



UNIVERSIDAD AUTÓNOMA DE SINALOA
FACULTAD DE INFORMÁTICA CULIACÁN



Problema 8 Puzzle por Heurísticas (Desordenados y Distancia Manhattan)

Materia:

Inteligencia Artificial

Carrera:

Facultad de informática - UAS

#Lenguaje de Programación JAVA – Heurística Desordenados y Distancia Manhattan

```
package Puzzle;
/**
 * 8Puzzle (Heuristica Desordenados y Distancia Manhattan)
 */
//LinkedList objeto que nos permite impletar la cola donde se introducirán abiertos
import java.util.LinkedList;
//HashSet – Conjunto de elementos NO Repetidos que guarda a los elementos cerrados
import java.util.HashSet;
//FrameWork de Java que nos permite manipular grupos de objetos
//Utilizado para reordenar la cola de abiertos cada que se aplican acciones(Arriba, Abajo, Derecha, Izquierda)
import java.util.Collections;

//Clase Puzzle que contiene los metodos para realizar la evaluacion de las dos heurísticas
class Puzzle {
```

Metodo – Evaluar Desordenados

```
//Metodo Evaluar recibe los elementos NODO INICIAL y NODO FINAL pasado a traves
//del metodo principal main
public void Evaluar(int[][] nodoI, int[][] nodoF) {
    //Variable para manejar la HEURISTICA en enteros
    int heuristica;
    //Se instancia un nuevo Objeto Nodos como estadoFinal
    Nodos estadoFinal = new Nodos(nodoF);
    //Se obtiene la heurística para el estado Inicial
    heuristica=getHeuristica(nodoI, estadoFinal.estado);
    //Se instancia el nuevo objeto (estadoInicial) de tipo nodos con el Nodo Inicial y su heuristica
    Nodos estadoInicial = new Nodos(nodoI,heuristica,null);
    //caminoFinal utilizado para la Impresion de los nodos Analizados
    //NOTA: Cada NODO GENERADO GUARDA a su PADRE (estado actual en ese momento)
    Nodos caminoFinal = null;
    //Se instancia el objeto CERRADOS Vacio
    HashSet<Nodos> cerrados = new HashSet<>();
    //Arreglo que nos servira para guardar la posicion del Elemento que se Movera
    int[] posNuevoEl = new int[2];
    //Arreglo que nos permitira guardar la posicion donde esta el elemento Vacio
    int[] posVacio = new int[2];
    //Se instancia la cola que tendra los elementos de abiertos
    LinkedList<Nodos> cola = new LinkedList<>();
    //Se le agrega el estado Inicial
    cola.add(estadoInicial);
    //Ciclo que se repitira mientras COLA NO SEA VACIA
```

```

while (!cola.isEmpty()) {
    //Hacemos Actual el Nodo(estado) que se encuentra en el cabezal de la cola
    Nodos estadoActual = cola.poll();
    /**
     * Si la heuristica del Nodo Actual es igual a 0 entonces igualar la variable
     * Nodos que incluye a todos los Nodos Analizados a caminoFinal y mandar a imprimir
     */
    if (estadoActual.heuristica==0) {
        caminoFinal = estadoActual;
        System.out.println("Evaluacion por Heuristica Desordenados: ");
        System.out.println(caminoFinal);
        break;
    }
    //ACCIONES
    //ARRIBA
    /**
     * Obtenemos la posicion de 0 en X de la matriz del estadoActual
     * y restamos -1 para checar que estamos dentro de los limites de la matriz
     * y posteriormente intercambiar 0 por el numero superior a este
     */
    if (getPosicionX(0, estadoActual.estado) - 1 >= 0) {
        /**
         * matrizAux de tipo enteros nos sirve para duplicar la matriz
         * del estadoActual y asi evitar que se sobreescriba o alterar su valores
         */
        int[][] matrizAux = duplicar(estadoActual.estado);
        //Variable temp utilizada para guardar el valor a substituir por la posicion 0
        int temp;
        /**
         * Obtenemos la posicion o coordenada que esta arriba del 0 y la asignamos a el
         * arreglo posNuevoEl
         */
        posNuevoEl[0] = getPosicionX(0, estadoActual.estado) - 1;
        posNuevoEl[1] = getPosicionY(0, estadoActual.estado);
        /**
         * Obtenemos el numero que esta arriba de 0 en la matriz de estadoActual
         * Y lo igualamos a la variable temp
         */
        temp = getNum(posNuevoEl[0], posNuevoEl[1], estadoActual);
        /**
         * Obtenemos la posicion del Cero(Vacio) en la matriz de estadoActual
         * y obtenemos sus cordenadas (X y Y) igualandolas en el arreglo posVacio
         */
        getPosicion(posVacio, 0, estadoActual.estado);
        /**
         * Intercambiamos los valores, estableciendo cero en la posicion arriba de el
         * en la matrizAux e intercambiando la posicion anterior de 0 por el numero

