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 aplicar estas reglas se parte de un tablero completo en el cual cada fila posee todos los números entre 1 y N. Esto consiste en un tablero que respeta únicamente la restricción de repetidos en filas, no así en columnas y visibilidad. Las reglas intercambian casilleros adyacentes de manera horizontal. Hay NxN reglas. Se crean todos los tableros posibles, ya que se modifica de a dos casilleros a la vez.

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. La cantidad de **conflictos** se definió 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.

- Por cada columna que tenga al menos un número repetido se suma 1 conflicto

Por lo tanto, la **máxima cantidad de conflictos por tablero** son: 2 * #filas + 2 * #columnas + #columnas = **5N**

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

- Intercambiando dos casilleros horizontalmente: 2 por la visibilidad de la fila involucrada + 2 por la visibilidad de cada columna involucrada + 1 por los posibles repetidos de cada columna involucrada.

En total, **idealmente en un swap resuelvo como máximo 8 conflictos**.

Las heurísticas elegidas buscan favorecer los escenarios en los que la cantidad de conflictos es menor.

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. 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 *8* conflictos), definimos

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

Por lo tanto la h elegida fue,

$$h(n) = \lceil (\#conflictos\ del\ tablero/8) \rceil * costo\ del\ swap <= h^*(n)$$

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:

$$h(n) = \lceil (\#conflictos\ del\ tablero/8) \rceil$$

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 “Swap” 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

- 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.

- 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

- 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

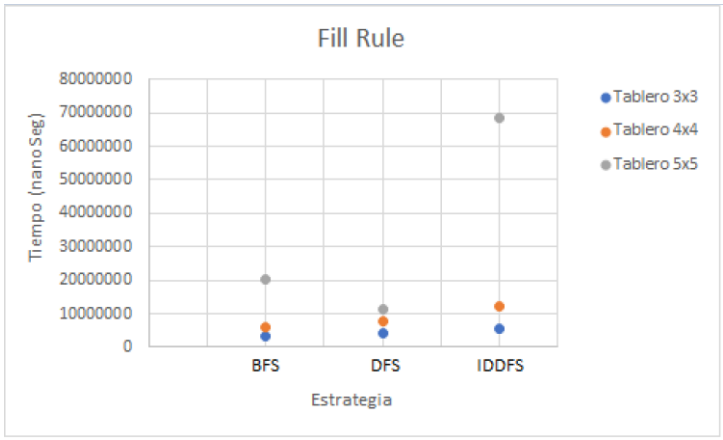


Figura 1: Gráfico búsqueda no informada con FillRule (estrategia contra tiempo de ejecución)

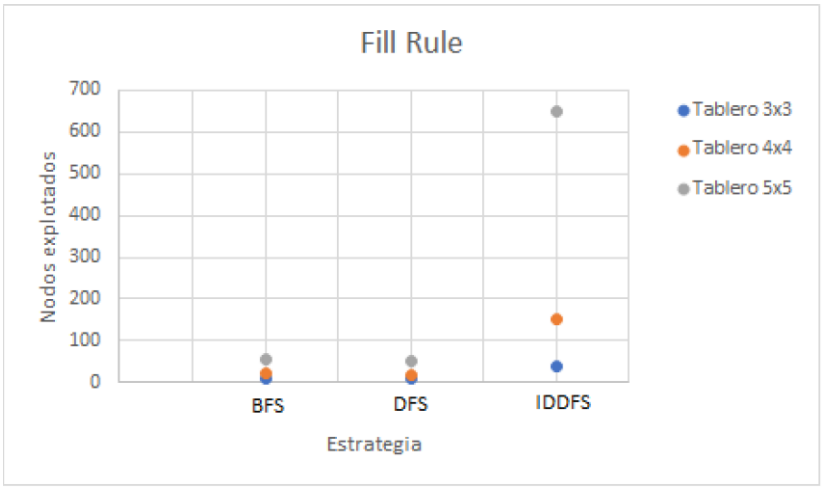


Figura 2: Gráfico búsqueda no informada con FillRule (estrategia contra nodos explotados)

