Sistemas de Inteligencia Artificial - 2019 1ºC

TPE Nº1: Algoritmos de Búsqueda

Integrantes

Nombre	*Correo*	*Legajo*
Dammiano, Agustín	adammiano@itba.edu.ar	57702
Donoso Naumczuk, Alan	adonoso@itba.edu.ar	57583
Sanz Gorostiaga, Lucas	lsanz@itba.edu.ar	56312
Torreguitar, José	jtorreguitar@itba.edu.ar	57519

1. Introducción

El trabajo que se detalla en este informe consistió en analizar un problema, representarlo adecuadamente e implementar un sistema que, basado en una API provista por la cátedra, permitiera encontrar, mediante distintos algoritmos de búsqueda (tanto informados como no informados), una solución al mismo. En nuestro caso el problema planteado fue el juego GridLock.

La API provista por la cátedra consiste en una serie de interfaces (State, Rule, Probem, Heuristic) que hacen que el sistema a implementar quede conformado por dos claros bloques modulares: la implementación del GPS y la implementación del problema. Permitiendo esto último, intercambiar cualquiera de los dos módulos por otra implementación que también cumpla con la API, por ejemplo: cambiar el módulo de implementación de problema por la implementación de otro problema que cumpla con la API y que nuestro módulo de GPS sepa resolverlo.

2. Problema

2.1. Descripción del problema

2.1.1 Descripción de los componentes del juego

El juego GridLock tradicionalmente cuenta con un tablero cuadrado, dividido en su interior en celdas iguales (como un tablero de ajedrez). Los bordes del tablero son paredes y cuenta con un hueco en uno de los bordes, que es la salida.

A su vez, el tablero cuenta con piezas que son bloques estrictamente rectangulares. Uno de estos bloques está distinguido de los demás, al que llamamos “Bloque objetivo” o “Goal block”.

2.1.2 Reglas del juego

Los bloques pueden moverse únicamente en la dirección paralela a su lado más largo, en ambos sentidos. Las paredes y los demás bloques son sólidos, por lo cual no pueden atravesarse entre sí, impidiendo ciertos movimientos en ciertas configuraciones. Tradicionalmente, mover un determinado bloque de una posición a otra se cuenta como un “turno”. En la sección de representación del problema hacemos una salvedad respecto al turno en nuestra implementación.

2.1.3 Objetivo del juego

El objetivo del juego es que el bloque objetivo llegue a la salida del tablero.

2.1.4 Configuración inicial del juego

La configuración inicial del juego es variable, dependerá del nivel de dificultad que se desee. En general, los bloques se encontrarán dispuestos de tal forma que el bloque objetivo encuentre el camino hacia la salida bloqueado por otros bloques. Obviamente, el bloque objetivo debe estar situado de tal forma que esté alineado con la salida, de lo contrario no podría lograrse el objetivo.

2.2. Representación del problema

2.2.1 Representación de componentes básicos

Una configuración de un instante del juego se encuentra representada por nuestra representación del tablero, mediante la clase `BoardGridLock` que cuenta con una matriz de números enteros (esta matriz no necesariamente es cuadrada como en la definición tradicional del juego). Cada bloque tiene un identificador,

un número mayor o igual a cero. Por lo tanto, cada celda de la matriz contendrá el número identificador que representa una parte de un bloque situada en dicho lugar del tablero, o un -1 en caso de ser una celda vacía.

A su vez, esta clase cuenta con un `Point` cuyos puntos (x, y) representan la (fila, columna) del tablero en la cual está situada la salida, y una lista de objetos `BlockGridLock`, que está instanciada como una `ArrayList` para optimizar la obtención de un determinado bloque, ya que en dicha lista, los bloques estarán ordenados con respecto a su número identificador.

`BlockGridLock` es nuestra representación de un bloque, cuenta su número identificador, con sus dos direcciones de movimiento y con sus posiciones de inicio y de fin, representadas por un `Point`. Para optimizar el proceso de chequeo de aplicabilidad de una regla, y el proceso de aplicación de la regla, tomamos la convención de que si el bloque es un rectángulo horizontal, su inicio será el extremo más a la izquierda y su fin el extremo más a la derecha. Mientras que para un bloque vertical, el inicio será el extremo situado más arriba y el fin el que se encuentre más abajo en el tablero.

2.2.2 Implementación de la API provista por la cátedra

2.2.2.1 State

Nuestra implementación, `StateGridLock`, cuenta únicamente con una variable de tipo `BoardGridLock`. Es decir, la representación de un estado esta delegada y totalmente vinculada a la representación de un tablero. El `equals` de nuestro `State` delega la comparación al `equals` de `BoardGridLock`, el cual compara celda a celda la matriz que representa el tablero y que coincidan los puntos de salida del mismo.

2.2.2.2 Rule

Contamos con dos reglas para cada bloque, una para cada dirección de movimiento que tiene permitido. Cada regla tiene el identificador del bloque y la respectiva dirección. Por ejemplo, para un bloque horizontal, estará la regla de moverlo a la derecha y la regla de moverlo a la izquierda.

A diferencia del problema tradicional, para tener menos reglas, definimos que en cada turno podemos mover un bloque únicamente una celda en una dirección. Cada regla tiene costo 1, que representa el costo de gastar un turno en realizar dicho movimiento.

2.2.2.3 Problem

En nuestra implementación, el método `isGoal` verifica que el bloque objetivo tenga alguno de sus extremos ubicados en el punto de salida.

2.2.2.4 Heuristic

Se implementaron dos heurísticas. La primera (denotada como la número 0), consiste en devolver la cantidad de celdas entre el bloque objetivo y la salida; la segunda (denotada como la número 1) consiste en devolver la cantidad de celdas entre el bloque objetivo y la salida sumado a la cantidad de bloques que se interponen en el camino del bloque objetivo hacia la salida.

Ambas heurísticas son admisibles: al tener bloques en el camino podemos asegurar que se van a tener que mover para liberar la salida, por lo tanto, como mínimo se necesitarán tantos turnos como número de bloques se interpongan. Y una vez que el camino se encuentre libre, el bloque objetivo va a tener que moverse la cantidad de celdas que hay desde donde se encuentra hasta la salida para poder llegar a la solución.

3. Motor de Inferencia

3.1. Búsqueda desinformada

3.1.1. Breadth-First Search (BFS)

El algoritmo de búsqueda BFS consiste en analizar todos los nodos de un mismo nivel antes de avanzar al siguiente. Gracias a esto, es un algoritmo de búsqueda óptimo, ya que si existe solución, la encontrará en el primer nivel en el que se haye. Se implementa utilizando como frontera una `LinkedList` utilizándola como una `Queue`.

3.1.2. Depth-First Search (DFS)

El algoritmo de búsqueda DFS consiste en analizar por ramas. Tomará una rama e irá lo más profundo posible dentro de la misma hasta que se encuentre con el objetivo o llegue a un nodo que sea una hoja. En caso de llegar a un nodo hoja, se realiza la acción conocida como backtracking. En nuestro caso, como evitamos estados repetidos, también será condición de backtracking llegar a un estado ya visitado anteriormente. Se implementó utilizando una `LinkedList` como frontera, utilizándola como si fuera un `Stack`.

3.1.3. Iterative Deepening Depth-First Search (IDDFS)

El algoritmo consiste en realizar iterativamente búsquedas DFS pero con profundidad limitada, iniciando con la profundidad limitada en 1 y terminando en `maxDepth`. El parámetro método `maxDepth` permite delimitar al usuario la profundidad máxima del árbol que se quiere analizar. Dado que la misma no se puede saber de antemano a menos que se encuentre la solución con otro método y se sepa en qué nivel se encuentra, se utiliza una profundidad máxima que sobreestime la profundidad real para que el algoritmo pueda encontrar la solución.

3.2. Búsqueda informada

3.2.1. Greedy Search

El método de búsqueda GREEDY intenta acercarse a la solución por medio de una heurística. Existen dos maneras de implementarlo: explotar el nodo de menor heurística de la frontera o el de mejor heurística de los nodos recientemente expandidos. En este trabajo se implementó la segunda. La frontera es una pila a la cual se le agrega los hijos ordenados dejando en el tope de la pila los de menor valor de heurística. Dicho orden surge de agregar los hijos a una cola de prioridades. En caso de haber empates, se resuelven aleatoriamente.

3.2.2. A*

El algoritmo de búsqueda A* escoge en cada paso el mejor nodo de la frontera basándose en la siguiente regla: el mejor nodo es aquel que tenga menor **heurística** + **costo**, en caso de existir un empate, será mejor el que cuente con menor **heurística**. En caso de empatar en este caso también, se define **aleatoriamente** entre los nodos empatados. La frontera está implementada como una `PriorityQueue` cuyo `Comparator` sigue las reglas detalladas anteriormente. Para el desempate aleatorio se enlistan los nodos empatados, y se genera un número aleatorio dentro del intervalo [0, cantidad de nodos empatados - 1], luego se escoge de la lista de empatados el nodo ubicado en la posición del número generado.

4. Procedimiento

Se realizaron 100 ejecuciones del programa para cada posible variación del mismo. Una de esta variaciones es la de la configuración inicial del tablero, el cual puede inicializarse en los niveles 1, 3, 8, 27 y 40 del juego GridLock provisto por la cátedra como ejemplo. También se variaron los métodos de búsqueda

entre los mencionados anteriormente. A su vez, para los métodos de búsqueda informada, se usaron dos heurísticas distintas, la número 0 y la número 1, explicadas en la sección 2.2.2.4.

5. Análisis de resultados

Se puede observar en los gráficos (ver anexo) de estados analizados, para cualquier nivel, que el método de búsqueda GREEDY analiza en promedio menos estados. Lo cual se debe a que dicho algoritmo intenta acercarse a la solución lo más posible, gracias a las heurísticas que le proporcionan un estimación de que tan lejos esta la solucion. Además, GREEDY es unos de los algoritmos más rápidos en promedio (ver gráfico de tiempo en el anexo). Esto tiene sentido, ya que el problema GridLock tiene varias formas de llegar a la solución y esto es muy favorable para GREEDY, porque aunque no se vaya por la rama de la solución óptima es posible que haya una solución en dicha rama.

Al mirar los gráficos de los costos de las soluciones, para cualquier nivel, se puede destacar que los algoritmos BFS, IDDFS y A* encuentran las mejores soluciones (siendo mejores las soluciones de menor costo). Entre estos 3, el algoritmo A* es que expande menos nodos pero es el que tarda más tiempo (ver gráficos de nodos expandidos y tiempo). El resultado del tiempo del algoritmo A* se puede deber a que las heurísticas que realizamos son costosas de aplicar, no estiman muy bien los pasos que faltan o por el desempate aleatorio. El algoritmo IDDFS es el que expande más nodos (ver gráfico de nodos explotados).

