Proyecto Number Link: Algoritmo de solución*

Katheryn Guasca Chavarro[a], Simon Diaz Monroy[b], Melissa Ruiz Barrera[c]

[a] *Pontificia Universidad Javeriana*
[b] *Pontificia Universidad Javeriana*
[c] *Pontificia Universidad Javeriana*

Abstract

Este documento presenta la implementación de una solución al juego*Number Link*, un rompecabezas NP-completo se utiliza una estrategia que combina métodos de búsqueda en grafos: por un lado, se emplean técnicas de *búsqueda no informada*, como BFS y DFS, para explorar las rutas que conectan cada par de números. Por otro lado, se incorpora un componente de *búsqueda informada* mediante una heurística que determina el orden en que se procesan los pares, priorizando aquellos que generan mayores restricciones sobre el tablero. Esta combinación permite reducir la exploración redundante y mejorar la eficiencia general del proceso de búsqueda.

Keywords: Number Link, backtracking, búsqueda informada, algoritmos, heurística

1. Análisis del problema

El juego *Number Link* se considera un problema de decisión combinatorio donde se analiza un subconjunto de pares que cumplen ciertas restricciones de clase np-completo.

Puede ser modelado como un problema de búsqueda en grafos, donde cada celda del tablero corresponde a un nodo y los movimientos posibles (arriba, abajo, izquierda y derecha) representan aristas. El objetivo es conectar cada par de números iguales mediante un camino simple, sin ciclos y sin que dos caminos compartan celdas.

Las decisiones tempranas en la construcción de los caminos afectan directamente la posibilidad de completar el resto del tablero, lo que produce un crecimiento exponencial del espacio de búsqueda. Cada estado del problema

*En este documento presenta soluciones algorítmicas para el problema de contar el número de subsecuencia palíndromas teniendo en cuenta la elección de índices

Email addresses: ksofia.guasca@javeriana.edu.co (Katheryn Guasca Chavarro), simondiaz@javeriana.edu.co (Simon Diaz Monroy), melissafruizb@javeriana.edu.co (Melissa Ruiz Barrera)

representa una configuración parcial con algunos caminos trazados y otros pendientes, mientras que los estados objetivo son aquellos donde todos los pares han sido conectados válidamente.

2. Diseño del problema

2.1. Entradas

- n, dimensión del tablero.
- T, tablero representado como una matriz de dimensión $n \times n$.

2.2. Salidas

- T', tablero solucionado si existe una conexión válida para todos los pares; de lo contrario, un tablero sin solución.

3. Solución algorítmica

Los casos base surgen cuando el tablero está completamente resuelto o cuando una configuración impide continuar sin violar las reglas. En los casos recurrentes, la búsqueda consiste en extender los caminos disponibles guiándose por la heurística reduciendo las combianciones.

La heurística empleada es:

$$H(p_1, p_2) = 1000\,E + 100\,B + (100 - d_M),$$

donde E y B indican si las posiciones del par están en esquinas o bordes, y d_M es la distancia Manhattan. Esta función prioriza los pares más restrictivos, ya sea por su ubicación o por su cercanía. Si bien las rutas se exploran mediante búsqueda en anchura (BFS) y búsqueda en profundidad (DFS), la heurística introduce un componente de *búsqueda informada* al ordenar previamente los pares, lo que reduce la exploración innecesaria y mejora la eficiencia general.

Algorithm 1 Se ordenan los pares de acuerdo a la heuristica establecidas

```
1: procedure OrdenarParesPorHeurística(T, n)
2:     pares ← se encuentran los pares T
3:     n ← número de filas de T
4:     m ← número de columnas de T
5:     P ← [ ]
6:     L ← |pares|
7:     for i ← 1 to |L| do
8:         (k, p1, p2) ← pares[i]
9:         (f1, c1) ← p1
10:        (f2, c2) ← p2
11:        E ← 0
12:        if (f1 = 0 ∨ f1 = n − 1) ∧ (c1 = 0 ∨ c1 = m − 1) then
13:            E ← E + 1
14:        end if
15:        if (f2 = 0 ∨ f2 = n − 1) ∧ (c2 = 0 ∨ c2 = m − 1) then
16:            E ← E + 1
17:        end if
18:        B ← 0
19:        if f1 = 0 ∨ f1 = n − 1 ∨ c1 = 0 ∨ c1 = m − 1 then
20:            B ← B + 1
21:        end if
22:        if f2 = 0 ∨ f2 = n − 1 ∨ c2 = 0 ∨ c2 = m − 1 then
23:            B ← B + 1
24:        end if
25:        d ← |f1 − f2| + |c1 − c2|
26:        H ← 1000E + 100B + (100 − d)
27:        P ← ∪(k, p1, p2, H)
28:    end for
29:    return Sort(P)
30: end procedure
```