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

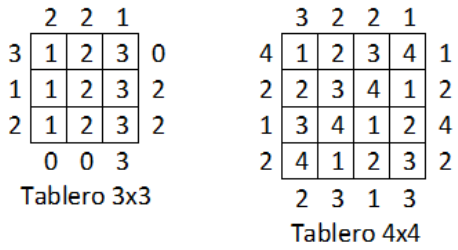
Tablero 4x4		
Estrategia	Nodos Explotados	Tiempo (nano Seg)
BFS	21	5843525
DFS	20	7613774
IDDFS	150	12199977

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

5.1. Gráficos Swap Rule

Tableros iniciales utilizados para los siguientes gráficos:



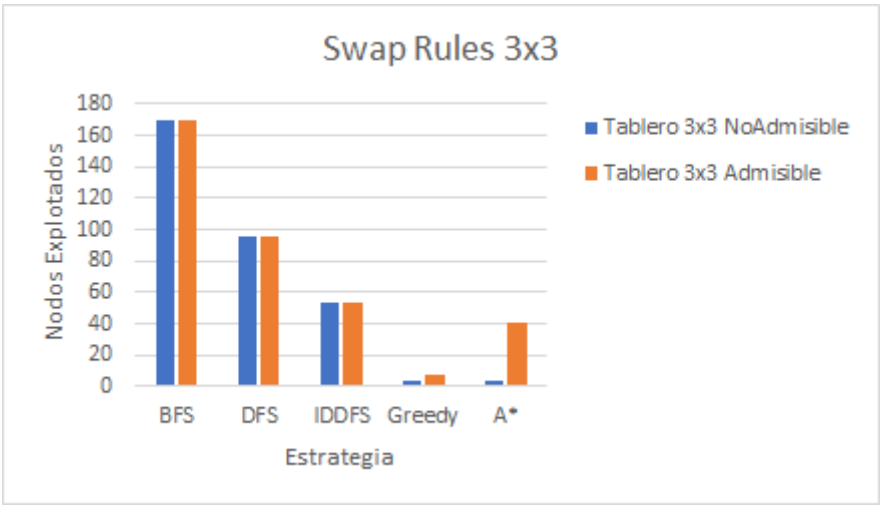


Figura 4: Gráfico búsqueda con Swap Rule

