Proyecto taquin Análisis de Algoritmos

Camilo Hernández Guerrero
Samy Felipe Cuestas Merchán
Juan Camilo Mendieta Hernández

2 de junio de 2022

1. Parte I: Análisis y diseño del algoritmo

1.1. Análisis

El problema consiste en crear un jugador automático para Taquín el cual es un juego que consiste en mover cuadrículas de una unidad con una figura o numeros en una matriz cuadrada hasta ordenar la figura o la secuencia de números. Se abordará este problema con un método eurístico donde se calcularán los posibles tableros dependiendo de cada jugada realizada, donde la probabilidad de ganar se calcula a partir de la distancia euclidiana que tiene el tablero actual con el tablero que ya está resuelto, buscando una ruta que de una posible solución. Debido a la gran cantidad de posibles caminos que pueden deribar de una jugada, una solución de fuerza bruta no es adecuada.

1.2. Diseño

Usando el análisis previo se sabe que para resolver el problema del jugador automatico de Taquín, se van a necesitar para el diseño del algoritmo en los componentes de las entradas y las salidas los siguientes datos para cada función a usar.

1.2.1. Verificar tablero

- Entrada
  Se necesita la matriz del tablero de juego actual y la matriz del tablero resuelto.
- Salida
  Un valor booleano indicando si se gano el juego.

1.2.2. Moverse hacia arriba, abajo, derecha e izquierda

- Entrada
  Se necesita la matriz del tablero de juego actual y las coordenadas del espacio en blanco.
- Salida
  Coordenas de la nueva ubicación del espacio en blanco.

1.2.3. Euristica

- Entrada
  Se necesita el numero de de movimientos que se han realizado y la distancia euclidiana para resolver el tablero.
- Salida
  Número que representa la medida de la eurística.

1.2.4. Buscar posición de un elemento

- Entrada
  Se necesita la matriz del tablero actual y el elemento a buscar en la matriz.
- Salida
  Coordenas en la matriz donde se ubica el elemento.

1.2.5. Calcular distancia euclidiana

- Entrada
  Se necesita la matriz del tablero de juego actual y la matriz del tablero resuelto.
- Salida
  Numero con el calculo de la distancia euclidiana.

1.2.6. Obtener posibles movimientos

- Entrada
  Se necesita la matriz del tablero de juego actua, la matriz del tablero resuelto y un nodo del arbol con los tableros posibles
- Salida
  Listado de movimientos que se pueden hacer.

1.2.7. Buscar mejor nodo

- Entrada
  Listado con posibles tableros para jugar
- Salida
  Nodo con un tablero que tiene la menor distancia con el tablero completado.

1.2.8. Movimiento automático

- Entrada
  Se necesita la matriz del tablero actual y la matriz con el table resuelto.
- Salida
  Listado de los movimientos a realizar para ganar.

1.2.9. Inicializar tablero

- Entrada
  Numero para generar la matriz cuadrada.
- Salida
  Una matriz con objetos desoredenados para empezar el juego.

2. Parte II: Pseudocódigo

Algorithm 1 Función que verifica si ganó el juego

```
1: procedure gameWon(table, tableWon, n)
2:     result ← true
3:     for row ← 0 to n do
4:         for column ← 0 to n do
5:             if table[row][column] ≠ tableWon[row][column] then
6:                 result ← false
7:             end if
8:         end for
9:     end for
10:    return result
11: end procedure
```

Algorithm 2 Función que mueve la pieza hacia arriba

```
1: procedure moveUp(table, blankSpace)
2:     if blankSpace[0] ≠ 0 then
3:         aux ← table[blankSpace[0] − 1][blankSpace[1]]
4:         table[blankSpace[0] − 1][blankSpace[1]] ← ""
5:         table[blankSpace[0]][blankSpace[1]] ← aux
6:         blankSpace ← (blankSpace[0] − 1, blankSpace[1])
7:         return blankSpace
8:     else
9:         return blankSpace
10:    end if
11: end procedure
```

