INSITUTO POLITECNICO
NACIONAL
ESCUELA SUPERIOR DE COMPUTO

Instituto Politécnico Nacional

Escuela Superior de Cómputo

Fundamentos de Inteligencia Artificial

Reporte de práctica 2

PROFESOR: Macario Hernandez Cruz **GRUPO: 4BM1**

INTEGRANTES:

Carmona Serrano Ian Carlo

Mendez Lopez Luz Fernanda

Rosales Benhumea Aldo

Práctica 2

Búsqueda no Informada

(Búsqueda primero en anchura y Búsqueda primero en profundidad)

RESUMEN

En esta práctica se implementaron dos algoritmos de búsqueda no informada donde pudimos observar los comportamientos que tienen el algoritmo primero en anchura y el algoritmo primero en profundidad, al resolver un puzzle.

Sabemos que el algoritmo primero en anchura es el más eficaz para encontrar la solución a los problemas, sin embargo, su mayor deficiencia es la cantidad de recursos de memoria y tiempo que consume, mientras que el algoritmo de primero en profundidad es más rápido, pero este no asegura encontrar un estado a la meta.

INTRODUCCIÓN

Búsqueda no Informada

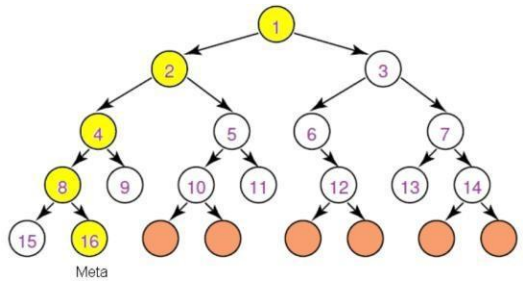
Los métodos de búsqueda no informado son estrategias de búsqueda en las cuales se van a generar estados siguientes y son evaluados con respecto al estado meta sin saber si mejoran su situación con respecto del caso anterior.

Búsqueda primero en anchura (BFS)

Este algoritmo de búsqueda no informado va a desarrollar por niveles todas las posibles alternativas de estados siguientes, el algoritmo se detiene hasta encontrar un nivel con una solución.

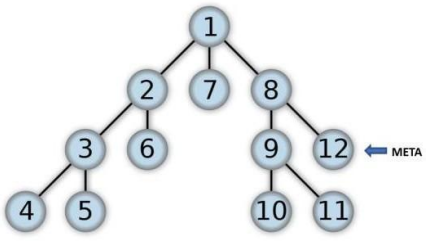
Su complejidad tanto espacial como temporal es exponencial; por ello, esta estrategia es sólo aplicable a problemas que no tienen una gran amplitud.

Este procedimiento es completo, ya que encontrará siempre una solución si es que la hay, pero en general no es óptimo.



Búsqueda primero en profundidad (DFS)

Este algoritmo de búsqueda no informado consiste en desarrollar las ramas de un árbol hasta llegar a la solución del problema, es más rápido que el algoritmo primero en anchura, ya que en este algoritmo no desarrolla todos los posibles estados siguientes, solo desarrolla uno hasta llegar al fondo, si no encuentra la solución va a desarrollar otra solución, de esta manera hasta encontrar el estado soluciones, el inconveniente de este algoritmo es que si existe solución y se mete por una rama infinita puede que no termine nunca. Tampoco es óptimo, ya que puede encontrar la solución pero haciendo un recorrido mayor del necesario en general.



DESARROLLO DE LA PRÁCTICA

La práctica se desarrolló utilizando el lenguaje de programación Java y sus capacidades de Programación Orientada a Objetos.

El programa consta de 4 clases las cuales son Pozole (la clase principal), Tablero (clase que instancia y muestra la interfaz gráfica), State (que almacena los estados del programa durante la ejecución) y por último Executor (que ejecuta un hilo que muestra al usuario como sería la resolución paso a paso de manera gráfica).