Podemos ver, comparando los algoritmos de búsqueda desinformada, que cada uno tiene sus ventajas y desventajas. Al ver DFS, por ejemplo, vemos que claramente es el algoritmo con la solución de mayor costo. Sin embargo, en cuanto a tiempos tiene uno de los mejores, siendo solo superado por el GREEDY de heurística número 0. Como se dijo anteriormente, el BFS siempre encuentra la solución óptima debido a cómo analiza los estados, sin embargo, siempre es el algoritmo con más estados analizados.

6. Conclusión

Concluimos que el algoritmo BFS sirve si se quiere encontrar una solución óptima pero no se puede formar una heurística admisible. De conocerse una heurística, entonces A* sería una mejor opción.

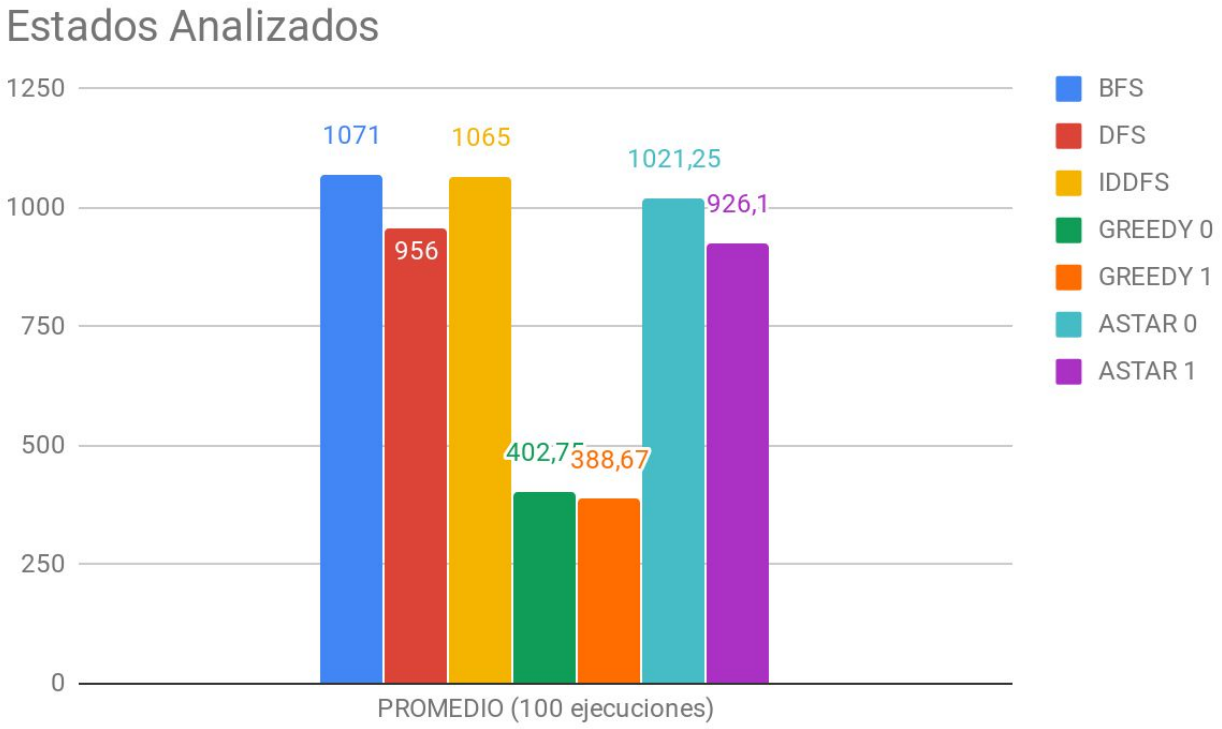
Por el contrario, si fuera de interés encontrar una solución, en poco tiempo, aceptando subóptimos, se podría usar DFS o GREEDY.

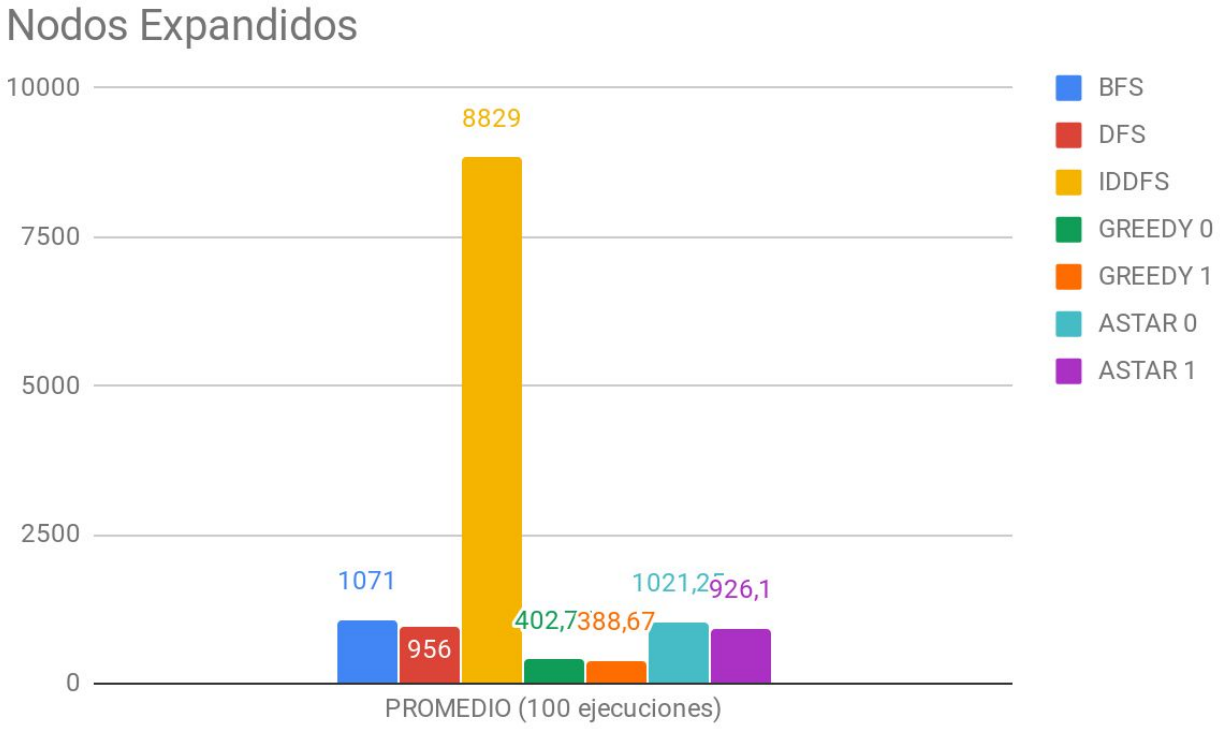
En caso de que se necesite encontrar una solución óptima y se tuviera un gran factor de ramificación, podría ser conveniente usar IDDFS ya que este solo guarda los nodos hermanos a los nodos que conforman el camino que se está recorriendo mediante DFS, en cambio BFS guardaría todo un nivel. Por ejemplo, suponiendo un factor de ramificación (FR) de 10, si nos encontramos en la profundidad 12 del árbol, en la frontera de IDDFS habrían **FR + ((NIVEL - 1) x (FR - 1)) = 10 + ((12 - 1) x (10 - 1)) = 109** nodos, mientras que en BFS habrían **FR elevado al NIVEL = 10 elevado a la 12** nodos.

En resumen, podemos decir que todos los algoritmos de búsqueda analizados en este trabajo son útiles, lo importantes es conocer el tipo de solución que buscamos y los recursos con los que contamos (tiempo, memoria, etc.), para luego escoger el algoritmo que se adapte mejor a nuestras necesidades, ya que cada uno ofrece una compensación en un aspecto a cambio de una pérdida en otro.