Construye recursivamente todos los caminos entre dos extremos hasta un límite L, avanza a las celdas vacías marcando las casillas visitadas y expande desde su posición actual a cada celda en su vecindario.

Algorithm 2 Construcción de los caminos en DFS

```
1: procedure DFS_RUTAS(nodo_i)
2:     if nodo_i = nodo_d then
3:         return C[ ] ← camino
4:     end if
5:     nodo_i ← visitado
6:     (i, j) ← nodo_i                                  ▷ Ubicación del nodo
7:     dir ← [(−1, 0), (1, 0), (0, −1), (0, 1)]   ▷ Direcciones que puede tomar el
   nodo
8:     for k ← 0 to 3 do
9:         (di, dj) ← dir[k]
10:        nodo_aux ← (i + di, j + dj)
11:        if nodo_aux en el tablero then
12:            if nodo_aux ∉ visitados then
13:                nodo_aux ∪ C[ ]
14:                nodo_aux ← visitado
15:                DFS_RUTAS(nodo_aux)
16:                C[] ← ∩ nodo_aux
17:                nodo_aux ← desmarcado
18:            end if
19:        end if
20:    end for
21:    nodo_i ← desmarcado
22: end procedure
```

GenerarCaminosIncremental explora rutas con BFS desde *inicio* hasta fin generando primero las más cortas. Arranca una cola con el camino inicial y, mientras haya elementos y no supere el límite L, toma el primer camino: si ya llegó a fin, lo entrega y sigue; si no, expande a vecinos ortogonales no visitados aún, encolando solo extensiones que pisan celdas vacías o el extremo final. En comparación a DFS_Rutas que explora en profundidad y puede generar rutas largas, BFS expande por capas de tal manera que entrega las rutas más cortas entre los extremos.

Algorithm 3 Expandir cada uno de los caminos generados por BFS

```
1: procedure GenerarCaminosIncremental(T, i, j)
2:     q ← [i]
3:     S_caminos ← [ ]
4:     while q ≠ ∅ do
5:         C[ ] ← q
6:         k ← C[ |C| − 1 ]
7:         (j, j) ← nodo
8:         if nodo = C[j] then
9:             C[ ] ← S_caminos
10:            continua con cada uno de los caminos
11:        end if
12:        dir ← [(−1, 0), (1, 0), (0, −1), (0, 1)]
13:        for k ← 0 to 3 do
14:            (di, dj) ← dir[k]
15:            nodo_aux ← (fila + di, columna + dj)
16:            if 0 ≤ nodo_aux.fila < n and 0 ≤ nodo_aux.columna < m then
17:                if nodo_aux ∉ C[ ] then
18:                    if T[nodo_aux] = ∅ ∨ nodo_aux = fin then
19:                        N_camino ← C[ ]
20:                        N_camino ∪ [ nodo_aux ]
21:                        N_camino ← q[ ]
22:                    end if
23:                end if
24:            end if
25:        end for
26:    end while
27:    return Sort(S_caminos)
28: end procedure
```

BFS recorre una región del tablero conformado por celdas transitables (vacías o extremos) desde una celda inicial: marca el id de componente en $comp_{id}$, y mientras expande por vecinos ortogonales cuenta cuántas celdas libres y cuántos extremos hay en esa componente. Al terminar devuelve ese resumen libres, extremos.

