Práctica 3: Misioneros y Puzzle con AIMA
Inteligencia Artificial

Autores:
José Javier Cortés Tejada
Pedro David González Vázquez

Parte 1: Misioneros con AIMA

Definición de heurísticas

Búsquedas no informadas

	Coste del camino	Nodos expandidos
Prof. iterativa		
Prof. limitada		

Búsquedas informadas

Voraz, A*

	Coste del camino	Nodos expandidos	Tam. de cola	Tam. de cola (máx)

Parte 2: Puzzle con AIMA

Primera heurística

La primera heurística se encuentra en la clase ***BWHeuristic1***, en el paquete ***blackWhitePuzzle***, donde tratamos de dar un coste máximo para todas las operaciones posibles sobre una configuración concreta del estado, en función de la distancia de cada pieza negra a una pieza blanca que no haya sido tenida en cuenta antes.

El siguiente fragmento de código define la implementación de esta heurística:

```
1  public double h(Object arg0) {
2      Piece[] pieces = ((BWPuzzleBoard)arg0).getState();
3      double acc = 0;
4      boolean marks[] = new boolean[7];
5          for (int i = 0; i < 3; i++) {
6              if (pieces[i] == Piece.BLACK) {
7                  for (int j = 3; j < 7; j++) {
8                      if (pieces[j] == Piece.WHITE &&
                          !marks[j]) {
9                          acc += (j - i > 2) ? 2 * (j - i) : 1;
10                         marks[j] = true;
11                         break;
12                     }
13                 }
14             }
15         }
16         return acc;
17     }
18 }
```

En primer lugar, hacemos una copia del estado actual, creamos una variable que contendra el coste y creamos un *array* de *boolean* donde indicaremos las piezas que hemos procesado poniendo su posición (en función de la que ocupen en el *array* que define el estado actual.

En el primer for miramos las 3 primeras posiciones del tablero y vemos si alguna es negra (si alguna está mal colocada). En caso de que ninguna sea negra, el valor devuelto será 0 pues la configuración de ese estado es la final, y en cualquier otro caso pasaremos a buscar una pieza blanca y no marcada (si está marcada ya la hemos procesado antes) desde la cuarta posición del tablero hasta la séptima con el fin de determinar el coste de ese intercambio.

Una vez hecho esto, marcamos la pieza blanca e incrementamos el contador, de tal manera que si la distancia entre ambas piezas es de dos casillas, el coste de dicho intercambio será el

doble de la distancia entre ambas, mientras que si se trata de un movimiento donde se salta una casilla o nos movemos a otra adyacente, el coste será 1. Tras haber determinado el coste de dicha operación, marcamos la pieza blanca y continuamos con la ejecución.

Segunda heurística

La segunda heurística se encuentra en la clase ***BWHeuristic2***, también en el paquete ***blackWhitePuzzle***, aunque en este caso nos interesa la distancia de la distancia de un par de piezas (una blanca y otra negra) al *hole*, con el fin de usar las distancias para determinar el coste de la operación (movemos la pieza negra al *hole*, la negra la intercambiamos con la blanca y por último movemos la blanca al *hole*). Esta heurística queda definida por el siguiente código:

```
1  public double h(Object arg0) {
2              Piece[] pieces = ((BWPuzzleBoard)
                  arg0).getState();
3              double acc = 0;
4              boolean marks[] = new boolean[7];
5
6              double posBlack = 0, posWhite= 0, posHole= 0;
7
8              for(int i = 0; i < 7; i++){
9                      if(pieces[i] == Piece.HOLE) {
10                             posHole = i;
11                             break;
12                             }
13             }
14
15             for (int j = 0; j < 3; j++) {
16                     if (pieces[j] == Piece.BLACK) {
17                                    marks[j] = true;
18                                    posBlack = j;
19                             }
20                     for (int i = 3; i < 7; i++) {
21                             if (pieces[i] == Piece.WHITE &&
                                  !marks[i]) {
22                                    marks[i] = true;
23                                    posWhite = i;
24                                    break;
25                             }
26                     }
27                     acc += Math.abs(posBlack - posWhite) +
                         Math.abs(posHole - posWhite);
28             }
```

```
29                  return acc;
30          }
```

En este caso, al igual que en el anterior, también vamos a hacer una copia del tablero, creamos un contador y un *array* de *boolean* que marcará las piezas blancas procesadas.

Empezamos con un *for* donde buscaremos la posición del *hole* dentro del tablero. Cuando la hayamos encontrado salimos del bucle y entramos a otro donde miramos las 3 primeras posiciones del tablero con el fin de encontrar una pieza negra (si hay alguna aquí, está mal colocada), y en caso de encontrar una, buscamos en la mitad derecha del tablero una pieza blanca no marcada.

Tanto de la pieza blanca como de la negra nos interesa su posición con respecto del tablero, luego guardamos este valor en variables auxiliares. Una vez hecho esto, nos quedamos con la suma de las distancias (en valor absoluto) de la pieza negra a la blanca y de la blanca al *hole*, la cual define determina el coste de cambiar la pieza negra con el *hole*, la blanca con la negra y por último la blanca con el *hole*.

Búsquedas no informadas

	Coste del camino	Nodos expandidos
Prof. iterativa	15	724476
Prof. limitada	17	647320

resumir mas, es redundante[oo]
DISCUSIÓN: los resultados obtenidos en ambas simulaciones son bastantes similares, luego cualquiera de los dos algoritmos nos ofrece un coste parecido, el cual compensa el tiempo de ejecución con la memoria consumida (y viceversa). Sin embargo, cabe destacar que en el caso de la profundidad iterativa, cuanto mayor sea el ámbito del problema obtendremos un *pathCost* bajo a costa de una gran cantidad de nodos expandidos, al contrario que con la profundidad limitada, donde tendremos un *pathCost* mayor (será el limite de profundidad), a cambio de un menor número de nodos expandidos, luego podemos concluir que para simulaciones pequeñas, los costes van a presentar ligeras variaciones, pero conforme incrementemos el espacio de búsqueda, no vamos a encontrar con el una de las dos variables crece enormemente con respecto de la otra.

Búsquedas informadas

Voraz, A*

DISCUSION PARA VORAZ: con la heurística definida en *BWHeuristic1.java* obtenemos en general mejores resultados que con la heurística de *BWHeuristic2.java*, de manera que en la primera solo tomamos el coste de cambiar una pieza blanca con una negra, sin embargo

	Coste del camino	Nodos expandidos	Tam. de cola	Tam. de cola (máx)
Voraz con BWHeuristic1	13	19	23	24
Voraz con BWHeuristic2	15	42	37	38
A* con BWHeuristic1	13	33	30	33
A* con BWHeuristic2	10	56	46	47

en la segunda hacemos el mismo movimiento pero fraccionado, es decir, cambiarmos una pieza negra por el *hole*, una blanca por la negra y luego la blanca con el *hole*, lo cual implica un coste mayor por operación, pues en el segundo caso es casi el doble que la primera, por tanto esto explica la diferencia de costes de una ejecución a otra, ya que estos son cerca del 25-50 por ciento mayores (50 para nodos exp y 25 cola y max cola). Resaltar tambien que tenermos un numero de pathCost muy similar y costes muy distintos, esto se debe a que el coste de cada operacion en la segunda heu ronda el doble que el de la primera.

DISCUSION PARA A*: lo mismo que en el anterior, aunque en este caso tenermos una heu que es más rápida a la hora de dar la solucion pero que consume muchos mas recursos. terminar bien