CLASE TABLERO

Esta clase es importante porque vamos a definir la configuración de este, de los elementos más importantes que podemos notar son **start:** que nos permite desordenar el tablero y **goal:** que es el estado final al que tiene que llegar.

```
public class Tablero extends JFrame {
    private final JButton[][] jBoard = new JButton[4][4];
    private final LinkedHashMap puzzle = new LinkedHashMap();
    private BufferedImage empty;
    private boolean deep = false;

    private final String start = "12C4EDB5AF369870";
    private final String goal = "12C4EDB5F036A987";

    private final JMenuItem solveB = new JMenuItem("Solve BFS");
    private final JMenuItem solveD = new JMenuItem("Solve DFS");

    private final int maxDeep = 100000; // Para limitar la profundidad del árbol
```

CLASE ImagePieces

Esta clase nos permite partir la imagen que vamos a utilizar, para esto necesitamos conocer el tamaño de la imagen en pixeles y cuántas veces la vamos a querer dividir, en este caso la teníamos que dividir en 4 pedazos por lo que, la cantidad de pixeles la dividimos entre 4, obteniendo así 125.

```
// Parte la imagen en piezas
private void imagePieces(String pathName) {
    try // Bloque "try" por que hay una tarea de lectura de archivo
    {
        BufferedImage buffer = ImageIO.read(new File(pathName));
        BufferedImage subImage;
        int n = 0;
        for (int i = 0; i < 4; i++)
            for (int j = 0; j < 4; j++) {
                subImage = buffer.getSubimage((j) * 125, (i) * 125, 125, 125); // Extrae un fragmento de la imagen
                String k = goal.substring(n, n + 1);
                puzzle.put(k, subImage); // Almacena las piezas etiquetándolas con base en el estado final
                n++;
            }
    } catch (IOException ex) {
        ex.printStackTrace(System.out);
    }

}
```

CLASE SOLVE

Este método contiene una sentencia **if** en la cual dependiendo de la elección del usuario, se resolverá el ejercicio, ya sea que el usuario elija resolver por anchura o profundidad.

```
if (success) // Si hubo éxito
{
    long elapsed = System.currentTimeMillis() - startTime;
    if (deep)
        this.setTitle("8-Puzzle (Deep-First Search)");
    else
        this.setTitle("8-Puzzle (Breadth-First Search)");
    JOptionPane.showMessageDialog(rootPane,
            "Success!! \nPath with " + path.size() + " nodes" + "\nGenerated nodes: " + totalNodes
                    + "\nDead ends: " + deadEnds + "\nLoops: " + m + "\nElapsed time: " + elapsed
                    + " milliseconds",
            "Good News!!!", JOptionPane.INFORMATION_MESSAGE);
    System.out.println("Success!");
    String thePath = "";
    int n = 0;
    int i = startState.getI();
    int j = startState.getJ();
    for (State st : path) {
        st.show();
        if (n > 0)
            thePath = thePath + st.getMovement();
        n++;
    }
    Executor exec = new Executor(jBoard, i, j, thePath, empty);
    exec.start();
} else {
    JOptionPane.showMessageDialog(rootPane, "Path not found", "Sorry!!!", JOptionPane.WARNING_MESSAGE)
    System.out.println("Path not found");
}
```

```
// Search loop

int m = 0;
long startTime = System.currentTimeMillis();
while (!queue.isEmpty() && !success && m < maxDeep) {
    int validStates = 0;
    m++;
    // System.out.println("Ciclo " + m);
    ArrayList<State> l = (ArrayList<State>) queue.getFirst();
    // System.out.println("Analizando Ruta de :" + l.size());
    // muestraEstados(l);
    State last = (State) l.get(l.size() - 1);
    // last.show();
    ArrayList<State> next = last.nextStates();
    // System.out.println("Se encontraron " + next.size()+ " estados sucesores
    // posibles");
    totalNodes += next.size();

    queue.removeFirst(); // Se elimina el primer camino de la estrutura
```

```
    for (State ns : next) {
        if (!repetido(l, ns)) // Se escribió un método propio para verificar repetidos
        {
            validStates++;
            ArrayList<State> nl = (ArrayList<State>) l.clone();
            if (ns.goalFunction(goalState)) {
                success = true;
                path = nl;
            }
            nl.add(ns);
            // muestraEstados(nl);
            if (deep)
                queue.addFirst(nl); // Si es en profundidad agrega al principio la nueva ruta
            else
                queue.addLast(nl); // Si es en anchura agrega al final
            // System.out.println("Agregé un nuevo camino con "+nl.size()+ " nodos");
        }
        // else System.out.println("Un nodo repetido descartado");
    }
    if (validStates == 0)
        deadEnds++; // Un callejón sin salida
}
```