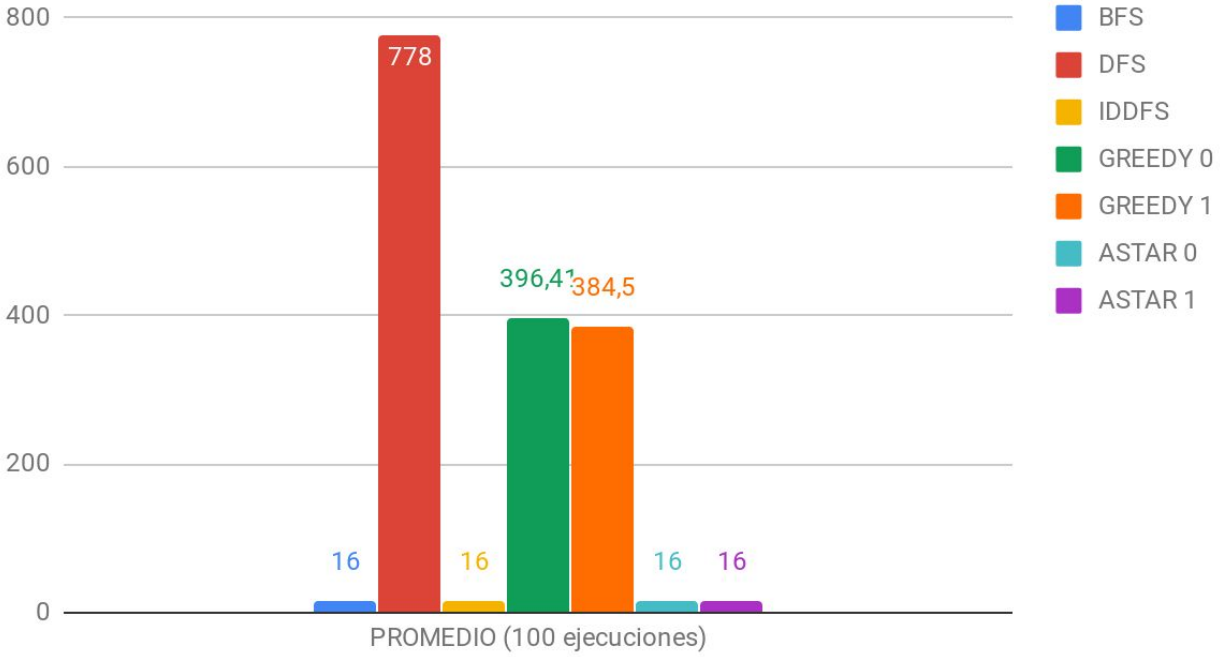
Anexo

Nivel 1

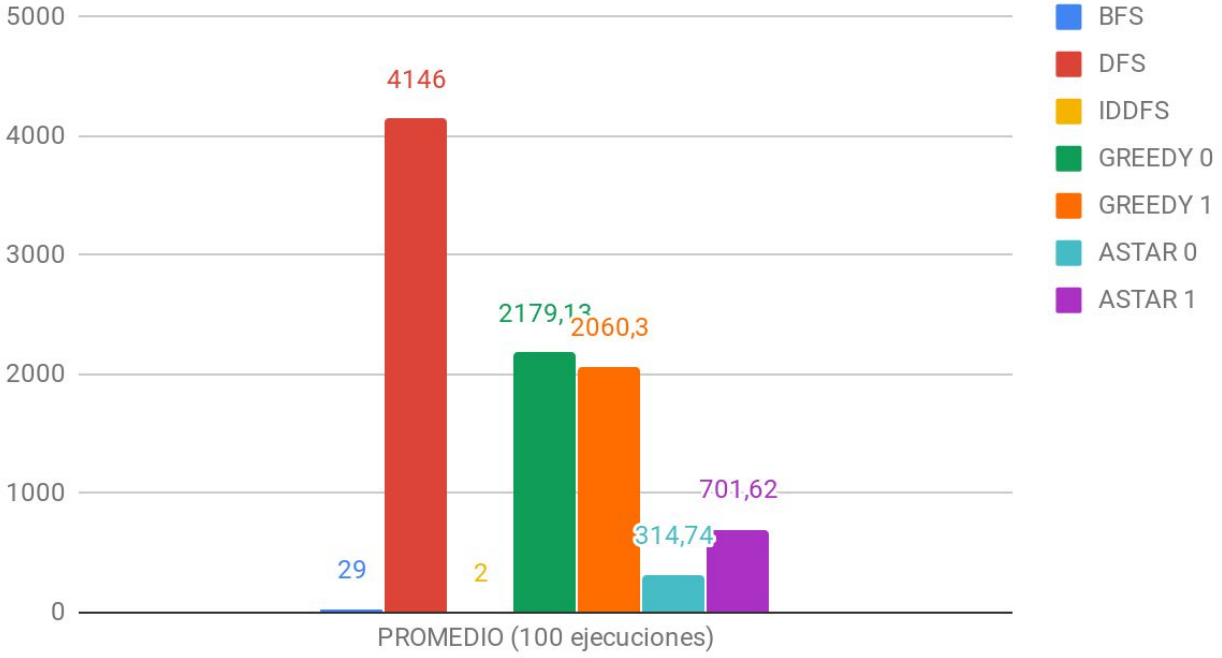


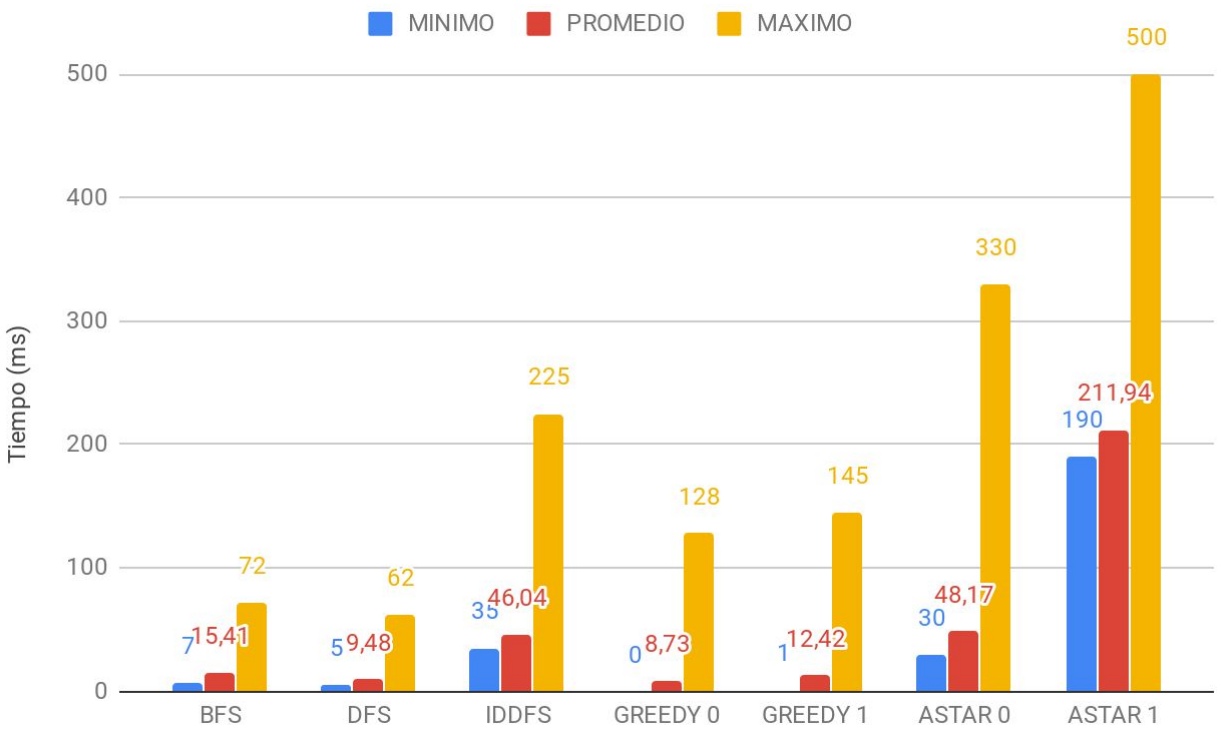


Costo de la solucion

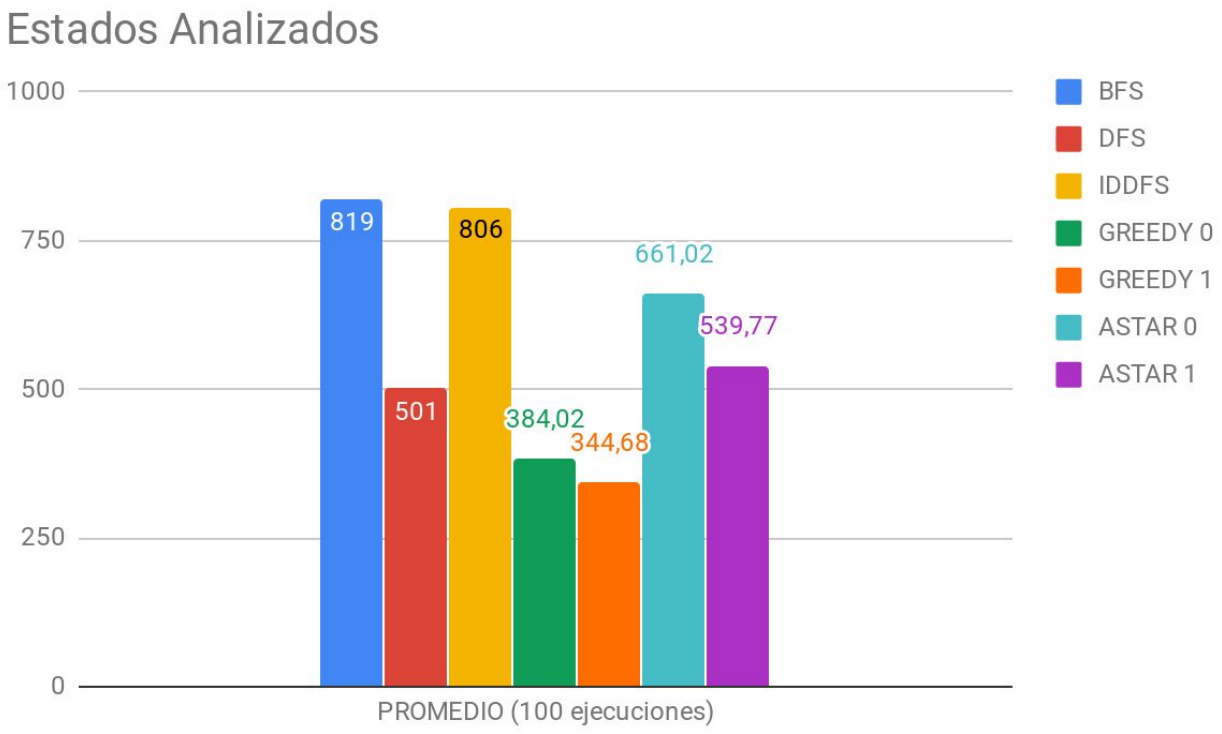


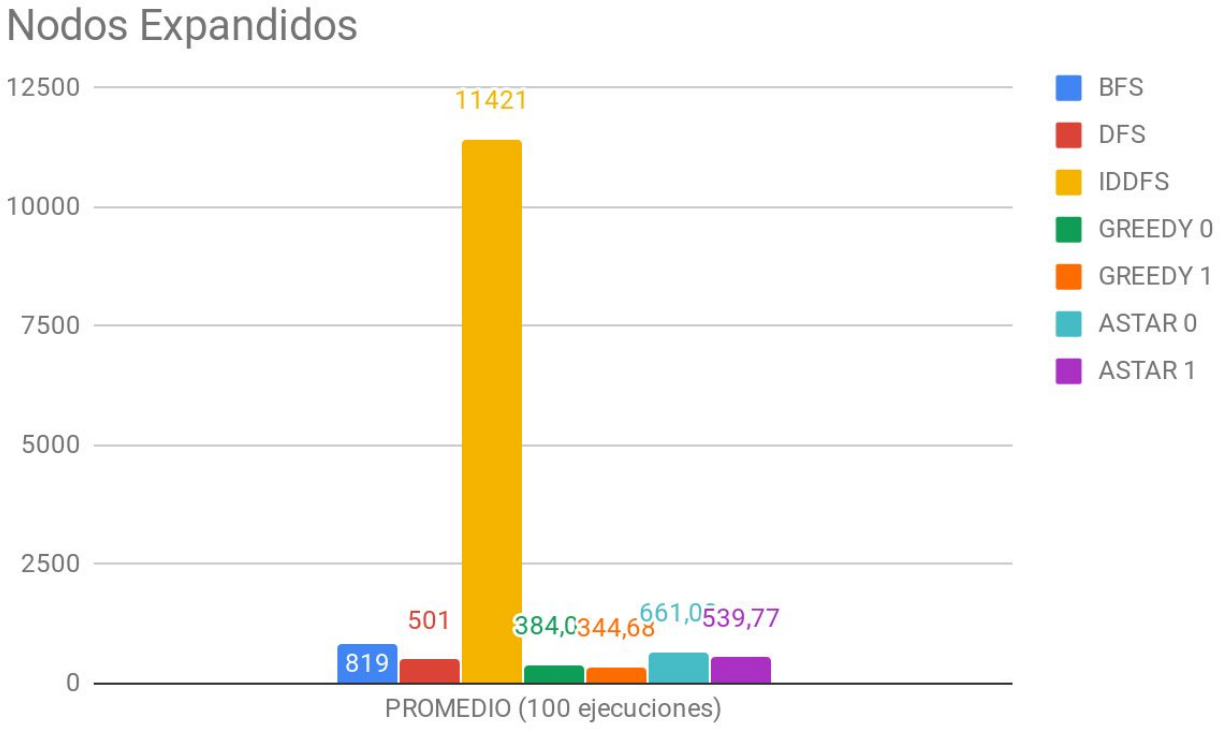
Nodos en la Frontera



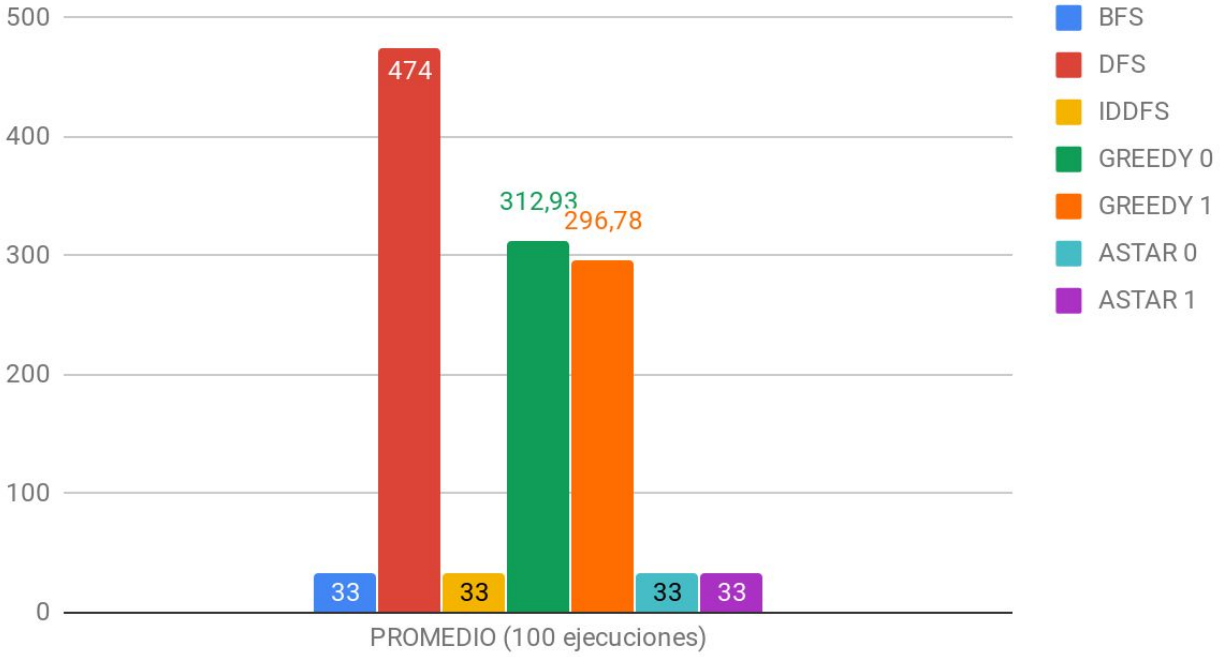

MINIMO
PROMEDIO
MAXIMO
Tiempo (ms)
500
400
300
200
100
0
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1
7
15,41
72
5
9,48
62
35
46,04
225
0
8,73