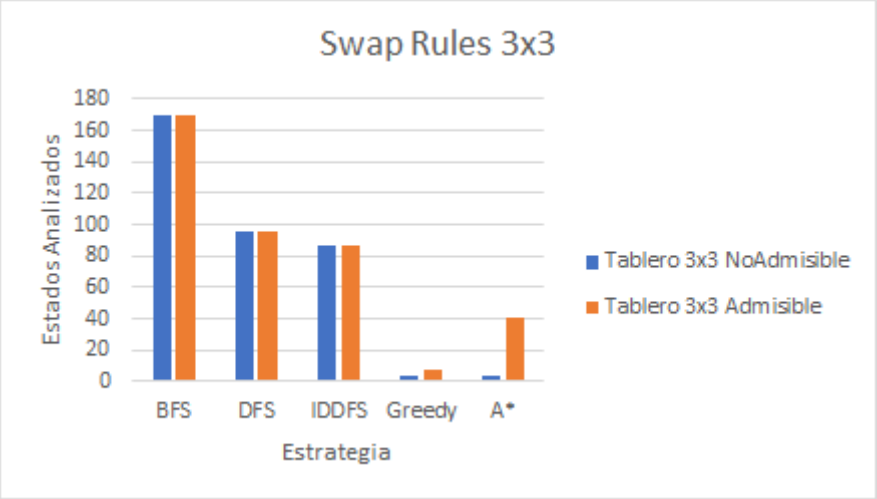


Figura 5: Gráfico búsqueda con Swap Rule

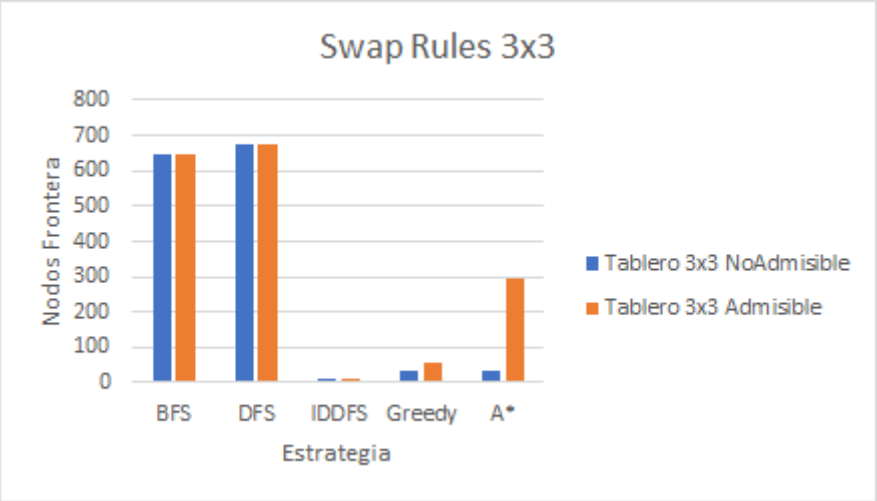


Figura 6: Gráfico búsqueda con Swap Rule

Tablero 3x3 NoAdmisible					
Estrategia	Depth	Cost	Nodos Explotados	Estados Analizados	Nodos Frontera
BFS	4	4	170	170	649
DFS	96	96	96	96	675
IDDFS	4	4	54	87	11
Greedy (No admisible)	4	4	4	4	32
A* (No admisible)	4	4	4	4	32
Greedy (Admisible)	4	4	7	7	56
A* (Admisible)	4	4	40	40	294