CLASE Repetido

Esta clase verifica cada vez que se crea un nuevo nodo del árbol si ese nodo ya fue creado en otra rama o en otro nivel, para no gastar memoria extra en duplicados de nodos.

```
// Compara para evitar nodos repetidos
public boolean repetido(ArrayList<State> l, State s) {
    boolean exist = false;
    for (State ns : l) {
        if (ns.isEqual(s)) // Un método propio para compaarar estados
        {
            exist = true;
            break;
        }
    }
    return exist;
}
```

Clase State

En estas funciones vamos a iterar con un ciclo for el tablero para obtener el caracter de cada pieza del puzzle y cuando encuentre la pieza con un valor de 0, va a calcular sus coordenadas en i y en j para saber dónde se encuentra.

```
public State(String sts)
{
    int n=0;
    this.board = new int[4][4];
    for(int i=0;i<4;i++)
        for(int j=0;j<4;j++)
        {
            char c=sts.charAt(n);
            board[i][j]=Character.getNumericValue(c);
            if(board[i][j]==0)
            {
                posI = i;
                posJ = j;
            }
            n++;
        }
}
public State (int[][] board, char movement)
{
    this.movement = movement;
    this.board = new int[4][4];
    for(int i=0;i<4;i++)
        for(int j=0;j<4;j++)
        {
            this.board[i][j]=board[i][j];
            if(board[i][j]==0)
            {
                posI = i;
                posJ = j;
            }
        }
```

CLASE SWAP

Esta clase nos permite que una vez detectado un cambio de movimiento, va a intercambiar el elemento vacío del puzzle a la nueva casilla vacía, para hacerlo necesitamos conocer sus coordenadas.

```
public void swap(int i,int j)
{
    int aux=board[i][j];
    board[i][j]=0;
    board[posI][posJ]=aux;
    posI=i;
    posJ=j;
}
```

CLASE NEXTSTATES

Esta función se encarga de realizar los movimientos, dependiendo del caso, ya sea abajo, arriba, derecha, izquierda; va a llamar a la función swap y va a dar la ilusión que se está moviendo el tablero.

```
public ArrayList nextStates()
{
    ArrayList<State> next =  new ArrayList();
    int[][] newBoard = new int[4][4];

    // Clone board
    for(int i=0;i<4;i++)
        for(int j=0;j<4;j++)
            newBoard[i][j]=this.board[i][j];


    // Move up
    if(posI>0)
    {
        int newI = posI-1;
        State newState = new State(newBoard,'u');
        newState.swap(newI, posJ); //
        next.add(newState);
    }

    // Move down
    if(posI<2)
    {
        int newI = posI+1;
        State newState = new State(newBoard,'d');
        newState.swap(newI, posJ);
        next.add(newState);
    }

    // Move left
    if(posJ>0)
    {
        int newJ = posJ-1;
        State newState = new State(newBoard,'l');
        newState.swap(posI, newJ);
        next.add(newState);
    }

    // Mover right
    if(posJ<2)
    {
        int newJ = posJ+1;
        State newState =  new State(newBoard,'r');
        newState.swap(posI, newJ);
        next.add(newState);
    }

    return next;
}
```

CLASE GOALFUNCTION

Esta clase nos permite comprobar si el puzzle está resuelto, lo que hace es recorrer el tablero y va comprobando casilla por casilla si sus valores coinciden y si esto es verdadero cumplió su objetivo y si es falso continúa ejecutando el programa.

```
public boolean goalFunction(State goal)
{
    boolean success=true;
    int[][] goalBoard = goal.getBoard();
    for(int i=0;i<4;i++)
        for(int j=0;j<4;j++)
        {
            if(goalBoard[i][j]!=board[i][j])
            {
                success = false;
                break;
            }
        }
    return success;
}
```

Demostración de Funcionamiento

Cabe resaltar que utilizaremos dos estados iniciales diferentes para cada método de solución pues puede tender a complicarse o extenderse mucho la búsqueda por profundidad dependiendo de la solución encontrada.