128
1
12,42
145
30
48,17
330
190
211,94
500

Nivel 3

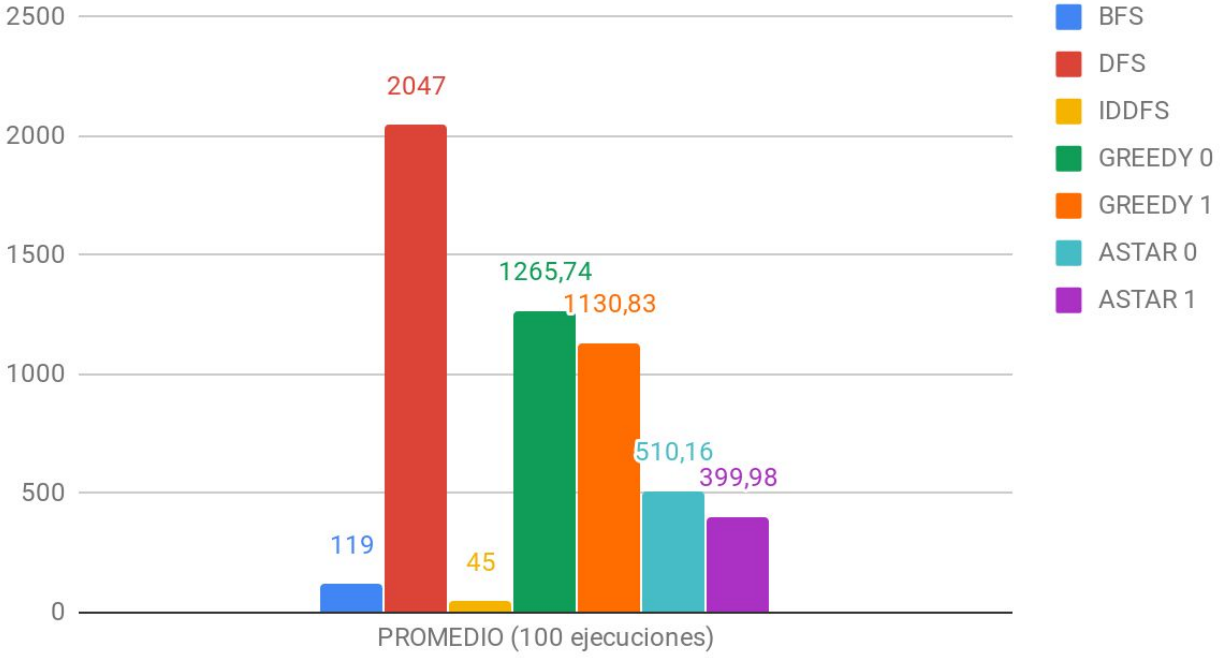

Estados Analizados
1000
750
500
250
0
819
501
806
384,02
344,68
661,02
539,77
PROMEDIO (100 ejecuciones)
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

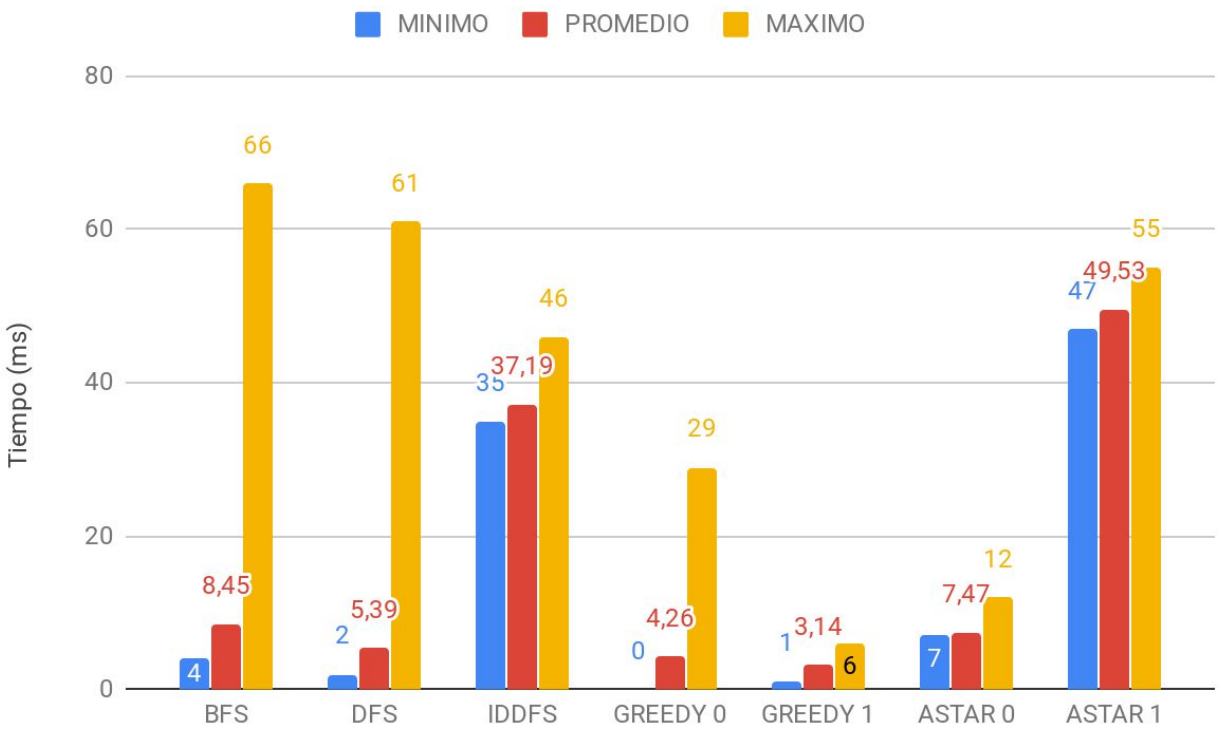

Nodos Expandidos
12500
10000
7500
5000
2500
0
819
501
11421
384,0
344,68
661,0
539,77
PROMEDIO (100 ejecuciones)
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