Figura 7: Tablas búsqueda con Swap Rule

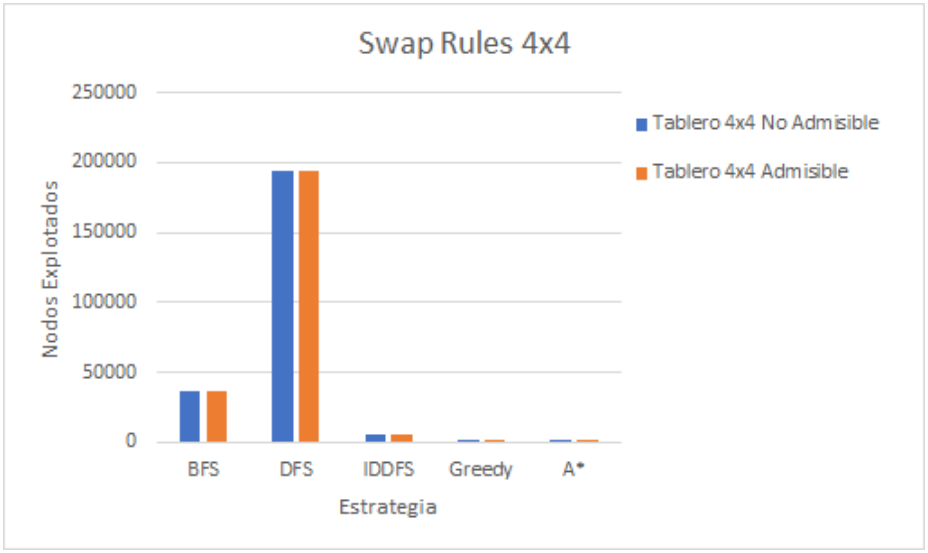


Figura 8: Gráfico búsqueda con Swap Rule

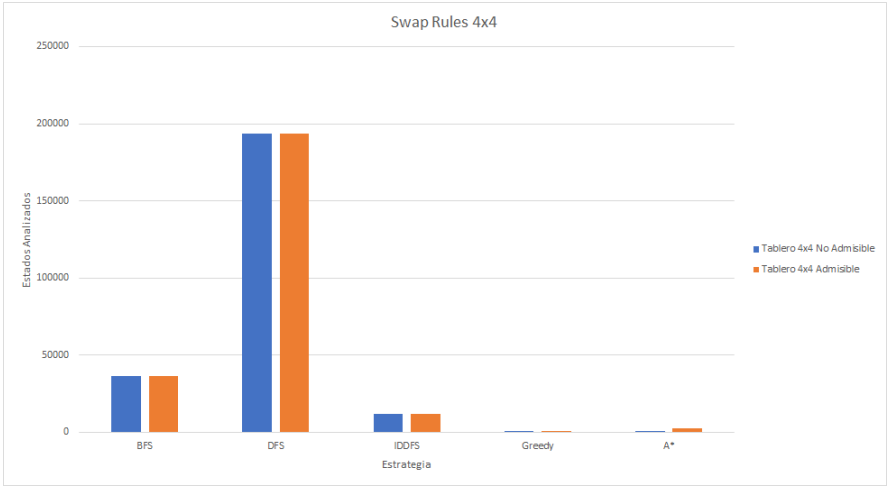


Figura 9: Gráfico búsqueda con Swap Rule

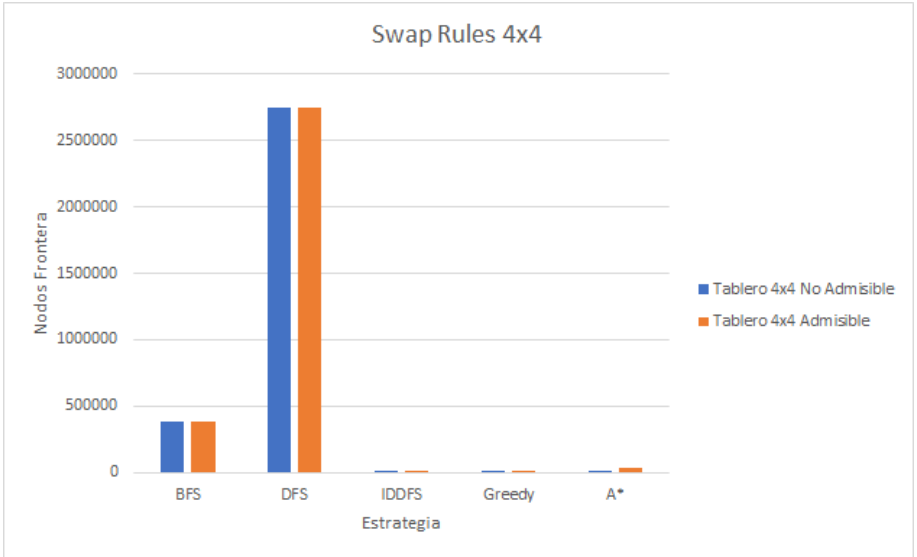


Figura 10: Gráfico búsqueda con Swap Rule

Tablero 4x4 No Admisible					
Estrategia	Depth	Cost	Nodos Explotados	Estados Analizados	Nodos Frontera
BFS	6	6	36373	36373	378385
DFS	193696	193696	193696	193696	2745437
IDDFS	6	6	5339	12048	32
Greedy (No Admisible)	10	10	15	15	222
A* (No Admisible)	6	6	12	12	177
Greedy (Admisible)	6	6	15	15	221
A* (Admisible)	6	6	2380	2380	33103

Figura 11: Tablas búsqueda con Swap Rule