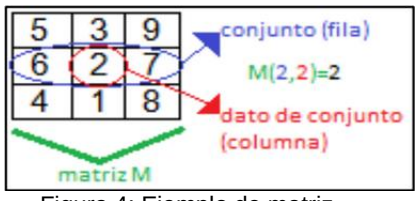

Figura 4: Ejemplo de matriz.

Un *Árbol* es una estructura de datos que se puede definir como una colección de nodos organizados de forma recursiva. Existe un nodo inicial llamado *raíz* el cual esta enlazado a otros nodos, cada una de esas conexiones se le llama *ramas,* de las cuales de ese nodo se pueden originar *subárboles*. Las raíces de los subárboles se denominan *hijos* de la raíz, y consecuentemente la raíz se denomina *padre* de las raíces de sus subárboles. Los nodos que no poseen hijos se les denomina *hojas.* En un árbol existe un único camino desde la raíz hasta cualquier otro nodo del árbol. La siguiente figura muestra como es un árbol:

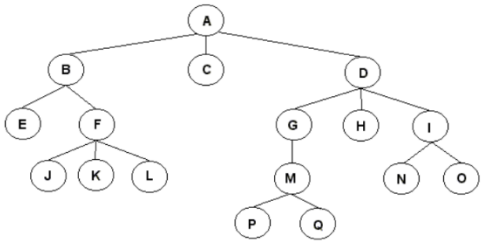

Figura 5: Ejemplo de Árbol.

4 DESCRIPCIÓN DE LA SOLUCIÓN

4.1 IDEA

Para resolver este problema por medio de *backtracking*, se tiene pensado el uso de listas enlazadas, matrices y árboles. En el archivo de entrada, los datos de cada línea son los datos que se guardarán en cada una de las posiciones de la matriz, por lo tanto, esa matriz será el nodo raíz del árbol. Después se debe verificar cuales son los posibles movimientos que puede hacer el X (espacio del puzle). Por ejemplo, si el X está al medio puede hacer 4 posibles movimientos (arriba, abajo, izquierda y derecha), entonces la raíz tendría 4 ramas, las cuales serían los 4 movimientos posibles y cada nodo hijo sería el nuevo estado del puzle realizando un movimiento.

Para poder realizar el acotamiento del árbol, se debe tomar en cuenta dos cosas, uno es la *distancia Manhattan,* la cual consiste en la suma de las distancias de todas fichas del puzle que no esté en su posición hasta llegar a la posición que corresponda, es decir, si el puzle es:

- 1 2 3
- 5 6 4
- X 8 7

La distancia Manhattan seria 8 porque seria la suma de la distancia de cada ficha desacomodada (las cuales son 4,5,6,7 y el X) a la posición a la que debería ir. Entonces se calcula la distancia Manhattan del nodo raíz y

después se calcula la distancia Manhattan para cada uno de los posibles movimientos que se pueden hacer. Para saber si seguir por una rama o no, la distancia Manhattan de ese nodo tiene que ser a lo mas 2 unidades mayor que el nodo padre, si no es así no se meterá por ese camino. Cuando llegue a un nodo que es igual a la solución, guardará los movimientos realizados en una lista enlazada. Cuando recorra todo el árbol entregará el mínimo número de movimientos para llegar a una solución.

4.2 REPRESENTACIÓN Y ESTRCUTURA

Los datos del archivo de texto de la entrada inicialmente son representados como strings, pero luego son transformados a enteros para ser trabajados. La lista enlazada va a poseer 2 tipos de datos para almacenar; el primero es de tipo char llamado *movimiento,* el cual contiene el movimiento realizado en el puzle y otra llamada *siguiente* para ver el siguiente nodo en la lista. Cuando se crean las combinaciones, donde cada combinación es una lista enlazada con los movimientos realizados, esta lista se reemplaza si se encuentra que la lista de la combinación actual tiene menor cantidad de movimientos que la anterior. Cuando se tiene la lista definitiva, se calcula el largo de la lista para ingresarlo en el archivo final, junto con las posiciones que se realizaron en el puzle para resolverlo .

4.3 PSEUDOCÓDIGO

Para el tipo de dato lista, posee alguna de sus funcionalidades básicas de Crear_lista(lista, entero, entero), obtener(lista, entero), insertar(lista, lista, entero), copiar_lista(lista) y Largo(lista). La lectura del archivo se hace en la función principal, la cual también crea el puzle inicial, la cual entrega la lista final. Pero la función más importante es la que realiza el método de *Backtracking*, la cual en el código se divide en dos partes, aunque esto es solo por temas de no iniciar con una lista vacía la recursión, entonces la primera parte solo verifica los posibles movimientos del nodo raiz.

