




Métodos de Búsqueda Desinformados e Informados

Gastón Lifschitz
Lucio Pagni



Trabajo Realizado



Se implementó un generador de soluciones para el juego Sokoban. Estas soluciones se generan según métodos de búsqueda desinformados (DFS , IDDFS y BFS) e informados (A^* , IDA^* y GGS). También se implementó una modificación de IDA^* (CUSTOM_ IDA^*).

Se desarrollaron 4 heurísticas, 3 de ellas admisibles y una de ellas no admisible.(ver diapo. 4).

También se buscó implementar algunos casos simples de deadlocks para optimizar la búsqueda. Estos casos se fijan de no estar en una esquina y de no estar contra una pared en la cual no hay storage. No siempre optimizan (ver tablas).



Resultados



Para poder estudiar los resultados de los diferentes algoritmos implementados se corrieron todos los algoritmos para un tablero fijo con distintos parámetros de configuración. De esta manera, podemos establecer cierta comparabilidad entre los distintos algoritmos.

Tablero 1



```
      ###  
      #.#  
    #####  #####  
   ##                ##  
 ##   #  #  #  #  ##  
#    ##                ##  
#  ##   ##   ##  
#.#    @$    .  #  
####   ##   ####  
      ####.####
```



	BFS	DFS	IDDFS
Heurística	-	-	-
Deadlocks	-	-	-
Resultado	Éxito	Éxito	Éxito
Profundidad/ Costo	20	236	26
Nodos Expandidos	37821	9511	444
Nodos Frontera	9288	2524	1206
Tiempo [s]	101.806	13.035	55.466



Vemos que BFS tiene menos profundidad que DFS. Consume más recursos pero encuentra una solución con menos costo.

DFS tiene muchos menos nodos expandidos y nodos frontera que BFS.

IDDFS tiene menos nodos frontera y menos nodos expandidos.

Una característica de estos algoritmos es que no hay nodos repetidos.



	GGs	A*	IDA*	C_IDA*
Heurística	0	0	0	0
Deadlocks	FALSE	FALSE	FALSE	FALSE
Resultado	Éxito	Éxito	Éxito	Éxito
Profundidad/ Costo	198	826	21	21
Nodos Expandidos	25164	18082	10789598	10971
Nodos Frontera	278	307	7035353	10582
Tiempo [s]	164.838	66.207	735.592	9.231



	GGs	A*	IDA*	C_IDA*
Heurística	1	1	1	1
Deadlocks	FALSE	FALSE	FALSE	FALSE
Resultado	Éxito	Éxito	Éxito	Éxito
Profundidad/ Costo	24	24	25	25
Nodos Expandidos	603	4016	627123	892
Nodos Frontera	267	1567	406454	839
Tiempo [s]	0.283	1.439	42.771	2.229



	GGs	A*	C_IDA*
Heurística	2	2	2
Deadlocks	FALSE	FALSE	FALSE
Resultado	Éxito	Éxito	Éxito
Profundidad/Costo	50	26	27
Nodos Expandidos	15671	45255	1894052
Nodos Frontera	4924	14654	1801092
Tiempo [s]	18.829	165.969	117.614



	GGs	A*	C_IDA*
Heurística	3	3	3
Deadlocks	FALSE	FALSE	FALSE
Resultado	Éxito	Éxito	Éxito
Profundidad/Costo	50	32	63
Nodos Expandidos	2410	2260	3485
Nodos Frontera	916	847	4220
Tiempo [s]	1.377	1.294	4.551



	GGs
Heurística	0
Deadlocks	TRUE
Resultado	Éxito
Profundidad/Costo	56
Nodos Expandidos	4248
Nodos Frontera	1342
Tiempo [s]	1.672



Vemos que con la heurística 0 los deadlock aportan una gran optimización al programa bajando el tiempo de procesamiento de 164 s a 1s. Pero esto no ocurre para todos los algoritmos ni para todas la heurísticas. En algunos podrían ser contraproducentes ya que la cantidad de nodos expandidos que tiene deadlock es muy baja.




	GGs	A*	IDA*	C_IDA*
Heurística	1	1	1	1
Deadlocks	TRUE	TRUE	TRUE	TRUE
Resultado	Éxito	Éxito	Éxito	Éxito
Profundidad/C osto	24	24	25	25
Nodos Expandidos	428	4871	627123	895
Nodos Frontera	193	1937	406454	843
Tiempo [s]	0.224	1.646	43.097	2.296



	GGs	A*	C_IDA*
Heurística	2	2	2
Deadlocks	TRUE	TRUE	TRUE
Resultado	Éxito	Éxito	Éxito
Profundidad/C osto	30	26	27
Nodos Expandidos	13749	39028	1803476
Nodos Frontera	4219	12282	1718790
Tiempo [s]	13.750	110.848	109.092



Implementación



Se buscó implementar los métodos de las heurísticas de manera mantenible y eficiente. Para calcular la heurística sobre un tablero no hace falta recorrerlo completamente porque se guardan y se mantienen los estados de los objetivos, las cajas y el sokoban por lo tanto no hay que recorrer la matriz.

Para facilitar el mantenimiento, se creó la clase `Cell.java` que es una abstracción de la lógica de cómo se representa cada celda en el tablero. Resuelve muchos problemas en los métodos `move`.

No generamos una estructura de árbol, todo está basado en la clase `Node`.

Para la heurística no admisible, no está permitido el deadlock, por lo tanto se ignorará.



Algoritmos:

- En Global Greedy y A* se utilizan colas de prioridad (Priority Queue) donde estan ordenadas segun la funcion de cada una.
- En DFS, BFS, Greedy y A* se tiene una estructura que contiene tableros repetidos para evitar que se vuelvan a visitar
- En custom ida star tenemos una lista llamada “path” donde realizamos backtracking sobre la rama actual. Por ejemplo, no se va a repetir el caso dud (abajo, arriba, abajo sin contacto con nada).
- Recursivos: DFS; IDDFS; IDA; Custom IDA;
- Iterativos (por uso de cola o cola de prioridad): Greedy; A*; BFS




Heurísticas

- 1) Retorna la menor distancia manhattan entre una caja y un objetivo (la caja sin estar en objetivo).
- 2) Retorna la suma de las distancia manhattan desde el sokoban hasta cada caja.
- 3) Agrupa las cajas con objetivos según cercanía y calcula la suma de distancias manhattan desde el sokoban a la caja y de esa caja con el objetivo asignado.
- 4) Retorna el valor de: $\text{heurística 1} * \text{heurística 2} * \text{heurística 3} * \text{cantidad de cajas}$, generando un promedio ponderado entre todas las heurísticas, haciendo que esta sea no admisible

Nota: cada uno toma el valor índice -1, tomando 0,1,2, y 3 como los valores de la heurística



Conclusiones



En todos los métodos de búsqueda informados el hecho de hacer una buena heurística aumenta la eficiencia del código.

La versión CUSTOM de IDA* es más rápida porque se ahorra los nodos que ya recorrió.