Costo de la solucion

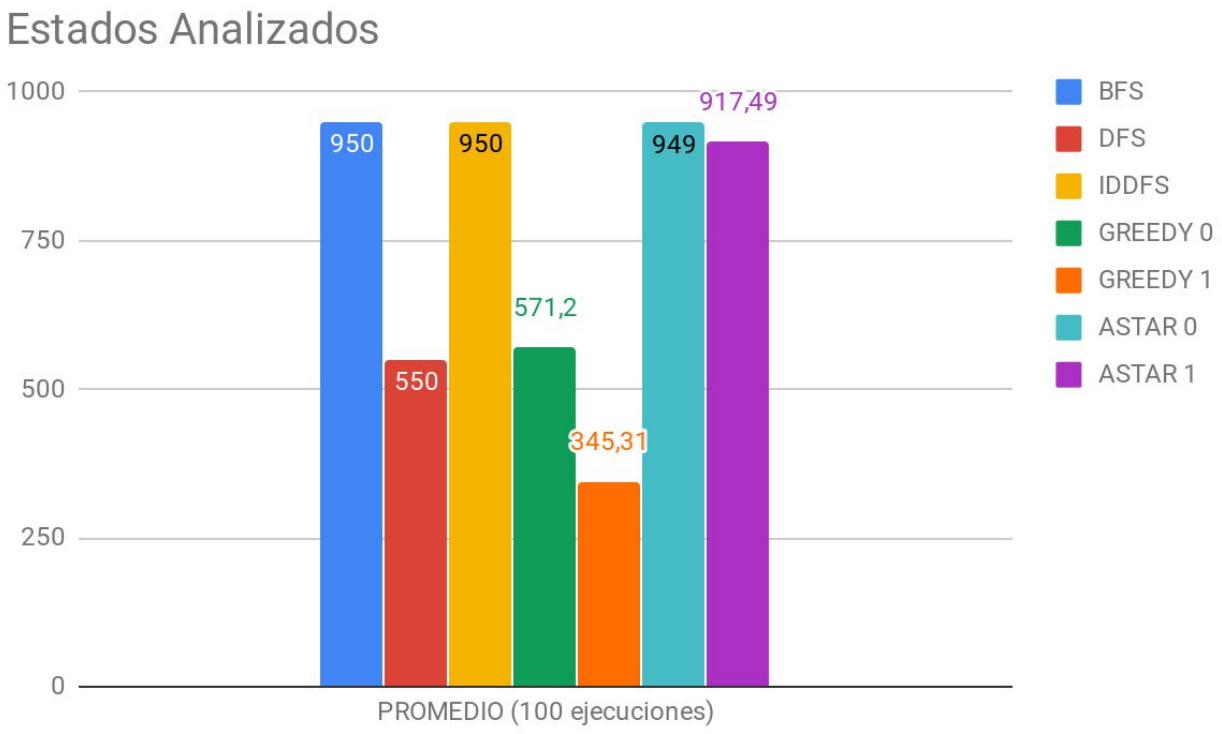


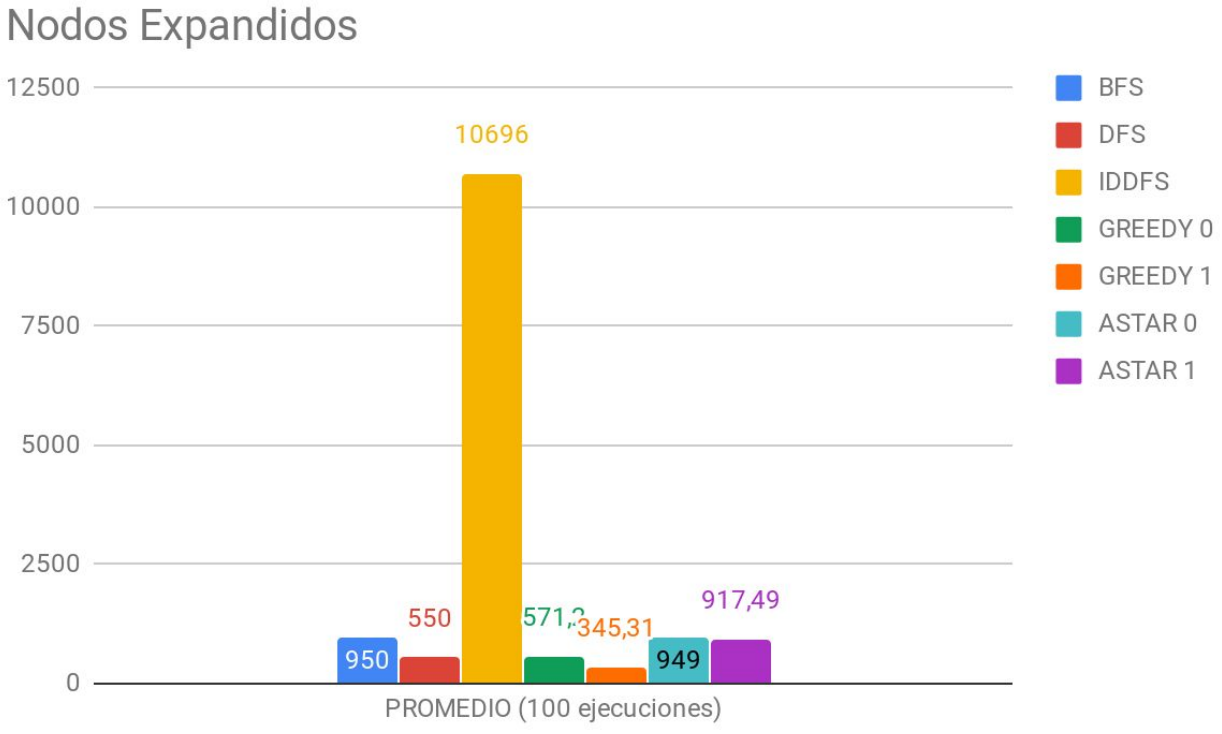
Nodos en la Frontera



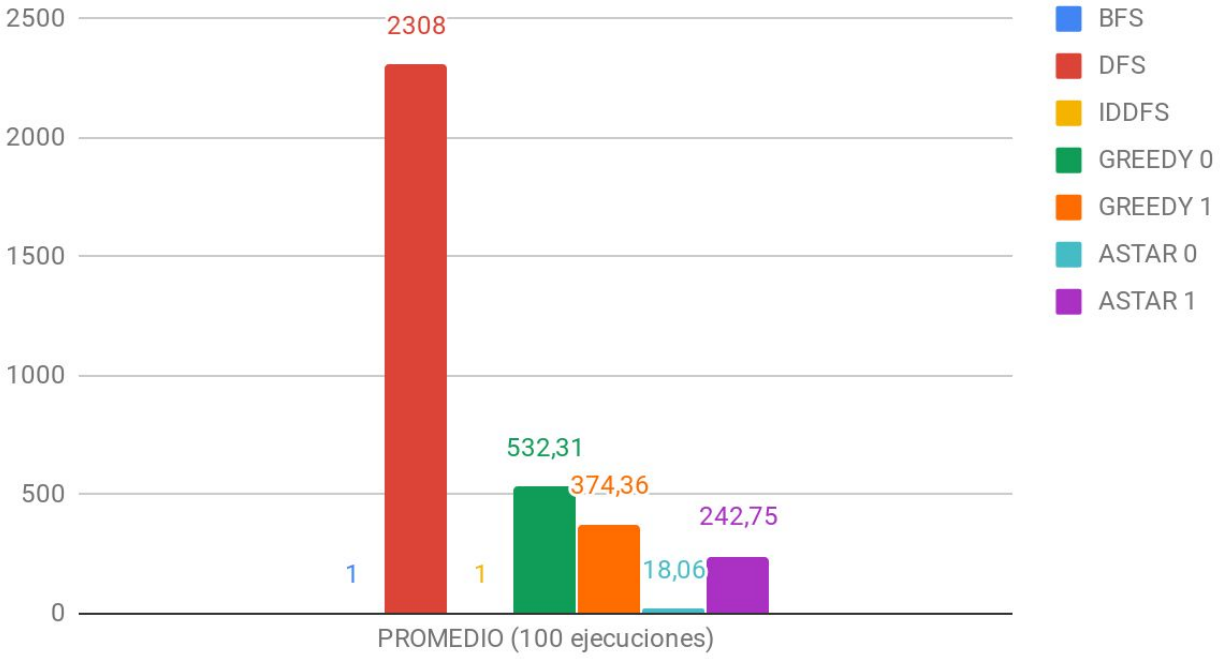

MINIMO
PROMEDIO
MAXIMO
Tiempo (ms)
80
60
40
20
0
66
61
46
29
12
55
8,45
5,39
37,19
4,26
3,14
7,47
49,53
4
2
35
0
1
6
7
47
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

Nivel 8

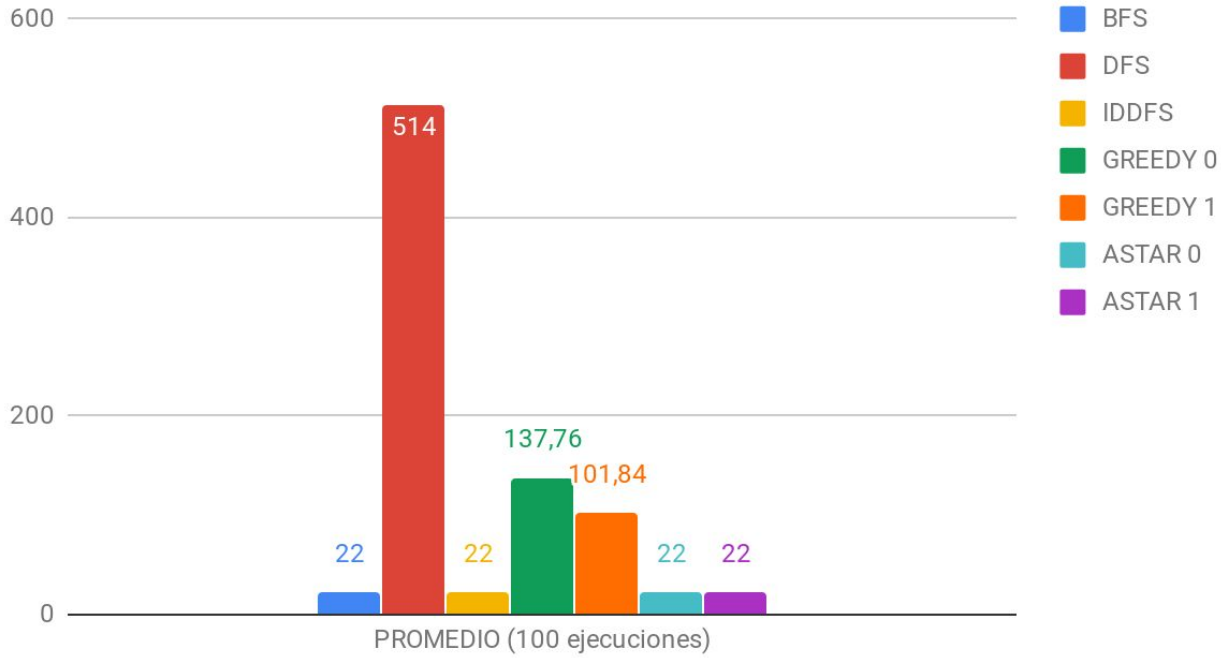

Estados Analizados
1000
750
500
250
0
950
550
950
571,2
345,31
949
917,49
PROMEDIO (100 ejecuciones)
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