Pantalla principal

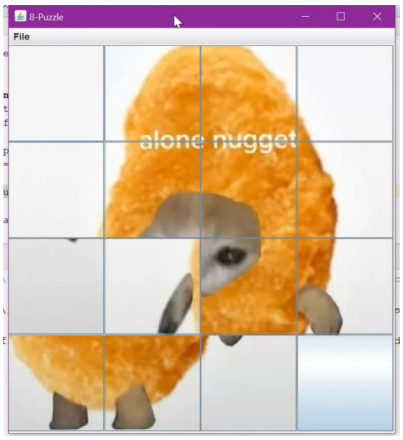


Ilustración 1. Pantalla de inicio búsqueda por anchura

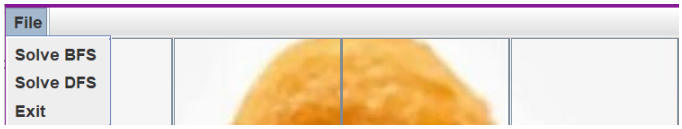


Ilustración 2. Menú de opciones

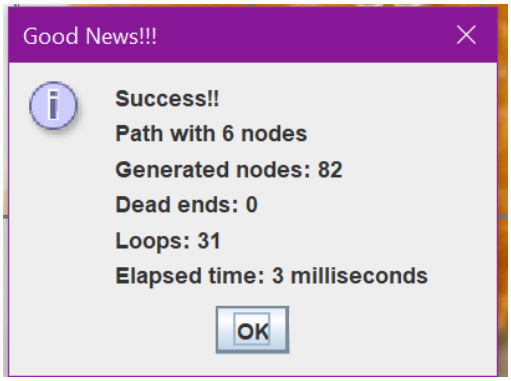


Ilustración 3. Mensaje de éxito

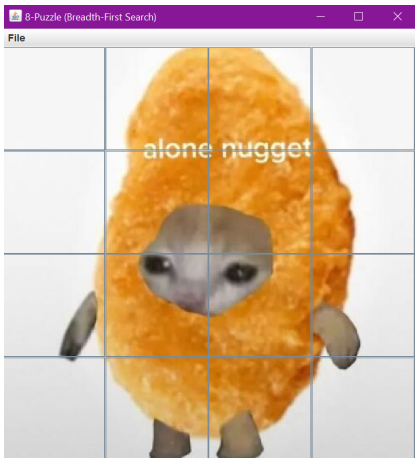


Ilustración 4. Puzzle resuelto

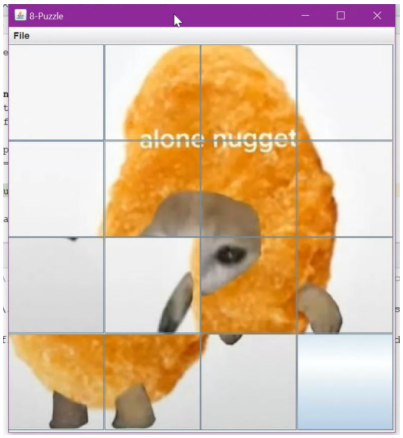


Ilustración 5. Pantalla inicial para búsqueda en profundidad

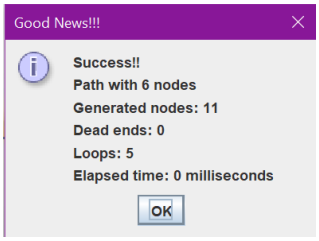


Ilustración 6. Mensaje de éxito

Ilustración 7. 

Puzzle resuelto

CONCLUSIÓN

Lo que aprendimos en esta práctica fue el funcionamiento del algoritmo de primero en anchura y el algoritmo en primero en profundidad, también notamos un gran problema con los algoritmos, ya que al hacer el puzzle más complicado la memoria de la computadora se va al límite por lo que podemos deducir que estos algoritmos aunque cumplen con su propósito no son los más óptimos para resolver el problema.