Trabajo Práctico 1:

Métodos de Búsqueda No Informados e Informados

Materia: Sistemas de Inteligencia Artificial

Grupo: 4

Integrantes: Dallas, Tomás
Dorado, Tomás

Fecha de entrega: 20/08/2020

Introducción

Este informe consiste en el desarrollo de una implementación del juego Sokoban, para así poder aplicar diferentes algoritmos para encontrar la solución de algún mapa del juego. Para ello se implementaron tres algoritmos de búsqueda no informada y 3 de búsqueda informada, para los cuales se implementaron 3 heurísticas diferentes. Luego se compararon los resultados de cada uno y se obtuvieron conclusiones.

Descripción del juego

El Sokoban es un juego simple, en donde se tiene al jugador, cuyos movimientos son arriba, abajo, izquierda y derecha. El jugador puede moverse por el tablero, el cual tiene tres tipos de tiles: tile vacío, tile con roca (o tile no caminable) o tile con objetivo. En el tablero se encuentran cajas las cuales el jugador debe empujar en sus movimientos hacia los tiles con objetivo. Una vez que se empujaron todas las cajas a sus respectivos tiles, el juego termina. El jugador solo puede empujar una caja a la vez en la dirección en la que se está moviendo.

Implementación

Se eligió dividir la implementación en dos partes:

1. Modelado del juego
2. Modelado de estrategias de resolución

Para el **modelado del juego**, se eligió utilizar un *Board* que contenía todos los datos necesarios para simular el juego:

- Un **set de coordenadas** para las posiciones finales
- Un **set de coordenadas** para las “piedras” o posiciones por las que no se podía *caminar*
- Un **set de coordenadas** para aquellas posiciones por las que se podía *caminar*
- Un **estado** inicial, representando la posición inicial de las cajas y del personaje.

El **estado** se utiliza en gran parte del código para poder realizar las simulaciones de movimientos necesarias y guardar el estado actual del juego como parte del árbol que construimos luego de realizar (o no) un movimiento.

Para las **estrategias de resolución** del *Sokoban* optamos por un modelo homogéneo para todas las estrategias. Es decir, cada estrategia, sea informada o no, respeta una estructura y un comportamiento. Esto nos dió una flexibilidad extra a la hora de agregar estrategias durante el desarrollo, y también nos ayudó mucho a la refactorización y robustez durante las iteraciones realizadas:

- Las estrategias están divididas en:

    - Informadas: BFS, DFS y IDDFS.
    - No informadas: Dentro de este grupo, se puede diferenciar entre aquellas que utilizan una estrategia de resolución con una cola de prioridades (A* y Greedy), y las que no (IDA*)
- Las heurísticas que planteamos fueron 5 (cinco):
    - ***Euclidean:*** Utilizando la distancia euclidiana, calculamos un **valor base**, buscando la distancia entre el *pusher* y la caja más cercana. Luego, para cada caja, le sumamos al **valor base** entre la caja y la posición final más cercana. ESCRIBIR ECUACION
    - ***Manhattan:*** Análogo a *Euclidean*, en este caso utilizamos la distancia de Manhattan.
    - ***GlobalMinManhattan:*** Análogo a Euclidean y Manhattan, pero en vez de retornar para el valor base la distancia entre el pusher y la caja más cercana, vamos guardando el estado de ese valor, y si encontramos un valor mínimo, retornamos el 95% de ese valor. Y utilizamos la misma lógica para las distancias calculadas entre las cajas y las posiciones finales más cercanas. Con esto nos aseguramos que los movimientos sean los más “cortos” posibles.
    - ***GlobalMinEuclidean:*** Análogo a *GlobalMinManhattan* pero utilizando la distancia *Euclideana*.
    - ***ManhattanCheckingHalf:*** Análogo a *Manhattan*, pero utilizando como premisa que creemos no es necesario calcular la distancia entre el *pusher* y cada caja, ni tampoco la distancia entre cada caja y una posición final, sino en solo la mitad (utilizando el piso si es un valor decimal) de ellas. Es decir, si tenemos 5 cajas, y por lo tanto 5 posiciones finales, buscamos el valor de sólo 2 cajas. Con esto buscamos fortalecer y aprovechar el optimismo de la heurística y ahorrar tiempo de ejecución, asegurándonos un buen trade-off entre buscar la menor cantidad de pasos y el menor tiempo de ejecución.

Las dos heurísticas admisibles pedidas son GlobalMinManhattan y GlobalMinEuclidean. La otra heurística ManhattanCheckingHalf no es admisible.

Para la implementación de todos los métodos se utilizó una poda de ramas de dos maneras:

- Por estados **visitados**: Cuando se llega a un estado ya visitado se corta directamente ahí la rama.
- Por estado en **deadlock**: Si alguna caja no quedó en una posición final y no hay manera de moverla, todo el estado ya no va a tener manera posible de ganar el juego, entonces se corta esa rama para no desperdiciar tiempo en ella. Esto se realizó de una manera bastante simple, en donde verificamos solo 8 casos, pero se podría ampliar todavía más y mejoraría la velocidad de todos los métodos.