Algorithm 3 Función que mueve la pieza hacia abajo

```
1: procedure moveDown(table, blankSpace)
2:     if blankSpace[0] < |table[0]| − 1 then
3:         aux ← table[blankSpace[0] + 1][blankSpace[1]]
4:         table[blankSpace[0] + 1][blankSpace[1]] ← ""
5:         table[blankSpace[0]][blankSpace[1]] ← aux
6:         blankSpace ← (blankSpace[0] + 1, blankSpace[1])
7:         return blankSpace
8:     else
9:         return blankSpace
10:    end if
11: end procedure
```

Algorithm 4 Función que mueve la pieza hacia la izquierda

```
1: procedure moveLeft(table, blankSpace)
2:     if blankSpace[1] > 0 then
3:         aux ← table[blankSpace[0]][blankSpace[1] − 1]
4:         table[blankSpace[0]][blankSpace[1] − 1] ← ""
5:         table[blankSpace[0]][blankSpace[1]] ← aux
6:         blankSpace ← (blankSpace[0], blankSpace[1] − 1)
7:         return blankSpace
8:     else
9:         return blankSpace
10:    end if
11: end procedure
```

Algorithm 5 Función que mueve la pieza hacia la derecha

```
1: procedure moveRight(table, blankSpace)
2:     if blankSpace[1] < |table[0]| − 1 then
3:         aux ← table[blankSpace[0]][blankSpace[1] + 1]
4:         table[blankSpace[0]][blankSpace[1] + 1] ← ""
5:         table[blankSpace[0]][blankSpace[1]] ← aux
6:         blankSpace ← (blankSpace[0], blankSpace[1] + 1)
7:         return blankSpace
8:     else
9:         return blankSpace
10:    end if
11: end procedure
```

Algorithm 6 Función que obtiene la posición del elemento

```
1: procedure getElementPosition(currentPuzzleTable, element)
2:     for i ← 0 to |currentPuzzleTable| do
3:         if elementincurrentPuzzleTable[i] then
4:             return (i, currentPuzzleTable[i].index(element))
5:         end if
6:     end for
7: end procedure
```

Algorithm 7 Función que obtiene distancia euclidiana entre tablas

```
1: procedure tableEuclidianDistance(currentPuzzleTable, tableWon)
2:     tableDistance ← 0
3:     for i ← 0 to |currentPuzzleTable| do
4:         for j ← 0 to |currentPuzzleTable| do
5:             positionTableWon ← getElementPosition(tableWon, currentPuzzleTable[i][j])
6:             tableDistance ← tableDistance + ABS(i − positionTableWon[0]) + ABS(j −
   positionTableWon[1])
7:         end for
8:     end for
9:     return tableDistance
10: end procedure
```