```

```

        * que estaba arriba de este anteriormente
        */
        setNum(0, posNuevoEl[0], posNuevoEl[1], matrizAux);
        setNum(temp, posVacio[0], posVacio[1], matrizAux);
    /**
     * Obtenemos la heuristica de acuerdo a los numero desordenados en la matrizAux
     * comparando esta con la matriz del Nodo estadoFinal
     */
    heuristica = getHeuristica(matrizAux, estadoFinal.estado);
    /**
     * Creamos un nuevo Nodo llamado estadoSiguiente que guarda la matrizAux con la
     * accion ya aplicada, asi mismo le mandamos la HEURISTICA de ese Estado o Nodo
     * y su PADRE que es el estadoActual apartir del cual se genero este NODO
     */
    Nodos estadoSiguiente = new Nodos(matrizAux, heuristica, estadoActual);
    //Verificamos que el estado sea unico(que no este ni en ABIERTOS ni en CERRADOS)
    verificarEstadosUnicos(cerrados, estadoSiguiente, cola);
}
//ABAJO
/**
 * Se realiza un proceso similar con el resto de acciones cambiando unicamente
 * los numeros que se pretende intercambiar en el estadoActual
 */
if (getPosicionX(0, estadoActual.estado) + 1 <= 2) {
    int[][] matrizAux = duplicar(estadoActual.estado);

    int temp;

    posNuevoEl[0] = getPosicionX(0, estadoActual.estado) + 1;
    posNuevoEl[1] = getPosicionY(0, estadoActual.estado);

    temp = getNum(posNuevoEl[0], posNuevoEl[1], estadoActual);

    getPosicion(posVacio, 0, estadoActual.estado);

    setNum(0, posNuevoEl[0], posNuevoEl[1], matrizAux);
    setNum(temp, posVacio[0], posVacio[1], matrizAux);

    heuristica = getHeuristica(matrizAux, estadoFinal.estado);

    Nodos estadoSiguiente = new Nodos(matrizAux, heuristica, estadoActual);
    verificarEstadosUnicos(cerrados, estadoSiguiente, cola);
}
//DERECHA
if (getPosicionY(0, estadoActual.estado) + 1 <= 2) {
    int[][] matrizAux = duplicar(estadoActual.estado);

```

```

        int temp;

        posNuevoEl[0] = getPosicionX(0, estadoActual.estado);
        posNuevoEl[1] = getPosicionY(0, estadoActual.estado) + 1;

        temp = getNum(posNuevoEl[0], posNuevoEl[1], estadoActual);

        getPosicion(posVacio, 0, estadoActual.estado);

        setNum(0, posNuevoEl[0], posNuevoEl[1], matrizAux);
        setNum(temp, posVacio[0], posVacio[1], matrizAux);

        heuristica = getHeuristica(matrizAux, estadoFinal.estado);

        Nodos estadoSiguiente = new Nodos(matrizAux, heuristica, estadoActual);
        verificarEstadosUnicos(cerrados, estadoSiguiente, cola);
    }
    //Izquierda
    if (getPosicionY(0, estadoActual.estado) - 1 >= 0) {
        int[][] matrizAux = duplicar(estadoActual.estado);

        int temp;

        posNuevoEl[0] = getPosicionX(0, estadoActual.estado);
        posNuevoEl[1] = getPosicionY(0, estadoActual.estado) - 1;

        temp = getNum(posNuevoEl[0], posNuevoEl[1], estadoActual);

        getPosicion(posVacio, 0, estadoActual.estado);

        setNum(0, posNuevoEl[0], posNuevoEl[1], matrizAux);
        setNum(temp, posVacio[0], posVacio[1], matrizAux);

        heuristica = getHeuristica(matrizAux, estadoFinal.estado);

        Nodos estadoSiguiente = new Nodos(matrizAux, heuristica, estadoActual);
        verificarEstadosUnicos(cerrados, estadoSiguiente, cola);
    }
    //Se realiza un reordenamiento de los elementos de la cola
    //NOTA: vease el metodo toCompare sobreescrito de java para verificar el funcionamiento
    Collections.sort(cola);
}
}