Para el pseudocódigo se mostrará el backtracking que realiza el código de forma general (no se separá en 2 partes), y cuales son los acotamientos para no realizar ese movimiento. También se acortara la cantidad de posibles casos que tiene el puzle, ya que, son muchos y alargaría mucho el pseudocódigo, se motrará de manera general.

```
backtracking(puzzle,movimientos,manhattan,iter
acion,listaFinal,puzzleAnterior): listaFinal
   #Las próximas condicionales son unos de
   #los acotamientos el árbol
   Si(puzle esta resuelto):
      verificarLargo(movimientos,listaFinal)
       Devolver listaFinal
   Si(iteración > 32): #Hace max 32 movimientos
       Devolver listaFinal
   Si(largo(movimientos)>largo(listaFinal)):
       Devolver listaFinal
   Sino:
     Si(puzle[0][0]==X):
        #Se realiza lo siguiente con todos
        #los posibles movimientos en esta
        #posicion y hace una recursion de
        #de todos los posible movimientos
        puzleNuevo = posibleMovimiento
        manhattanNuevo =obtenerMan(puzleNuevo)
        #El siguiente es otro acotamiento
        Si(manhattanNuevo< manhattan+2):
          Si(puzzle != puzzleAnterior):
            NuevoMov = crearNuevoMovimiento
            agregarMov(movimientos,NuevoMov)
            listaFinal=backtracking(puzleNuevo
            ,manhattanNuevo,iteración +
            listaFinal,puzle)
     Si(puzle[0][1]==X):
       #Se realiza lo mismo que el anterior
     Si(puzle[0][2]==X):
       #Se realiza lo mismo que el anterior
     .
     .
     .
     Si(puzle[2][2]==X):
      #Se realiza lo mismo que en anteriores

  Devolver listaFinal

verificarLargos(movimientos,listaFinal):
  Si(largo(movimientos) < largo(listaFinal):
    listaFinal= movimientos
```

La última parte verifica si el largo de la lista actual es menor a la lista final (siempre y cuando esta no esté vacía). Si es menor se reemplaza la lista final por la otra, así se guardará la lista con menor cantidad de movimientos.

4.4 TRAZA

Para la traza de dará un caso ejemplo, en el que se ingresa un puzle cualquiera (en cual se mostrará en la siguiente imagen). En esta sección mostraremos el funcionamiento de la función printCurrent, la cual va mostrando el puzle de cada nodo del árbol, junto con la cantidad de movimientos que se han realizados y la distancia manhattan de ese puzle.

- Puzle inicial: Se tiene inicialmente el puzle ingresado en el archivo (el 0 indica la X), junto con su distancia manhattan.

```
Movimientos realizados: 0  Distancia manhattan: 12
2 |5 |1
7 |6 |3
X |8 |4
```

- Primer caso: después e obtiene el primer movimiento posible por el puzle, que cumpla con las restricciones. En este caso mueve el X para arriba.

```
Movimientos realizados: 1  Distancia manhattan: 12
2 |5 |1
X |6 |3
7 |8 |4
```

- Segundo caso: el algoritmo puede iterar muchas veces, para este caso vemos que se han realizado 26 movimientos y no ni cerca de la solución, por lo tanto, parece que tomará después otra rama del arbol.

```
Movimientos realizados: 25  Distancia manhattan: 16
6 |X |1
7 |2 |3
8 |5 |4
```

4.5 ORDEN DE COMPLEJIDAD

Anteriormente en el resumen se explicó que el algoritmo es recursivo, ya que, utiliza el método de backtracking se descompone en máximo 4 subproblemas de tamaño n-1, el tiempo para definir el subproblema es de 1, entonces el tiempo es de:

$$T(n) = 4T(n-1) + O(1)$$

Entonces si luego se utiliza la fórmula para algoritmos recursivos por sustracción. El algoritmo finalmente tiene un orden de :

$$\sim O(4^n)$$

Para todas las funciones básicas de la lista enlazada tiene complejidad de orden O(n), debido a que n corresponde a la cantidad de veces que se recorre la lista.

Para el algoritmo de printCurrent su complejidad tambien es O(n) debido a que imprime todos los elementos de una lista.