Algorithm 8 Función que retorna los movimientos posibles

```
1: procedure getPossibleMoves(node, currentPuzzleTable, tableWon)
2:     listOfMoves ← ∅
3:     blankCoordinates ← getElementPosition(node.currentPuzzleTable, "")
4:     sizeCurrentPuzzle ← |currentPuzzleTable|
5:     newPosition ← (blankCoordinates[0] − 1, blankCoordinates[1])
6:     if 0 ≤ newPosition[0] < sizeCurrentPuzzle and 0 ≤ newPosition[1] < sizeCurrentPuzzle
   then
7:         newPuzzleTable ← deepcopy(node.currentPuzzleTable)
8:         newPuzzleTable[blankCoordinates[0]][blankCoordinates[1]] ←
   node.currentPuzzleTable[newPosition[0]][newPosition[1]]
9:         newPuzzleTable[newPosition[0]][newPosition[1]] ← ""
10:        listOfMoves ← listOfMoves∪Node(newPuzzleTable, node.currentPuzzleTable, node.numberOfMoves+
   1, tableEuclidianDistance(currentPuzzleTable, tableWon), "U")
11:    end if
12:    newPosition ← (blankCoordinates[0] + 1, blankCoordinates[1])
13:    if 0 ≤ newPosition[0] < sizeCurrentPuzzle and 0 ≤ newPosition[1] < sizeCurrentPuzzle
   then
14:        newPuzzleTable ← deepcopy(node.currentPuzzleTable)
15:        newPuzzleTable[blankCoordinates[0]][blankCoordinates[1]] ←
   node.currentPuzzleTable[newPosition[0]][newPosition[1]]
16:        newPuzzleTable[newPosition[0]][newPosition[1]] ← ""
17:        listOfMoves ← listOfMoves∪Node(newPuzzleTable, node.currentPuzzleTable, node.numberOfMoves+
   1, tableEuclidianDistance(currentPuzzleTable, tableWon), "D")
18:    end if
19:    newPosition ← (blankCoordinates[0], blankCoordinates[1] + 1)
20:    if 0 ≤ newPosition[0] < sizeCurrentPuzzle and 0 ≤ newPosition[1] < sizeCurrentPuzzle
   then
21:        newPuzzleTable ← deepcopy(node.currentPuzzleTable)
22:        newPuzzleTable[blankCoordinates[0]][blankCoordinates[1]] ←
   node.currentPuzzleTable[newPosition[0]][newPosition[1]]
23:        newPuzzleTable[newPosition[0]][newPosition[1]] ← ""
24:        listOfMoves ← listOfMoves∪Node(newPuzzleTable, node.currentPuzzleTable, node.numberOfMoves+
   1, tableEuclidianDistance(currentPuzzleTable, tableWon), "R")
25:    end if
26:    newPosition ← (blankCoordinates[0], blankCoordinates[1] − 1)
27:    if 0 ≤ newPosition[0] < sizeCurrentPuzzle and 0 ≤ newPosition[1] < sizeCurrentPuzzle
   then
28:        newPuzzleTable ← deepcopy(node.currentPuzzleTable)
29:        newPuzzleTable[blankCoordinates[0]][blankCoordinates[1]] ←
   node.currentPuzzleTable[newPosition[0]][newPosition[1]]
30:        newPuzzleTable[newPosition[0]][newPosition[1]] ← ""
31:        listOfMoves ← listOfMoves∪Node(newPuzzleTable, node.currentPuzzleTable, node.numberOfMoves+
   1, tableEuclidianDistance(currentPuzzleTable, tableWon), "L")
32:    end if
33:    return listOfMoves
34: end procedure
```

Algorithm 9 Función que obtiene el mejor nodo

```
1: procedure getBestNode(dictionaryOfNodes)
2:     i ← 0
3:     bestEuristic ← ∞
4:     for node ← 0 to dictionaryOfNodes.values() do
5:         if i = 0 or node.euristic() < bestEuristic then
6:             i ← i + i + 1
7:             bestNode ← node
8:             bestEuristic ← bestNode.euristic()
9:         end if
10:    end for
11:    return bestNode
12: end procedure
```

Algorithm 10 Función de movimiento

```
1: procedure movement(actualTable, tableWon)
2:     moves ← str(actualTable) : Node(actualTable, actualTable, 0, tableEuclidianDistance(actualTable, tableWon), "
3:     movesToWin ← ∅
4:     while true do
5:         testMove ← getBestNode(moves)
6:         movesToWin[str(testMove.currentPuzzleTable)] ← testMove
7:         if testMove.currentPuzzleTable = tableWon then
8:             auxNode ← movesToWin[str(tableWon)]
9:             auxMoves ← list()
10:            while auxNode.direction do
11:                auxMoves ← auxMoves∪auxNode.directionasdirection, auxNode.currentPuzzleTableasnode
12:                auxNode ← movesToWin[str(auxNode.previousPuzzleTable)]
13:            end while
14:            auxMoves.reverse()
15:            return auxMoves
16:        end if
17:        possibleMoves ← getPossibleMoves(testMove, actualTable, tableWon)
18:        for node ← 0 to |possibleMoves| do
19:            if str(node.currentPuzzleTable)inmovesToWin.keys() or
    str(node.currentPuzzleTable)inmoves.keys() and moves[str(node.currentPuzzleTable)].euristic() <
    node.euristic() then
20:                continue
21:            end if
22:            moves[str(node.currentPuzzleTable)] ← node
23:        end for
24:        delete moves[str(testMove.currentPuzzleTable)]
25:    end while
26: end procedure
```