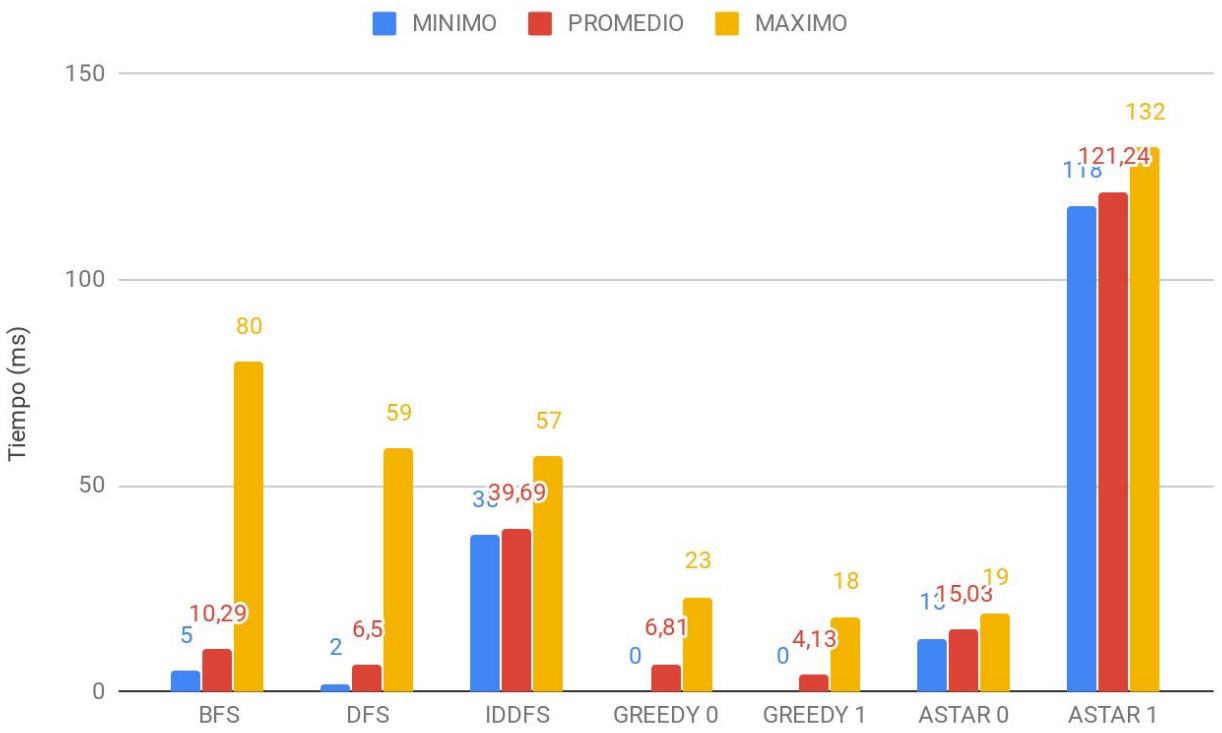

Nodos Expandidos
12500
10000
7500
5000
2500
0
10696
950
550
571,2
345,31
949
917,49
PROMEDIO (100 ejecuciones)
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

