INSTITUTO TECNOLÓGICO DE BUENOS AIRES

SISTEMAS DE INTELIGENCIA ARTIFICIAL

INGENIERÍA EN INFORMÁTICA

Métodos de búsqueda

Primer trabajo práctico especial

Titular: Parpaglione, Cristina
Semestre: 2019 1C
Grupo: 5
Repositorio: https://bitbucket.org/itba/sia-2019-1c-05/src/master/search-algorithms/
Entrega: 3 de marzo de 2019
Autores: Banfi, Micaela (57293)
Guzzetti, Clara (57100)
Ritorto, Bianca (57082)
Vidaurreta, Ignacio (57250)

1. Introducción

En este trabajo especial se utiliza un motor de inferencia para poder resolver problemas de búsqueda tanto informada como no informada. El proyecto se divide en dos secciones, la implementación de dicho motor, y la implementación de la lógica de un juego en particular, "Edificios" (o *Skyscraper*), que se interconectan mediante interfaces. Ya que la implementación del motor de búsqueda está separada por completo de la del juego específico, se podría resolver cualquier problemática de búsqueda utilizándolo correctamente.

2. Juego

2.1. Descripción

El juego "SkyscrapersPuzzle" consiste en llenar un tablero de dimensión NxN con números del 1 al N. En cada fila y en cada columna se debe poner un edificio con altura variable del 1 al N, todos los números deben estar presentes, sin repetir ninguno en la fila o en la columna. Fuera del tablero se encuentran números también del 1 al N que determinan la "visibilidad" desde cada posición. Un edificio con altura i tapa a uno con altura i-1.

2.2. Implementación

La implementación del juego consiste principalmente de la clase Board. Se tiene, dentro de Board, una matriz de dimensión NxN de Skyscrapers, una clase que contiene un objeto Point (de Java) y un int que representa la altura del Skyscraper.

Board tiene un método que devuelve un BoardValidator que se fija cuantos conflictos hay en una matriz determinada. Esto sirve para las heurísticas del juego explicadas luego.

Se desarrollaron dos tipos de reglas para aplicar en un tablero.

1. **FillRule:** Para aplicar estas reglas se comienza con un tablero vacío y las reglas de los costados. Hay una regla por cada casillero del tablero (NxN). Llena un casillero vacío con un número del 1 al N.
2. **SwapRule**: Para poder aplicar estas reglas se debe comenzar con un tablero válido, es decir, con un tablero que contenga en sus filas y columnas permutaciones de números del 1 a N sin repetir ningún número en ellas. Hay Nx(N-1) reglas por tablero.[1]
    a. **SwapColRule:** intercambia de lugar una columna con otra. Hay Nx(N-1)/2 reglas por tablero.
    b. **SwapRowRule:** intercambia de lugar una fila con otra. Hay Nx(N-1)/2 reglas por tablero.

2.3. Heurísticas

Se crearon heurísticas para las reglas de **SwapRule** únicamente por temas de simplicidad y porque son las que resuelven rápidamente un tablero dado.

Se utilizó la clase de **BoardValidator** para determinar y contar la cantidad de conflictos que puede haber en un casillero, y por ende, en un tablero. Ya que los tableros que se utilizan cuando se aplican las reglas de **SwapRule** son siempre válidos (no repiten valores en filas y columnas), la cantidad de **conflictos** se definió únicamente teniendo en cuenta conflictos de visibilidad de edificios, y de la siguiente manera:

- Por cada fila que no cumpla la restricción, se puede sumar hasta 2 conflictos: 1 por la izquierda y 1 por la derecha.

- Por cada columna que no cumpla la restricción, se puede sumar hasta 2 conflictos: 1 por arriba y 1 por abajo.

[1] Aclaración: esta regla no funcionó completamente. Más detalles en el análisis de resultados.

Por lo tanto, la máxima cantidad de conflictos por tablero son: filas *2 +columnas*2 = N*2 + N*2 = 4N, siendo N la dimensión del tablero.