Algorithm 11 Función principal

```
1: procedure slicePuzzle( )
2:     n ← 3
3:     randomPuzzle ← ∅
4:     puzzle ← ∅
5:     puzzleWon ← ∅
6:     letnumbersInOrder[0..(n ∗ n)]beasequence.
7:     letnumbersWithoutOrder[0..(n ∗ n)]bearandomsequence.
8:     blank ← (0, 0)
9:     i ← 0
10:    for numberA ← 0 to n do
11:        aux ← ∅
12:        for numberB ← 0 to n do
13:            if numbersWithoutOrder[i] ≠ (n ∗ n) − 1 then
14:                aux ← aux ∪ str(numbersWithoutOrder[i])
15:            else
16:                aux ← aux ∪ ""
17:                blank ← (numberA, numberB)
18:            end if
19:            i ← i + 1
20:        end for
21:        randomPuzzle ← randomPuzzle ∪ aux
22:    end for
23:    i ← 0
24:    for numberA ← 0 to n do
25:        aux ← ∅
26:        for numberB ← 0 to n do
27:            if numbersInOrder[i] ≠ (n ∗ n) − 1 then
28:                aux ← aux ∪ str(numbersInOrder[i])
29:            else
30:                aux ← aux ∪ ""
31:                blank ← (numberA, numberB)
32:            end if
33:            i ← i + 1
34:        end for
35:        puzzleWon ← puzzleWon ∪ aux
36:    end for
37:    puzzle ← randomPuzzle
38:    blank ← getElementPosition(puzzle, "")
39:    movimientos ← movement(puzzle, puzzleWon)
40:    turn ← 0
41:    while gameWon(puzzle, puzzleWon) = False do
42:        command ← movimientos[turn]["direction"]
43:        turn ← turn + 1
44:        if command = "U" then
45:            blank ← moveUp(puzzle, blank)
46:        else if command = "D" then
47:            blank ← moveDown(puzzle, blank)
48:        else if command = "L" then
49:            blank ← moveLeft(puzzle, blank)
50:        else if command = "R" then
51:            blank ← moveRight(puzzle, blank)
52:        end if
53:    end while
54:    print puzzle as Matrix
55: end procedure
```

3. Parte III: Análisis de complejidad

3.1. Verificar tablero

La complejidad de esta función es de O(n^2)*debidoasusdosciclos*.

3.2. Moverse hacia arriba, abajo, derecha e izquierda

La complejidad de estas funciones es de O(1) debido a su falta de ciclos.

3.3. Euristica

3.4. Buscar posición de un elemento

La complejidad de esta función es de O(n) debido a su unico ciclo.

3.5. Calcular distancia euclidiana

La complejidad de esta función es de O(n^2)*debidoasusdosciclos*.

3.6. Obtener posibles movimientos

La complejidad de estas funciones es de O(1) debido a su falta de ciclos.

3.7. Buscar mejor nodo

La complejidad de esta función es de O(n^2)*debidoasusdosciclos*.

3.8. Inicilizar tablero

La complejidad de esta función es de O(n^2)*debidoasusdosciclos*.

4. Parte IV: Pruebas

Para realizar las pruebas se utilizaron distintos tableros, aumentando la dificultad respecto a la cantidad de movimientos (turnos) necesaria teóricamente para resolver el tablero. Específicamente se evaluaron ocho tableros que difieren de dificiltad.