Nodos en la Frontera

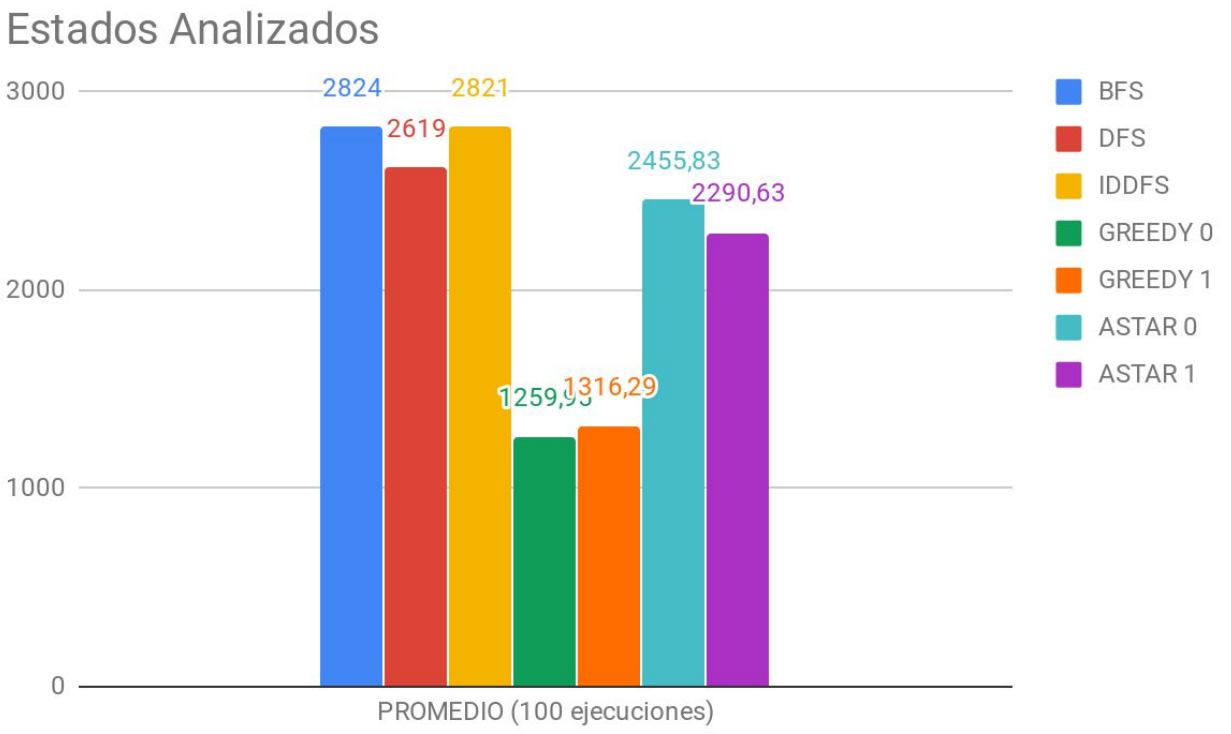


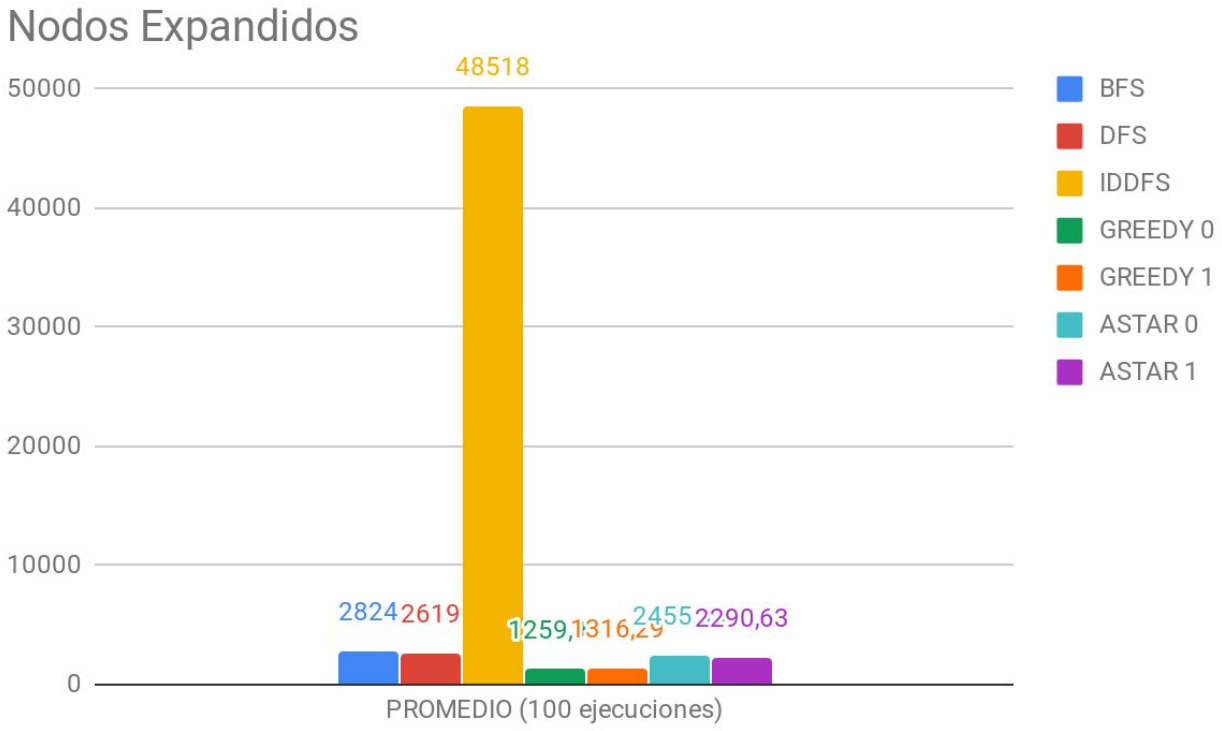
Costo de la solucion



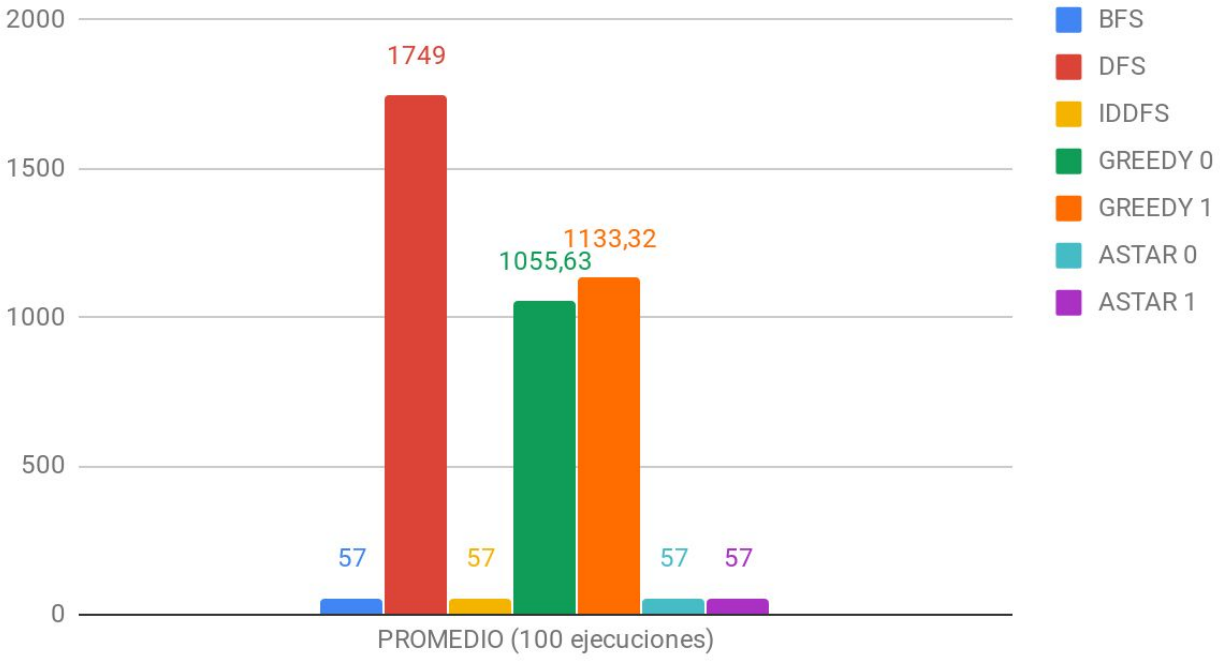

MINIMO
PROMEDIO
MAXIMO
150
100
50
0
Tiempo (ms)
5
10,29
80
2
6,5
59
39,69
57
0
6,81
23
0
4,13
18
15,03
19
121,24
132
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

Nivel 27

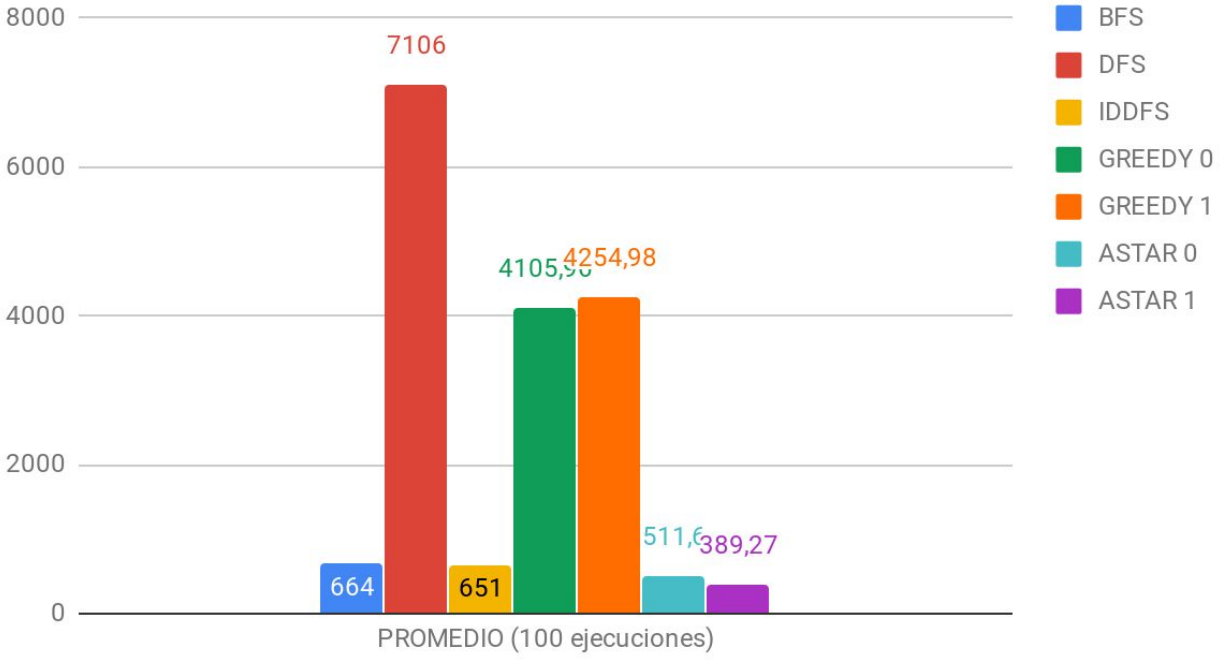

Estados Analizados
3000
2000
1000
0
2824
2619
2821
1259,99
1316,29
2455,83
2290,63
PROMEDIO (100 ejecuciones)
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

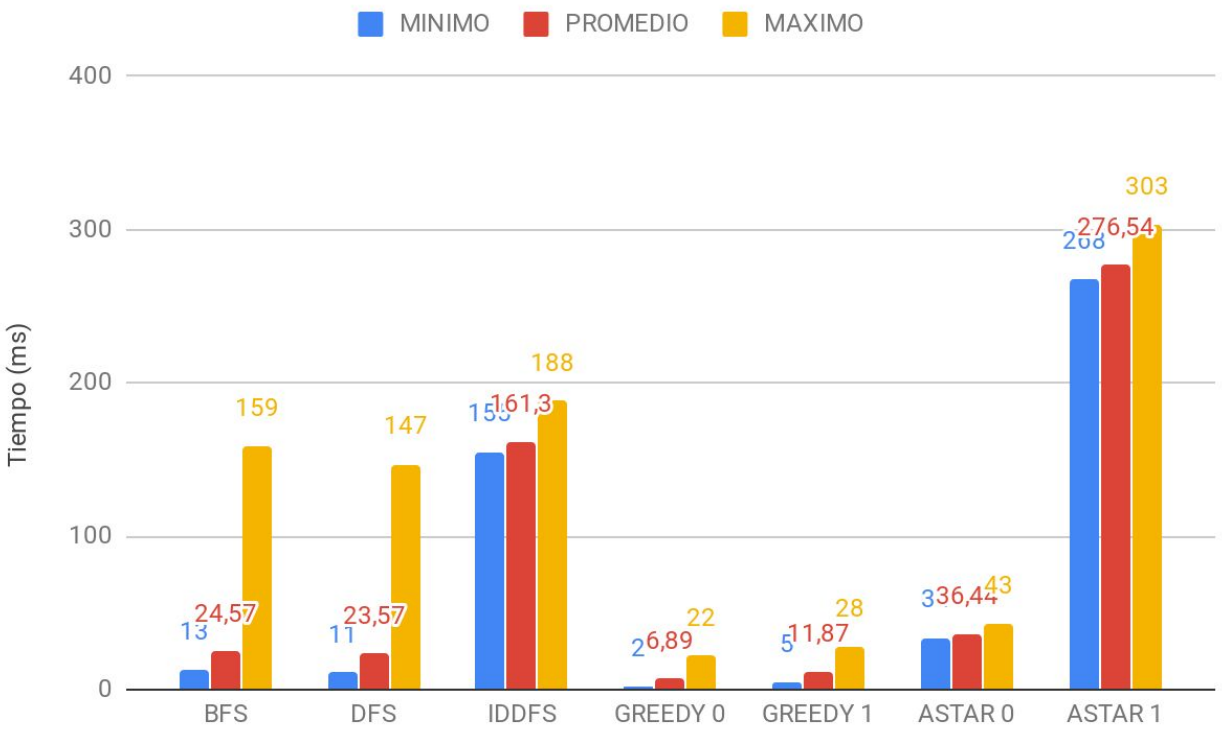

Nodos Expandidos
50000
40000
30000
20000
10000
0
48518
2824
2619
1259,1
1316,29
2455
2290,63
PROMEDIO (100 ejecuciones)
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