Teniendo esto en cuenta, se calculó la máxima cantidad de conflictos que se pueden resolver en un "swap":

- Si intercambio filas: 2 por cada fila involucrada, es decir, (4) + N columnas (ya que los conflictos de las columnas podrían justo ser resueltos con un solo swap en una de las posiciones).
- Si intercambio columnas: 2 por cada columna involucrada, (4) + N filas ya que los conflictos de las filas podrían justo ser resueltos con un solo swap en una de las posiciones).

En total, idealmente en un swap resuelvo la máxima cantidad de conflictos posibles: 4 + N.

2.3.1. Admisible

Entre las condiciones para que A* encuentre el camino óptimo al objetivo, se pide que $h(n) \leq h^*(n)$ para todo nodo n. Al no contar con una h* conocida, se debió buscar una h' que cumpla que $h(n) \leq h'(n) \leq h^*(n)$ para todo nodo n, es decir, que asegure que esté acotada por h*(n) y por lo tanto, sea admisible.

Sabiendo que

$$h^*(n) = \#swaps\ para\ solucionar\ todos\ los\ conflictos\ *\ costo\ del\ swap$$

y habiendo definido una cota máxima suponiendo que un swap resuelve todos los conflictos que podría resolver (es decir, resuelve 4 conflictos), definimos

$$h'(n) = ideal\ swaps\ para\ solucionar\ todos\ los\ conflictos*\ costo\ del\ swap$$

Por lo tanto,

$$h'(n) = \lceil (\#conflictos\ del\ tablero/4+N) \rceil * costo\ del\ swap <= h^*(n)$$

Encontramos una cota inferior de h*, y la h elegida fue:

$$h(n) = (\#conflictos\ del\ tablero/4+N) * costo\ del\ swap$$

Para simplificar las cuentas, y por consistencia con las reglas de **FillRule** en las que el costo de llenar un casillero es 1, el costo de hacer un swap, es 1 también. Finalmente, queda:

$$h(n) = (\#conflictos\ del\ tablero/4+N)$$

2.3.2. No admisible

Para la heurística no admisible se intentó hacer un promedio de la cantidad de swaps necesarios para realmente resolver todos los conflictos de un tablero. Se corrieron varias pruebas y, estadísticamente se encontró que el motor resolvía aproximadamente un conflicto cada vez que hacía un swap. Por lo tanto, nuestra heurística no admisible fue la de:

$$h_i(n) = \#conflictos * costo\ del\ swap = \#conflictos$$

3. Motor

3.1. Estructura

El motor cuenta con una clase principal en la que están definidos los diferentes algoritmos de búsqueda, y que según el caso utiliza uno u otro. También cuenta con una clase de Nodo, que utilizan las diferentes estrategias de

búsqueda para crear sus árboles. Se creó una clase adicional llamada EngineFactory que recibe parámetros tal como estrategia elegida y heurística crea un motor con las características deseadas para la resolución del problema.

El motor utiliza 4 interfaces para entender el problema: Rule, Heuristic, Problem y State, que son implementadas en la parte de la lógica del juego que será explicada más adelante.

3.2. Búsqueda desinformada

Se implementaron tres métodos de búsqueda desinformada: Breadth First Search, Depth First Search y Profundización Iterativa. Los mismos utilizan las clases Stack y ArrayList de Java para encolar y desencolar los hijos correspondientes adecuadamente.

3.3. Búsqueda informada

Se implementó el algoritmo Greedy utilizando una PriorityQueue que compara valores teniendo en cuenta la heurística del nodo analizado. Se implementó el algoritmo A* utilizando una PriorityQueue que compara valores teniendo en cuenta la heurística del nodo analizado y también el costo de tal nodo.