Algorithm 4 Explorar componente transitable con BFS

```
1: procedure BFS(tablero, comp_id, inicio, comp_idx, extremos, filas, cols)
2:     q ← [inicio]
3:     comp_id[inicio] ← comp_idx
4:     libres ← 0
5:     extremos_comp ← 0
6:     while q ≠ ∅ do
7:         i, j ← extraer frente de q
8:         if i, j ∈ extremos then
9:             extremos_comp ← extremos_comp + 1
10:        else if tablero[i][j] = ' ' then
11:            libres ← libres + 1
12:        end if
13:        vecinos ← ObtenerVecinos(i, j, filas, cols) ▷ Vecinos ortogonales
14:        for (ni, nj) in vecinos do
15:            if comp_id[ni][nj] ≠ −1 then
16:                continue
17:            end if
18:            if EsTransitable(tablero, extremos, ni, nj) then  ▷ Revisar si
   se puede usar como camino
19:                comp_id[ni][nj] ← comp_idx
20:                agregar ni, nj a q
21:            end if
22:        end for
23:    end while
24:    return {libres, extremos_comp}
25: end procedure
```

AnalizarComponentes revisa si el tablero parcial sigue siendo una solución viable: toma los extremos pendientes y recorre las celdas transitables (vacías o extremos) armando componentes con BFS, contando libres y extremos por componente. Si alguna componente tiene libres pero cero extremos, o un número impar de extremos, declara el tablero inválido.

Algorithm 5 Validar paridad de extremos y conectividad de cada par

```
1: procedure AnalizarComponentes(tablero, paresRestantes)
2:     if |paresRestantes| = 0 then
3:         return true                                   ▷ Tablero válido sin pares
4:     end if
5:     filas ← |tablero|, columnas ← |tablero[0]|
6:     extremos ← RecolectarExtremos(paresRestantes)    ▷ Obtener pares pendientes
7:     comp_id ← matriz filas × columnas con −1
8:     comp_info ← [ ]                                  ▷ Libres y extremos por componente
9:     idx ← 0
10:    for i ← 1 to filas do
11:        for j ← 1 to columnas do
12:            if comp_id[i][j] = −1 and Transitable(tablero, extremos, i, j) then
13:                info ← BFS(tablero, comp_id, i, j, idx, extremos)
14:                comp_info[j] ← info
15:                idx ← idx + 1
16:            end if
17:        end for
18:    end for
                                   ▷ Validar paridad y presencia de extremos
19:    for info in comp_info do
20:        if info.libres > 0 and info.extremos = 0 then
21:            return false                              ▷ Hueco aislado
22:        end if
23:        if info.extremos es impar then
24:            return false                              ▷ Paridad inválida
25:        end if
26:    end for
                                   ▷ Verificar conectividad de pares
27:    for p1, p2 in paresRestantes do
28:        if comp_id[p1.fila][p1.col] ≠ comp_id[p2.fila][p2.col] then
29:            return false
30:        end if
31:    end for
32:    return true
33: end procedure
```

El *Backtrack* toma el tablero y los pares pendientes, incrementa el contador de intentos y, si ya no quedan pares, comprueba si el tablero está lleno. Selecciona los pares más restringidos y sus rutas candidatas; para cada candidato, pinta temporalmente una ruta y revisa que no haya cuellos, exista una conectividad de componentes válida y cada par restante tenga algún camino básico.
Si cumple todas las condiciones , recurre con el resto de pares; si alguna rama llega a solución, retorna verdadero. Si todas las rutas fallan, desmarca y prueba la siguiente; si no queda ninguna opción viable, devuelve falso.

Algorithm 6 Backtracking guiado con podas

```
1: procedure Backtrack(tab, pRest, tPares, intent, lim, verb)
2:     intent ← intent + 1                                   ▷ contador de intentos
3:     if |pRest| = 0 then
4:         return TableroLleno(tab)
5:     end if
6:     cand ← ObtenerCandidatosPares(tab, pRest, MAX_CAND, lim)   ▷ pares más restringidos
7:     if |cand| = 0 then
8:         return false
9:     end if
10:    for c ← 1 to |cand| do
11:        idx, caminos ← cand[c]                            ▷ índice del par y sus rutas
12:        sim, p1, p2 ← pRest[idx]                          ▷ símbolo y extremos del par
13:        rest ← pRest sin el elemento en idx               ▷ pares restantes tras conectar éste
14:        limCam ← mín(lim, |caminos|)                      ▷ cuántas rutas probar como máximo
15:        for k ← 1 to limCam do
16:            cam ← caminos[k]                              ▷ ruta candidata
17:            MarcarCamino(tab, cam, sim)
18:            Cuellos ← DetectarCuellos(tab, rest)
19:            Componentes ← AnalizarComponentes(tab, rest)
20:            Conectividad ← HayCaminoParaPares(tab, rest)   ▷ el par pendiente tiene caminos disponibles
21:            if not Cuellos and Componentes and Conectividad then
22:                if Backtrack(tab, rest, tPares, intent, lim, verb) then
23:                    return true
24:                end if
25:            end if
26:            DesmarcarCamino(tab, cam, sim)
27:        end for
28:    end for
29:    return false
30: end procedure
```

DetectarCuellos recorre las celdas libres (que no son extremos) y revisa su vecindario ortogonal. Marca como "cuello de botella" cualquier celda libre que tenga 0 o 1 vecino transitable. Un vecino transitable es una celda por la que se puede avanzar: está vacía o es un extremo pendiente. Una celda con pocos vecinos transitables solo puede formar un callejón sin salida o un punto de paso único, si se usa, separa el tablero en una isla o segmento aislado y bloquea la conexión de pares

Algorithm 7 Detectar cuellos de botella en el tablero

```
1: procedure DetectarCuellos(tablero, pares_restantes)
2:     if pares_restantes = ∅ then
3:         return false
4:     end if
5:     filas ← |tablero|
6:     cols ← |tablero[0]|
7:     extremos ← RecolectarExtremos(pares_restantes)        ▷ Obtener pares pendientes
8:     for i ← 0 to filas − 1 do
9:         for j ← 0 to cols − 1 do
10:            if tablero[i][j] ≠ ' ' or (i, j) ∈ extremos then
11:                continue
12:            end if
13:            vecinos_libres ← 0
14:            vecinos ← ObtenerVecinos((i, j), filas, cols)
15:            for (ni, nj) in vecinos do
16:                if tablero[ni][nj] = ' ' or (ni, nj) ∈ extremos then
17:                    vecinos_libres ← vecinos_libres + 1
18:                end if
19:            end for
20:            if vecinos_libres ≤ 1 then
21:                return true
22:            end if
23:        end for
24:    end for
25:    return false
26: end procedure
```

3.0.1. Algoritmo para verificar el tablero

$VerificarTablero$ toma el tablero y valida cada ruta: agrupa las celdas por símbolo, exige que cada ruta tenga al menos dos celdas, calcula el grado de cada celda (vecinos con el mismo símbolo) para contar extremos (grado 1) y descartar intersecciones (grado >2), comprueba que haya exactamente dos extremos por ruta

Algorithm 8 Verificar grados, extremos y continuidad de todas las rutas

```
1: procedure VerificarTablero(tablero)
2:     rutas ← ObtenerRutas(tablero)
3:     for simbolo, posiciones in rutas do
4:         if |posiciones| < 2 then
5:             return false                          ▷ ruta sin ambos extremos
6:         end if
7:         grados ← [ ]
8:         extremos ← 0
9:         for i, j in posiciones do
10:            g ← Grado(tablero, i, j) ▷ cuenta vecinos con el mismo símbolo
11:            grados ← g
12:            if g = 1 then
13:                extremos ← extremos + 1
14:            else if g > 2 then
15:                return false                      ▷ intersección detectada
16:            end if
17:        end for
18:        if extremos ≠ 2 then
19:            return false      ▷ cada ruta debe tener exactamente 2 extremos
20:        end if
21:        if not RutaConectada(tablero, simbolo, posiciones) then ▷ BFS:
    todas las celdas del símbolo están conectadas
22:            return false      ▷ celdas del símbolo no forman camino continuo
23:        end if
24:    end for
25:    return true                                   ▷ todas las rutas válidas
26: end procedure
```

3.1. Análisis de complejidad

3.1.1. OrdenarParesPorHeurstica

Hace un recorrido lineal sobre los pares en el tablero y utiliza un ordenamiento que se definira en la sección de notas de implementación por lo que su complejidad sera:

$$T(|pares|) = O(|pares|) + O(Sort(|pares|)) \tag{1}$$

3.1.2. DFS_Rutas

Cada llamada puede explorar hasta máximo 4 vecinos y como el algoritmo construye caminos simples, la profundidad máxima de la recursión está limitada por la cantidad total de celdas del tablero, pues una celda no puede visitarse más de una vez dentro de un mismo camino. En el peor de los casos, el espacio de búsqueda incluye un número exponencial de caminos posibles, por lo que la complejidad temporal del procedimiento es:

$$T(n) = O(4^n) \tag{2}$$

Donde n es el número de celdas en el tablero.

3.1.3. GenerarCaminosIncremental

Este procedimiento utiliza una búsqueda en anchura (BFS), donde la cola almacena caminos completos en lugar de nodos individuales. Cada expansión considera hasta 4 vecinos y la longitud máxima posible de un camino simple es el número total de celdas del tablero. En el peor caso, la expansión BFS debe procesar todas las rutas simples de longitud hasta n, lo que da una complejidad temporal exponencial:

$$T(n) = O(4^n) \tag{3}$$

Al igual que en el caso de DFS_Rutas, la complejidad sigue siendo exponencial en el tamaño del tablero debido al número potencial de caminos simples que se pueden generar.

3.1.4. BFS

Cada celda visitada se marca con un identificador de componente y se encola a lo sumo una vez. Expande como máximo 4 vecinos ortogonales, verificando si estos son transitables y si aún no pertenecen a una componente. El costo total está acotado por el número de celdas del tablero:

$$T(n) = O(n) \tag{4}$$

donde n representa el número total de celdas del tablero.

3.1.5. AnalizarComponentes

Este procedimiento inicializa una matriz de identificadores de componentes de tamaño *filas* $\times$ *columnas* y se examina cada celda mediante un doble ciclo anidado, lo que implica procesar a lo sumo $n \cdot m$ celdas, donde n es el número de filas y m el número de columnas del tablero.

Cuando se encuentra una celda transitable que aún no pertenece a ninguna componente, se ejecuta un BFS a partir de ella. En el peor de los casos, el algoritmo procesa todas las celdas del tablero por lo que su complejidad temporal es:

$$T(n, m) = O(n \cdot m) \tag{5}$$

3.1.6. Backtrack

Cada llamada recursiva selecciona uno de los pares pendientes, obtiene un conjunto de rutas candidatas para conectarlo y, por cada ruta, genera una nueva configuración del tablero sobre la cual se aplican diversas podas: $DetectarCuellos$, $AnalizarComponentes$ y $HayCaminoParaPares$. Cada una de estas verificaciones requiere recorrer el tablero completo en el peor caso, lo que implica un costo de:

$$T(n, m) = O(n \cdot m)$$

Para cada par, el algoritmo puede elegir entre hasta k rutas distintas, y cada elección conduce a una nueva llamada recursiva que debe resolver los $p-1$ pares restantes. De esta forma, la cantidad total de configuraciones posibles en el peor caso se comporta como:

$$T(n, m) = O(k^p \cdot n \cdot m) \tag{6}$$

Donde k son las ramificaciones del árbol dependiendo de los caminos considerados por par y p la cantidad de parejas total en el tablero.

3.1.7. DetectarCuellos

Para cada celda libre se inspeccionan como máximo 4 vecinos ortogonales con el fin de determinar cuántos de ellos son transitables. Si alguna celda tiene 0 o 1 vecino transitable, se detecta un cuello y la rama se descarta.

En el peor caso, el algoritmo debe examinar las $n \cdot m$ celdas del tablero y, para cada una, revisar su vecindario inmediato, lo cual implica un costo constante por celda. La complejidad temporal del procedimiento es:

$$T(n, m) = O(n \cdot m) \tag{7}$$

3.1.8. VerificarTablero

Valida la estructura final del tablero revisando cada ruta completa asociada a un símbolo. *ObtenerRutas* recorre el tablero una vez para agrupar las celdas por símbolo, lo que equivale a un costo lineal en el número de celdas. Luego, para cada ruta, se recorre la lista de posiciones asociadas y se calcula el grado de cada celda mediante el procedimiento *Grado*, el cual inspecciona un número constante de vecinos por celda.

Finalmente, el método *RutaConectada* realiza un recorrido en anchura sobre las celdas de esa ruta para comprobar que todas pertenecen a un único componente conexo. El costo total de la verificación está determinado por recorrer el tablero y procesar todas las celdas que pertenecen a rutas, lo que en conjunto implica un costo de:

$$T(n, m) = O(n \cdot m) \tag{8}$$

3.1.9. ***Complejidad de la heurística***

3.2. Invariante

El invariante del algoritmo garantiza que, en todo momento durante la exploración del tablero, se cumple que:

- Cada camino parcial construido es válido, es decir, no atraviesa celdas ocupadas, no sale del tablero y respeta la conexión entre puntos del mismo color.

- Ningún camino interfiere con el progreso de otro. Las celdas ya asignadas a un par de puntos se consideran bloqueadas de manera permanente.
- El estado del tablero siempre representa una configuración alcanzable sin contradicciones: cada decisión tomada mantiene la posibilidad de completar los caminos restantes.

En otras palabras, el algoritmo nunca avanza hacia una configuración imposible. La invariante asegura que, incluso al retroceder (backtracking), todas las asignaciones previas cumplen las reglas del Numberlink, manteniendo la coherencia del tablero y evitando explorar ramas inválidas.

4. Pruebas

Para las pruebas se tuvieron en cuenta tableros de diferentes dimensiones proporcionando escenarios triviales, promedio y difíciles. El entorno de pruebas utilizado tiene las siguientes especificaciones:

Especificación	**Detalle**
Modelo del equipo	MacBook Pro 16" (2019)
Chip / CPU	Intel 9 y 2.40GhZ
Memoria RAM	32 GB
Almacenamiento	512 GB SSD
Sistema Operativo	macOS Sonoma 26.0.1
Arquitectura	x86-64
Editor / IDE	Visual Studio Code
Lenguaje / Versión usada	Python 3.11

Cuadro 1: Especificaciones del entorno de pruebas

Resultados

Por medio de la siguiente tabla se presentan los resultados en tiempos de ejecución:

Entrada	Dimensión	Tiempo	N.º Intentos	N.º Pares	Completado
1	7x7	2.2968	6	5	sí
2	5x5	0.008	7	6	sí
3	9x9	90.0548	11	9	sí
4	7x7	0.0891	8	7	sí
5	8x8	11.9824	8	5	sí
6	10x10	n.a	n.a	12	no
7	10x10	n.a	n.a	6	no
8	9x9	n.a	n.a	6	no
9	2x3	0.0002	3	2	sí
10	8x8	68.0985	252	7	sí

Cuadro 2: Resultados de las entradas del experimento

Por medio de la siguiente gráfica es posible visualizar el comportamiento del algoritmo propuesto para estos escenarios:

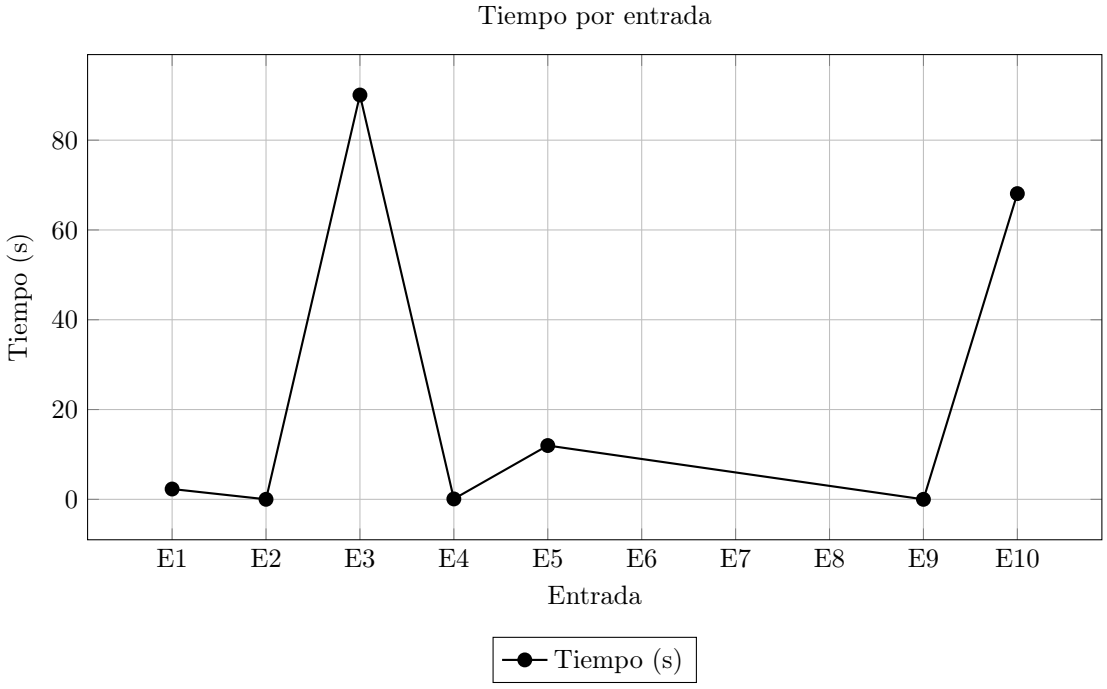

Figura 1: Tiempo de ejecución por entrada.

En la siguiente gráfica se visualiza la relación entre el número de pares en el tablero y el tiempo de ejecución:

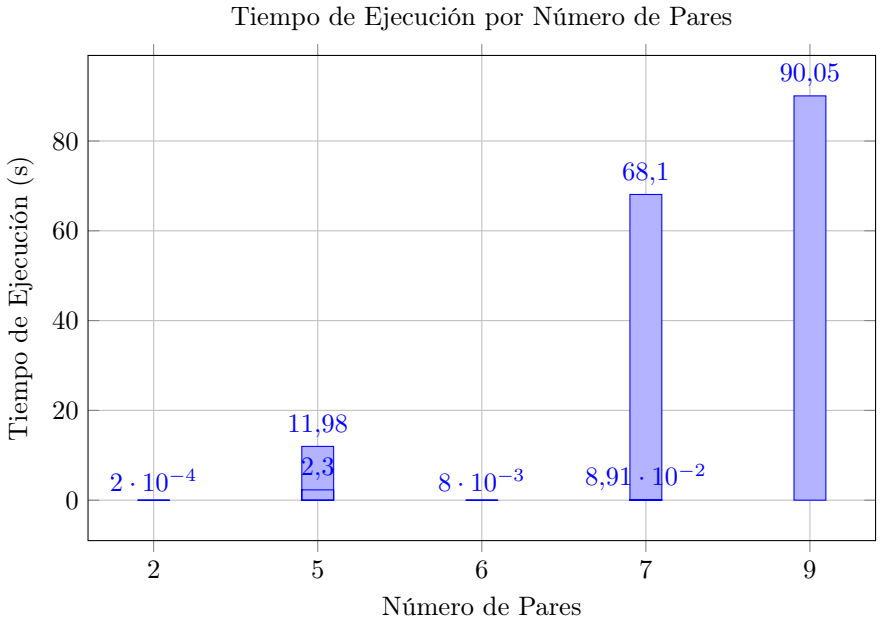

Figura 2: Tiempo de ejecución por número de pares.

