SIA - TP1

Métodos de Búsqueda

Ejercicio 1
8-Puzzle

Caracterización del 8-Puzzle

- **Estados:** Un estado especifica la ubicación de cada una de las ocho piezas, y del espacio, en las 9 celdas de la matriz.
- **Estado inicial:** Cualquier distribución de las 8 piezas en el tablero.
- **Acciones:** Movimiento de las fichas adyacentes al espacio en blanco hacia arriba, abajo, izquierda o derecha.
- **Transición:** Dado un estado y una acción, se obtiene su estado resultante.
- **Goal test:** Verificación si el estado coincide con el estado objetivo.
- **Path cost:** Cada paso cuesta 1, el path cost es el número de pasos en el path. En otras palabras, el costo es uniforme

Heurísticas admisibles encontradas

Suma de distancias Manhattan:

Esta heurística consiste en calcular la suma de las distancias Manhattan de cada pieza a su posición objetivo.

Esta es admisible ya que se deberá mover cada pieza por lo menos su norma para llegar a su posición objetivo.

Cantidad de piezas fuera de lugar:

Otra heurística admisible es contar la cantidad de piezas que no están en su posición objetivo.

Esta es admisible ya que todas las piezas que contribuyen a la heurística deben ser movidas por lo menos una vez para llegar a su posición objetivo.

Estructura, algoritmos y heurística elegida

- Matriz de enteros de 3x3
- Elegimos los algoritmos A* e IDA*
- Elegimos como heurística la suma de distancias Manhattan

Justificación: Si los algoritmos A* e IDA* utilizan una heurística admisible, entonces encontrarán una solución óptima a dicho problema.

Comparación

IDA*:

- No necesita mantener un conjunto de estados tentativos por visitar, por lo tanto, su consumo de memoria está abocado a las variables locales de la función recursiva.

A*

- Utiliza programación dinámica y, por lo tanto, no explora nodos ya visitados.

Conclusión:
Si se dispone de suficiente memoria y se busca la solución más rápida posible, A* es la mejor opción. Si se dispone de una cantidad limitada de memoria y se desea encontrar la solución óptima sin consumir demasiados recursos, IDA* es la mejor opción.

Ejercicio 2
Fillzone

Fill Zone

¿En qué consiste?

- El juego consiste de una grilla de NxN, donde cada celda guarda colores.
- Hay una cantidad fija de colores.
- Al iniciar el juego, la grilla se llena con colores de forma aleatoria.
- Cada jugada se cambia el color del nodo de arriba a la izquierda, y de todos los nodos adyacentes que sean del mismo color, iterativamente.
- El objetivo del juego es “conquistar” todas las celdas, transformándolas todas en el mismo color.
- Una vez que se conquista una celda, esta queda conquistada para siempre.

Representación de un estado del juego

- El estado del juego se representa por una matriz de NxN enteros almacenada en el atributo `grid`.
- Los colores se guardan como un número entero en el atributo `color_count`. Si el juego tiene 5 colores, entonces se guardan como 0, 1, 2, 3 y 4.
- El tamaño del tablero, N, se almacena como un entero en el atributo `grid_size`.

```python
class FillzoneState:

    def __init__(self, grid_size: int, color_count: int) -> None:
        self.grid_size = grid_size
        self.color_count = color_count
        self.grid = [[0 for _ in range(grid_size)] for _ in range(grid_size)]
```

Transición de un estado a otro

- Definimos el método `play_color(color_index)`, que recibe el color a jugar como entrada y retorna una nueva estructura `FillzoneState`, con el resultado de jugar dicho color.
- Realiza una búsqueda iterativa, comenzando en la celda superior izquierda, para encontrar todas las celdas conquistadas y actualizar su color.
- Si el jugador elige el mismo color que eligió en la jugada anterior, el estado no varía.

Heurísticas admisibles

Cantidad de colores en el juego - 1:

Esta heurística consiste en contar cuántos colores distintos hay en el tablero, y restarle 1.

Da 0 si y sólo si queda un solo color, que ocurre si y sólo si conquistaste todo.

Es admisible, pues si quedan 5 colores vas a tener que elegir por lo menos 4 para conquistarlos todos.

Cantidad de colores diferentes en la frontera:

Una heurística similar a la anterior, pero solo cuenta colores en la frontera, excluyendo celdas conquistadas.

Si no hay frontera, no hay colores en frontera y por ende da 0.

Justificación de admisible análoga al caso anterior.

Heurísticas admisibles

Encontrar la celda con mayor cantidad de colores distintos en celdas adyacentes, excluyendo el color actual del jugador, y tomar dicha cantidad

Cantidad de colores distintos en las diagonales, excluyendo el color actual del jugador

Ambas se pueden justificar como admisibles de forma simple; ambas heurísticas encuentran que hay por lo menos X colores aparte del color actual del jugador, y por ende se precisa hacer por lo menos X movidas para conquistarlos todos.

Estas heurísticas son muy parecidas a las anteriores, pero pueden mantener efectividad a un mucho menor costo al no necesitar escanear la grilla entera.