3.4. Costo

Se definió el costo de llenar un casillero con un número como 1. Esto funciona para la regla que tenemos en el juego de "Fill". Para la regla que intercambia una columna con otra, o una fila con otra, se definió el costo de hacer dicho movimiento como 1, ya que en un movimiento intercambia todo lo que tiene que cambiar.

4. Conclusiones

4.1. Análisis de resultados

A continuación se intentan exponer los aspectos más significativos del problema y sus resultados:

Error en la regla de **SwapRule**:

- Ya avanzado el trabajo, se comenzó a testear el programa con matrices de 4x4 y 5x5, para corroborar su funcionamiento. Se notó que algunas no funcionaban correctamente por el hecho de que nunca encontraba la matriz solución. Luego de mucho tiempo de análisis detallado, se pudo concluir que lo que estaba sucediendo era que la aplicación de las reglas definidas sobre un tablero válido no generaban todos los tableros necesarios posibles.

- Dependiendo de con qué tablero se iniciara el árbol, el motor de búsqueda iba a encontrar (o no) la solución que le correspondía al problema. Es por esto que algunas de las matrices de prueba funcionaban correctamente, por lo que el equipo continuó desarrollando el proyecto en base a estas reglas que resultaron ser erróneas.

- Para los tableros de 4x4 se desarrolló un "fix" aceptable: Se crea la menor cantidad posible de matrices iniciales como para asegurar que el motor encontrará una solución con al menos una de ellas, y se corre el motor sobre todas, hasta que alguna la encuentre.

- Ya con poco tiempo para arreglar todo, se tomó la decisión de analizar la mayor cantidad de matrices que tenían un árbol "ganador" y en base a estos datos sacar conclusiones.

- Se debe aclarar, y esto es **muy importante**, que estos datos, al ser incompletos, no son totalmente válidos.

- Finalmente, se propone un nuevo set de reglas que, luego de analizarlas, se cree que funcionarán bien y requerirían pocas modificaciones en el programa como está:

- **SwapRule** pero que en vez de intercambiar filas o columnas enteras, intercambie dos casilleros entre sí.

- Se comienza con una matriz "válida" (misma definición de válida que se viene utilizando en todo el informe: permutaciones del 1 al N que no se repitan ni en filas ni en columnas).

- Se intercambiarán solo dos casilleros adyacentes "horizontalmente". Esto ahorraría el checkeo de las filas por repetidos. Solo faltaría chequear las columnas y la visibilidad en los costados de la matriz.

- Se crearían todos los tableros posibles, ya que se modifica de a dos casilleros a la vez. La heurística quedaría muy parecida a la que ya está implementada, solo que por cada swap se resolverán como máximo 1 por la fila y 2 por las columnas involucradas. Es decir, 3.

Otras conclusiones:

- Luego de analizar los tiempos de ejecución, se puede ver claramente que definir un buen costo de aplicar una regla es esencial, ya que permite llegar a una solución mucho más rápidamente y con menos esfuerzo.

- Utilizar un mapa de estados es clave para que los algoritmos puedan terminar correctamente y no quedarse en loops infinitos. Además, se reduce el tiempo de ejecución drásticamente.

- Mejor tiempo de ejecución en búsqueda no informada (**FillRule**): DFS. Como DFS visita primero nodos más profundos, el fill se cumple más rápido ya que en cada paso de DFS llena un nuevo casillero vacío. En cambio en BFS evalúa todas las opciones de llenar ese casillero primero.

- Mejor tiempo de ejecución en búsqueda informada (**SwapRule**): No estaría bien sacar una conclusión de esto ya que las reglas estaban incorrectamente pensadas. Pero a partir de los resultados que pudimos obtener, el mejor tiempo de ejecución lo tiene Greedy.

- Definir bien el hash y el equals para todas las clases necesarias es imperativo para que el algoritmo de búsqueda funcione bien y pueda determinar correctamente cuáles estados son exactamente iguales y cuales no.

4.2. Mejoras posibles

- Utilizar la clase **BoardValidator** en la función **isGoal()** para evitar la repetición de código.