5 ANÁLISIS DE LA SOLUCIÓN

5.1 ANÁLISIS DE IMPLEMENTACIÓN

Se puede comprobar que el algoritmo se detiene, entregando el resultado si el archivo ingresado es correcto con los requerimientos. Esto quiere decir que el algoritmo se detiene cuando entrega el resultado operando con los parámetros correctos, que sería un archivo de texto que contenga los datos necesarios. Por el momento, se puede considerar como un algoritmo no eficiente pues la complejidad no es un polinomio, aunque el algoritmo en ningún

caso demoró tanto tiempo en encontrar el resultado, además con los acotamientos dados reduce considerablemente las ramas del arbol. Puede que el algoritmo fuera más eficiente si se conocieran más funcionalidades disponibles que reduzcan el tiempo y complejidad, como también otros métodos para elaborar algoritmos. Otro método que también hubiese funcionado es utilizar Goloso, ya que devuelve la primera solución que encuentra, puede que no sea la óptima, pero para resolver un puzle por lo general no hay tantos caminos para solucionarlo.

Se realizaron varias pruebas en el programa, se utilizó un gráfico para mostrar la solución de una prueba, el cual resuelve el problema en 8 movimientos. Crea aproximadamente 3240 estados, el tiempo de solución es de aproximadamente 4 segundos para este caso.

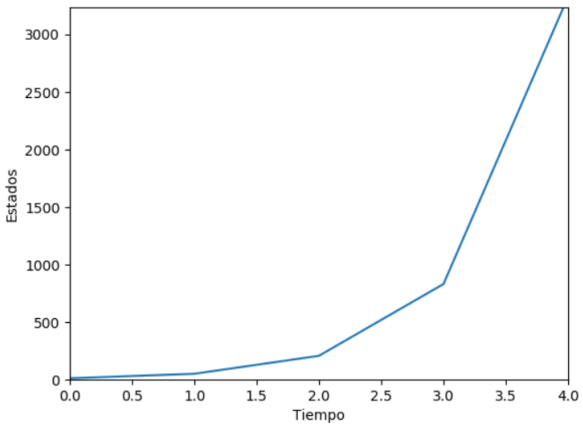

Figura 6: Gráfico de estados vs tiempo.

5.2 EJECUCIÓN

El algoritmo fue implementado en el lenguaje de programación C. Además, se trabajó en el sistema operativo Windows 10, por lo tanto, para la compilación del programa se debe agregar en la consola lo siguiente:

```
Avanzados\Lab2>gcc backtracking.c -o ejecutable.exe
```

En el caso de que se quiere compilar implementando el DEBUG, se agrega lo siguiente:

```
Avanzados\Lab2>gcc backtracking.c -DDEBUG -o ejecutable.exe
```

Por ultimo, para ambos casos de compilación, la ejecución es la siguiente, en la cual se deben ingresar el archivo de entrada y el nombre del archivo de salida:

```
Avanzados\Lab2>ejecutable.exe entrada.in solucion.txt
```

6 CONCLUSIONES

Tras el desarrollo y análisis del algoritmo, se puede extraer que fue fácil de implementar el código, y que se logra de buena forma el método de Backtracking, consiguiendo el resultado esperado. Sin embargo, siendo Backtracking un método donde la generación de posibles soluciones se basa en restricciones para filtrar esas mismas, en este algoritmo al ser recursivo con estados pendientes genera muchos estados para buscar la solución de este, depende mucho si encuentra una solución rápidamente para realizar una buena filtración de estados, sino puede que itere casi igual que Fuerza Bruta.

En comparación con el algoritmo implementado por mis compañeros de grupo de este enunciado, utilizando el método de Fuerza Bruta, la traza que realizan es la misma, con la diferencia es que asumen que aproximadamente siempre el árbol generará 2 hijos, y yo siempre asumo el peor caso es que generará 4 hijos, por eso las diferencias de complejidad, pero si asumiéramos lo mismo seria la misma complejidad. La otra diferencia es su tiempo de ejecución es demasiado comparado con el de este algoritmo, entonces se concluye que realizando un acotamiento cambia significadamente el tiempo del algoritmo.

En conclusión se puede decir que el método de Backtracking, es recomendable para este tipo de problemas ya que el tiempo de resolverlos es aceptable a pesar de no tener una buena complejidad.

7 REFERENCIAS

Enlaces de donde se ha basado para el desarrollo del documento. El formato debe ser

- “Algoritmos: Teoría y aplicaciones” , Mónica Villanueva ,2002 , disponible en: http://www.udesantiagovirtual.cl/moodle2/pluginfile.php?file=%2F117134%2Fmod_resource%2Fcontent%2F1%2FAlgoritmos-apuntes.pdf