```

Metodo - Evaluar Distancia Manhattan

```
/**
 * Casi identico al metodo anterior solo cambiando su heuristica por Distancia a Manhattan
 * Nota: vease metodo de la heuristicaManhattan dentro de esta misma clase
 */
public void EvaluarManhattan(int[][] nodoI, int[][] nodoF) {
    int heuristica;
    Nodos estadoFinal = new Nodos(nodoF);
    heuristica=getHeuristicaManhattan(nodoI, estadoFinal.estado);
    Nodos estadoInicial = new Nodos(nodoI,heuristica,null);
    Nodos caminoFinal = null;
    HashSet<Nodos> cerrados = new HashSet<>();

    int[] posNuevoEl = new int[2];
    int[] posVacio = new int[2];

    LinkedList<Nodos> cola = new LinkedList<>();

    cola.add(estadoInicial);

    while (!cola.isEmpty()) {
        Nodos estadoActual = cola.poll();

        if (estadoActual.heuristica==0){
            caminoFinal = estadoActual;
            System.out.println("Evaluacion por Heuristica Manhattan: ");
            System.out.println(caminoFinal);
            break;
        }

        //ACCIONES
        //ARRIBA
        if (getPosicionX(0,estadoActual.estado) - 1 >= 0) {
            int[][] matrizAux = duplicar(estadoActual.estado);

            int temp;

            posNuevoEl[0] = getPosicionX(0, estadoActual.estado) - 1;
            posNuevoEl[1] = getPosicionY(0, estadoActual.estado);

            temp = getNum(posNuevoEl[0], posNuevoEl[1], estadoActual);

            getPosicion(posVacio, 0, estadoActual.estado);

            setNum(0, posNuevoEl[0], posNuevoEl[1], matrizAux);
            setNum(temp, posVacio[0], posVacio[1], matrizAux);
        }
    }
}
```

```

    heuristica = getHeuristicaManhattan(matrizAux, estadoFinal.estado);

    Nodos estadoSiguiente = new Nodos(matrizAux, heuristica, estadoActual);
    verificarEstadosUnicos(cerrados, estadoSiguiente, cola);
}
//ABAJO
if (getPosicionX(0, estadoActual.estado) + 1 <= 2) {
    int[][] matrizAux = duplicar(estadoActual.estado);

    int temp;

    posNuevoEl[0] = getPosicionX(0, estadoActual.estado) + 1;
    posNuevoEl[1] = getPosicionY(0, estadoActual.estado);

    temp = getNum(posNuevoEl[0], posNuevoEl[1], estadoActual);

    getPosicion(posVacio, 0, estadoActual.estado);

    setNum(0, posNuevoEl[0], posNuevoEl[1], matrizAux);
    setNum(temp, posVacio[0], posVacio[1], matrizAux);

    heuristica = getHeuristicaManhattan(matrizAux, estadoFinal.estado);

    Nodos estadoSiguiente = new Nodos(matrizAux, heuristica, estadoActual);
    verificarEstadosUnicos(cerrados, estadoSiguiente, cola);
}
//DERECHA
if (getPosicionY(0, estadoActual.estado) + 1 <= 2) {
    int[][] matrizAux = duplicar(estadoActual.estado);

    int temp;

    posNuevoEl[0] = getPosicionX(0, estadoActual.estado);
    posNuevoEl[1] = getPosicionY(0, estadoActual.estado) + 1;

    temp = getNum(posNuevoEl[0], posNuevoEl[1], estadoActual);

    getPosicion(posVacio, 0, estadoActual.estado);

    setNum(0, posNuevoEl[0], posNuevoEl[1], matrizAux);
    setNum(temp, posVacio[0], posVacio[1], matrizAux);

    heuristica = getHeuristicaManhattan(matrizAux, estadoFinal.estado);

    Nodos estadoSiguiente = new Nodos(matrizAux, heuristica, estadoActual);