- Se podrían haber llenado los tableros con las reglas de **FillRule** teniendo en cuenta los lugares en donde la visibilidad era o 1 o N, ya que esto restringe a que, donde es 1, justo al lado debe haber un edificio de altura N. Y donde es N, justo al lado debe haber uno de altura 1.

- Se podría haber desarrollado una mejor GUI, con una interfaz visual y una mejor representación de la matriz del juego, pero debido a que el tiempo apremiaba, se decidió dar prioridad a la implementación y la lógica del juego y del motor.

5. Apéndice

5.1. Gráficos Fill Rule

Tableros iniciales utilizados para los siguientes gráficos:

	3	0	1	
3	0	0	0	1
0	0	0	0	0
1	0	0	0	2
	0	0	2	

Tablero 3x3

	4	0	2	1	
4	0	0	0	0	1
3	0	0	0	0	2
0	0	0	0	0	0
1	0	0	0	0	2
	1	2	0	2	

Tablero 4x4

	5	4	3	2	1	
5	0	0	0	0	0	1
0	0	0	0	0	0	0
3	0	0	0	0	0	2
2	0	0	0	0	0	2
0	0	0	0	0	0	0
	1	0	0	2	2	

Tablero 5x5

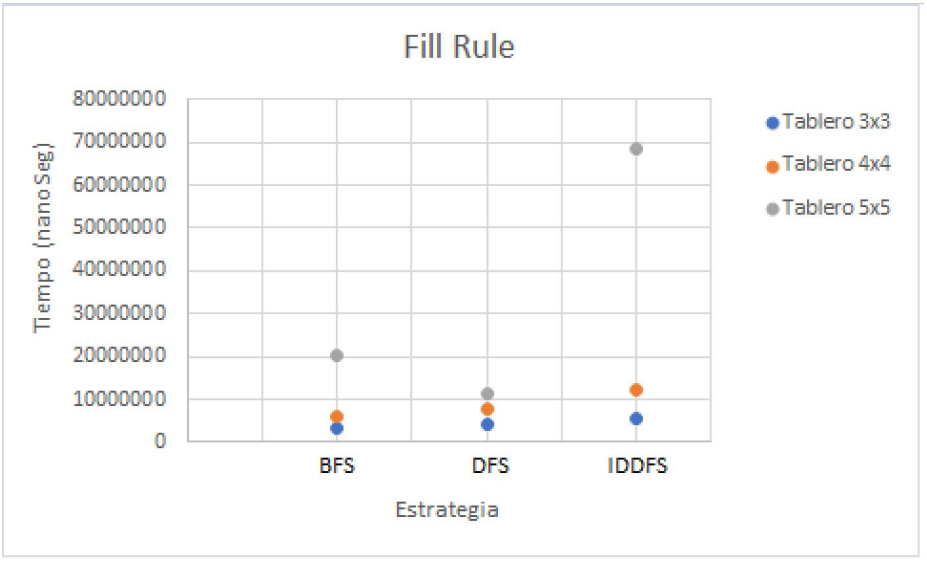


Figura 1: Gráfico búsqueda no informada con FillRule (estrategia contra tiempo de ejecución)