En la siguiente gráfica es posible observar la tendencia de los tiempos de ejecución según la dimensión de cada tablero de prueba:

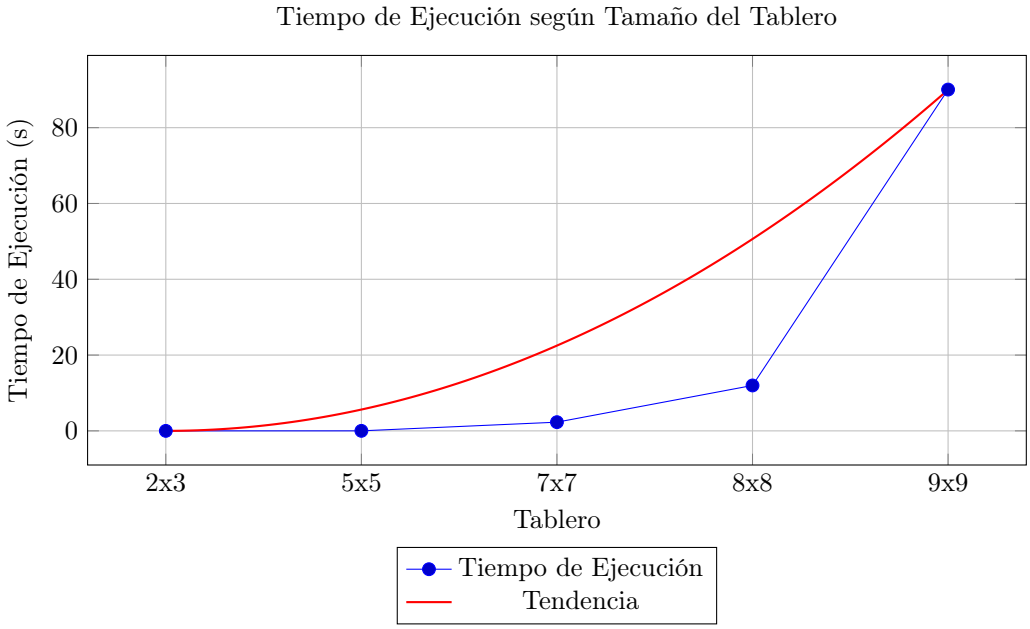

Figura 3: Tiempo de ejecución según el tamaño del tablero con línea de tendencia.

Conclusiones de los resultados

A partir de los resultados obtenidos, se puede concluir que el algoritmo propuesto maneja correctamente escenarios triviales y de dificultad promedio. Esto se evidencia en los tiempos de ejecución, siendo el menor 0.0002 segundos para un tablero de 2×3 y el mayor 90.0548 segundos para un tablero de 9×9. Sin embargo, algunos tableros de mayor dimensión, como los de 10×10, no pudieron completarse, por lo que sus tiempos y número de intentos no fueron registrados.

Al analizar la relación entre el número de pares de cada tablero y sus tiempos de ejecución (Figura 2), se observa que un mayor número de pares en tableros de gran dimensión incrementa el tiempo de procesamiento. En escenarios de dificultad promedio, los tiempos pueden variar según la dimensión del tablero (Figura 3), mostrando una tendencia aproximada semejante a una exponencial que crece con el tamaño del tablero.

En conclusión, la estrategia propuesta, basada en una heurística que prioriza esquinas, bordes y combina búsquedas en anchura y profundidad, ofrece un desempeño eficiente en escenarios básicos y de dificultad media. No obstante, en escenarios más complejos se evidencian incrementos significativos en los tiempos de ejecución, lo que sugiere posibles limitaciones del algoritmo para tableros de gran tamaño.

4.1. Notas de implementación

La implementación del algoritmo se desarrolló en Python. Aunque conceptualmente el tablero puede entenderse como una matriz, internamente se utiliza una estructura flexible que me permitiera recorrer posiciones, verificar restricciones y actualizar el estado sin depender estrictamente de una representación matricial fija.

Algunas decisiones importantes de implementación fueron:

- Representar el tablero como una lista de listas para facilitar el acceso por coordenadas.
- Utilizar una función recursiva para explorar los caminos mediante backtracking, garantizando que cada paso respeta la invariante del problema.
- Mantener una estructura auxiliar para registrar qué celdas están ocupadas en cada momento, simplificando el retroceso cuando una ruta resulta inválida.
- Permitir que la función de búsqueda opere con cualquier tamaño de tablero, evitando dependencias rígidas en la forma en que fue generado.

Estas decisiones permitieron obtener una implementación modular, clara y adaptable, independiente de cómo se haya generado el tablero internamente.