Costo de la solucion

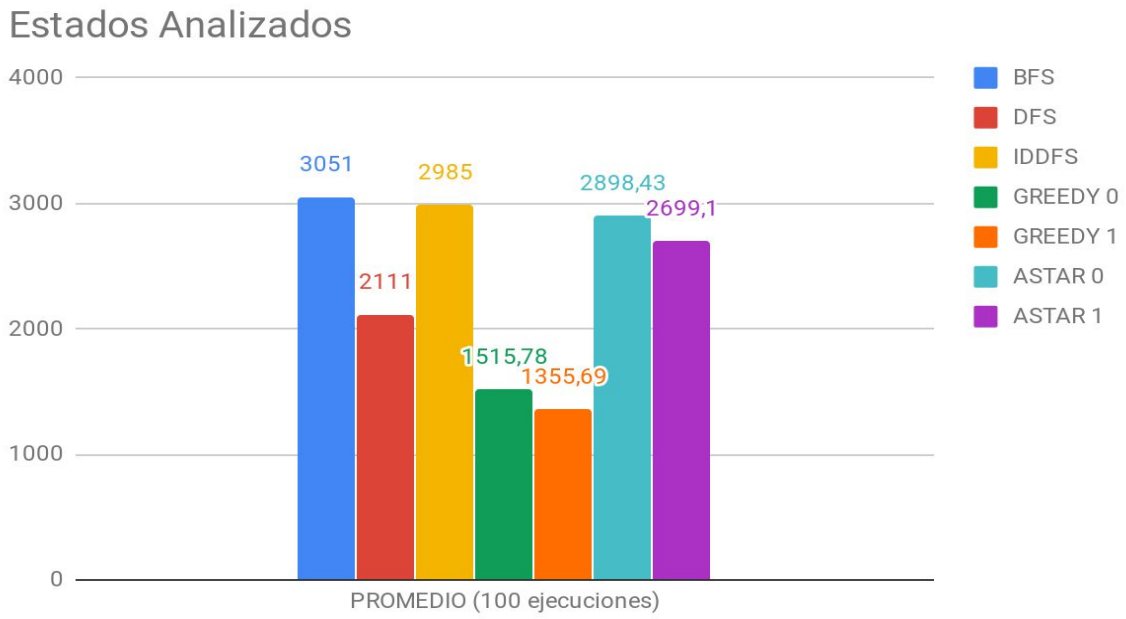


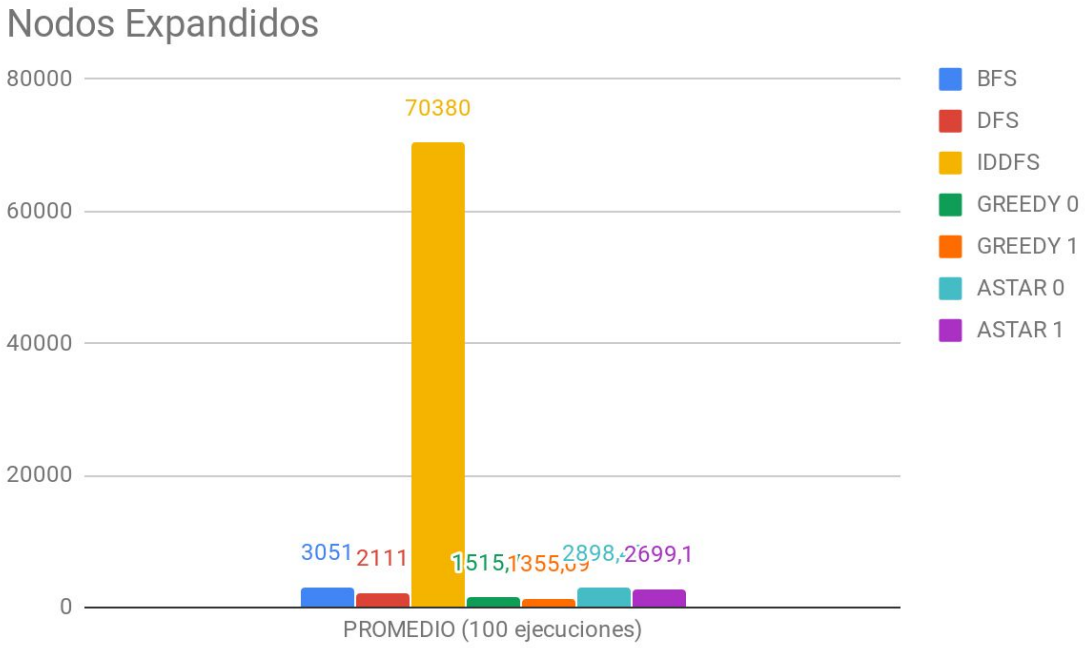
Nodos en la Frontera



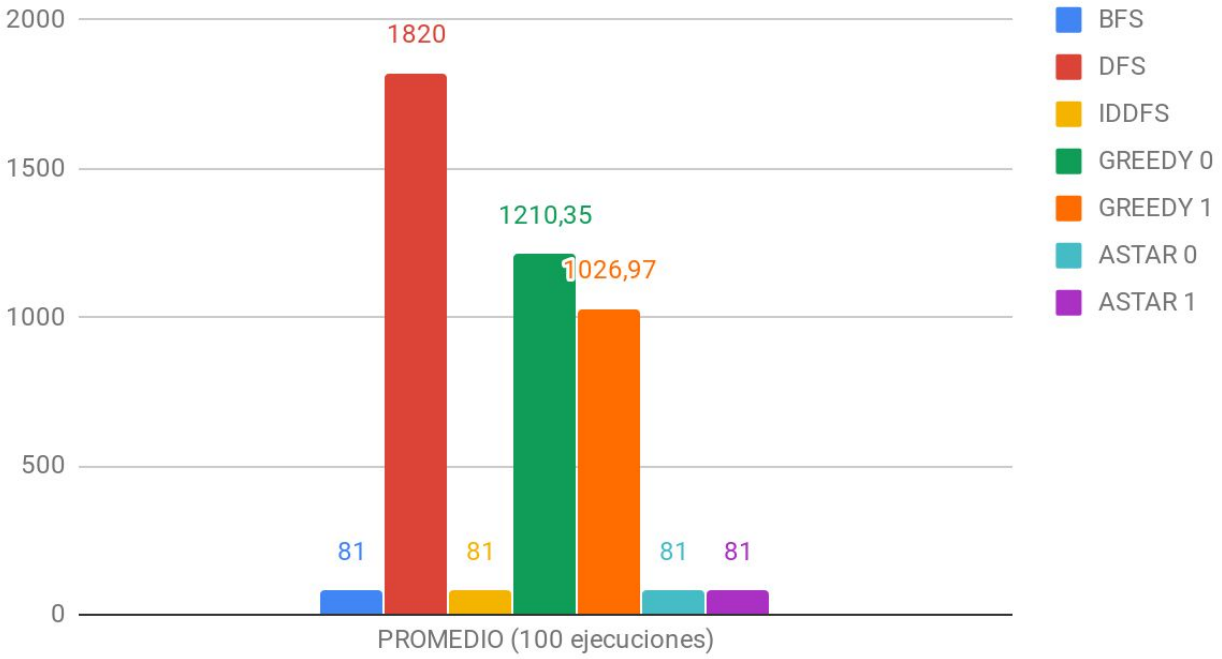

MINIMO
PROMEDIO
MAXIMO
Tiempo (ms)
400
300
200
100
0
13
24,57
159
11
23,57
147
153
161,3
188
2
6,89
22
5
11,87
28
336,44
43
303
276,54
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

Nivel 40

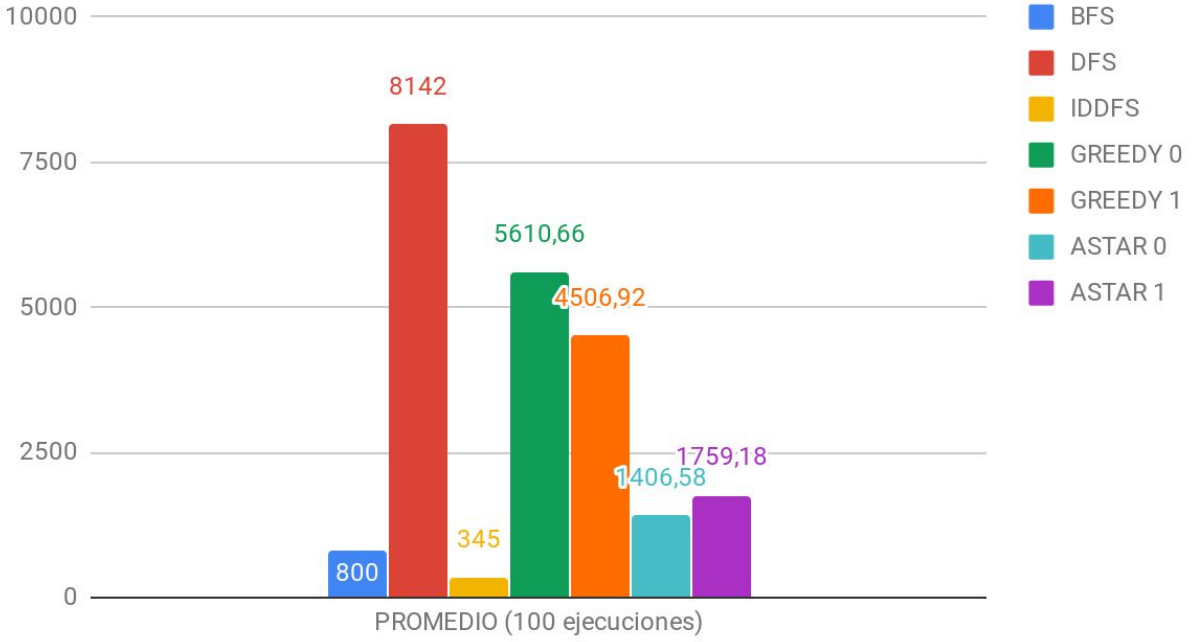

Estados Analizados
4000
3000
2000
1000
0
3051
2111
2985
1515,78
1355,69
2898,43
2699,1
PROMEDIO (100 ejecuciones)
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

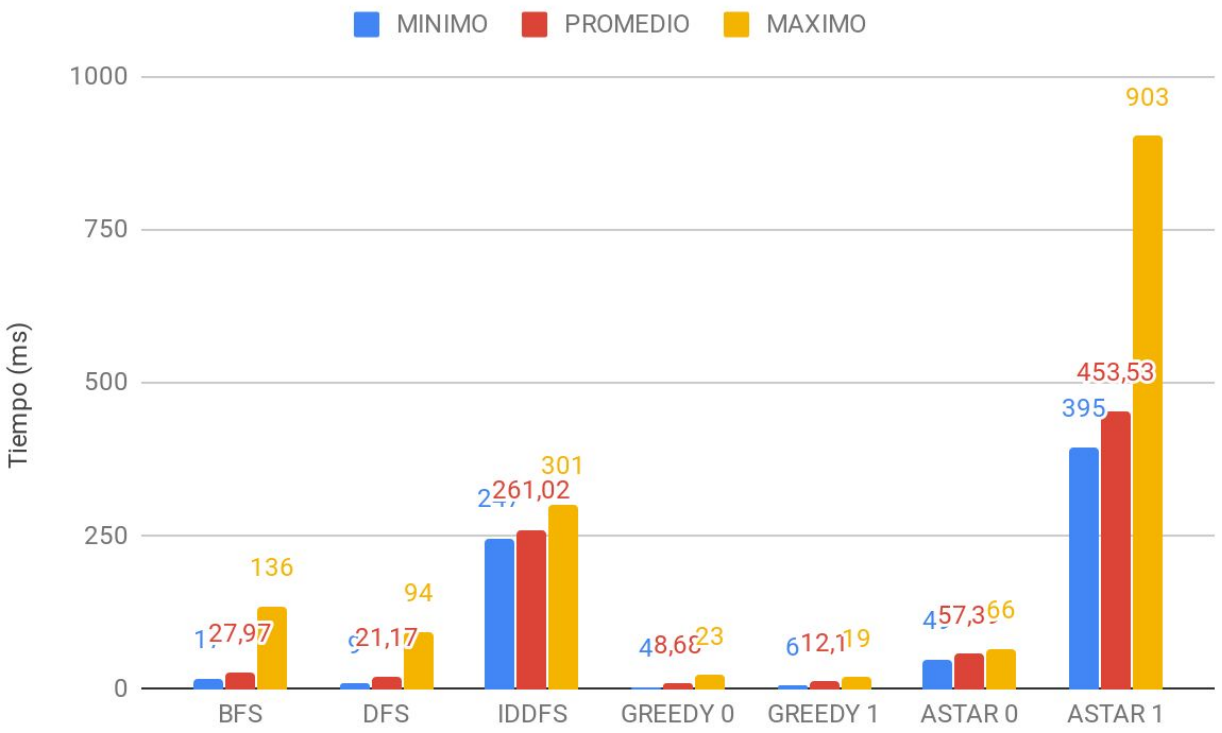

Nodos Expandidos
80000
60000
40000
20000
0
70380
3051
2111
2699,1
PROMEDIO (100 ejecuciones)
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1

Costo de la solucion



Nodos en la Frontera




MINIMO
PROMEDIO
MAXIMO
1000
750
500
250
0
Tiempo (ms)
903
453,53
395
301
261,02
136
94
BFS
DFS
IDDFS
GREEDY 0
GREEDY 1
ASTAR 0
ASTAR 1