Durante el desarrollo hicimos mucho énfasis en tener una buena cobertura de tests para evitar conflictos, ya que creímos que el tiempo para realizar el trabajo práctico no era bastante holgado y teníamos poco margen de error, por ello intentamos mantener una cobertura arriba del 75% en todos los aspectos (lineas, clases, métodos). Esto nos dio una robustez muy grande a la hora de realizar refactors para optimizar y abstraer el código

dentro de lo posible. Creemos que logramos con esa abstracción pocas líneas de código dentro del proyecto, alrededor de las 1600.

Problemas encontrados

Al principio del desarrollo, elegimos implementar el juego primero, y luego los algoritmos. Esto nos llevó a pensar el juego como un juego, con el modelado ideal para eso, pero luego a la hora de simular movimientos, no era algo que estábamos contemplado en un principio, lo que nos llevó a refactorizar todo ese modelado.

Tuvimos también problemas con el manejo de colecciones dentro del árbol que armamos para las diferentes estrategias, lo que nos llevó a inconsistencias en los primeros resultados. Cuando hacíamos alguna expansión de nodo y modificamos algún objeto del estado actual, también se modificaban los estados predecesores del mismo nodo.

Respecto de los tiempos de ejecución, algunas pruebas realizadas en mapas de complejidad muy alta demandaban mucho tiempo para luego terminar con un *OutOfMemoryError*.

Para las heurísticas nos topamos con muchas trabas respecto con nuestra imaginación, nos fue difícil pensar heurísticas que no fueran triviales, y a la vez que sean admisibles, por eso intentamos a partir de la distancia Euclidiana y de Manhattan plantear una variedad de ellas y ver cómo las búsquedas reaccionaban a cada una. Al principio nos costó confirmar que alguna de ellas sea admisible, si bien en un principio todas parecían serlo, cuando las usábamos había alguna que performaba mejor que otra (ya sea por tiempo de ejecución o por cantidad de movimientos), y eso nos hacía dudar.

El último algoritmo que implementamos fue IDA*, en cuanto a complejidad fue el que más nos costó realizar. A la hora de correrlo en alguno de los mapas que tenemos predefinidos tardaba mucho más tiempo de lo esperado, y la solución, en cuanto a cantidad de movimientos, no era la mejor. Por eso sabíamos que teníamos un error en el algoritmo, debido a que IDA* con una heurística admisible debería ser el más óptimo de todos los algoritmos de búsqueda.

Datos

Se tomaron los datos de 5 mapas, los cuales están detallados en cada tabla, H1 es ***GlobalMinManhattan***, H2 es ***GlobalMinEuclidean*** y H3 es ***ManhattahCheckingHalf***.

MAPA 1 (maps/m4.txt)

Algoritmo	Tiempo(ms)	Costo	Longitud	Nodos expandidos	Nodos frontera	Solución encontrada
BFS	69	86	86	1577	2070	SUCCESS
DFS	53	106	106	2300	1158	SUCCESS
IDDFS	1373	146	146	334242	199183	SUCCESS
Greedy (H1)	63	120	120	1224	1621	SUCCESS
Greedy (H2)	69	120	120	1224	1621	SUCCESS
Greedy (H3)	69	96	96	919	1186	SUCCESS
A* (H1)	85	86	86	1570	2065	SUCCESS
A* (H2)	84	86	86	1570	2065	SUCCESS
A* (H3)	85	90	90	1419	1871	SUCCESS
IDA* (H1)	170	86	86	3444	5173	SUCCESS
IDA* (H2)	169	86	86	3444	5173	SUCCESS
IDA* (H3)	95	98	98	1929	2950	SUCCESS

MAPA 2 (maps/m5.txt)						
Algoritmo	Tiempo(ms)	Costo	Longitud	Nodos expandidos	Nodos frontera	Solución encontrada
BFS	216	23	23	5873	7364	SUCCESS
DFS	254	153	153	19152	9761	SUCCESS
IDDFS	586	31	31	99056	47798	SUCCESS
Greedy (H1)	470	499	499	12221	16926	SUCCESS
Greedy (H2)	454	499	499	12221	16926	SUCCESS
Greedy (H3)	69	25	25	886	1060	SUCCESS
A* (H1)	401	23	23	6541	8284	SUCCESS
A* (H2)	339	23	23	1570	2065	SUCCESS
A* (H3)	169	37	37	3936	4805	SUCCESS
IDA* (H1)	640	23	23	17226	26118	SUCCESS
IDA* (H2)	734	23	23	17226	26118	SUCCESS
IDA* (H3)	92	35	35	1926	2789	SUCCESS

MAPA 3 (maps/m6.txt)