```

```

        verificarEstadosUnicos(cerrados, estadoSiguiente, cola);
    }
    //Izquierda
    if (getPosicionY(0, estadoActual.estado) - 1 >= 0) {
        int[][] matrizAux = duplicar(estadoActual.estado);

        int temp;

        posNuevoEl[0] = getPosicionX(0, estadoActual.estado);
        posNuevoEl[1] = getPosicionY(0, estadoActual.estado) - 1;

        temp = getNum(posNuevoEl[0], posNuevoEl[1], estadoActual);

        getPosicion(posVacio, 0, estadoActual.estado);

        setNum(0, posNuevoEl[0], posNuevoEl[1], matrizAux);
        setNum(temp, posVacio[0], posVacio[1], matrizAux);

        heuristica = getHeuristicaManhattan(matrizAux, estadoFinal.estado);

        Nodos estadoSiguiente = new Nodos(matrizAux, heuristica, estadoActual);
        verificarEstadosUnicos(cerrados, estadoSiguiente, cola);
    }
    Collections.sort(cola);
}

/**
 * Permite realizar una copia del estadoActual para realizar el intercambio de
 * operaciones con una nueva matriz duplicada o clonada y al final la retorna
 */
public int[][] duplicar(int[][] estadoActual) {
    //Instancia una nueva matriz llamada copia con el largo en x y y de la matriz estadoActual
    int[][] copia = new int[estadoActual.length][estadoActual.length];
    for (int x = 0; x <= 2; x++) {
        for (int y = 0; y <= 2; y++) {
            copia[x][y] = estadoActual[x][y];
        }
    }
    return copia;
}

//Obtiene el numero segun una posicion dada en el nodo proporcionado y lo retorna
public int getNum(int posX, int posY, Nodos estadoActual) {
    int temp = 0;
    temp = estadoActual.estado[posX][posY];
    return temp;
}

//Obtiene la posicion de un numero dada una matriz e igualandola a un arreglo dado en X y Y

```



```

public void getPosicion(int posXY[], int num, int[][] estado) {
    for (int x = 0; x < 3; x++) {
        for (int y = 0; y < 3; y++) {
            if (estado[x][y] == num) {
                posXY[0] = x;
                posXY[1] = y;
                break;
            }
        }
    }
}

//establece un numero en la posicion X y Y de la matriz proporcionada
public void setNum(int num, int x, int y, int[][] matriz) {
    matriz[x][y] = num;
}

```

Metodo - Heurística Desordenados

```

/**
 * Obtiene la heuristica de acuerdo a los elementos desordenados
 * comparando a la matrizAux dada con el estadoFinal
 */
public int getHeuristica(int[][] matrizAux, int[][] estadoFinal) {
    int heuristica = 0;
    for (int x = 0; x < 3; x++) {
        for (int y = 0; y < 3; y++) {
            if (matrizAux[x][y] != estadoFinal[x][y]) {
                if(matrizAux[x][y]==0){
                }
                else{
                    heuristica = heuristica + 1;
                }
            }
        }
    }
    return heuristica;
}

//Obtiene la posicion en X de un numero dado mediante una matriz
public int getPosicionX(int num, int[][] estado) {
    int posX = 0;
    for (int x = 0; x < 3; x++) {
        for (int y = 0; y < 3; y++) {
            if (estado[x][y] == num) {
                posX = x;
                break;
            }
        }
    }
}

```

```

    }
}
return posX;
}
//Obtiene la posicion en Y de un numero dado mediante una matriz
public int getPosicionY(int num, int[][] estado) {
    int posY = 0;
    for (int x = 0; x < 3; x++) {
        for (int y = 0; y < 3; y++) {
            if (estado[x][y] == num) {
                posY = y;
                break;
            }
        }
    }
    return posY;
}
}

```

Metodo Heurística - Distancia Manhattan

```

/**
 * Obtiene la heuristica de la distancia Manhattan de las matrices dadas
 * "matrizAux con la accion aplicada y recibiendo la matriz del estadoFinal"
 * Retornando un entero con la heuristica obtenida
 */
public int getHeuristicaManhattan(int[][] matrizAux, int[][] estadoFinal) {
    int heuristica = 0;
    //Dos arreglos para obtener los valores en (X,Y) de la estado Generado por accion y del estadoFinal
    int[] coordenadasMatrizAux = new int[2];
    int[] coordenadasEstadoFinal = new int[2];

    for (int x = 1; x <= 9; x++) {
        //Obtiene las posiciones en (X,Y) del estado Generado por accion recibiendo x como numero
        coordenadasMatrizAux[0] = getPosicionX(x,matrizAux);
        coordenadasMatrizAux[1] = getPosicionY(x,matrizAux);
        //Obtiene las posiciones en (X,Y) del estadoFinal recibiendo x como numero
        coordenadasEstadoFinal[0] = getPosicionX(x,estadoFinal);
        coordenadasEstadoFinal[1] = getPosicionY(x,estadoFinal);
    }
    /**
     * Obtenemos los numeros absolutos (enteros positivos) de las posiciones reespectivas
     * y los sumamos a los valores preeviamente obtenidos en la variable heuristica local
     * Ejemplo:
     * matriz(estadoActual)   matriz(estado Generado)   matriz(estadoFinal)
     * 2 8 3                   2 8 3                   1 2 3
     * 1 6 4   H=5             1 0 4   H=4               8 0 4   H=0
     */
}

