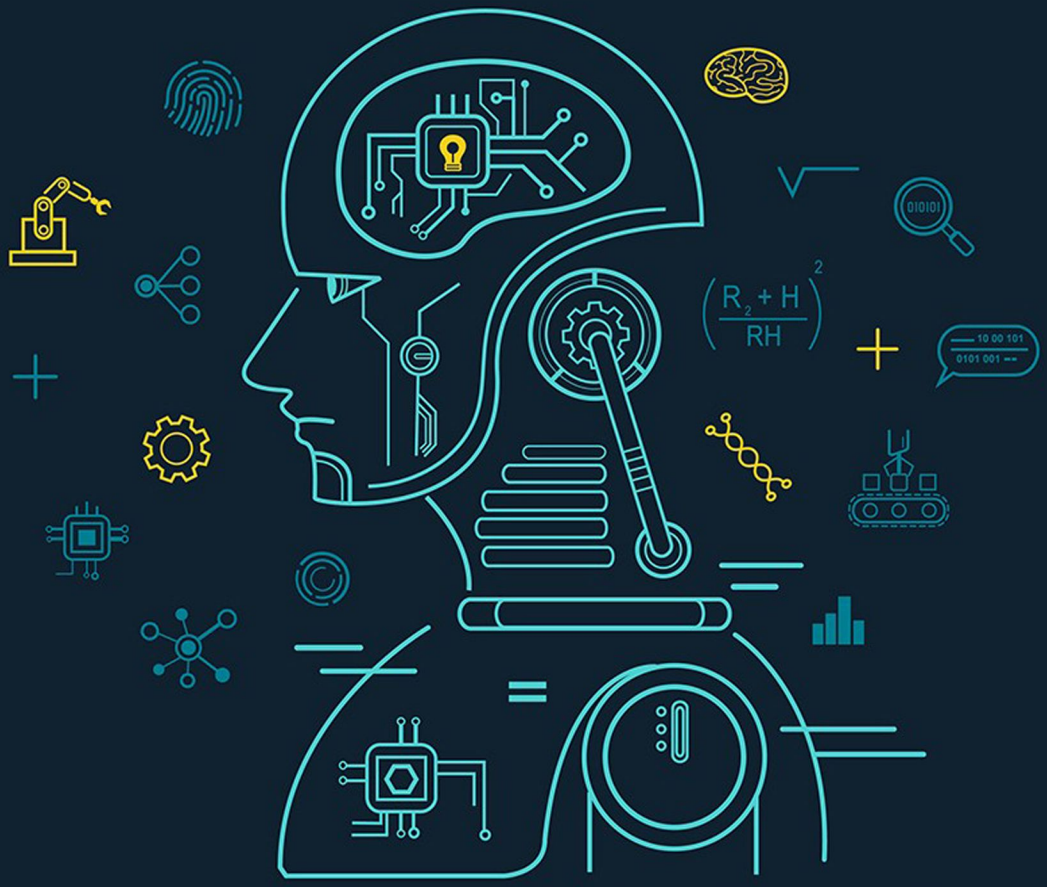
RUBÉN MARÍN LUCAS
PEDRO MIGUEL CARMONA



PROYECTO IASI 14-PUZZLE

CURSO 2019/2020

INDICE

PROYECTO DE LA ASIGNATURA DE INTELIGENCIA ARTIFICIAL Y SISTEMAS INTELIGENTES 2019/2020

ALUMNOS: (MÁXIMO 3 ALUMNOS)

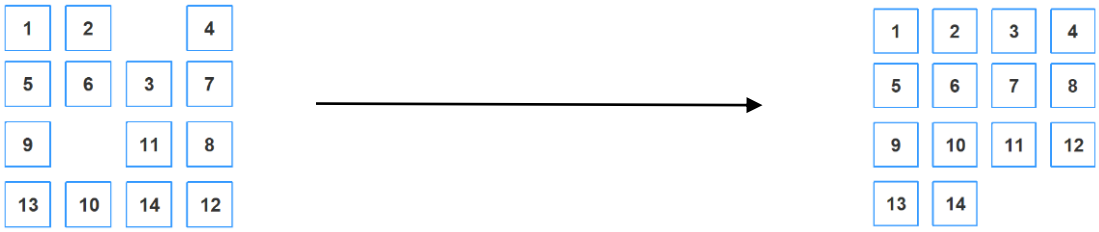
- Pedro Miguel Carmona Broncano
- Rubén Marín Lucas

1. RESUMEN DEL PROBLEMA DE LA PRÁCTICA:

Extrayendo del enunciado proporcionado, nos queda el siguiente resumen:

"Se pide diseñar e implementar un sistema relacionado con los movimientos en un tablero de 4 x 4, que se da como parámetro de entrada, y conseguir como salida un tablero ordenado de números. Se ***deben utilizar técnicas de búsquedas*** *de forma que proporcione una* ***solución***, ***consistente*** *en* ***indicar*** *la* ***secuencia de acciones*** *(movimientos) desde el tablero origen al tablero destino. Se parte desde el supuesto de que* ***se dispone*** *de* ***muy poco tiempo*** *para encontrar la solución, por lo que interesa utilizar búsquedas con heurísticas, y se quiere encontrar una solución que* ***no tenga muchos movimientos***."

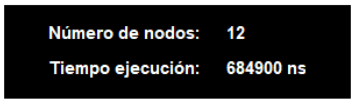
El objetivo de forma gráfica:



UNA POSIBLE SOLUCION:

3N 10N 7O 14O 8N 12N

Además, se pide los datos del número de nodos y el tiempo que ha tardado el algoritmo usado:

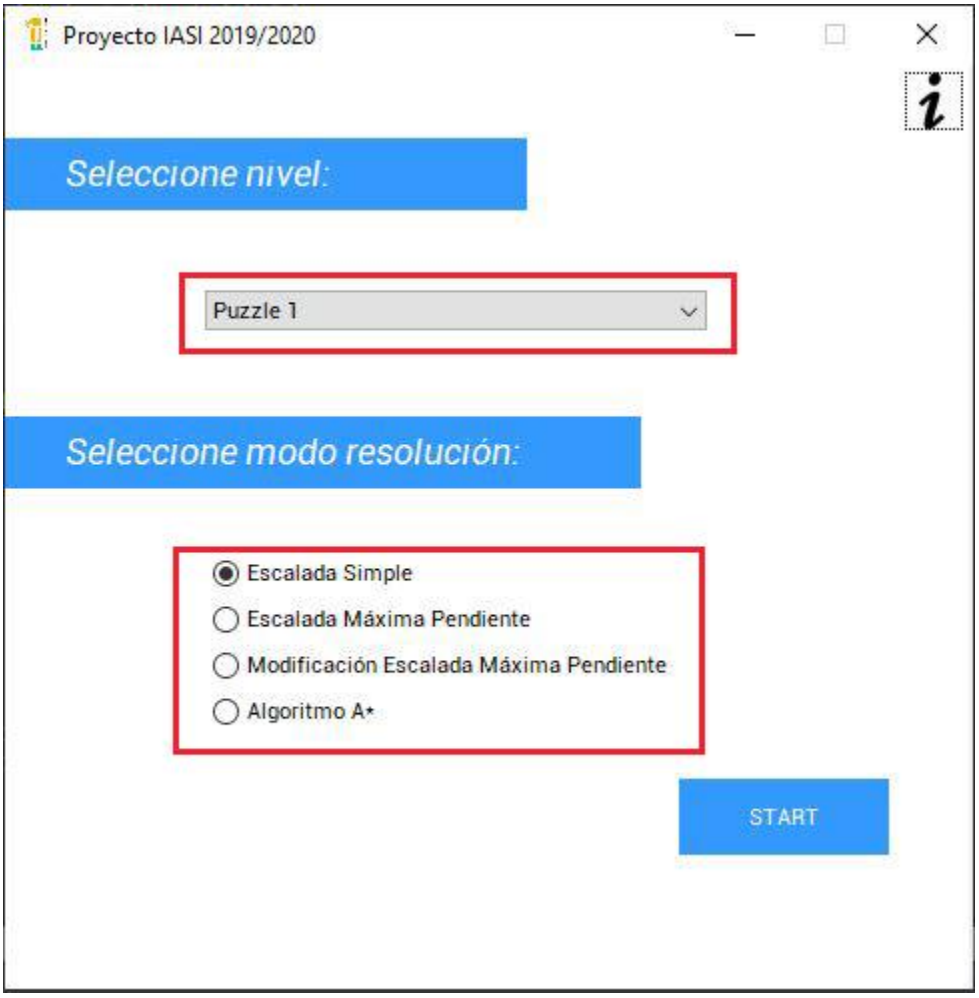


2. DESCRIPCIÓN DE LA INSTALACIÓN, INTERFAZ Y MANEJO DE LA PRÁCTICA

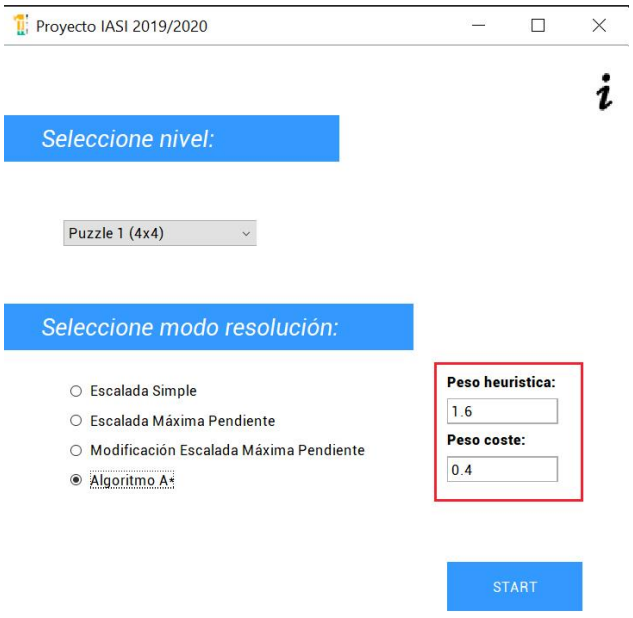
Lo requisitos para ejecutar la interfaz son:

- Versión del JDK de java 11 o superior
- Entorno de programación para ejecutar la interfaz, nosotros hemos utilizado eclipse, pero cualquiera que pueda ejecutar java vale.

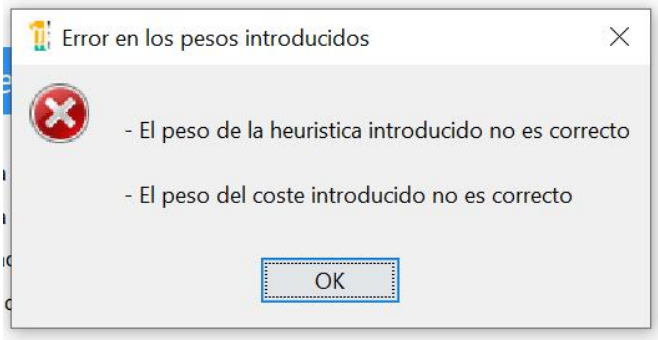
La interfaz se compone de una pantalla principal que te da la opción para elegir el puzzle que se desea resolver, así como por que método de resolución se desea resolver este:



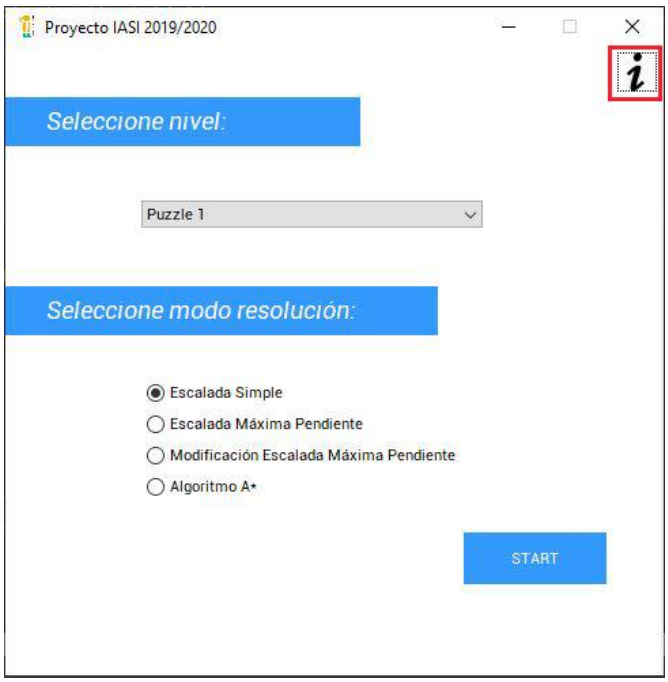
Si seleccionamos *Algoritmo A** se nos aparecerá dos campos para indicar los pesos, tanto de la heurística, como del coste.



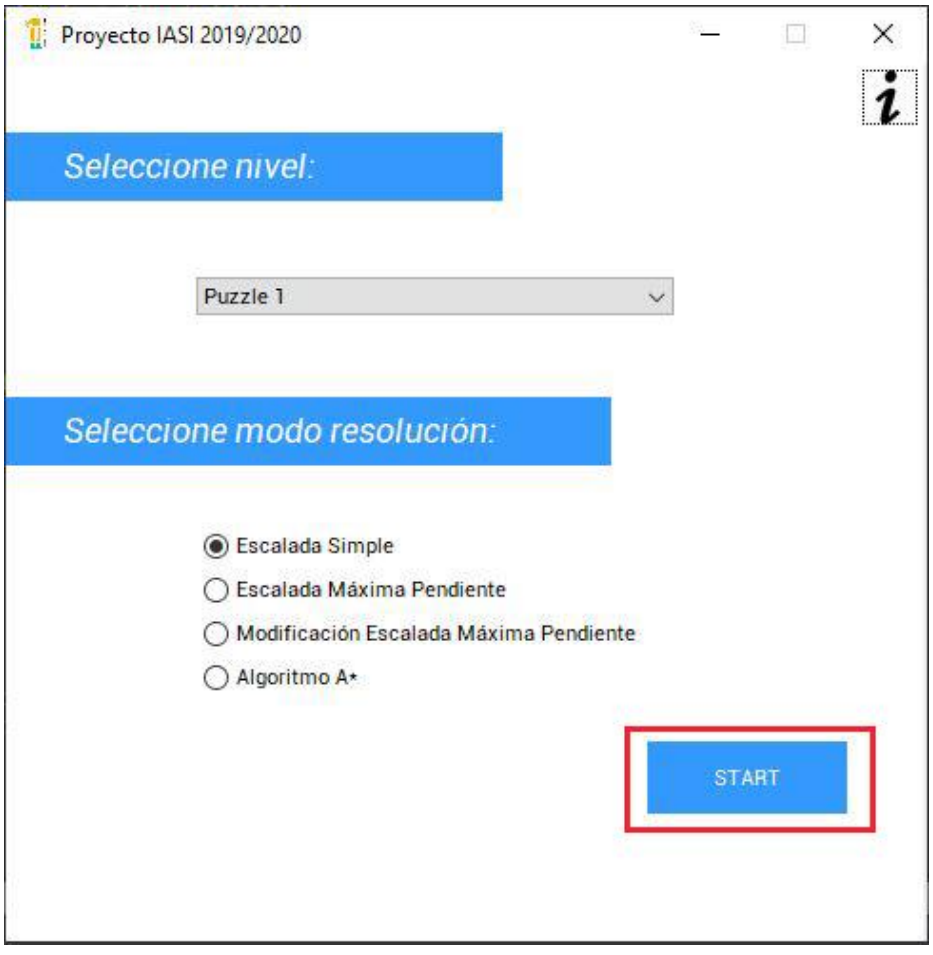
Si metemos un carácter que no sea un número o el punto decimal se bloqueará el campo. Y si metemos un número mal formado nos aparecerá un error informando en que campos hemos fallado.



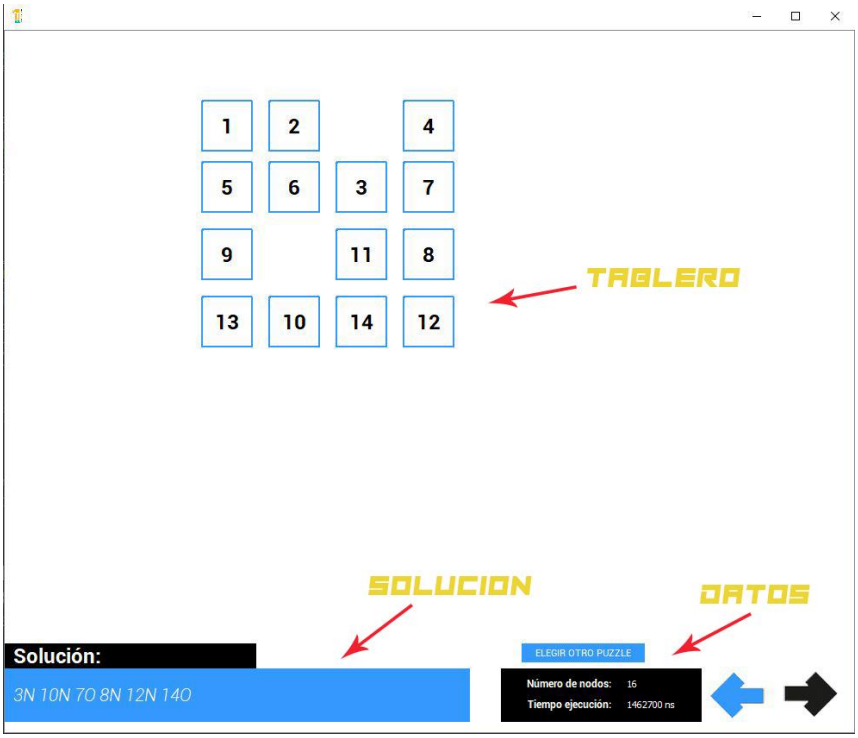
En esta misma ventana si le damos al botón de información *i* se nos abrirá otra ventana que nos proporciona información de los autores y de las ampliaciones realizadas:



