# Práctica 3: Misioneros y Puzzle con AIMA
# Inteligencia Artificial

**Autores (Grupo 5):**
**José Javier Cortés Tejada**
**Pedro David González Vázquez**

*Nota: en el documento se hace referencia en varias ocasiones a unos archivos externos, los cuales se han usado para razonar varias cuestiones de las expuestas en esta práctica. Dichos archivos contienen una batería de pruebas para cada puzzle (en el archivo BWPuzzleMTH.java se ha implementado la heurística Manhattan, con el fin de comparar el resultado de nuestras heurísticas con ella y ver así si conseguimos un pathCost mínimo y demás).*
*Además, en la entrega del proyecto también se han incluido dos ficheros, BWTest.java y MCTest.java en el paquete Test, los cuales generan los archivos indicados al principio, por si se quisiese comprobar que funcionan correctamente.*

# Parte 1: Misioneros con AIMA

## Primera heurística

Asignamos un coste estimado para reposicionar un misionero, el cual será 3, mientras que el coste de reposicionar un caníbal será 6, debido a la restricción de más misioneros que caníbales en la misma orilla.

```
1  public double h(Object state) {
2      MCPuzzleBoard board = (MCPuzzleBoard)arg0;
3      int[] st = board.getState();
4      return st[0] * 3 + st[1] * 6 ;
5  }
```

**¿Es admisible?** No es admisible pues al no tener en consideración el lado de la barca, en cada iteración aumentaremos el valor heurístico de dicho nodo provocando que el éste quede con un coste estimado mayor respecto del anterior, por consiguiente, no es admisible. Pese a ello, sí podemos observar que nos da los costes mínimos, pues en el caso de solo evaluar los nodos válidos, el coste del nodo siguiente siempre va a ser menor que el del caso anterior.

**¿Es consistente?** No es admisible, luego tampoco es consistente.

**¿Es óptima para A*?** Si es óptimo pues en las exploraciones realizadas hemos detectado siempre obtenemos el mínimo *pathCost*.

## Segunda heurística

Asignamos un coste igual al numero de misioneros más el número de caníbales en una orilla por dos (como los vamos a mover dos veces, estimamos un coste el doble de grande), más el coste de mover la barca.

```
1  public double h(Object state) {
2      MCPuzzleBoard board = (MCPuzzleBoard)state;
3      int[] st = board.getState();
4      return (st[0] + st[1]) * 2 + st[2];
5  }
```

**¿Es admisible?** A pesar de considerar la barca en este caso, tampoco es admisible pues no damos una estimación adecuada para todas las configuraciones

**¿Es consistente?** No es admisible, luego tampoco es consistente.

**¿Es óptima para A*?** Si es óptima pues en las exploraciones realizadas hemos detectado siempre obtenemos el mínimo *pathCost*.

## Búsquedas no informadas

| | Coste del camino | Nodos expandidos |
|---|---|---|
| Prof. iterativa | 11 | 14017 |
| Prof. limitada | 17 | 3568 |

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, nos vamos a encontrar con el que una de las dos variables crece enormemente con respecto de la otra.

## Búsquedas informadas

| | Coste del camino | Nodos expandidos | Tam. de cola | Tam. de cola (máx) |
|---|---|---|---|---|
| Voraz [1] | 11 | 21 | 7 | 9 |
| Voraz [2] | 11 | 19 | 8 | 9 |
| A* [1] | 11 | 22 | 6 | 9 |
| A* [2] | 11 | 25 | 3 | 9 |

Las ejecuciones de los dos algoritmos con diferentes heurísticas nos da tanto un *pathCost* como un tamaño de cola máximo idénticos, aunque presenta ligeras variaciones de cara al número de nodos expandidos y al tamaño de la cola.