4.1. Tablero 3x3 resuelto en tres movimientos

```
puzzle2=[["0", "1", "2"],
         [" ", "4", "5"],
         ["3", "6", "7"]]
```

Figura 1: Tablero 3x3 fácil sin resolver

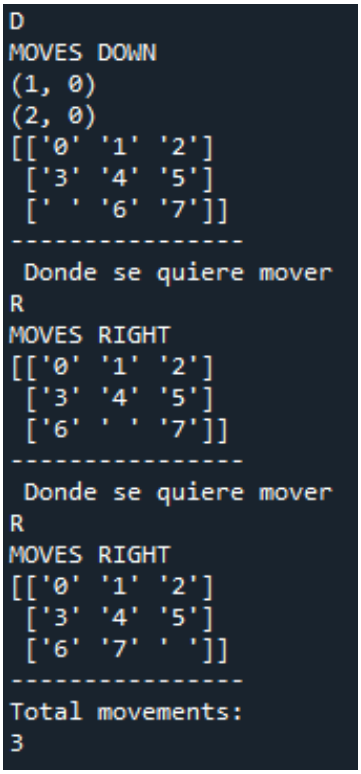

Figura 2: Tablero 3x3 fácil resuelto

4.2. Tablero 3x3 resuelto en doce movimientos

```
puzzle6=[["4", "1", " "],
         ["0", "6", "2"],
         ["7", "3", "5"]]
```

Figura 3: Segundo tablero 3x3 fácil sin resolver

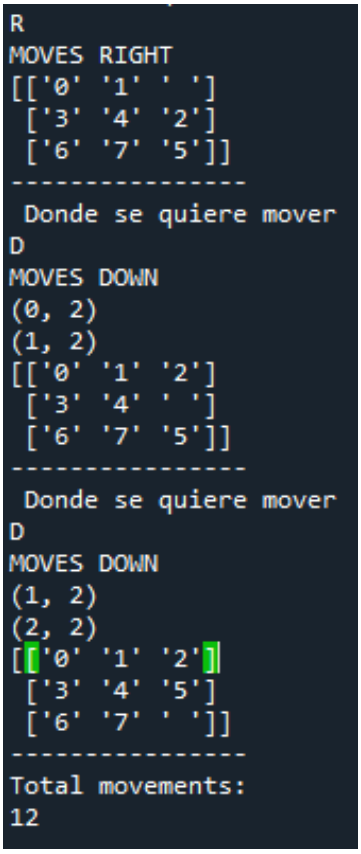

Figura 4: Segundo tablero 3x3 fácil resuelto

4.3. Tablero 3x3 resuelto en catorce movimientos

```
puzzle7=[["3", "7", "0"],
         ["6", "4", "1"],
         [" ", "5", "2"]]
```

Figura 5: Tercer tablero 3x3 fácil sin resolver

```
R
MOVES RIGHT
[['0' '1' ' ']
 ['3' '4' '2']
 ['6' '7' '5']]
---------------
 Donde se quiere mover
D
MOVES DOWN
(0, 2)
(1, 2)
[['0' '1' '2']
 ['3' '4' ' ']
 ['6' '7' '5']]
---------------
 Donde se quiere mover
D
MOVES DOWN
(1, 2)
(2, 2)
[['0' '1' '2']
 ['3' '4' '5']
 ['6' '7' ' ']]
---------------
Total movements:
14
```

Figura 6: Tercer tablero 3x3 fácil resuelto

4.4. Tablero 3x3 resuelto en dieciocho movimientos

Desde este tablero en adelante, los tableros empiezan a tardar considerablemente más, este tardó treinta y dos segundos en completarse utilizando como CPU un Intel i5 9600K corriendo a 4.6Ghz, cabe resaltar que los anteriores tableros no tardaban ni un segundo en resolverse.

```
puzzle4=[["1", "5", "6"],
         ["0", " ", "7"],
         ["4", "2", "3"]]
```