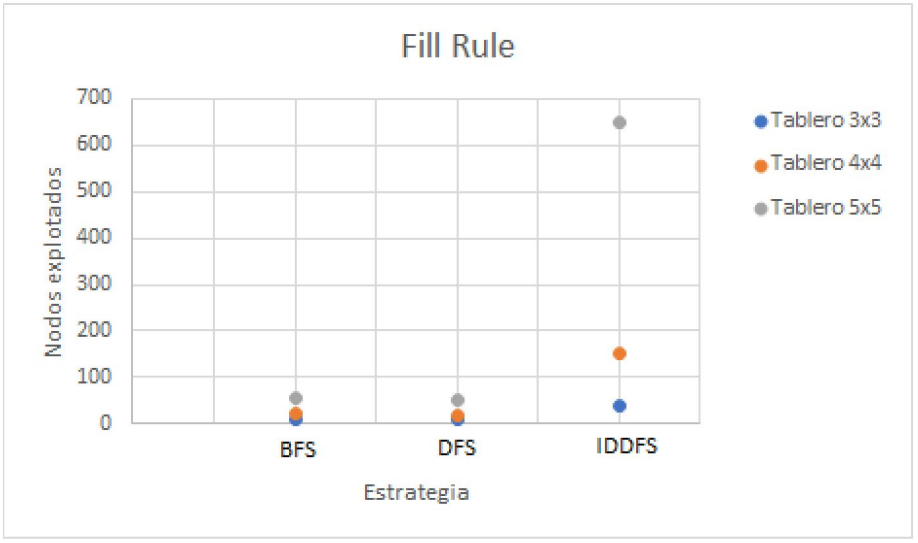


Figura 2: Gráfico búsqueda no informada con FillRule (estrategia contra nodos explotados)

Tablero 3x3		
Estrategia	Nodos Explotados	Tiempo (nano Seg)
BFS	10	3477704
DFS	10	4116452
IDDFS	39	5741589

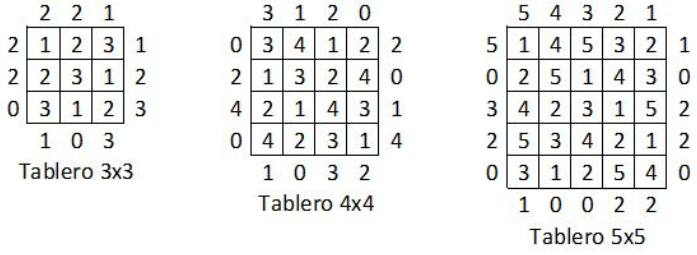
Tablero 4x4		
Estrategia	Nodos Explotados	Tiempo (nano Seg)
BFS	21	5843525
DFS	20	7613774
IDDFS	150	12199977

Tablero 5x5		
Estrategia	Nodos Explotados	Tiempo (nano Seg)
BFS	56	20274597
DFS	52	11544580
IDDFS	649	68547061

Figura 3: Tablas búsqueda no informada con FillRule

5.2. Gráficos Swap Rule

Tableros iniciales utilizados para los siguientes gráficos:



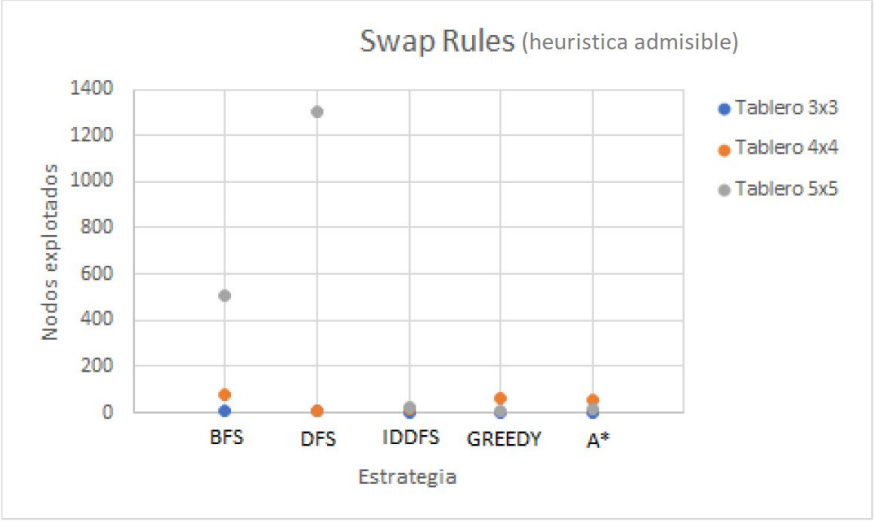


Figura 4: Gráfico búsqueda con SwapRule heurística admisible (nodos explotados)

