Análisis de algoritmos : Proyecto final

Fabián Rojas[1] Julián Tarazona[1]

[1]Pontificia Universidad Javeriana
{fabian-rojasm, jdtarazona}@javeriana.edu.co
June 2, 2022

1 Resumen

En este documento se presenta el análisis, diseño e implementación de un algoritmo para un jugador automático para el juego de Taquin o sliding-puzzle haciendo uso de arboles, búsqueda en profundidad y de una función heurística.

2 Análisis y diseño

2.1 Análisis

El Taquin o Sliding puzzle, son rompecabezas que desafían al jugador a deslizar las piezas de forma que este pueda establecer un patrón objetivo. Las piezas para mover pueden ser patrones de números, letras, colores, etc. Los tableros más conocidos de este juego son los de tamaño 3x3 o 4x4.[1]

El problema del jugador automático radica en ordenar los elementos de un tablero de tamaño nxn de forma que en cada una de sus filas los elementos queden ordenados de forma ascendente, tal como se muestra en la figura 1. Para el desarrollo de este algoritmo, los elementos del tablero se definen como $C = \langle c \in \mathbb{N}; 1 \leq c \leq n * n \rangle$ donde n es el tamaño del tablero.

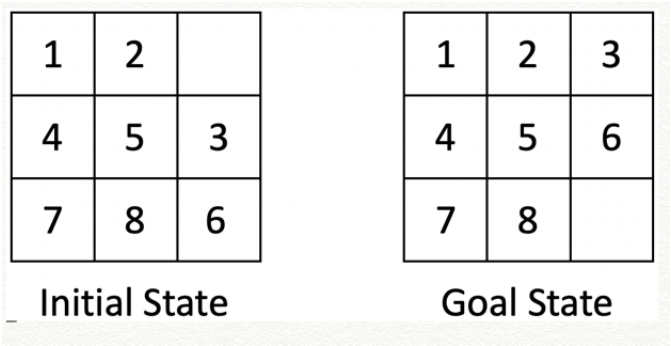

Figure 1: Ejemplo de un juego de Taquin en tablero de 3x3

El jugador automático solo puede mover las casillas que tengan a su lado un espacio vacío, pero no en sus diagonales, limitando el movimiento de las piezas en 4 posibles opciones: derecha, izquierda, arriba y abajo.

Para que el jugador automático pueda decidir que movimiento realizar que lo acerquen a la victoria se hará uso de árboles, búsqueda en profundidad y de una función heurística. Para la estructura del árbol se define que los nodos estarán constituidos por los siguientes elementos:

- El tablero de Taquin.
- El movimiento que se realizo para llegar al estado actual del tablero.

- El estado del tablero anterior.
- Todos los posibles movimientos que se pueden realizar con el estado del tablero actual.

El árbol tendrá un máximo de 4 niveles para limitar el espacio en memoria ocupado. En el cuarto nivel del árbol se encontrarán los nodos hoja que contarán con su respectivo cálculo de la heurística, cabe aclarar que en los niveles superiores no se realiza el cálculo de dicha función.

Para la función heurística se calcula la distancia a la cual se encuentra cada pieza de su posición objetivo, se muestra un ejemplo en la figura 2. Cuando ya se tengan todos estos valores, se procede a realizar la sumatoria de los valores que dará como resultado el valor de la función heurística.

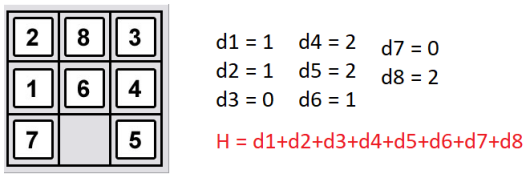

Figure 2: Ejemplo de heuristica en un tablero de 3x3

Al tener el árbol construido con todos sus respectivos nodos, se realiza una búsqueda en profundidad de los nodos hojas, al encontrar estos nodos se toma aquel que tenga el valor de la función heurística más bajo, de este se realiza backtracking con el fin de obtener los movimientos que se realizaron para llegar a ese nodo hoja. Este proceso se repite hasta que se encuentre el patrón ganador del juego.

2.2 Diseño

1. **Entradas:**
   - Una matriz cuadrada de tamaño n dentro de la cual hay un conjunto de elementos C definidos como $C = \langle c \in \mathbb{N}; 1 \leq c \leq n * n \rangle$
2. **Salidas:**
   - Una lista de acciones que permiten ganar el juego.

3 Algoritmos

Algorithm 1 Jugador automático

```
1:  procedure JUGAR(tablero)
2:      pila_de_movimientos ← ∅
3:      arbol ← Nodo(tablero)
4:      while ¬seHaGanado() do
5:          extenderArbol(arbol, 2)
6:          nodos4toNivel ← ∅
7:          buscarHijos4toNivel(arbol, nodos4toNivel)
8:          min_heuristica ← −∞
9:          nodoEscogido ← ∞
10:         for hijo ∈ nodos4toNivel do
11:             if hijo.heuristica < minHeuristica then
12:                 nodoEscogido ← hijo
13:             end if
14:         end for
15:         arbol ← nodoEscogido
16:     end while
17:     crearPilaDeMovimientos(nodoEscogido, pila_de_movimientos)
18:     Reverse(pila_de_movimientos)
19:     return pila_de_movimientos
20: end procedure
```

En el algoritmo 1 se muestra el procedimiento que realiza el jugador automático para jugar Taquin. Comienza tomando el estado inicial del tablero y procede a crear el árbol con todas las posibles jugadas en la función extenderArbol, cuando se finaliza la creación del árbol se procede a buscar los nodos hojas del cuarto nivel que tendrán los valores de la función heurística para cada estado del tablero. Al conocer todos los nodos hoja se busca cual de estos tiene el valor de la heurística mas bajo y partir de este crea la pila de movimientos realizando el backtracking hasta el nodo raíz. Por ultimo la función reverse invierte el orden de la pila de movimientos.

Algorithm 2 Calculo de heuristica

```
1:  procedure HEURISTICA(tablero)
2:      H ← 0
3:      for i ← 1to|tablero| ∗ |tablero| do
4:          PosCasillaActual ← coordenadas(fila, columna) de i
5:          PosCasillaGanador ← coordenadas(fila, columna) de i
6:
7:          DistFila ← |PosCasillaGanador[0] − PosCasillaActual[0]|
8:          DistColumna ← |PosCasillaGanador[1] − PosCasillaActual[1]|
9:
10:         H ← H + (DistFila + DistColumna)
11:     end for
12:     return H
13: end procedure
```

Algorithm 3 Extensión del árbol

```
1: procedure EXTENDERARBOL(nodo, nivel)
2:     movimientos ← getMovimientosPosibles()
3:     if nivel = 4 then
4:         for movimiento ∈ movimientos do
5:             hijo ← hacerMovimiento(movimiento, nodo.tablero)
6:             if ¬nodo.padre.tablero = hijo then
7:                 nodo.hijos.add(Nodo(hijo, movimiento, nodo, heuristica(hijo)))
8:             end if
9:         end for
10:    else
11:        for movimiento ∈ movimientos do
12:            hijo ← hacerMovimiento(movimiento, nodo.tablero)
13:            if ¬nodo.padre = ∅ then
14:                if ¬nodo.padre.tablero = hijo then
15:                    nuevoNodo ← Nodo(hijo, movimiento, nodo, ∅)
16:                    nodo.hijos.add(nuevoNodo)
17:                    extenderArbol(nuevoNodo, nivel + 1)
18:                end if
19:            else
20:                nuevoNodo ← Nodo(hijo, movimiento, nodo, ∅)
21:                nodo.hijos.add(nuevoNodo)
22:                extenderArbol(nuevoNodo, nivel + 1)
23:            end if
24:        end for
25:    end if
26: end procedure
```

En el algoritmo 3 se muestra el procedimiento para crear el árbol de posibles jugadas. Se comienza creando una lista con los movimientos que se pueden ejecutar, para validar que estos son posibles se utiliza la función getMovimientosPosibles que verifica según la posición de la casilla vacía que movimientos son prohibidos y los remueve de la lista de movimientos. Continua con la parte recursiva del algoritmo, en esta sección se crean tableros a partir de la lista de movimientos y dichos tableros se convierten en nodos para luego ser incluidos en la lista de posibles movimientos del nodo actual. Cuando se llega al nivel 4 del árbol se repite el procedimiento anterior pero con la diferencia de que los tableros hijos se les incluye el valor de su función heurística.

4 Implementación

Anexo a este documento se encuentra el algoritmo del juego Taquin en el lenguaje Python.

5 Complejidad

Para el calculo de la complejidad de este algoritmo se tuvo en cuenta la complejidad de crear el árbol de posibles jugadas, esto tendría un resultado de $O(n^3)$ siendo n la cantidad de hijos que puede tener un nodo (máximo 4 hijos ya que son solo 4 movimientos) y que el arbol tiene 3 niveles sin contar la raiz del arbol. La búsqueda en profundidad que se debe realizar para hallar los nodos hoja del árbol. Dicha complejidad sería un valor de O(v) siendo v la cantidad de nodos del arbol. Por lo tanto la complejidad del algoritmo seria $O(v * n^3)$.

6 Resultados obtenidos

Tras realizar la implementación del algoritmo se pudo evidenciar que este no lograba llegar al estado objetivo del juego, por lo que se piensa que la razón de esto sea por la función heurística, la cual no tiene en cuenta que, para llevar a ciertas casillas a sus posiciones, se debe mover otras que ya estaban en su posición objetivo. Esto hace que el algoritmo se mantenga en un ciclo infinito.

7 Bibliografía

[1] “Sliding puzzle”, Wikipedia. el 29 de mayo de 2022. Consultado: el 31 de mayo de 2022. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Sliding$_puzzleoldid = 1090507304$