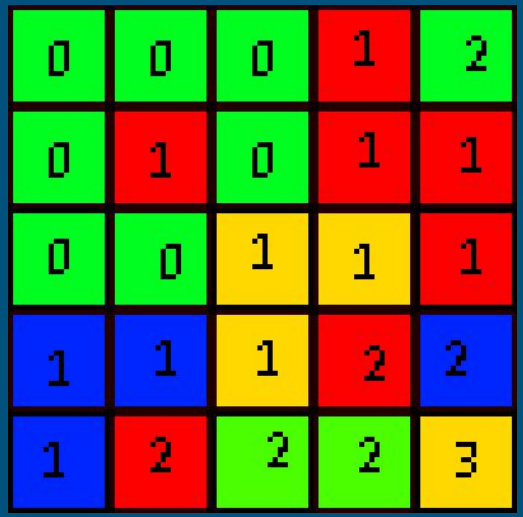
Heurísticas admisibles

“Distancia Buscaminas”:

Se calcula para cada celda la mayor cantidad de jugadas que se deben hacer para conquistar dicha celda. Esto se puede calcular de forma iterativa:

→ Marcar las celdas conquistadas con 0, las demás celdas empiezan como no marcadas.
→ contador = 1
→ while (quedan celdas sin marcar):
- Calcular los “Grupos de celdas”: todos los grupos disjuntos de celdas que son adyacentes y del mismo color.
- Todos los grupos de celdas no-marcados adyacentes a una celda marcada, se les marcan todas las celdas con el valor de “contador”.
- Si quedan celdas sin marcar, contador += 1

→ Retornar “contador”, el valor de la celda con mayor número



Heurísticas no admisibles

Cantidad celdas no conquistadas:

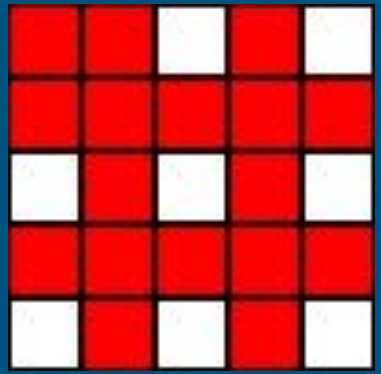
Calculado como N*N - conquistadas.

Cantidad de grupos de celdas distintos, excluyendo las conquistadas:

Parecida a la anterior, pero celdas adyacentes del mismo color se cuentan como una.

Ambas se pueden demostrar no admisibles con el siguiente contraejemplo:

El juego se resuelve en una movida, pero las heurísticas dan ambas 8.



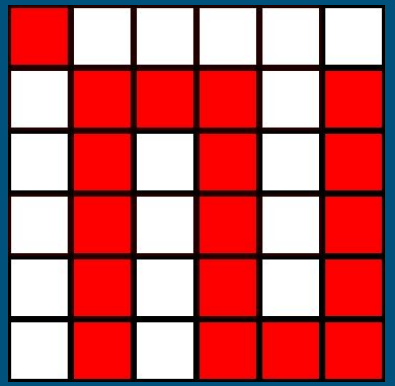
Heurísticas no admisibles

Cantidad de filas que no tienen todas las celdas del mismo color

Análogamente:

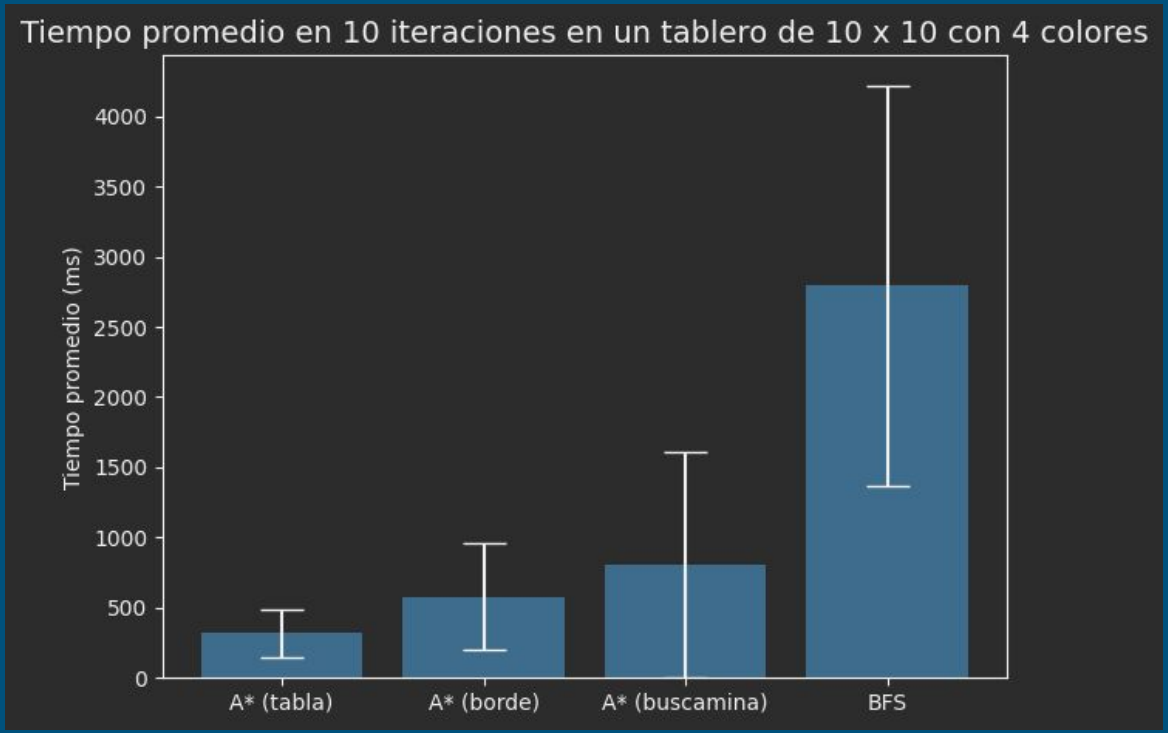
Cantidad de columnas que no tienen todas las celdas del mismo color

Demostramos que ambas no son admisibles con el mismo contraejemplo:

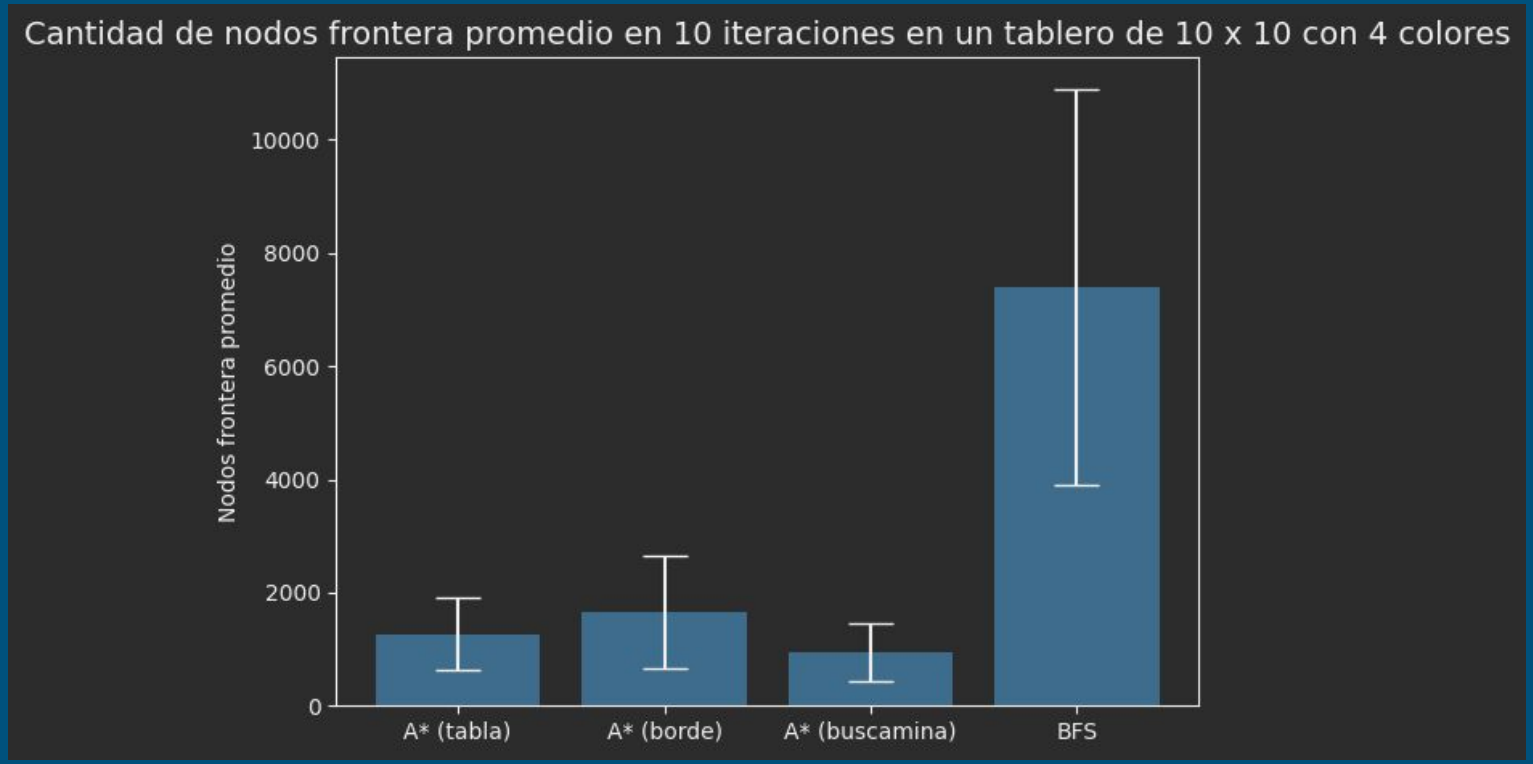


Ninguna filas o columnas tiene todas las celdas iguales, por ende ambas heurísticas dan 6. Sin embargo, se puede resolver el juego en 2 jugadas.

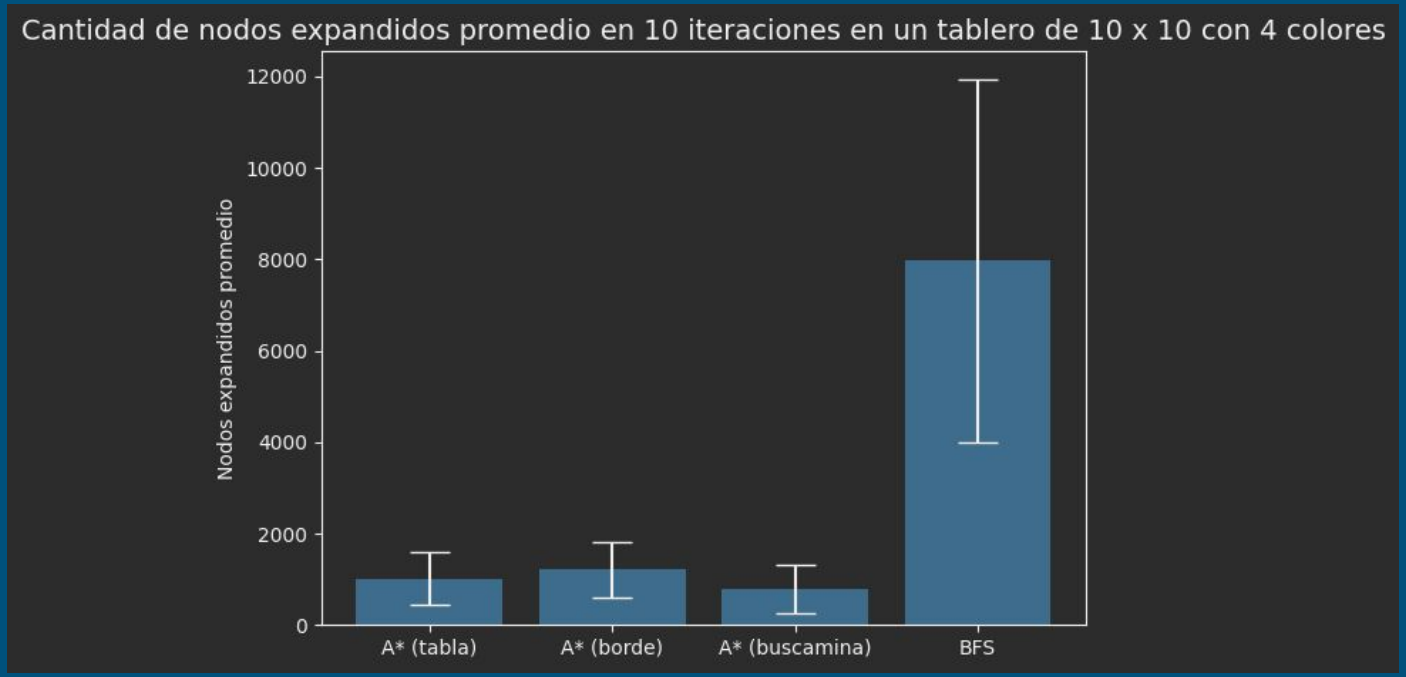
Resultados obtenidos - Alg. óptimos



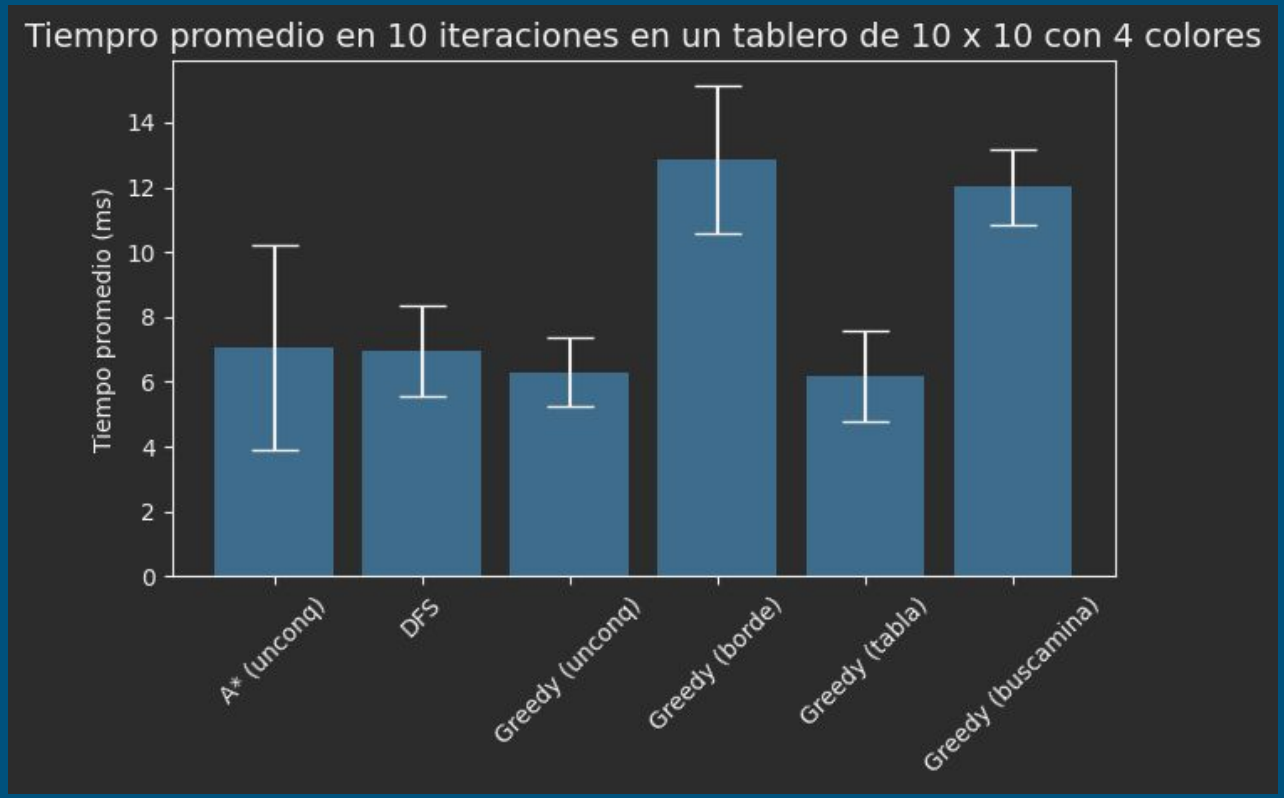
Resultados obtenidos - Alg. óptimos



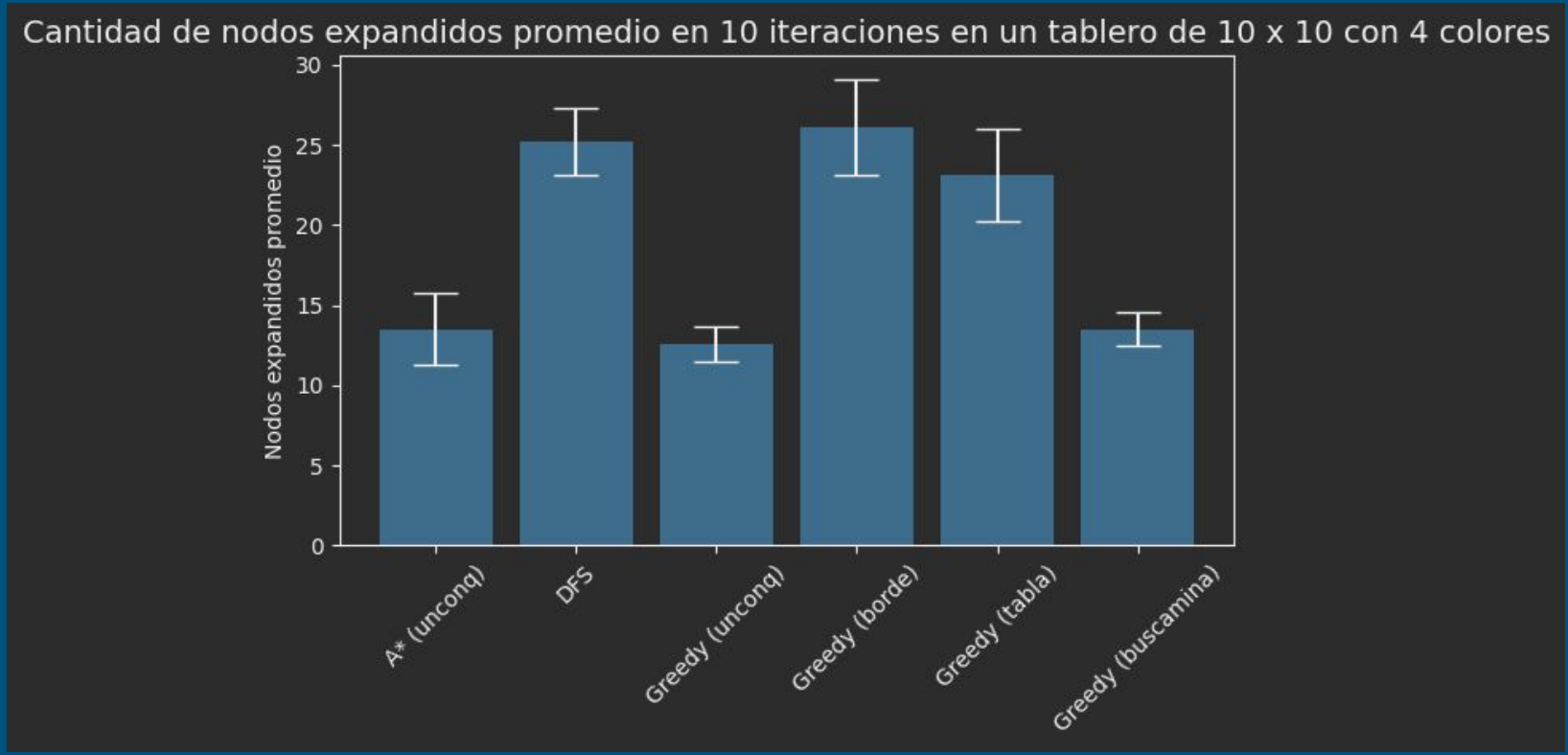
Resultados obtenidos - Alg. óptimos



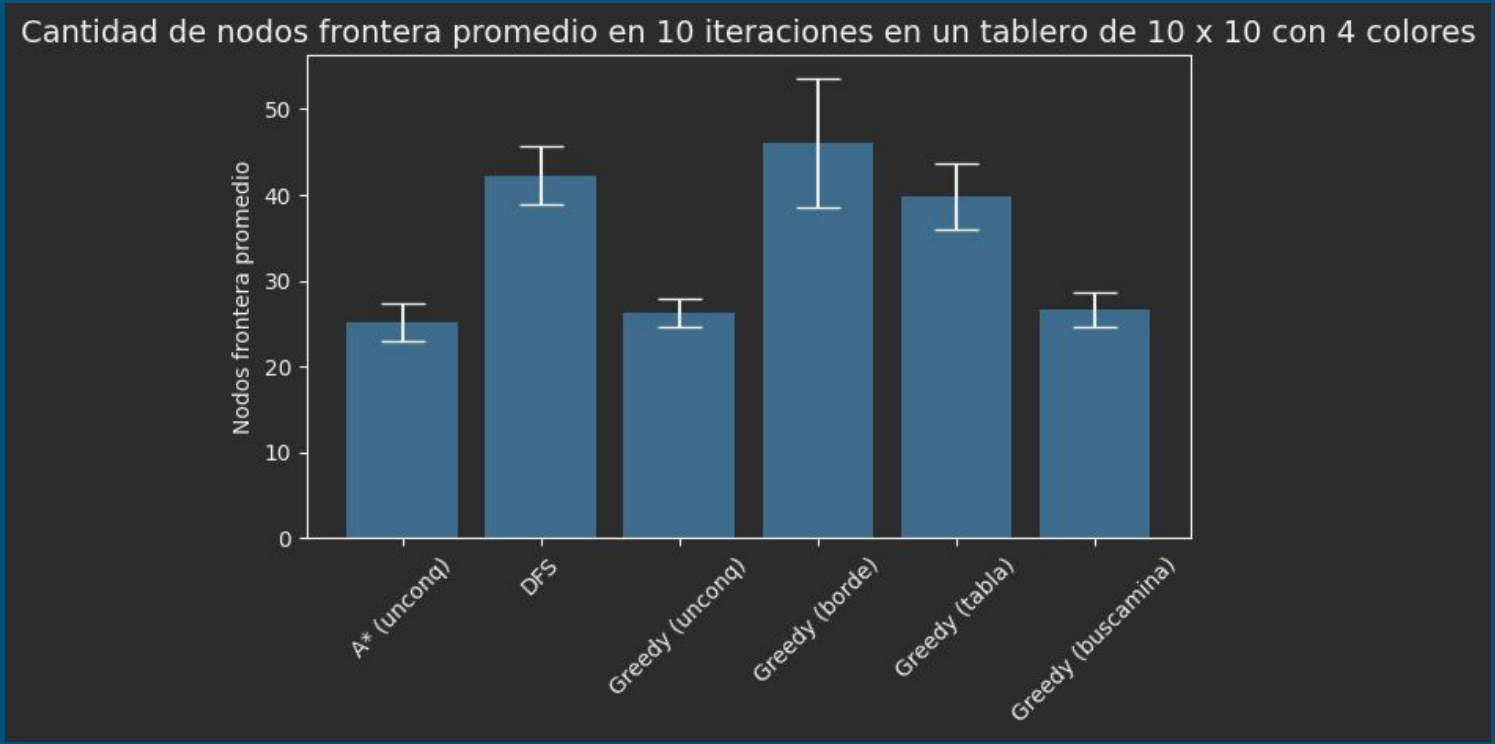
Resultados obtenidos - Alg. no óptimos



Resultados obtenidos - Alg. no óptimos

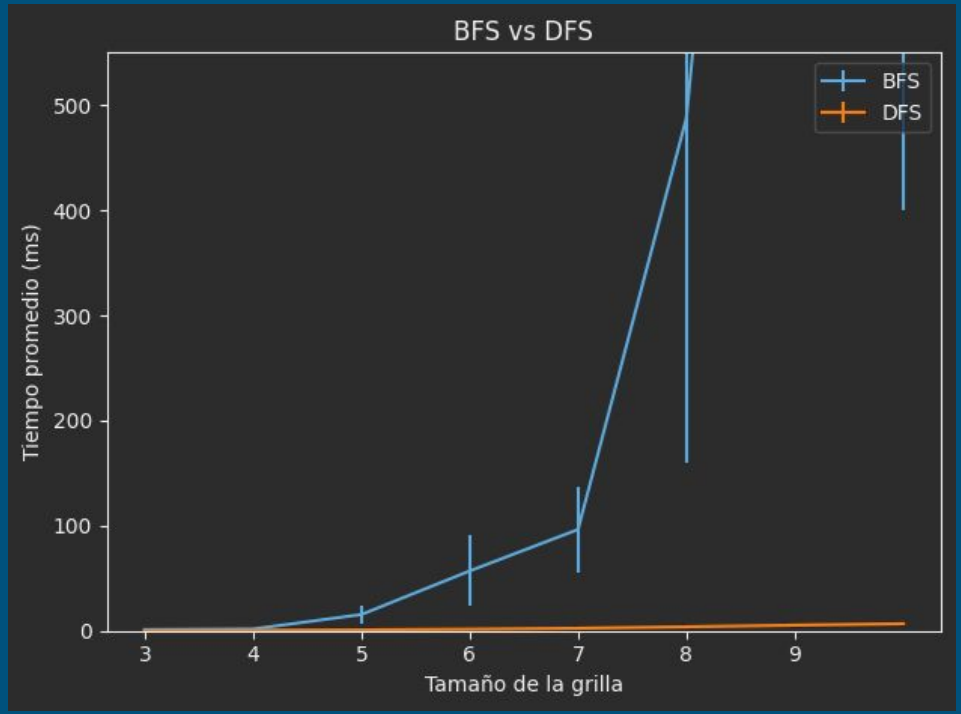


Resultados obtenidos - Alg. no óptimos



Resultados obtenidos - BFS vs DFS

DFS tuvo un rendimiento superior a BFS. A medida que aumenta el *grid_size*, el árbol generado por BFS se expande exponencialmente, ya que busca siempre encontrar una solución óptima.



Resultados obtenidos - BFS vs DFS

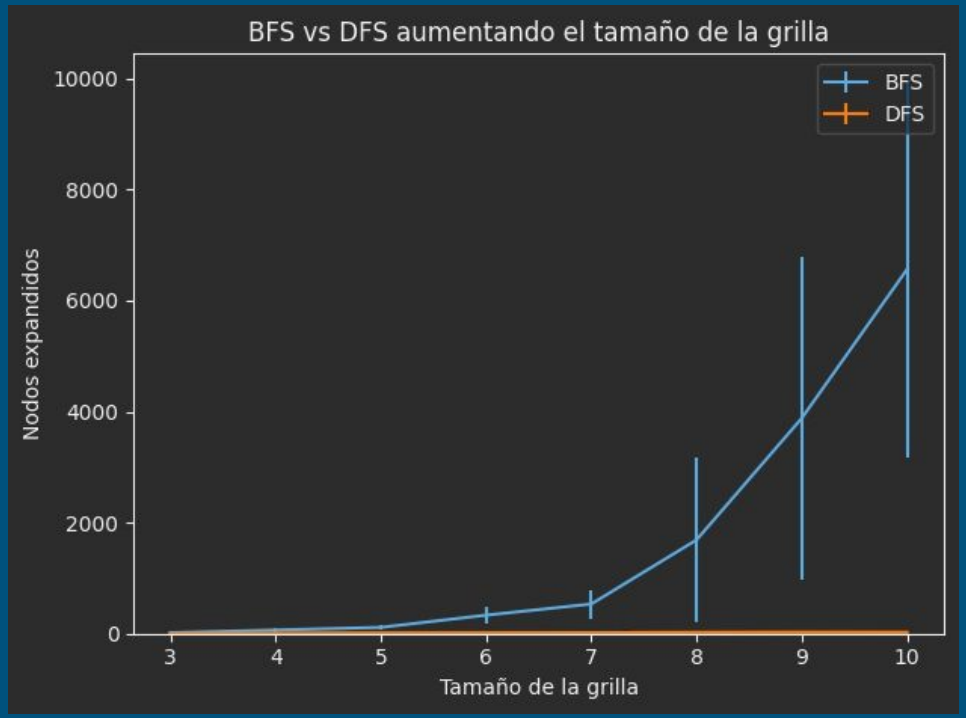
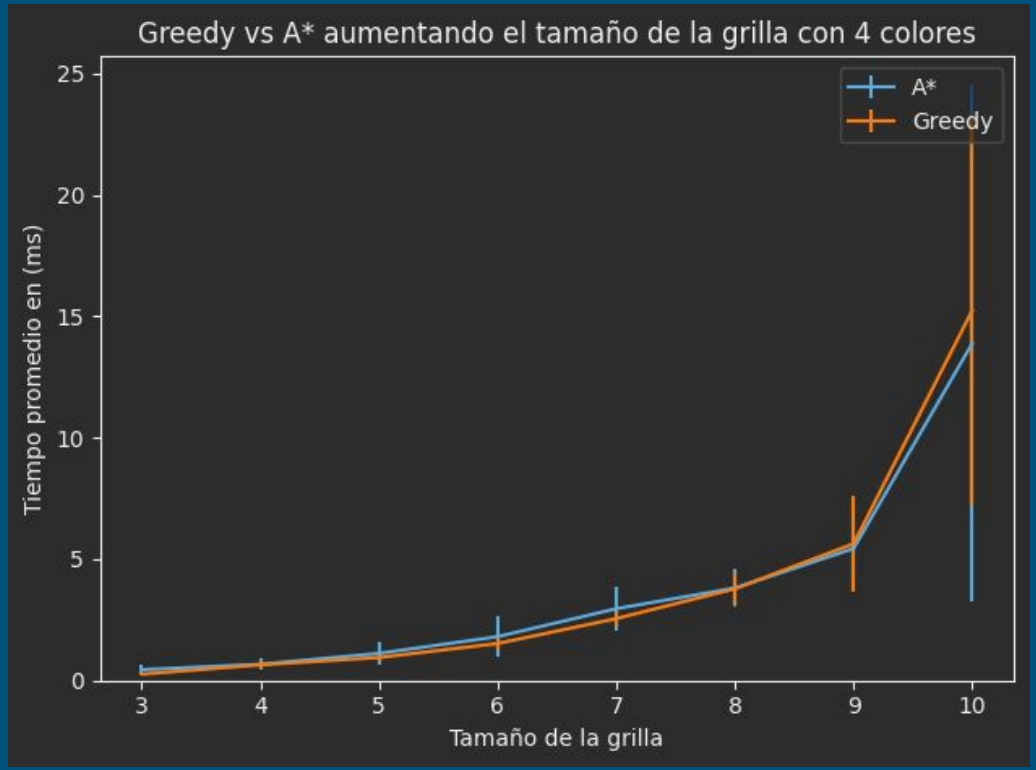