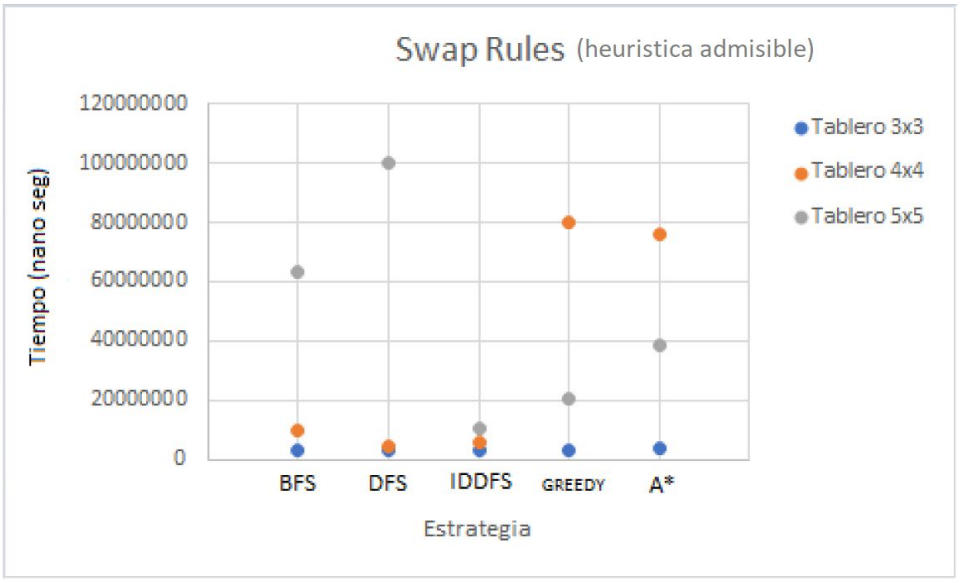


Figura 5: Gráfico búsqueda con SwapRule heurística admisible (tiempo de ejecución)

Tablero 3x3			
Estrategia	Nodos Explotados	Tiempo	Depth
BFS	11	2895258	2
DFS	8	2766330	8
IDDFS	1	2540522	2
Greedy	2	3099270	2
A*	3	3476545	2

Tablero 4x4			
Estrategia	Nodos Explotados	Tiempo	Depth
BFS	84	7201206	3
DFS	87	9462771	87
IDDFS	14	5662702	3
Greedy	4	5581681	3
A*	15	8831907	3

Tablero 5x5			
Estrategia	Nodos Explotados	Tiempo	Depth
BFS	504	63325171	3
DFS	1301	100107502	1301
IDDFS	22	10585832	3
Greedy	6	20578829	5
A*	15	38357565	3