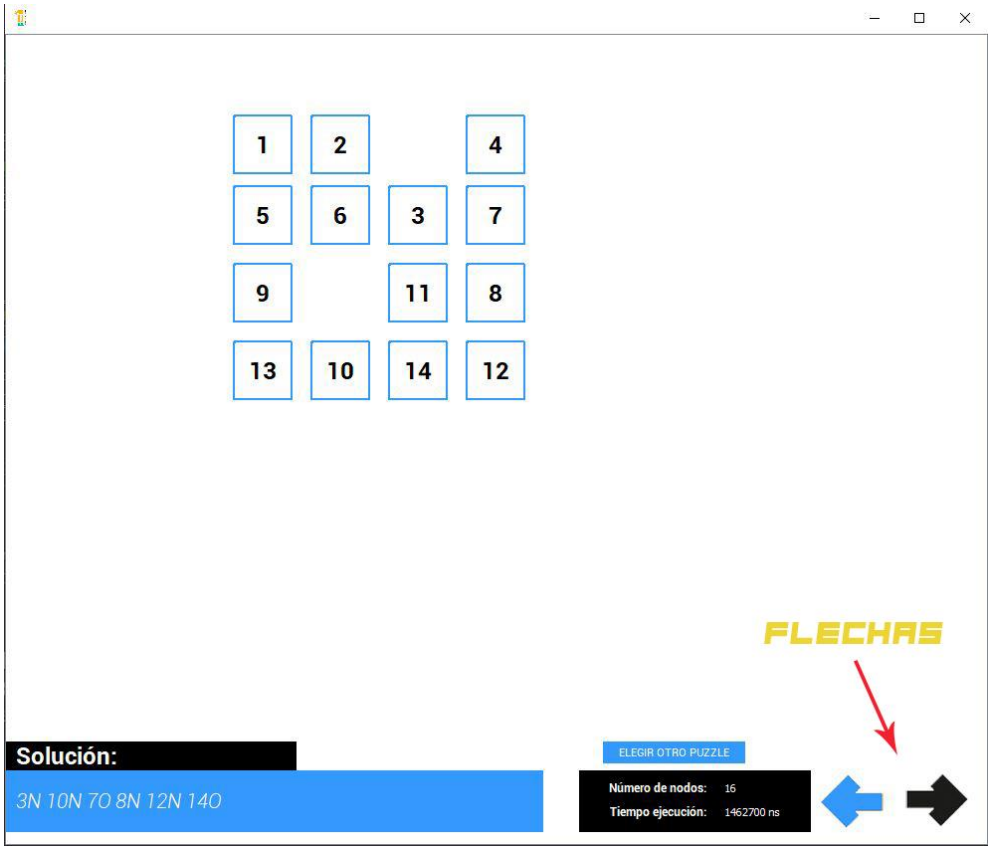
Una vez elegido el puzzle a resolver y el método de resolución para resolverlo le daremos al botón que dice START para que empiece a resolverlo:

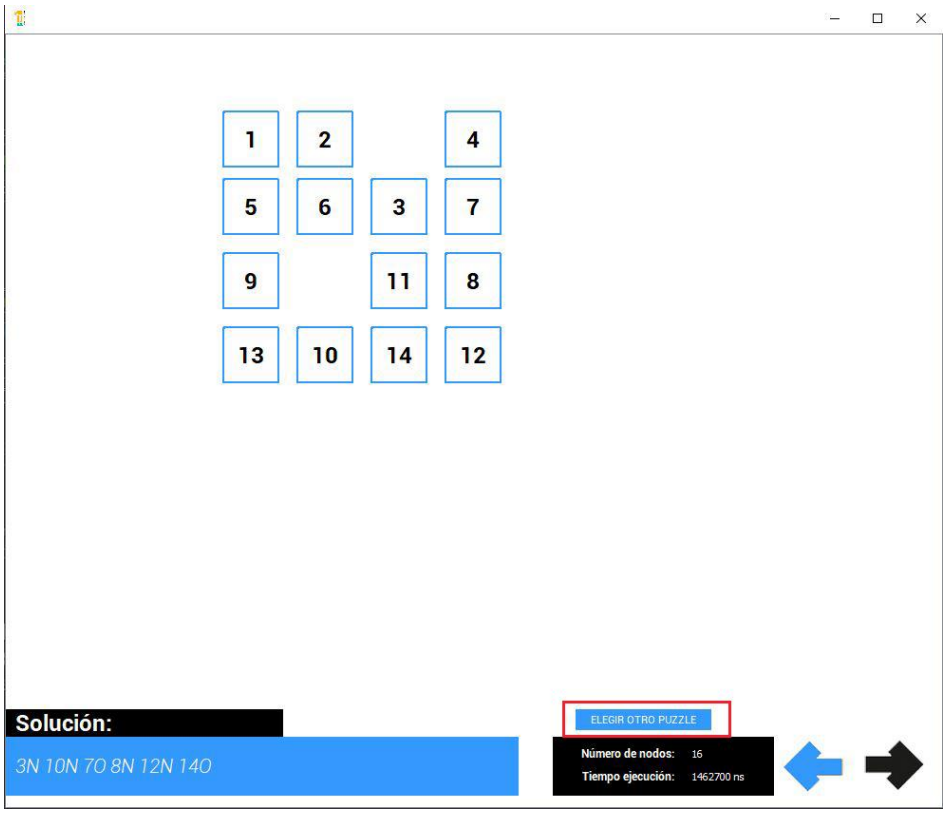


Se nos abrirá otra ventana que nos mostrará un tablero del tamaño del tablero del puzzle con el estado inicial de este, la solución que se ha calculado previamente, información de los nodos generados, así como el tiempo que ha tardado el algoritmo en encontrar una solución:



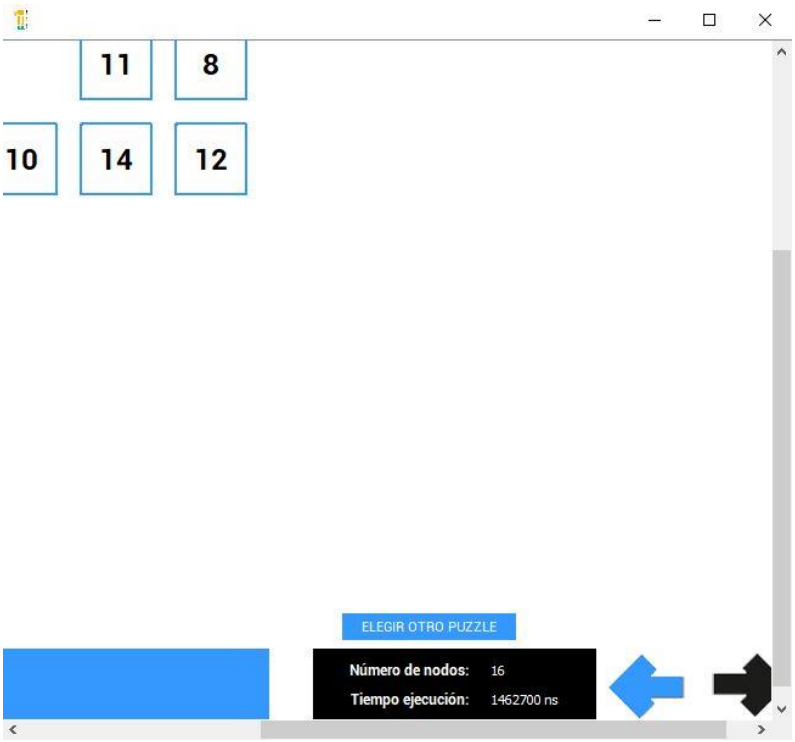
También se dispone de unas flechas para que el usuario pueda retroceder o avanzar, para ver el movimiento de las casillas.





También tenemos el botón *ELEGIR OTRO NIVEL* que al pulsarlo volveríamos a la pantalla inicial.

Por ultimo para todas las pantallas se ha diseñado con el objeto *JScrollPane* que proporciona el paquete Swing de Java para que el usuario pueda mover la ventana en caso de que el diseño no se ajuste a su pantalla. Solo tienes que situar el ratón en el borde de la ventana y arrastrarla para cambiarla de tamaño.



DESCRIPCIÓN DE LA CLASE *INIT_DATA*

Los parámetros de la clase son:

- g (GestorSolucion): lo utilizaremos para acumular el tamaño del tablero y copiar el puzzle del fichero correspondiente.

- Puzzle (String): indica el tablero elegido por el usuario.

Se utiliza la clase *Scanner* para abrir el puzzle que se pasa por parámetro. Se va leyendo el fichero por líneas y se trocea con la función *Split* para luego ir pasando valor a valor al tablero de la clase GestorSolucion.

Todo esto envuelto en un bloque Try catch para cazar la excepción en caso de no encontrar el archivo.

Hemos exportado el proyecto en forma de .JAR y .EXE, se puede encontrar en la raíz de la carpeta proyecto (Si se ejecuta el .jar en ubuntu y no funciona, por favor ejecutarlo desde Eclipse o desde el programa que se utilice para ejecutarlo)

Pedimos perdón de antemano si hay algún fallo gráfico, ya que en este proyecto no se tenía en cuenta la interfaz, nos hemos sacrificado por hacerla para aprender sobre interfaces gráficas en Java y es nuestra primera vez haciendo una.

3. Descripción de los algoritmos de resolución usados en la solución

Todos los algoritmos implementados como se nos dice en el enunciado pertenecen a la categoría de algoritmos de búsqueda heurística, que tienen una serie de características comunes:

- No garantizan que se encuentre una solución, aunque existan soluciones.
- Si encuentran una solución, no se asegura que ésta tenga las mejores propiedades (que sea de longitud mínima o de coste óptimo).
- En algunas ocasiones (que en general, no se podrán determinar a priori) encontrarán una solución (aceptablemente buena) en un tiempo razonable.

Sobre esta base partimos de una serie de algoritmos como son:

- Generación y prueba.
- Métodos de escalada (simple, máxima pendiente, enfriamiento simulado).
- Búsqueda del primero mejor.
- Algoritmo A*.

De ellos se han escogido 2 algoritmos de escalada: simple, y máxima pendiente, y el algoritmo A*, cuyos principios se explican a continuación.

Desde un primer momento descartamos la implementación del algoritmo de generación de prueba puesto que opinamos que depende demasiado de la aleatoriedad, y que no te asegura la resolución de ninguno de los puzzles, aunque tampoco otros

algoritmos te aseguran una solución segura, pero preferíamos realizar un algoritmo que tuviera un mayor grado de certeza a la hora de la resolución de un puzzle. El algoritmo de enfriamiento simulado fue descartado por el hecho de realizar algún algoritmo que no fuera de esclada. Por último el algoritmo de búsqueda del primero mejor fue descartado porque preferíamos hacer el algoritmo A* que parecía un mayor reto al no tener en cuenta solo la heurística, sino también el coste, haciendo que las solcuiones halladas por lo f¡general sean mejores, es decir, necesiten menos movimientos para llegar al estado final.

3.1. ESCALADA SIMPLE

En general los algoritmos de escalada exploran el camino buscando un estado mejor que el actual (que tengan un mejor h'), y siguen una estrategia de control irrevocable, es decir, una vez escogido un estado, no tienen en cuenta las anteriores posibilidades. Si desde un estado, todas las posibilidades son peores que el actual estado, el algoritmo para.

En concreto la escalada simple se caracteriza por escoger el primer estado que mejore la función heurística actual. En el caso de que desde un estado no se encontrara ningún estado mejor (con mejor heurística), el algoritmo para.

Suele dar soluciones rápidas, aunque no aseguran optimalidad del coste, también es fácil que pare puesto que al guiarse solo de h' que no deja de ser una función estadística, es bastante probable que se encuentre en algún estado en el que no mejore su h'.

3.2. ESCALADA MÁXIMA PENDIENTE

Ya se han mencionado los principios de los algoritmos de escalada, por lo que nos disponemos a describir las peculiaridades de la escalada de máxima pendiente.

Desde un estado se aplican todos los operadores para pasar a un nuevo estado, que será aquel que tenga mejor h', teniendo en cuenta que esa h' también tiene que ser mejor que la del estado actual, en caso contrario pararía.

Suele tardar más que el algoritmo anterior, pero a la vez puede asegurar mayor certeza de resolución.

3.3. ALGORITMO A*

Se trata de un algoritmo que tiene en cuenta de forma proporcional a la vez los costes y las distancias al objetivo(h').

La idea principal es tener en cuenta dos valores a la vez: la estimación de lo cerca que estamos del objetivo (h'), y el coste de operadores (longitud del camino) que se han empleado en llegar a ese estado (g).

La función f' será la función que tenga en cuenta la estimación de la cercanía del objetivo y los costes de llegar al nodo. ($f'(n) = c0 * g(n) + c1 * h'(n)$)

El algoritmo trabaja con dos listas formadas por nodos, donde se almacena un puntero al nodo padre y el valor de la f' cuando se expandió el nodo:

- ABIERTOS: Nodos que se han generado, pero que aún no han sido expandidos. Es en realidad una cola con prioridad, donde los elementos de mayor prioridad son los que tienen una mejor f'.
- CERRADOS: Nodos que ya se han expandido. Se mantienen en memoria para evitar que, si varios nodos tienen el mismo sucesor, se pueda expandir un mismo nodo varias veces.

El algoritmo expande el nodo mejor de la lista de ABIERTOS y lo pasa a CERRADOS. Inserta los nodos expandidos en la lista de ABIERTOS, comprobando que no existan en las dos listas.

Si en el proceso de expansión se encuentra un nodo ya explorado situado en ABIERTOS o CERRADOS, debe compararse su f' actual con la f' almacenada. Si f' actual es mejor que f' almacenada, debe actualizarse la f' y el puntero al nuevo nodo padre. Además, deben recorrerse todos los nodos de las listas de ABIERTOS y CERRADOS actualizando los descendientes del nodo actualizado con nuevos valores de f'. Luego debe reordenarse la lista de ABIERTOS.

4. Descripción de las soluciones seguidas para resolver el problema

En primer lugar, cabe destacar que la resolución de todos los algoritmos recae en 2 clases:

- En primer lugar, explicamos lo principal de la clase *Tablero*:

En esta clase recae toda aquella funcionalidad que tiene que ver con la generación de movimientos aleatorios principalmente. Podemos empezar comentando sus atributos.

La constante *MAX_TAM* indica el tamaño máximo que puede admitir la matriz y *MAX_MOVIMIENTOS* guarda el número máximo de movimientos que puede realizar una casilla. Después nos encontramos con los atributos *matrizTablero[][]* que

representa como su nombre indica la matriz por la que queda definido el tablero; *tamTablero* que indica el tamaño que tiene la matriz; *nulos* que se trata de un ArrayList de Nulos (sirve para guardar la posición de cada 0 en el tablero); *padre* que se trata de un puntero de tipo Tablero, sirve para ir guardando la solución; *movimiento* que se trata de una cadena que indica el movimiento que se ha hecho para pasar del tablero al que apunta padre al actual; y *coste*, *h* y *función*, que almacenan los valores de coste, función heurística y función (combinación de coste y heurística) respectivamente.

Quedan señalados los métodos usados para generar los movimientos en cada uno de los algoritmos implementados, pero éstos los explicaremos más adelante. Ahora nos centraremos en explicar como funcionan una serie de métodos usados en todos estos algoritmos para generar movimientos aleatorios, entre otros:

1) Primero cabe destacar los métodos relacionados con los movimientos: A partir de un *Nulo*, comprueba si se puede realizar el movimiento, y en el caso afirmativo lo realiza. Están definidos pensando en el movimiento que realiza el 0, es decir, *moverNorte(Nulo n),* por ejemplo lo que hace es mover hacia arriba el 0 representado por el Nulo n.

2) También debemos destacar los métodos empleados para generar movimientos aleatorios, generarMovimientos() y generarMovNulos(). Ambos se basan en el uso de un ArrayList en el que se añaden los valores desde 0 hasta el máximo de movimientos, o el número de Nulos respectivamente. Después los 2 algoritmos usan el método shuffle heredado de la interfaz Collections para ordenar de forma aleatoria dichos valores. Esto se realiza con el objetivo de que en los algoritmos de generación de movimientos si vamos obteniendo la primera posición de cualquiera de los 2 ArrayList empleados estemos obteniendo valores aleatorios, puesto que el orden de dichos valores es aleatorio.

Se muestra un ejemplo, para mejor comprensión:

generarMovimientos() → 1º | 0 | 1 | 2 | 3 | 2º (shuffle) | 1 | 3 | 0 | 2 |

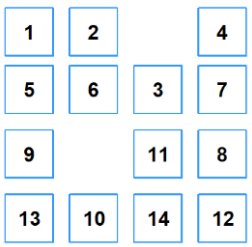
*Cada número representa un movimiento : 0 → , 1→ , 2 → , 3→

generarMovNulos() → 1º | n1 | n2 | n3 | 2º (shuffle) | n2 | n1 | n3 |

*Caso de que tuvieramos 3 nulos

3) Por último, tenemos que señalar las funciones heurísticas empleadas: FuncionHeuristica1() → Suma de las distancias a la que se encuentran cada una de las casillas de su posición. Esta función se corresponde con la llamada *Distancia Manhattan*.

Ejemplo:



Hay 6 piezas fuera de su posición:

3 → distancia = 1
7 → distancia = 1
8 → distancia = 1
10 → distancia = 1
12 → distancia = 1
14 → distancia = 1

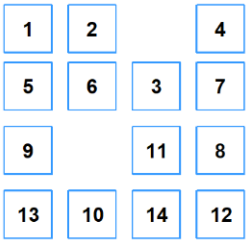
h' = 1+1+1+1+1+1 = 6

*La distancia se mide en función de los movimientos que hay que realizar, es decir si hay que realizar 2 movimientos para poner la casilla en su sitio, distancia = 2, y así para el resto de casos.

FuncionHeuristica2() → Número de casillas en su posición

Ejemplo:

En



Hay 6 piezas que no están en su sitio, por tanto h' = 6

segundo lugar, explicamos lo principal de la clase GestorSolucion:

En esta clase se implementan los algoritmos propiamente dichos, y recoge los datos que se piden a parte de solucionar el puzzle, como el almacenamiento del tiempo, o del conjunto de movimientos en una cadena. El sentido de esta clase es hacer un nexo entre la clase *Tablero* y la salida a través de la interfaz gráfica.

Para este cometido contiene los atributos siguientes: *t,* el tablero que representa el estado inicial de nuestros algoritmos; *numNodos* contiene el valor del número de nodos generados para la resolución del problema; *tiempoEjecucion* guarda el tiempo transcurrido en la ejecución de los algoritmos implementados; *solucionFinal* se trata de la cadena que contiene el conjunto de movimientos que forman parte dela solución; el resto de atributos son utilidades para la resolución de ciertos algoritmos, en concreto para A* (*abiertos* y *cerrados*) y para la modificación de Escalada de máxima pendiente (*nodoIgual*, *nodoPeor*).

Los algoritmos señalados son los que explicaremos más detalladamente en los próximos subapartados.

4.1. ESCALADA SIMPLE

Como ya hemos introducido anteriormente el algoritmo se resume en 2 métodos. En primer lugar toca comentar el realizado en la clase *Tablero,* **MejorMovientoES(Tablero mejor, GestorSolucion g)**, que en resumidas cuentas de lo que se encarga es de realizar **aleatoriamente movimientos** sobre el **tablero** en el que nos encontramos actualmente (**this**), y en el caso de que **encuentre** un **tablero** cuya **heurística** sea **mejor** que la anterior termina el proceso y **retorna** ese **tablero** para guardarlo en la solución. Exponemos un pseudocódigo para mejor comprensión:

```
boolean enc = false; Tablero aux;
int mejorh = this.funcionHeuristica();

Conjunto <Integer> nulo = generarMovNulos(); //ya se explicó como funciona

	while(!nulo.isEmpty() && !enc) {

		Conjunto <Integer> mov = generarMovimientos(); //ya se explicó como funciona

		Integer num = nulo.get(0); //Extraemos el nulo para realizar el movimiento

			while(!mov.isEmpty() && !enc) {

				switch(movimiento) { //Ya explicamos como funcionaban los metodos de
                                     realizar movimientos
					case 0:
						if(aux.moverNorte(n)) g.addNodos();//gestor cuenta
                                                           nodo generado
						break;

					case 1:
						if(aux.moverEste(n)) g.addNodos();
						break;

					case 2:
						if(aux.moverOeste(n)) g.addNodos();
						break;

					case 3:
						if(aux.moverSur(n)) g.addNodos();
						break;

				}

				int h = aux.funcionHeuristica();

				if(h < mejorh) {//Si la heurística es mejor
					enc = true;//encontramos un nodo mejor, para de generar
					mejor = aux;
				}
			mov.remove(0);//ya hemos relizado ese movimiento
			}
	nulo.remove(0);//Ya hemos relizado todos los movimientos con ese nulo
	}

if(!enc) mejor =  null;

return mejor;
```

Este método es llamado desde la clase gestor, desde el método **escaladaSimple()**, que resumidamente lo que hace **llamar** en bucle a ***MejorMovientoES****(Tablero mejor, GestorSolucion g)*, este bucle **acabará** cuando este método **devuelva null**, es decir, **no** se haya **encontrado** un **tablero** con **mejor heurística**, **o** cuando el **tablero encontrado** sea el **estado final**, en otras palabras, cuando el valor de la heurítica sea el deseado (0 si es minimizante, o el valor máximo si es maximizante). Exponemos un pseudocódigo para mejor comprensión:

```
long initTime = tiempoInicial();


boolean fin = false; Tablero nuevo; Tablero actual = t;//Obtenemos el tablero inicial

		while(nuevo != null && !fin) {

			nuevo = actual.MejorMovimientoES(nuevo, this);

			if(nuevo != null) {

				nuevo.setPadre(actual);//Establecemos puntero, para guardarlo para
				                       solucion
				actual = nuevo;//Actualizamos
				String solucion += actual.getMovimiento() + " ";//Añadimos el
				                   movimiento a la cadena de solucion
				if(actual.getHeuristica() == 0) fin = true;//Si es ultimo estado
			}


		}

		long endTime = tiempoFinal();
		long tiempo = endTime - initTime;//Tiempo que se tarda en ejecutar algoritmo

return fin;
```

4.2. ESCALADA MÁXIMA PENDIENTE

Primero debemos comentar el método realizado en la clase realizado en la clase *Tablero,* **MejorMovientoEMP(Tablero mejor, GestorSolucion g)**, el cual se encarga de realizar **movimientos aleatorios** sobre el **tablero** en el que nos encontramos actualmente (**this**), y en el caso de que **encuentre** un **tablero** cuya **heurística** sea **mejor** que la anterior **actualiza** el tablero **actual**, pero no finaliza la generación de nodos (como sí pasaba en el anterior algoritmo). Exponemos un pseudocódigo para mejor comprensión:

```
boolean enc = false; Tablero aux;
int mejorh = this.FuncionHeuristica();

Conjunto <Integer> nulo = generarMovNulos(); //ya se explicó como funciona

	while(!nulo.isEmpty()) {

		Conjunto <Integer> mov = generarMovimientos(); //ya se explicó como funciona

		Integer num = nulo.get(0); //Extraemos el nulo para realizar el movimiento

			while(!mov.isEmpty()) {

				switch(movimiento) { //Ya explicamos como funcionaban los metodos de
				                     realizar movimientos
					case 0:
```

```
                                        if(aux.moverNorte(n)) g.addNodos();//gestor cuenta
                                                                            nodo generado
                                        break;

                                    case 1:
                                        if(aux.moverEste(n)) g.addNodos();
                                        break;

                                    case 2:
                                        if(aux.moverOeste(n)) g.addNodos();
                                        break;

                                    case 3:
                                        if(aux.moverSur(n)) g.addNodos();
                                        break;

                                }

                                int h = aux.funcionHeuristica();

                                if(h < mejorh) {//Si la heurística es mejor
                                    enc = true;//encontramos un nodo mejor
                                    mejor = aux;
                                }
                            mov.remove(0);//ya hemos relizado ese movimiento
                        }
                nulo.remove(0);//Ya hemos relizado todos los movimientos con ese nulo
            }

if(!enc) mejor =  null;

return mejor;
```

Este último método es llamado desde la clase gestor, desde el método escaladaMaximaPendiente(), que resumidamente lo que hace llamar en bucle a *MejorMovientoEMP(Tablero mejor, GestorSolucion g)*, este bucle acabará cuando este método devuelva null, es decir, no se haya encontrado un tablero con mejor heurística, o cuando el tablero encontrado sea el estado final. Exponemos un pseudocódigo para mejor comprensión:

```
long initTime = tiempoInicial();
boolean fin = false; Tablero nuevo; Tablero actual = t;//Obtenemos el tablero inicial

                while(nuevo != null && !fin) {

                        nuevo = actual.MejorMovimientoES(nuevo, this);

                        if(nuevo != null) {

                                nuevo.setPadre(actual);//Establecemos puntero, para guardarlo para
                                                        solucion
                                actual = nuevo;//Actualizamos
                                String solucion += actual.getMovimiento() + " ";//Añadimos el
                                                        movimiento a la cadena de solucion
                                if(actual.getHeuristica() == 0) fin = true;//Si es ultimo estado
                        }


                }

                long endTime = tiempoFinal();
                long tiempo = endTime - initTime;//Tiempo que se tarda en ejecutar algoritmo

return fin;
```

4.2.1. MODIFICACIÓN

De este último algoritmo explicado se ha realizado una modificación que no supone demasiados cambios en el código, pero requieren ser explicados. El objetivo era conseguir que cuando se realizan todos los movimientos posibles en *MejorMovientoEMP(Tablero mejor, GestorSolucion g)* y no se encuentre un estado mejor, se siga hacia adelante un número determinado de veces, ya sea con un estado con igual o con peor heurística. Además, se usa una lista para comprobar que no se vuelve a generar un nodo que ya se incluye en la solución y cuando se halla el estado final se dejan de generar nodos.

Para ello se han realizado unos cuantos cambios en el método de la clase Tablero quedando recogidos en el método *MejorMovientoEMP1(Tablero mejor, GestorSolucion g)*, del cual se muestra el pseudocódigo a continuación:

```
boolean enc = false, fin = false, igual = false;
Tablero aux; int mejorh = this.funcionHeuristica();

Conjunto <Integer> nulo = generarMovNulos(); //ya se explicó como funciona

	while(!nulo.isEmpty() && !enc) {

		Conjunto <Integer> mov = generarMovimientos(); //ya se explicó como funciona

		Integer num = nulo.get(0); //Extraemos el nulo para realizar el movimiento

			while(!mov.isEmpty() && !enc) {

				switch(movimiento) { //Ya explicamos como funcionaban los metodos de
                                     realizar movimientos
					case 0:
						if(aux.moverNorte(n)) g.addNodos();//gestor cuenta
                                                           nodo generado
						break;

					case 1:
						if(aux.moverEste(n)) g.addNodos();
						break;

					case 2:
						if(aux.moverOeste(n)) g.addNodos();
						break;

					case 3:
						if(aux.moverSur(n)) g.addNodos();
						break;

				}
				if(!isConjunto(aux)) {//Si no está repetio
					int h = aux.funcionHeuristica();

					if(h < mejorh) {
					    mejorh = h;
					    enc = true;
					    mejor = aux;
					    if(mejorh == 0) fin = true;
					}else if(h == mejorh && !enc) { //Si es igual y no ha habido
					     mejorh = h;                mejor
					     igual = true;
					     mejor = aux;
					}else if(h > mejorh && !enc && !igual) {//Si es igual y no
					     mejorh = h;                    ha habido mejor ni igual
					     peor = true;
					     mejor = aux;
				}
```

```
                    mov.remove(0);//ya hemos relizado ese movimiento
                }
            nulo.remove(0);//Ya hemos relizado todos los movimientos con ese nulo
        }

        if(!enc) {
            if(igual) g.addNodoIgual(); //Si es igual el estado elegido
            else g.addNodoPeor(); //Si es peor el estado elegido

            if(g.getNodoIgual() == 80 && g.getNodoPeor() == 50) mejor = null; //Si se alcanza
                                                                              límite termina
        }

        addConjunto(mejor);//Lo añadimos para comprobar que no se vuelve a generar

        return mejor;
```

En el método de la clase *GestorSolución* no se han relizado cambios, sin embargo, se ha añadido un método escaladaMaximaPendiente1() con el objetivo de poderlo ejecutar por separado desde la interfaz gráfica como otro algoritmo más.

4.3. ALGORITMO A*

Primero debemos comentar el método realizado en la clase realizado en la clase *Tablero,* GenerarMovimientosA(Gestor g, int coste), el cual se encarga de realizar movimientos aleatorios sobre el tablero en el que nos encontramos actualmente (this), en el caso de que ese tablero no se haya generado con anterioridad se añade a la lista de abiertos, en caso contrario se comprueba que dicho tablero tiene mejor valor de f que el que habíamos generado con anterioridad, si es así y se encontraba en cerrados hay que modificar el tablero y sus hijos, si los tiene, en caso de que se encontrara en abiertos solo hay que modificar dicho tablero, puesto que no puede tener hijos. Realizar un pseudocódigo de todo este método sería muy extenso, así que se realiza un resumen de este código:

```
boolean enc = false, fin = false, igual = false;
Tablero aux; int mejorh = this.funcionHeuristica();

Conjunto <Integer> nulo = generarMovNulos(); //ya se explicó como funciona

        while(!nulo.isEmpty() && !enc) {

                Conjunto <Integer> mov = generarMovimientos(); //ya se explicó como funciona

                Integer num = nulo.get(0); //Extraemos el nulo para realizar el movimiento

                        while(!mov.isEmpty() && !enc) {

                                switch(movimiento) { //Ya explicamos como funcionaban los metodos de
                                                     realizar movimientos
                                        case 0:
                                          if(aux.moverSur(n)) {
                                                g.addNodos();
                                                b0 = g.isCerrado(aux); b1 = g.isAbierto(aux);

                                                  if(!b0 &&  !b1){
                                                        aux.setPadre(this);
                                                        g.addAbierto(aux);
                                                        if(aux.getHeuristica() == 0) fin = true;
```

```
                                        }else{
                                          if(b1) {
                                              t = g.getRepetidoAbiertos(aux);
                                              if(aux.getFuncion() < t.getFuncion()) {
                                                   t.modificar();
                                                   t.setPadre(this);
                                                   g.sortAbiertos();
                                              }else {
                                                   t = g.getRepetidoCerrados(aux);
                                                 if(aux.getFuncion() < t.getFuncion()) {
                                                    t.modificar();
                                                    t.setPadre(this);
                                                    g.addHijo(t, repetidos);
                                                    while(repetidos.size() > 0) {
                                                      repetidos = g.Busqueda(repetidos);
                                                    }
                                                  }
                                                }
                                            }
                                        }

                                    case 1:
                                           if(aux.moverEste(n)) (...)//igual que el anterior
                                           break;

                                    case 2:
                                           if(aux.moverOeste(n)) (...)//igual
                                           break;

                                    case 3:
                                           if(aux.moverSur(n)) (...)//igual
                                           break;

                                }

                            mov.remove(0);//ya hemos relizado ese movimiento
                        }
            nulo.remove(0);//Ya hemos relizado todos los movimientos con ese nulo
        }
```

Este último método es llamado desde la clase gestor, desde el método **algoritmoA()**, que resumidamente lo que hace es **llamar** a ***GenerarMovimientosA****(int coste, GestorSolucion g)* en bucle que **acabará** cuando el primer **tablero** de **abiertos** sea el **estado final**. Exponemos un pseudocódigo para mejor comprensión:

```
boolean fin = false;
Tablero actual; addAbierto(actual);
addNodos();

        while(!fin) {
           actual = primerAbierto();
           removeAbierto();
           addCerrado(actual);
           if(actual.FuncionHeuristica1() == 0) fin = true;
           actual.GenerarMovimientosA1(this, actual.getCoste()+1);
           sortAbiertos();
        }

int i = 0;
int index = cerrados.size();
String[] solucion = new String[index];
Tablero t = cerrados.get(index-1);

while(t != null) {
        solucion[index-i-1] = t.getMovimiento();
        t = t.getPadre();
        i++;
}

for (String s : solucion) {
```

```
        if(s != null)
        solucionFinal += s + " ";
}


long endTime = tiempoFinal();
long tiempo = endTime - initTime;//Tiempo que se tarda en ejecutar algoritmo

return fin;
```

4.3.1. MODIFICACIÓN (SIN “ATAJOS”)

Este algoritmo es exactamante igual que el anterior, lo único que cambia es que en el momento en que genera un tablero repetido, no lo tiene en cuenta y no lo añade a ninguna de las listas. Se adjunta un pseudocódigo del algoritmo que cambia:

```
boolean enc = false, fin = false, igual = false;
Tablero aux; int mejorh = this.funcionHeuristica();

Conjunto <Integer> nulo = generarMovNulos(); //ya se explicó como funciona

        while(!nulo.isEmpty() && !enc) {

                Conjunto <Integer> mov = generarMovimientos(); //ya se explicó como funciona

                Integer num = nulo.get(0); //Extraemos el nulo para realizar el movimiento

                        while(!mov.isEmpty() && !enc) {

                                switch(movimiento) { //Ya explicamos como funcionaban los metodos de
                                                     realizar movimientos
                                        case 0:
                                                if(aux.moverNorte(n)){
                                                  if((!g.isCerrado(aux) &&  !g.isAbierto(aux))) {
                                                                aux.setPadre(this);
                                                                g.addNodos();
                                                                g.addAbierto(aux);
                                                        }
                                                }

                                        case 1:
                                                if(aux.moverEste(n)){
                                                  if((!g.isCerrado(aux) &&  !g.isAbierto(aux))) {
                                                                aux.setPadre(this);
                                                                g.addNodos();
                                                                g.addAbierto(aux);
                                                  }
                                                }
                                                break;

                                        case 2:
                                                if(aux.moverOeste(n)){
                                                   if((!g.isCerrado(aux) &&  !g.isAbierto(aux))) {
                                                                aux.setPadre(this);
                                                                g.addNodos();
                                                                g.addAbierto(aux);
                                                  }
                                                }l
                                                break;

                                        case 3:
                                                if(aux.moverSur(n)){
                                                  if((!g.isCerrado(aux) &&  !g.isAbierto(aux))) {
                                                                aux.setPadre(this);
                                                                g.addNodos();
                                                                g.addAbierto(aux);
```

```
                                    }
                                }
                                 break;

                        }

                        mov.remove(0);//ya hemos relizado ese movimiento
                    }
    nulo.remove(0);//Ya hemos relizado todos los movimientos con ese nulo
}
```

5. Resultados obtenidos en las baterías de pruebas para cada una de las soluciones

Para realizar las baterías de pruebas se han usado los 10 puzzles proporcionados por el profesorado, además hemos añadido algunos de tamaño más grande 5x5, para mejorar esas pruebas. Se han realizado pruebas con todos los algoritmos explicados.

En el proyecto se han adjuntado ficheros de texto con la información de muchas ejecuciones realizadas para cada puzzle con todos los algoritmos, pero aquí comentaremos los datos más relevantes.

En primer lugar, se observan diferencias con el uso de una u otra función heurística, como era de esperar la función que implementa la *distancia Manhattan* da mejores resultados que la otra función que se ha usado (resuelve más puzzles para un mismo algoritmo). Era obvio que ayudara a resolver más puzzles, ya que da una información más detallada sobre la situación por el tablero, por otra parte, usar esta función también influye en el tiempo de ejecución, puesto que es más compleja de calcular.

En cuanto a los algoritmos, se muestra los puzzles que ha resuelto al que se le añade info adicional:

5.1. ATENDIENDO A ALGORITMOS

Se muestran los puzzles que ha resuelto cada algoritmo, añadiéndole información de relevancia.

5.1.1. ESCALADA SIMPLE

Puzzle 1 → Tiempo medio empleado: 45564.311 ns // Nodos generados media: 16 // Nº resueltos: 1000 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 6

Puzzle 2 → Tiempo medio empleado: 11737.07 ns // Nodos generados media: 5 // Nº resueltos: 1000 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 2

Puzzle 5 → Tiempo medio empleado: 23800.488 ns // Nodos generados media: 16 // Nº resueltos: 517 // Nº no resueltos: 483

Tasa de acierto: 51.7% Número de movimientos medio: 5

Puzzle 7 → Tiempo medio empleado: 40870.368 ns // Nodos generados media: 28 // Nº resueltos: 58 // Nº no resueltos: 942

Tasa de acierto: 5'8% Número de movimientos medio: 10

Puzzle 8 → Tiempo medio empleado: 27421.586 ns // Nodos generados media: 22 // Nº resueltos: 142 // Nº no resueltos: 858

Tasa de acierto: 14'2% Número de movimientos medio: 7

Observamos cómo se han conseguido resolver puzzles que necesitan pocos movimientos para hallar una solución.

5.1.2. ESCALADA MÁXIMA PENDIENTE

Puzzle 1 → Tiempo medio empleado: 36474.399 ns // Nodos generados media: 39 // Nº resueltos: 1000 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 2

Puzzle 2 → Tiempo medio empleado: 11048.055 ns // Nodos generados media: 11 // Nº resueltos: 1000 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 4

Puzzle 5 → Tiempo medio empleado: 27640.987 ns // Nodos generados media: 32 // Nº resueltos: 518 // Nº no resueltos: 482

Tasa de acierto: 51.8% Número de movimientos medio: 5

Puzzle 7 → Tiempo medio empleado: 51597.703 ns // Nodos generados media: 63 // Nº resueltos: 71 // Nº no resueltos: 929

Tasa de acierto: 7'1% Número de movimientos medio: 10

Puzzle 8 → Tiempo medio empleado: 41426.672 ns // Nodos generados media: 48 // Nº resueltos: 145 // Nº no resueltos: 855

Tasa de acierto: 14'5% Número de movimientos medio: 7

Hemos conseguido resolver los mismos puzzles que con escalada simple, pero generando un mayor número de nodos. Estos resultados han sido posibles gracias a usar la distancia de Manhattan como función heurística, sino los algoritmos resueltos, o incluso las tasas de aciertos serían menores.

5.1.2.1. ESCALADA MÁXIMA PENDIENTE (MOD)

Puzzle 1 → Tiempo medio empleado: 0.039 ms // Nodos generados media: 36 // Nº resueltos: 1000 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 6

Puzzle 2 → Tiempo medio empleado: 0.01 ms // Nodos generados media: 9 // Nº resueltos: 1000 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 2

Puzzle 3 → Tiempo medio empleado: 0.524 ms // Nodos generados media: 513 // Nº resueltos: 9 // Nº no resueltos: 991

Tasa de acierto: 0,9% Número de movimientos medio: 92

Puzzle 4 → Tiempo medio empleado: 0.494 ms // Nodos generados media: 519 // Nº resueltos: 12 // Nº no resueltos: 988

Tasa de acierto: 1'2% Número de movimientos medio: 96

Puzzle 5 → Tiempo medio empleado: 0.102 ms // Nodos generados media: 112 // Nº resueltos: 914 // Nº no resueltos: 86

Tasa de acierto: 91.4% Número de movimientos medio: 20

Puzzle 6 → Tiempo medio empleado: 0.535 ms // Nodos generados media: 536 // Nº resueltos: 10 // Nº no resueltos: 990

Tasa de acierto: 10% Número de movimientos medio: 98

Puzzle 7 → Tiempo medio empleado: 0.375 ms // Nodos generados media: 367 // Nº resueltos: 511 // Nº no resueltos: 489

Tasa de acierto: 51'1% Número de movimientos medio: 64

Puzzle 8 → Tiempo medio empleado: 0.321 ms // Nodos generados media: 329 // Nº resueltos: 544 // Nº no resueltos: 456

Tasa de acierto: 54'4% Número de movimientos medio: 57

Puzzle 9 → Tiempo medio empleado: 0.524 ms // Nodos generados media: 542 // Nº resueltos: 14 // Nº no resueltos: 986

Tasa de acierto: 1'4% Número de movimientos medio: 99

Puzzle 10 → Tiempo medio empleado: 0.57 ms // Nodos generados media: 565 // Nº resueltos: 13 // Nº no resueltos: 987

Tasa de acierto: 1'3% Número de movimientos medio: 102

Este algoritmo ha conseguido resolver todos los algoritmos al menos alguna vez, hemos conseguido que en alguna ocasión pueda resolver los puzzles 3, 4, 6, 9 y 10. Podemos observar además que en estos puzzles tiene una tasa de acierto muy pequeña,

pero es normal porque depende totalmente de la aleatoriedad de los puzzles que elija, en esta modificación lo que hicimos simplemente es que siguiera un número de veces determinado en el caso de no encontrar ningún tablero con mejor heurística.

5.1.3. ALGORITMO A*

Puzzle 1 → Tiempo medio empleado: 0.18 ms // Nodos generados media: 41 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 6

Puzzle 2 → Tiempo medio empleado: 0.03 ms // Nodos generados media: 13 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 2

Puzzle 3 → Tiempo medio empleado: 1330.12 ms // Nodos generados media: 16970 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 59

Puzzle 4 → Tiempo medio empleado: 4267.86 ms // Nodos generados media: 21931 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 78

Puzzle 5 → Tiempo medio empleado: 0.02 ms // Nodos generados media: 45 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 7

Puzzle 6 → Tiempo medio empleado: 637.73 ms // Nodos generados media: 10646 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 69

Puzzle 7 → Tiempo medio empleado: 0.26 ms // Nodos generados media: 160 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 19

Puzzle 8 → Tiempo medio empleado: 0.19 ms // Nodos generados media: 118 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 14

Puzzle 9 → Tiempo medio empleado: 928.97 ms // Nodos generados media: 10871 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 67

Puzzle 10 → Tiempo medio empleado: 1466.9 ms // Nodos generados media: 13265 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 76

Con este algoritmo hemos conseguido algo importante, y es hallar la solución de todos los puzzles, además con un tasa de acierto del 100%. Como era de esperar, el algoritmo A* resuelve todos los algoritmos aunque emplee más tiempo. Debemos mencionar además que la f usada es f = (int) (1.6 * h + 0.4 * g). En otro apartado se analizarán resultados con distintos pesos en f.

5.1.3.1. ALGORITMO A* (SIN "ATAJOS")

Puzzle 1 → Tiempo medio empleado: 0.24 ms

Nodos generados media: 33
Numero movimientos media: 6
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 2 → Tiempo medio empleado: 0.03 ms

Nodos generados media: 11
Numero movimientos media: 2
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 3 → Tiempo medio empleado: 948.09 ms

Nodos generados media: 8546
Numero movimientos media: 60
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 4 → Tiempo medio empleado: 3732.91 ms

Nodos generados media: 12401
Numero movimientos media: 77
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 5 → Tiempo medio empleado: 0.04 ms

Nodos generados media: 35
Numero movimientos media: 7
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 6 → Tiempo medio empleado: 573.98 ms

Nodos generados media: 6095
Numero movimientos media: 70
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 7 → Tiempo medio empleado: 0.26 ms

Nodos generados media: 113
Numero movimientos media: 18
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 8 → Tiempo medio empleado: 0.16 ms

Nodos generados media: 91
Numero movimientos media: 15
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 9 → Tiempo medio empleado: 515.13 ms

Nodos generados media: 5166
Numero movimientos media: 67
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 10 → Tiempo medio empleado: 1121.46 ms

Nodos generados media: 8083
Numero movimientos media: 76
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Podemos observar cómo los resultados entre el algoritmo A* sin "atajos" no difieren demasiado del algoritmo A* canónico. En lo que más se nota es en el número de nodos generados, pero sigue dando soluciones similares (con número de movimientos parecidos) en un tiempo equivalente.

5.1.3.2. ATENDIENDO A LA F

Dentro del algoritmo de A* tenemos que tener en cuenta la f usada, por tanto, a continuación, se realizan las siguientes pruebas, con carácter de hallar lo relevante que es esta función f.

5.1.3.2.1. F = (int) (1.6 * h + 0.4 * g)

Puzzle 1 → Tiempo medio empleado: 0.18 ms // Nodos generados media: 41 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 6

Puzzle 2 → Tiempo medio empleado: 0.03 ms // Nodos generados media: 13 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 2

Puzzle 3 → Tiempo medio empleado: 1330.12 ms // Nodos generados media: 16970 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 59

Puzzle 4 → Tiempo medio empleado: 4267.86 ms // Nodos generados media: 21931 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 78

Puzzle 5 → Tiempo medio empleado: 0.02 ms // Nodos generados media: 45 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 7

Puzzle 6 → Tiempo medio empleado: 637.73 ms // Nodos generados media: 10646 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 69

Puzzle 7 → Tiempo medio empleado: 0.26 ms // Nodos generados media: 160 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 19

Puzzle 8 → Tiempo medio empleado: 0.19 ms // Nodos generados media: 118 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 14

Puzzle 9 → Tiempo medio empleado: 928.97 ms // Nodos generados media: 10871 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 67

Puzzle 10 → Tiempo medio empleado: 1466.9 ms // Nodos generados media: 13265 // Nº resueltos: 100 // Nº no resueltos: 0

Tasa de acierto: 100% Número de movimientos medio: 76

5.1.3.2.1. F = (int) (1.5 * h + 0.5 * g)

Puzzle 1 → Tiempo medio empleado: 0.22 ms

Nodos generados media: 38
Numero movimientos media: 6
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 2 → Tiempo medio empleado: 0.03 ms

Nodos generados media: 10
Numero movimientos media: 2
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 3 → Tiempo medio empleado: 16677.66 ms

Nodos generados media: 29143
Numero movimientos media: 54
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 4 → Tiempo medio empleado: 36719.02 ms

Nodos generados media: 44743
Numero movimientos media: 68
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 5 → Tiempo medio empleado: 0.12 ms

Nodos generados media: 41
Numero movimientos media: 7
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 6 → Tiempo medio empleado: 2124.8 ms

Nodos generados media: 15255
Numero movimientos media: 64
Numero de resueltos: 100
Numero de no resueltos: 0

Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 7 → Tiempo medio empleado: 0.52 ms

Nodos generados media: 204
Numero movimientos media: 20
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 8 → Tiempo medio empleado: 0.22 ms

Nodos generados media: 119
Numero movimientos media: 15
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 9 → Tiempo medio empleado: 614.37 ms

Nodos generados media: 8277
Numero movimientos media: 62
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 10 → Tiempo medio empleado: 2611.94 ms

Nodos generados media: 17910
Numero movimientos media: 65
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

5.1.3.2.1. F = (int) (1.6 * h + 0.2 * g)

Puzzle 1 → Tiempo medio empleado: 0.89 ms

Nodos generados media: 40
Numero movimientos media: 6
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 2 → Tiempo medio empleado: 0.01 ms

Nodos generados media: 13
Numero movimientos media: 2
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 3 → Tiempo medio empleado: 26681.55 ms

Nodos generados media: 31060
Numero movimientos media: 73
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 4 → Tiempo medio empleado: 41601.74 ms

Nodos generados media: 46416
Numero movimientos media: 106
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 5 → Tiempo medio empleado: 0.06 ms

Nodos generados media: 50
Numero movimientos media: 7
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 6 → Tiempo medio empleado: 564.46 ms

Nodos generados media: 9167
Numero movimientos media: 85
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 7 → Tiempo medio empleado: 0.47 ms

Nodos generados media: 181
Numero movimientos media: 19
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 8 → Tiempo medio empleado: 0.26 ms

Nodos generados media: 140
Numero movimientos media: 15

Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 9 → Tiempo medio empleado: 1176.75 ms

Nodos generados media: 9668
Numero movimientos media: 83
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 10 → Tiempo medio empleado: 5824.57 ms

Nodos generados media: 17024
Numero movimientos media: 94
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

5.1.3.2.1. F = (int) (1.5 * h + 0.6 * g)

Puzzle 1 → Tiempo medio empleado: 0.17 ms

Nodos generados media: 33
Numero movimientos media: 6
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 2 → Tiempo medio empleado: 0.01 ms

Nodos generados media: 11
Numero movimientos media: 2
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 3 → Tiempo medio empleado: 59191.11 ms

Nodos generados media: 43894
Numero movimientos media: 48
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 4 → Tiempo medio empleado: 52312.73 ms

Nodos generados media: 34284

Numero movimientos media: 64
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 5 → Tiempo medio empleado: 0.06 ms

Nodos generados media: 35
Numero movimientos media: 7
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 6 → Tiempo medio empleado: 2008.54 ms

Nodos generados media: 11111
Numero movimientos media: 60
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 7 → Tiempo medio empleado: 0.53 ms

Nodos generados media: 130
Numero movimientos media: 19
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 8 → Tiempo medio empleado: 0.22 ms

Nodos generados media: 88
Numero movimientos media: 14
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 9 → Tiempo medio empleado: 515.56 ms

Nodos generados media: 6056
Numero movimientos media: 58
Numero de resueltos: 100
Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Puzzle 10 → Tiempo medio empleado: 11256.18 ms

Nodos generados media: 15672
Numero movimientos media: 61
Numero de resueltos: 100

Numero de no resueltos: 0
Tasa de aciertos: 100 %
Tasa de error: 0 %

Hemos podido observar en este último apartado que la función f es muy importante a la hora de implementar el algoritmo A*, de esta función depende que la solución se mejores (con menor número de movimientos) pero que emplee mayor tiempo o viceversa. Así observamos cómo cuando usamos una f con más peso en g, halla soluciones mejores empleando más tiempo, y sin embargo, cuando empleamos un mayor peso en h, halla soluciones más rápido pero peores. Estos datos se observan muy bien en los puzzles 3 y 4 por ejemplo, que son unos de los que más movimientos necesita para la resolución.

Hemos realizado una media de 1000 pruebas, para cada puzzle y por método para todo menos en el Algoritmo A* que hemos realizado 100, que se pueden encontrar en la carpeta "pruebas" del proyecto, donde se pueden ver más datos y más detallado.

6. Conclusiones sobre los resultados para cada una de las técnicas de resolución empleadas.

En este apartado se incluyen una serie de conclusiones, obtenidas a partir de la comparación del apartado anterior:

- En primer lugar, hay que destacar que la función heurística de la *distancia de Manhattan* proporciona una mayor información sobre el estado del tablero, y lo demuestra dando una mayor tasa de aciertos en nuestros algoritmos. Por ello, en todos los algoritmos implementados se usa esta función.
- Podemos afirmar que los algoritmos de escalada proporcionan soluciones rápidas, pero a la vez sólo resuelven los algoritmos que necesitan pocos movimientos para llegar a la solución, a excepción de la modificación realizada, que tiene una pequeña probabilidad de resolver otros puzzles.
- En cuanto a la diferencia entre el algoritmo A* canónico, y el algoritmo A* sin "atajos" no se aprecian grandes diferencias. Causa por la cual se ha decidido dejar implementada exclusivamente la versión con "atajaos", puesto que siempre tiene la mayor probabilidad de conseguir mejores soluciones.
- Lo siguiente que hemos podido observar es la diferencia de soluciones en puzzles que necesitan bastantes movimientos, como son el 3, 4, 6, 9 y 10, en función a la f usada en el algoritmo A*. La primera conclusión es que para que se halen soluciones en un tiempo aceptable (inferiores al minuto) necesitamos que haya un mayor peso en h que en g, no se han incluido pruebas con valores inferiores en h porque el

tiempo asciende considerablemente, y no se disponía del tiempo suficiente para realizar este tipo de pruebas.

- Relacionado con esto, podemos ver como se suele reducir el número de nodos generados usando mayores pesos de h. Ejemplo:

f = (int) (1.6 * h + 0.4 * g)

Puzzle 4 → Tiempo medio empleado: 36719.02 ms // Nodos generados media: 44743

f = (int) (1.5 * h + 0.5 * g)

Puzzle 4 → Tiempo medio empleado: 4267.86 ms // Nodos generados media: 21931

f = (int) (1.6 * h + 0.2 * g)

Puzzle 4 → Tiempo medio empleado: 2008.54 ms // Nodos generados media: 46416

Se puede observar cómo en los 2 algoritmos que han usado 1.6 de peso en la h, generan más nodos de media, aunque también cambian los valores de g, que también influye. Pero podemos afirmar que no influyen tanto, ya que la diferencia entre el primer y tercer algoritmo en cuanto al número de nodos generados son ínfimas y sin embargo el peso en g si que varía considerablemente. Todo esto nos lleva a afirmar que lo que más influye en el tiempo de ejecución es el peso usado en h.

- En cuanto a la variación de g, se puede observar que influye bastante en los movimientos requeridos para solucionar el puzzle:

f = (int) (1.6 * h + 0.4 * g)

Puzzle 4 → Número de movimientos medio: 78 // Nodos generados media: 44743

f = (int) (1.5 * h + 0.6 * g)

Puzzle 4 → Número movimientos media: 64 // Nodos generados media: 21931

f = (int) (1.6 * h + 0.2 * g)

Puzzle 4 → Número de movimientos medio: 103 // Nodos generados media: 66585

Sobre todo, entre la comparación del primer y el tercer ejemplo se aprecia de manera más clara de variación del número de movimientos de media, puesto que tienen un valor fijo de peso en h. Aun así la función que tiene mayor peso en g obtiene los mejores resultados en el número de movimientos de media.

- En cuanto a los algoritmos con mayor tamaño no se han incluido pruebas con muchas repeticiones porque el tiempo aumenta considerablemente y no se dispone del tiempo suficiente para realizar este tipo de pruebas.

Por último destacar que dependiendo del tipo de puzzle que queramos resolver mejor un algoritmo u otro. Si queremos una solución rápida y nos enfrentamos a puzzles con pocos movimientos probablemente el algoritmo que mejor nos venga sería el de escalada simple, o incluso escalada máxima pendiente. Sin embargo, si nos interesa más poder resolver puzzles con más movimientos pues nos interesa el algoritmo A*, porque probablemente ninguno de escalada lo resuelva, y dentro de A*, dependiendo si nos interesa una solución con pocos movimientos o una solución más rápida habrá que usar unos pesos u otros en la función f. Así que dependiendo de lo que queramos unos algoritmos nos vienen mejor que otros.

7. Referencias.

www.github.com

https://docs.oracle.com/en/java/javase/11/docs/api/index.html

http://unestudiantedeinformatica.blogspot.com/2014/07/medir-el-tiempo-de-ejecucion-en-java.html

http://lineadecodigo.com/java/crear-un-fichero-en-java/

Apuntes de la asignatura