```

```

* 7 0 5          7 6 5          7 6 5
*
*      8 esta en cord (0,1)      La posicion deseada de 8 es (1,0)
* Realizando la siguiente operacion seria:
* Math.abs(0-1)+(1-0) el valor absoluto nos regresaria un (1) + (1)
* siendo 2 la distancia de este elemento respecto a la posicion del estado Generado por accion
* la suma del resto de estas operaciones con el resto de numeros-
* de la matriz da la heuristica Final
*/
heuristica = heuristica + (Math.abs(coordenadasMatrizAux[0]-coordenadasEstadoFinal[0])+
                          Math.abs(coordenadasMatrizAux[1]-coordenadasEstadoFinal[1]));
}
return heuristica;
}
//Metodo que verifica que los elementoEvaluar o Siguiende no este en cerrados ni en abiertos
//Si no se encuentra en ambos lo agrega a la cola de abiertos
void verificarEstadosUnicos(HashSet<Nodos> cerrados, Nodos estadoEvaluar, LinkedList<Nodos> cola) {
    if (!cerrados.contains(estadoEvaluar) && !cola.contains(estadoEvaluar)) {
        cola.add(estadoEvaluar);
    }
}
}

```

Clase Nodos

```

package Puzzle;
/**
 * Clase Nodos que implementa la clase Comparable de Java para reordenar
 * los elementos de la lista en la cola de abiertos
 */
class Nodos implements Comparable<Nodos>{
    //Se declara la matriz bidimensional de un tamaño maximo de 2
    int estado[][] = new int[2][2];
    //Variable Global para Heuristica
    int heuristica;
    //Objeto de tipo Nodos Padre
    Nodos padre;
    //Metodo Constructor que recibe la matriz
    public Nodos(int[][] estado){
        this.estado=estado;
    }
    //Metodo Sobreescrito del Constructor que recibe tambien la heuristica y al padre
    public Nodos(int[][] estado, int heuristica, Nodos padre){
        this.estado=estado;
        this.heuristica= heuristica;
        this.padre= padre;
    }
}

```

```

//Metodo sobre escrito de Java para implementar la impresion de las matrices
@Override
public String toString() {
    StringBuilder constructor = new StringBuilder();
    if (padre != null) {
        constructor.append(padre);
    }
    for(int x=0; x<=2; x++){
        for(int y=0; y<=2; y++){
            constructor.append(estado[x][y]).append(" ");
        }
        if(x==1){
            constructor.append("    H=").append(heuristica);
        }
        constructor.append("\n");
    }
    constructor.append("\n");
    return constructor.toString();
}
//Metodo Sobreescrito la clase Comparable que ordena los elementos
//de la cola de abiertos de menor a mayor segun su heuristica
@Override
public int compareTo(Nodos t) {
    if(this.heuristica<t.heuristica){
        return -1;
    }
    else if(this.heuristica>t.heuristica){
        return 1;
    }
    return 0;
}
}

```

Metodo Main

```

//Metodo main
public static void main(String[] args) {
    int eInicial[][] = {{2,8,3},{1,6,4},{7,0,5}};
    int eFinal[][] = {{1,2,3},{8,0,4},{7,6,5}};
    //Metodo para Evaluar por Heuristica desordenados
    new Puzzle().Evaluar(eInicial, eFinal);
    //Metodo para Evaluar por Heuristica Manhattan
    new Puzzle().EvaluarManhattan(eInicial, eFinal);
}
}

```

```
Output - 8Puzzle (run)
run:
Evaluacion por Heuristica Desordenados:
2 8 3
1 6 4    H=4
7 0 5

2 8 3
1 0 4    H=3
7 6 5

2 0 3
1 8 4    H=3
7 6 5

0 2 3
1 8 4    H=2
7 6 5

1 2 3
0 8 4    H=1
7 6 5

1 2 3
8 0 4    H=0
7 6 5
```

```
Output - 8Puzzle (run)
Evaluacion por Heuristica Manhattan:
2 8 3
1 6 4    H=5
7 0 5

2 8 3
1 0 4    H=4
7 6 5

2 0 3
1 8 4    H=3
7 6 5

0 2 3
1 8 4    H=2
7 6 5

1 2 3
0 8 4    H=1
7 6 5

1 2 3
8 0 4    H=0
7 6 5
```

Salidas: