Number Link | Proyecto Final de Análisis de Algoritmos

Diego Alejandro Albarracín Maldonado
Jorge Esteban Martínez Clavijo

04 de junio de 2025

Abstract

En este documento encontrará el análisis, diseño, pseudocódigo, invariante, complejidad y pruebas unitarias relacionadas al algoritmo que se propone como solución al problema de implementar un jugador automático para una representación en el lenguaje de programación *Python* del juego *NumberLink*.

Parte I
Análisis y diseño del problema

1 Análisis

El juego Number Link consiste, de manera informal, en conectar automáticamente todos los pares de números iguales ubicados en una matriz de tamaño nn. Cada par debe ser conectado mediante una línea única continua, sin que estas líneas se sobrepongan entre sí ni atraviesen otras celdas que contengan números. Los números deben ubicarse exclusivamente al inicio y al final de cada línea y como condición final todas las celdas del tablero estén ocupadas.

Este problema se clasifica como un problema $NP - Completo$, por lo que no existe un algoritmo eficiente conocido que garantice encontrar la solución óptima en todos los casos. Por ello, se adopta una estrategia basada en probar permutaciones de pares de puntos, dado que el orden en que se conectan los pares afecta el resultado final.

2 Diseño

Con las observaciones presentadas en el análisis anterior, podemos escribir el diseño de un algoritmo que solucione el problema . A veces este diseño se co-

noce como el *contrato* del algoritmo, o las *precondiciones* y *pos-condiciones* del algoritmo. El diseño se compone de las siguientes entradas y salidas:

Definición. Entradas:

1. Objeto $Tablero$ de tamaño $n*m$, $Tablero = \langle\langle x_1, x_2, ..., x_n\rangle, \langle y_1, y_2, ..., y_n\rangle\rangle$ y una secuencia de pares de elementos que representan los elementos a conectar dentro del tablero, $P = \langle\langle x_i, y_i\rangle, ..., \langle x_n, y_n\rangle\rangle$ en donde $\forall_i X_i \wedge Y_i \in \mathbb{N}$ y $\mathbb{N}$ y $x_i < m$ y $y_i < n$ y $|P|$.

Definición. Salidas:

1. Objeto $Tablero$ de tamaño $n*m$, $Tablero = \langle\langle x_1, x_2, ..., x_n\rangle, \langle y_1, y_2, ..., y_n\rangle\rangle$ con los elementos dados en la secuencia $P = \langle\langle x_i, y_i\rangle, ..., \langle x_n, y_n\rangle\rangle$ conectados y que los caminos que conectan a los elementos dados en la secuencia P no se sobrepongan.

Parte II
Algoritmos

3 Jugador automático

3.1 Cambios con Respecto a la Primera Entrega

Para la primera entrega, se tenía una implementación donde se manejaban variables globales y el tablero de juego se representaba como una matriz; en esta nueva entrega se creó una clase *Tablero*, en la que se maneja toda la lógica de movimientos, restricciones y demás funciones del tablero.

3.2 Descripción y Pseudocódigo

Primero, se crea una instancia de la clase Tablero llamada tablero_original, cargando el contenido desde un archivo con el método load_table. Luego, mediante la función encontrar_pares(tablero_original), se obtienen todas las permutaciones posibles de pares de posiciones que deben conectarse. Cada permutación representa un orden distinto para intentar resolver el tablero.

- Convertir la permutación en una lista de pares pares.
- Copiar el tablero original y almacenarlo en tablero_temp con el objetivo de trabajar con él sin modificar el tablero original.
- Inicializar elementos auxiliares: bloqueados (lista para almacenar posiciones bloqueadas), intentos (contador de los intentos para resolver la permutación actual), e i (índice para recorrer los pares de permutación).

Dentro de un bucle while, se intenta conectar los puntos de cada par usando el algoritmo A* (*astar()*), evitando posiciones bloqueadas. Si no se encuentra camino, se descarta la permutación y se pasa a la siguiente.

Si se encuentra un camino, se actualiza el tablero temporal, registrando las coordenadas usadas, aplicando los movimientos con *revisar_movimiento()* y marcando la conexión con *aumentar_conectados()*. Luego, se verifica si la situación del tablero impide continuar (*salidas_pares()*); si hay bloqueos, se reinicia la búsqueda desde cero con el tablero original.

Finalmente, si todos los pares se conectan exitosamente y el tablero queda completo, se valida con *check_terminado()* y se retorna la solución.

Algorithm 1 JugadorAutomático

```
1: procedure JUGAR(tablero_original)
2:     permutaciones ← ENCONTRARPARES(tablero_original)
3:     for perm_index ← 1 to |permutaciones| do
4:         perm ← permutaciones[perm_index]
5:         pares ← lista de pares de perm
6:         tablero_temp ← copia de tablero_original
7:         bloqueados ← [ ]
8:         i ← 0
9:         while i < |pares| do
10:            (dato, (inicio, fin)) ← pares[i]
11:            copia ← copia de tablero_temp.matriz
12:            camino ← ASTAR(copia, dato, bloqueados, inicio, fin)
13:            if camino = None then
14:                break
15:            end if
16:            (fila, col) ← inicio
17:            Agregar inicio a tablero_temp.coordenadas_usadas
18:            for all mov ∈ camino do
19:                direccion ← DIR_INVERSA(mov)
20:                (_, fila, col) ← REVISARMOVIMIENTO(fila, col, direccion,
   dato)
21:            end for
22:            AUMENTARCONECTADOS(dato, fin) en tablero_temp
23:            i ← i + 1
24:            if SALIDASPARES(bloqueados, tablero_temp, pares) then
25:                tablero_temp ← copia de tablero_original
26:                i ← 0
27:            end if
28:        end while
29:        if CHECKTERMINADO(tablero_temp) then
30:            return tablero_temp
31:        end if
32:    end for
33:    return None
34: end procedure
```

Algoritmo A* con heurística de distancia Manhattan, que permite estimar eficientemente la distancia entre dos posiciones considerando solo movimientos horizontales y verticales, tal como lo permite el tablero.

Algorithm 2 AStar

```
1: procedure ASTAR(tablero, dato, bloqueados, inicio, fin)
2:     heap ← lista vacía
3:     PUSH(heap, (heurística(inicio, fin), 0, inicio, [ ]))
4:     visitados ← ⟨⟩
5:     while |heap| > 0 do
6:         (f, g, actual, camino) ← POP(heap)
7:         if actual ∈ visitados then
8:             continue
9:         end if
10:        Agregar actual a visitados
11:        if actual = fin then
12:            return camino
13:        end if
14:        for all delta ∈ VALORES(MOVIMIENTOS) do
15:            nueva_pos ← (actual[0] + delta[0], actual[1] + delta[1])
16:            if nueva_pos está dentro de los límites del tablero then
17:                if nueva_pos ∉ visitados y (dato, nueva_pos) ∉ bloqueados
    then
18:                    celda ← tablero[nueva_pos[0]][nueva_pos[1]]
19:                    if celda = 0 o nueva_pos = fin then
20:                        nuevo_g ← g + 1
21:                        nuevo_f ← nuevo_g+ HEURÍSTICA(nueva_pos, fin)
22:                        PUSH(heap, (nuevo_f, nuevo_g, nueva_pos, camino+
    [delta]))
23:                    end if
24:                end if
25:            end if
26:        end for
27:    end while
28:    return None
29: end procedure
```

Algoritmo encargado de bloquear el punto que menor heuristica tenga respecto al punto final.

Algorithm 3 BloquearMenorHeuristica

```
1: procedure BloquearMenorHeuristica(fin, bloqueos)
2:     for all (bloqueos_aux, libre) ∈ bloqueos do
3:         if libre = 0 then
4:             menor ← None
5:             min_heur ← infinito
6:             for all (dato, punto) ∈ bloqueos_aux do
7:                 h ← Heurística(fin, punto)
8:                 if h < min_heur then
9:                     min_heur ← h
10:                    menor ← (dato, punto)
11:                end if
12:            end for
13:            return menor
14:        end if
15:    end for
16:    return None
17: end procedure
```

Algoritmo que permite revisar si un número está completamente encerrado por otro y a pesar de haber bloqueado el primer camino, permite abrir otro paso bloqueando el camino bloqueador, para ajustar las rutas.

Algorithm 4 RevisarMovimientoDesdeBloqueo

```
1: procedure RevisarMovimientoDesdeBloqueo(tablero, bloqueado)
2:     (f, c) ← bloqueado
3:     fila ← Longitud(tablero)
4:     colu ← Longitud(tablero[0])
5:     aux ← [ ]
6:     salidas ← 0
7:     for all (df, dc) ∈ Valores(MOVIMIENTOS) do
8:         (nf, nc) ← (f + df, c + dc)
9:         if 0 ≤ nf < fila y 0 ≤ nc < colu y (nf, nc) ≠ bloqueado then
10:            dato_revisar ← tablero[nf][nc]
11:            if dato_revisar = 0 then
12:                salidas ← salidas + 1
13:            end if
14:            if dato_revisar < 0 then
15:                Agregar (Abs(dato_revisar), (nf, nc)) a aux
16:                Imprimir "Siguiente bloqueado dato dato_revisar por camino (nf, nc)"
17:            end if
18:        end if
19:    end for
20:    if salidas = 0 then
21:        return aux[0]
22:    end if
23:    return None
24: end procedure
```

Algoritmo que permite evaluar si cada par de puntos tiene al menos una salida disponible.

Algorithm 5 SalidasPares

```
1: procedure SalidasPares(bloqueados, copia, pares)
2:     for all (dato, (inicio, fin)) ∈ DeepCopy(pares) do
3:         for all punto ∈ {inicio, fin} do
4:             bloqueos ← RevisarSalidas(dato, copia.matriz, punto)
5:             tupla_bloqueo ← BloquearMenorHeuristica(fin, bloqueos)
6:             if tupla_bloqueo ≠ None y no EstaConectado(copia, dato)
   then
7:                 desespero ← RevisarMovimientoDesdeBloqueo(copia.matriz, tupla_bloqueo[1])
8:                 if desespero ≠ None then
9:                     Agregar desespero a bloqueados
10:                end if
11:                Agregar tupla_bloqueo a bloqueados
12:                return True
13:            end if
14:        end for
15:    end for
16:    return False
17: end procedure
```

Algoritmo que permite identificar las salidas disponibles que tiene un número, revisando si las celdas adyacentes de la matriz tienen un valor de cero, regresando una secuencia de puntos donde hay bloqueos y la cantidad de caminos libres.

Algorithm 6 RevisarSalidas

```
1: procedure RevisarSalidas(dato, tablero, coord)
2:     (f, c) ← coord
3:     fila ← Longitud(tablero)
4:     colu ← Longitud(tablero[0])
5:     bloqueados ← [ ]
6:     libre ← 0
7:     bloqueados_aux ← [ ]
8:     for all (df, dc) ∈ Valores(MOVIMIENTOS) do
9:         (nf, nc) ← (f + df, c + dc)
10:        if 0 ≤ nf < fila y 0 ≤ nc < colu then                ▷ Revisar límites
11:            dato_revisar ← tablero[nf][nc]
12:            if dato_revisar = 0 then
13:                libre ← libre + 1
14:            end if
15:            if dato_revisar < 0 y dato_revisar ≠ −dato then        ▷ Solo
    bloquea si es otro dato
16:                Agregar (Abs(dato_revisar), (nf, nc)) a bloqueados_aux
17:            end if
18:        end if
19:    end for
20:    Agregar (bloqueados_aux, libre) a bloqueados
21:    return bloqueados
22: end procedure
```

Permite calcular las permutaciones de puntos.

Algorithm 7 Permutaciones

```
1: procedure Permutaciones(diccionario)
2:     pares ← [ ]
3:     for all k ∈ Claves(diccionario) do
4:         Agregar (k, diccionario[k]) a pares
5:     end for
6:     return Permutaciones(pares)
7: end procedure
```

Heurística de Manhattan permite medir la distancia a dos puntos entre la matriz, permitiendo considerar solo movimientos válidos de forma vertical y horizontal.

Algorithm 8 Heuristica

```
1: procedure Heuristica(a, b)
2:     return Abs(a[0] − b[0]) + Abs(a[1] − b[1])
3: end procedure
```

3.3 Complejidad

La complejidad del algoritmo propuesto para dar solución a la implementación de un jugador automático es de $O(n! * n * m * k)$, en donde n es el número de pares que deben ser conectados, m es el número de filas del tablero y k es el número de columnas del tablero. La complejidad en el peor de los casos llega a ser factorial respecto al número de pares ($n!$) y lineal respecto al tamaño del tablero ($m * k$), en otras palabras, la solución propuesta puede, en el peor de los casos, llegar a ser muy costosa a nivel computacional para tableros muy grandes con demasiados pares, ya que el número de permutaciones crece rápidamente.

3.4 Invariante

- **Inicio**: El tablero inicial y la secuencia de pares P están definidos, y ningún elemento está conectado.
- **Avance**: Durante la ejecución, el algoritmo conecta los pares en P respetando las restricciones del tablero y evitando caminos sobrepuesto.
- **Terminación**: El tablero final refleja las conexiones realizadas, cumpliendo las restricciones, o permanece sin cambios si no se encuentra una solución válida.

3.5 Notas de Implementación

El algoritmo fue implementado en el lenguaje de programación *Python*. Para conocer más detalles sobre la implementación, en los archivos anexados al trabajo diríjase al directorio `AA_Proyecto2/` y revise los archivos `interfaz.py` y `tablero.py`, además de los archivos que contienen los datos que le permitirán probar el algoritmo: `prueba1.txt`, `prueba2.txt`, `prueba3.txt`, `prueba4.txt`, `prueba5.txt` y `prueba6.txt`.

Parte III
Pruebas unitarias

Para probar el correcto funcionamiento de la solución propuesta, se utilizaron los ejemplos de entrada `prueba1.txt` y `prueba2.txt` brindados para el desarrollo del proyecto de la asignatura. Las pruebas unitarias comprueban que cada una de las funcionalidades expuestas en la sección *Algoritmos* funcionan correctamente. Las coordenadas que permiten conectar el table exitosamente para cada caso de prueba se muestran a continuación:

3.6 Archivo `prueba1.txt`

El archivo contiene elementos en el siguiente formato:

- 3,3
- 1 ,2, 1
- 1 ,3, 1
- 1 ,1, 2
- 0 ,2, 2
- 2 ,2, 3
- 3 ,3, 3

Para el archivo `prueba1.txt` se esperan las siguientes conexiones como resultado:

- 1 => (1,2), (1,3).
- 2 => (1,1), (2,1), (3,1), (3,2).
- 3 => (2,2), (2,3), (3,3).

3.7 Archivo `prueba2.txt`

El archivo contiene elementos en el siguiente formato:

- 5,5
- 2 ,1, 1
- 3 ,4, 1
- 2 ,2, 2
- 4 ,4, 2
- 3 ,1, 3
- 5 ,1, 3
- 3 ,2, 4
- 5 ,3, 4
- 1 ,5, 5
- 5 ,4, 5

Para el archivo `prueba2.txt` se esperan las siguientes conexiones como resultado:

- 1 => (2,1), (1,1), (1,2), (1,3), (1,4), (2,4), (3,4).

- 2 => (2,2), (2,3), (3,3), (4,3), (4,4)
- 3 => (3,1), (4,1), (5,1)
- 4 => (3,2) , (4,2), (5,2), (5,3)
- 5 => (5,4), (5,5), (4,5), (3,5), (2,5), (1,5)

3.8 Archivo `entrada.txt`

El archivo contiene elementos en el siguiente formato:

- 7 ,7
- 1 ,4 ,4
- 3 ,4 ,3
- 2 ,2 ,3
- 2 ,5 ,2
- 2 ,6 ,5
- 3 ,5 ,1
- 4 ,4 ,5
- 6 ,3 ,1
- 7 ,1 ,2
- 7 ,5 ,4

Para el archivo `entrada.txt` se esperan las siguientes conexiones como resultado:

- 1 => (3,5), (3,4), (3,3), (3,2), (3,1), (4,1), (5,1), (6,1), (6,2), (6,3)
- 2 => (2,5), (2,4), (2,3), (2,2), (2,1), (3,1), (4,1), (5,1), (6,1), (7,1)
- 3 => (2,2), (2,3), (3,3), (3,4)
- 4 => (1,4), (1,5), (2,5), (3,5), (4,5), (5,5), (6,5), (7,5)
- 5 => (2,6), (3,6), (4,6), (4,5), (4,4)

3.9 Resultados de las Pruebas

En la *Figura3.1* se puede ver la forma en que se validan todos los casos de prueba para ambos ejemplos y es posible confirmar que, en efecto, el código da un resultado de salida esperado, al menos para estos ejemplos. A continuación se muestra el resultado en pantalla de las pruebas unitarias, teniendo en cuenta la información presentada previamente.

Figura 3.1: Resultados en consola de comandos de la ejecución de las pruebas unitarias.

3.10 Notas de Implementación

Las pruebas unitarias fueron desarrolladas en el lenguaje de programación *Python*. Para conocer más detalles sobre la implementación, en los archivos anexados al trabajo o en el repositorio de *Git hub* diríjase al directorio `AA_Proyecto2/` y diríjase al archivo `unit_tests_interfaz.py`.