[1] *con MCPuzzleHeuristic1*
[2] *con MCPuzzleHeuristic2*

# Parte 2: Puzzle con AIMA

## Primera heurística

La primera heurística se encuentra en la clase ***BWPuzzleHeuristic1***, 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 state) {
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 }
```

El *array* de *booleans* sirve para evitar intercambiar dos fichas negras con la posición válida mas cercana (una blanca no marcada al final del *array*), pues en el caso de no marcarla siembre calcularíamos el intercambio con la misma pieza (la más cercana), provocando un cálculo mal ajustado. Los valores asociados a la acumulación del coste están estimados en función de la distancia entre $i$ y $j$, y multiplicados por un coeficiente que, en caso de tener que hacer un salto de distancia mayor de 2 posiciones implica hacer como mucho $j - i$ saltos, luego queda multiplicado por 2 (son saltos de dos).

**¿Es admisible?** Es admisible e idónea pues evaluando el peor caso posible de intercambio de piezas (pieza negra en posición 0 del tablero y pieza blanca en posición 6 del tablero), en este caso el número de salto a realizar es mayor, y por consiguiente nuestro coste estimado es 2*$distancia\ de\ las\ fichas$, con lo que tenemos un coste estimado de 12. A la hora de realizar la simulación a mano, vemos que cuesta exactamente 12, que es menor o igual que el coste que habíamos estimado, por lo tanto, dicha estimación es válida para todos los casos ya que vale para el caso peor, luego sí es admisible.

**¿Es consistente?** No, porque al hacer varias ejecuciones hay casos en los que no tomamos el mejor camino para llegar a la solución.

**¿Es óptima para A*?** No es óptima en ningún caso, pues al lanzar diferentes simulaciones, obtenemos que para algunos algoritmos da valores mejores y para otros menores (ver el archivo adjunto a la entrega, ahí se aprecian los diferentes *pathCost* de las ejecuciones con diferentes algortimos).

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

```
26            acc += Math.abs(posBlack - posWhite) +
                  Math.abs(posHole - posWhite);
27        }
28        return acc;
29    }
```

El *array* de *booleans* sirve para evitar intercambiar dos fichas negras con la posición válida mas cercana (una blanca no marcada al final del *array*), pues en el caso de no marcarla siembre calcularíamos el intercambio con la misma pieza (la mas cercana), provocando un calculo mal ajustado. Los valores asociados a la acumulación del coste están estimados en función de la distancia entre la pieza negra y la blanca, más la distancia de la piewza blanca al *hole*.

**¿Es admisible?** Si pues evaluando el peor caso posible de intercambio de piezas (pieza negra en posición 0 del tablero y pieza blanca en posición 6 del tablero), en este caso el número de saltos a realizar es mayor, y por consiguiente nuestro coste estimado es *distancia de las fichas* + *distancia de la blanca al hole*, luego es admisible.

**¿Es consistente?** No, porque al hacer varias ejecuciones hay casos en los que no tomamos el mejor camino para llegar a la solución.

**¿Es óptima para A*?** No, pues al haber hecho varias ejecuciones vemos que obtenemos valores para el *pathCost* diferentes

## Búsquedas no informadas

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

Pasa lo mismo que con las búsquedas no informadas de la parte 1, a medida que obtengamos un *pathCost* menor, obtenemos un mayor número de nodos expandidos y viceversa para el otro algoritmo.

## Búsquedas informadas

| | Coste del camino | Nodos expandidos | Tam. de cola | Tam. de cola (máx) |
|---|---|---|---|---|
| Voraz [1] | 13 | 19 | 23 | 24 |
| Voraz [2] | 15 | 42 | 37 | 38 |
| A* [1] | 13 | 33 | 30 | 33 |
| A* [2] | 10 | 56 | 46 | 47 |

De cara a cada algoritmo, vemos que hemos obtenido resultados peores en aquellos que empleen la segunda heurística, luego la primera nos proporciona una estimación mejor. Sin embargo, de cara a ambos algoritmos, apreciamos que el voraz nos da un coste menor en cuanto a memoria y tiempo, pues busca la primera solución válida.

[1] *con BWPuzzleHeuristic1*
[2] *con BWPuzzleHeuristic2*