Gráfico que muestra la cantidad de nodos expandidos que explica la expansión del árbol.

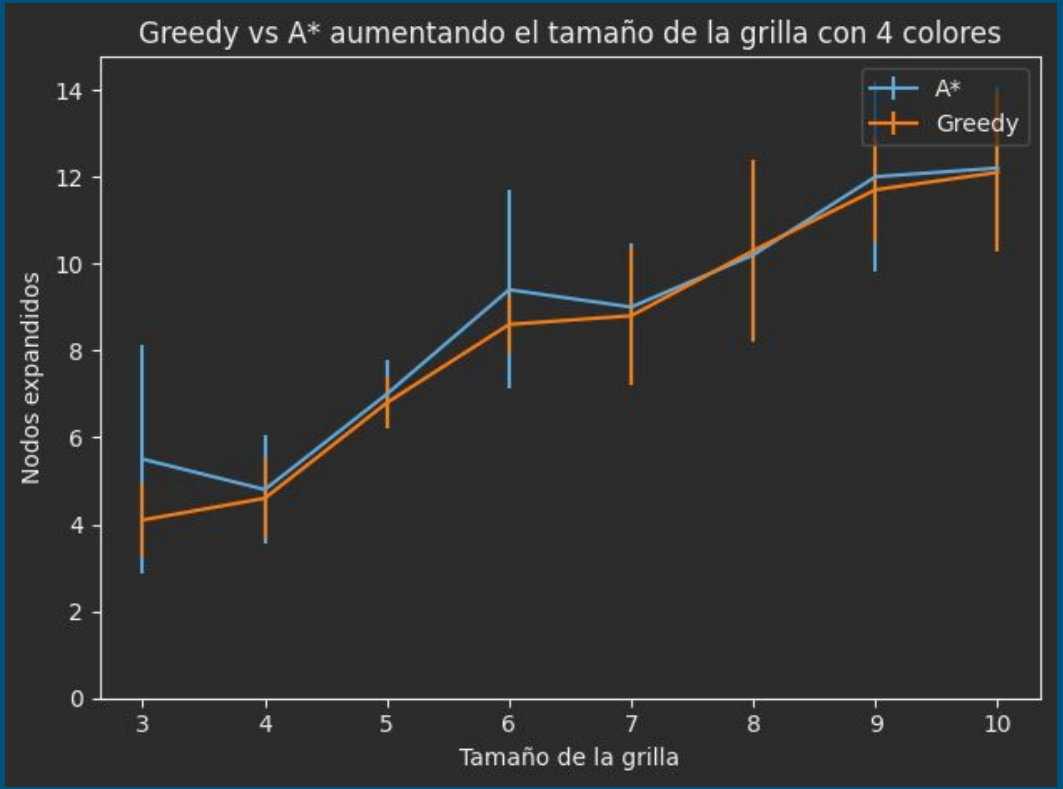
Resultados obtenidos - Greedy vs A*



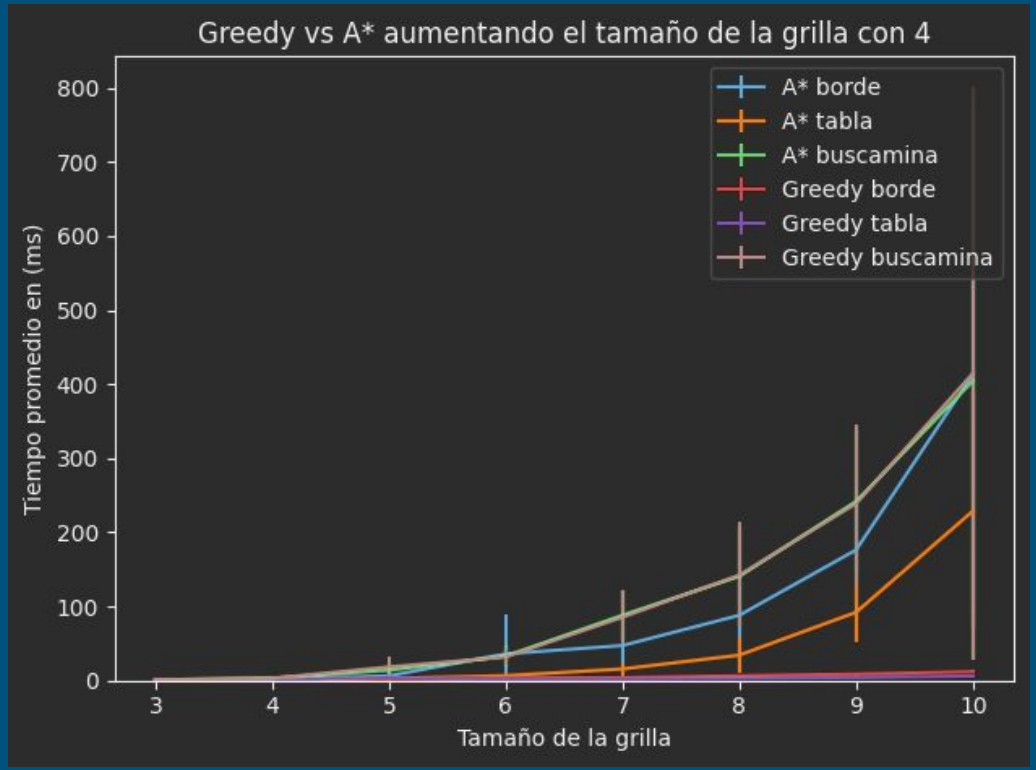
Utilizando la heurística no admisible *cantidad de celdas no conquistadas*, vemos que el algoritmo A* en promedio le lleva un poco más de tiempo en completarse

Resultados obtenidos - Greedy vs A*

Utilizando la heurística no admisible cantidad de celdas no conquistadas, vemos que el algoritmo A* expande más nodos que el algoritmo Greedy.



Resultados obtenidos - Greedy vs A*

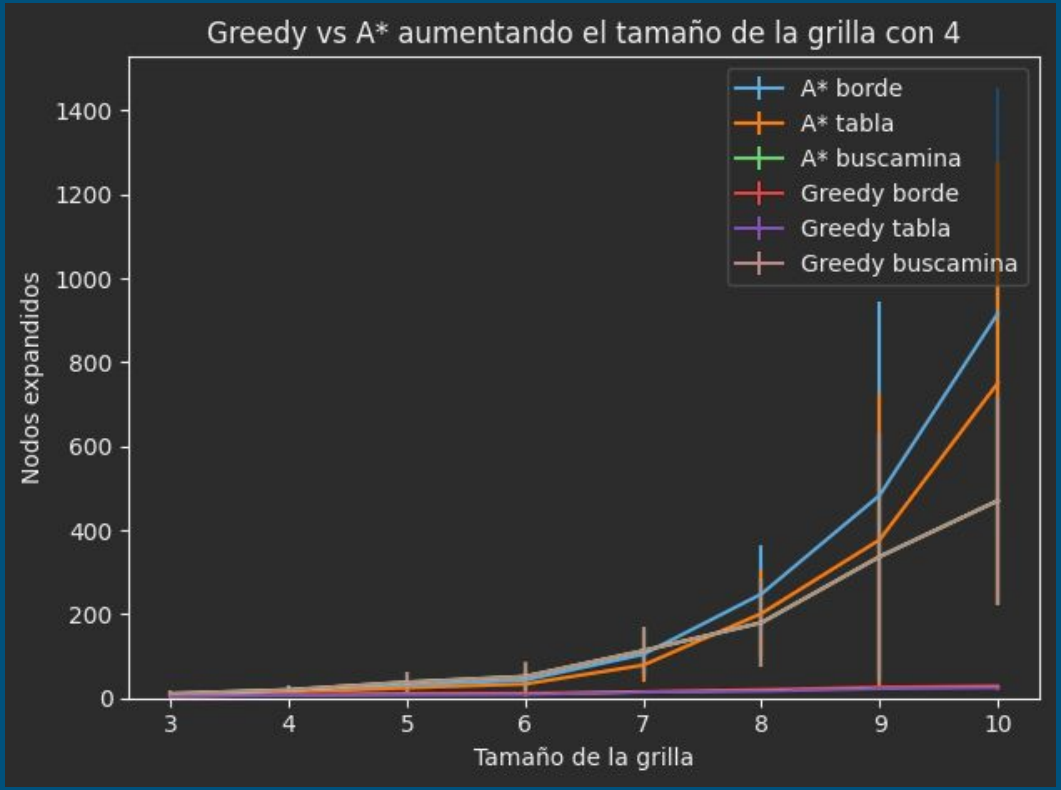


Algo interesante que nos muestra este gráfico, es que para que estos algoritmos sean eficientes en tiempo, la heurística debe ser fácil de calcular.
Podemos ver que la heurística de buscaminas, que es complicada de calcular, produce los resultados más lentos en ambos A* y Greedy.

Resultados obtenidos - Greedy vs A*

Utilizando heurísticas admisibles vemos que la comparación entre A* y Greedy empieza a tener mayor semejanza a la comparación entre BFS y DFS.

A* busca soluciones óptimas y por ende precisa expandir más nodos para encontrarla, mientras Greedy evita hacer backtracking y confía en que la heurística lo guíe por un buen camino.



Conclusiones - Óptimo vs Rápido

Una característica del problema del Fillzone es que ninguna acción puede alejarnos de la solución. Eligiendo colores de cualquier forma, verificando de no repetir estados, eventualmente siempre se llega a una solución.

Los algoritmos BFS y A* con una heurística admisible son capaces de garantizarnos soluciones óptimas. Sin embargo, por la naturaleza del problema, los algoritmos DFS y Greedy nos permiten conseguir una solución probablemente no óptima, pero de forma mucho más rápida.

Preguntas?