Figura 6: Tablas búsqueda con SwapRule heurística admisible

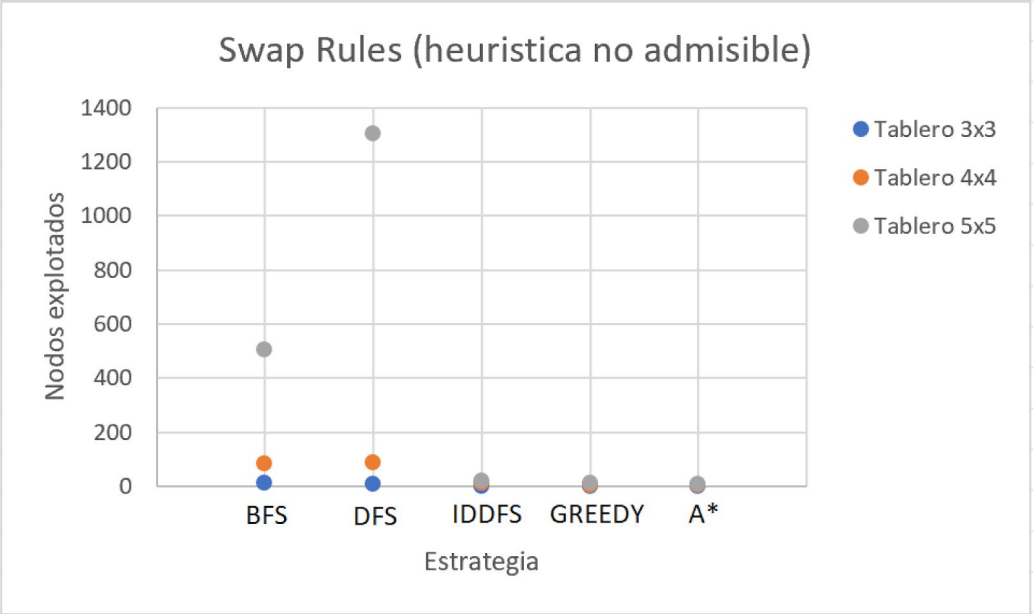


Figura 7: Gráfico búsqueda con SwapRule heurística no admisible (nodos explotados)

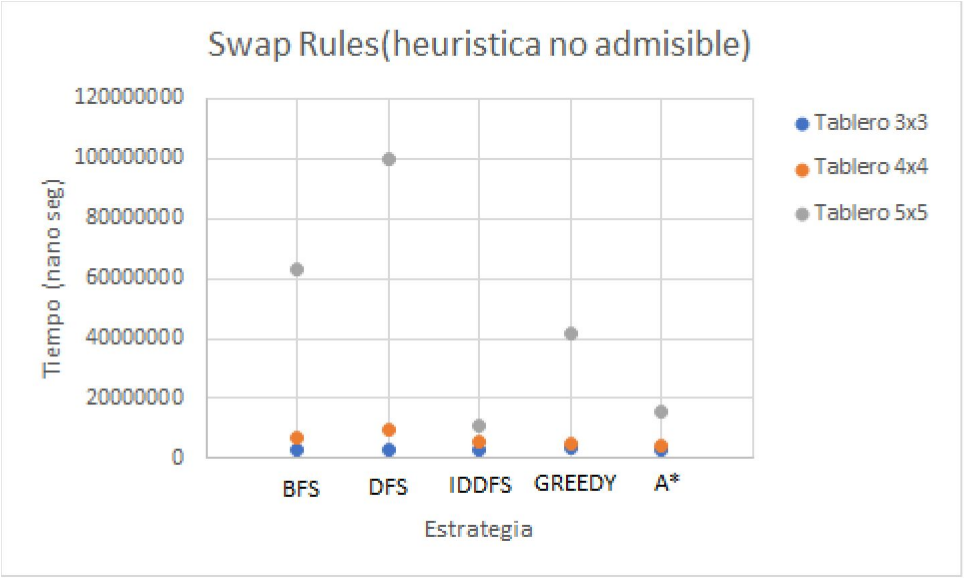


Figura 8: Gráfico búsqueda con SwapRule heurística no admisible (tiempo de ejecución)

Tablero 3x3			
Estrategia	Nodos Explotados	Tiempo	Depth
BFS	11	2895258	2
DFS	8	2766330	8
IDDFS	1	2540522	2
Greedy	2	3721554	2
A*	2	2959258	2

Tablero 4x4			
Estrategia	Nodos Explotados	Tiempo	Depth
BFS	84	7201206	3
DFS	87	9462771	87
IDDFS	14	5662702	3
Greedy	3	4813357	3
A*	3	3997310	3

Tablero 5x5			
Estrategia	Nodos Explotados	Tiempo	Depth
BFS	504	63325171	3
DFS	1301	100107502	1301
IDDFS	22	10585832	3
Greedy	12	41841577	7
A*	8	15405728	5

Figura 9: Tablas búsqueda con SwapRule heurística no admisible