Figura 7: Cuarto tablero 3x3 sin resolver

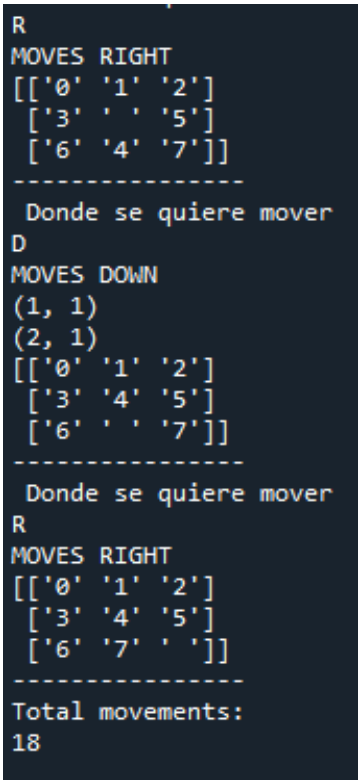

Figura 8: Cuarto tablero 3x3 resuelto

4.5. Tablero 3x3 resuelto en veinte movimientos

Este tablero tardó en completarse dos minutos, cuarenta y ocho segundos.

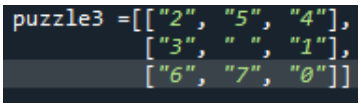

Figura 9: Quinto tablero 3x3 sin resolver

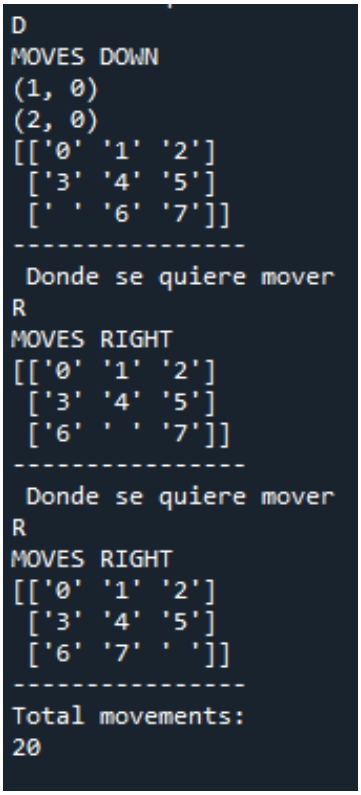

Figura 10: Quinto tablero 3x3 resuelto

4.6. Tablero 3x3 resuelto en veintidos movimientos

Este tablero tardó en completarse cuatro minutos y treinta segundos.

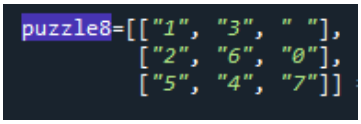

Figura 11: Sexto tablero 3x3 sin resolver

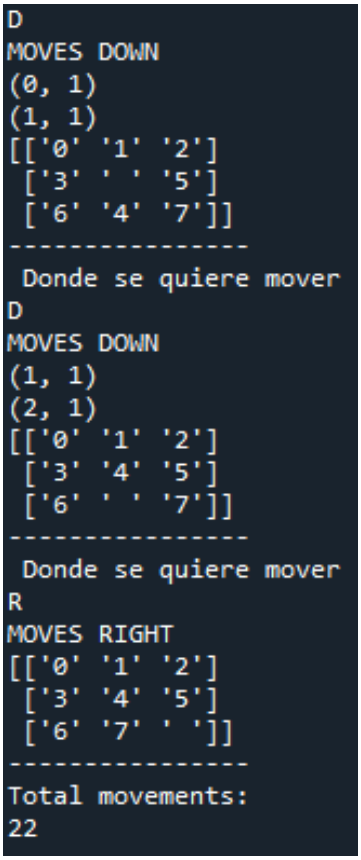

Figura 12: Sexto tablero 3x3 resuelto

4.7. Tablero 3x3 resuelto en veinticuatro movimientos

Este tablero tardó en completarse diez minutos y diez segundos.

```
puzzle=[["5", "1", "7"],
        ["3", "6", "0"],
        [" ", "2", "4"]]
```

Figura 13: Séptimo tablero 3x3 difícil sin resolver

```
D
MOVES DOWN
(0, 1)
(1, 1)
[['0' '1' '2']
 ['3' ' ' '5']
 ['6' '4' '7']]
---------------
 Donde se quiere mover
D
MOVES DOWN
(1, 1)
(2, 1)
[['0' '1' '2']
 ['3' '4' '5']
 ['6' ' ' '7']]
---------------
 Donde se quiere mover
R
MOVES RIGHT
[['0' '1' '2']
 ['3' '4' '5']
 ['6' '7' ' ']]
---------------
Total movements:
24
```

Figura 14: Séptimo tablero 3x3 difícil resuelto

4.8. Tablero 4x4 resuelto en trece movimientos

Este tablero tardó en completarse cuarenta y siete segundos.

```
puzzle8=[["1", "3", " "],
         ["2", "6", "0"],
         ["5", "4", "7"]]
```

Figura 15: Tablero 4x4 sin resolver

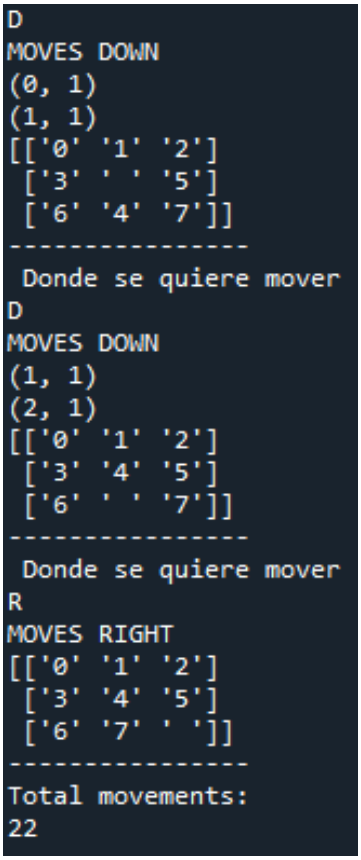

Figura 16: Tablero 4x4 resuelto

5. Parte V: Conclusión

Para este proyecto el análisis previo del problema que se realizó fue muy importante ya que nos dio indicaciones del camino a seguir para desarrollar un algoritmo que funcionara, ya que en un inicio no se tenía muy claro como abordar la gran cantidad de opciones implicadas en la resolución del tablero de taquin. Una vez finalizado el algoritmo para resolver el tablero de taquin se pudo comprobar que aunque si cumplía con este objetivo, las pruebas mostraban que en varios casos el tiempo que tardaba en resolverse un tablero era demasiado alto. Se pudo determinar con diferentes pruebas que esto se debe probablemente a un error en la forma en que se le da un peso a una opción por encima de otra, que es el mecanismo con el cual el algoritmo encuentra una solución, ya que en lugar de intentar todas las posibles opciones de movimiento, se prueban las más prometedoras para resolver el tablero. Se pudo verificar que aunque no se evaluan todos los posibles movimientos, cuando se tienen que resolver tableros donde es necesario intercambiar varias fichas de posición el algoritmo toma mucho más tiempo.En tableros donde los cambios a realizar son menos complicados estos se resuelven rapidamente, lo que indica que sí se le asigna una prioridad a cierto conjunto de movimientos para resolver el tablero de forma rapida, sin embargo estas medidas de prioridad comienzan a fallar a medida que los cambios que deben realizarse son mas complicados (llevar números a su posición deseada para posteriormente desordenarlos con el objetivo de ordenar otros números), esto lleva a que la resolución de varios tableros sea demasiado lenta. También se pudo verificar que la velocidad a la que se resuelven los tableros se relaciona con la distribución de los números en estos y no con la cantidad de movimientos que hay que realizar o el tamaño del tablero de forma directa.