Algoritmo	Tiempo(ms)	Costo	Longitud	Nodos expandidos	Nodos frontera	Solución encontrada
BFS	69	69	69	1023	1132	SUCCESS
DFS	38	139	139	1128	553	SUCCESS
IDDFS	370	71	71	49483	26014	SUCCESS
Greedy (H1)	47	119	119	692	763	SUCCESS
Greedy (H2)	53	119	119	692	763	SUCCESS
Greedy (H3)	53	95	95	491	531	SUCCESS
A* (H1)	69	69	69	1032	1144	SUCCESS
A* (H2)	62	69	69	1032	1144	SUCCESS
A* (H3)	62	69	69	981	1078	SUCCESS
IDA* (H1)	85	69	69	1571	1881	SUCCESS
IDA* (H2)	91	69	69	1571	1881	SUCCESS
IDA* (H3)	54	83	83	833	978	SUCCESS

MAPA 4 (maps/m7.txt)						
Algoritmo	Tiempo(ms)	Costo	Longitud	Nodos expandidos	Nodos frontera	Solución encontrada
BFS	6957	72	72	233621	325222	SUCCESS
DFS	254	262	262	29458	14586	SUCCESS
IDDFS	55538	194	194	14059724	7171460	SUCCESS
Greedy (H1)	101	228	228	2455	3303	SUCCESS
Greedy (H2)	100	228	228	2455	3303	SUCCESS
Greedy (H3)	987	212	212	31993	41756	SUCCESS
A* (H1)	11816	74	74	229990	320087	SUCCESS
A* (H2)	12009	74	74	229990	320087	SUCCESS
A* (H3)	1429	122	122	67292	88822	SUCCESS
IDA* (H1)	297103	74	74	6525631	11504379	SUCCESS
IDA* (H2)	282670	74	74	6525631	11504379	SUCCESS
IDA* (H3)	5776	150	150	240956	392990	SUCCESS

MAPA 5 (maps/m8.txt)						
Algoritmo	Tiempo(ms)	Costo	Longitud	Nodos expandidos	Nodos frontera	Solución encontrada
BFS	13974	26	26	270347	425489	SUCCESS
DFS	84	408	408	6902	3369	SUCCESS
IDDFS	2791	36	36	735613	288041	SUCCESS
Greedy (H1)	103	272	272	2296	3661	SUCCESS
Greedy (H2)	100	272	272	2296	3661	SUCCESS
Greedy (H3)	323	82	82	8718	12120	SUCCESS
A* (H1)	27224	30	30	279101	443737	SUCCESS
A* (H2)	27322	30	30	279101	443737	SUCCESS
A* (H3)	2507	52	52	80914	113318	SUCCESS
IDA* (H1)	194222	26	26	2503531	5046784	SUCCESS
IDA* (H2)	230177	26	26	2503531	5046784	SUCCESS
IDA* (H3)	1038	70	70	45066	70082	SUCCESS

Análisis de Datos

Para tomar los datos de las tablas, todos se tomaron ejecutando el programa en la misma PC, para así evitar diferencias por hardware. El tiempo para resolver el mapa se considera desde que el algoritmo comienza a buscar una solución hasta que encuentra una o no.

Los datos se tomaron de los mapas que se encuentran en la carpeta */maps* en donde solo se tomaron del archivo m4.txt al m8.txt, ya que los primeros son mapas muy simples solo de testeo. En los casos en donde terminó en FAILURE fue debido a que el programa lanzó una excepción de *out of memory*.

En todos los casos se pudo observar que BFS tuvo un menor tiempo para resolver que los demás métodos, dando el camino de menor costo. Esto es porque los mapas son pequeños pero en casos donde los mapas sean más grandes o con más cajas podría cambiar ampliamente. Lamentablemente todos los mapas que intentamos correr, ya sean más grandes o con más cajas nunca pudieron terminar porque el programa terminó lanzando nuevamente una excepción de *out of memory*.

Un buen punto a destacar, es la velocidad de la heurística *ManhattahCheckingHalf,* que si bien no lanzó el camino más corto usando el método A*, el tiempo para resolver fue bastante corto a comparación de las heurísticas con los mejores caminos y utilizando el

método Greedy, normalmente dió un camino mejor que las otras. Tambíen se puede ver que la cantidad de nodos expandidos es considerablemente menor que las otras.

En cuanto a comparaciones de los métodos no informados, se puede ver claramente que BFS es el mejor en cuanto al camino, y si bien DFS es el que menos tiempo toma el camino siempre es de los peores en costo. Por otro lado IDDFS no genera mucha ganancia, pues la cantidad de nodos expandidos es mayor a todos los demás métodos y en tiempo también es el que más consume, y el camino fue bueno en algunos mapas y muy malo en otros, teniendo en cuenta el tiempo que tuvo.

Conclusiones

Lo primero a destacar creemos que es que debido al alto coverage de tests, y a la abstracción del código, sería muy fácil en un futuro agregar más y mejores métricas.

Dado que el problema de buscar una solución para el Sokoban es un problema NP-hard, donde el espacio de soluciones crece exponencialmente, cuando se intenta buscar una solución para mapas relativamente grandes, con una gran cantidad de cajas y posibles movimientos, cada nivel que se expande en el árbol para cualquier estrategia es cada vez más grande (exponencialmente), hasta llegado el punto en el que se llegan a errores de memoria.

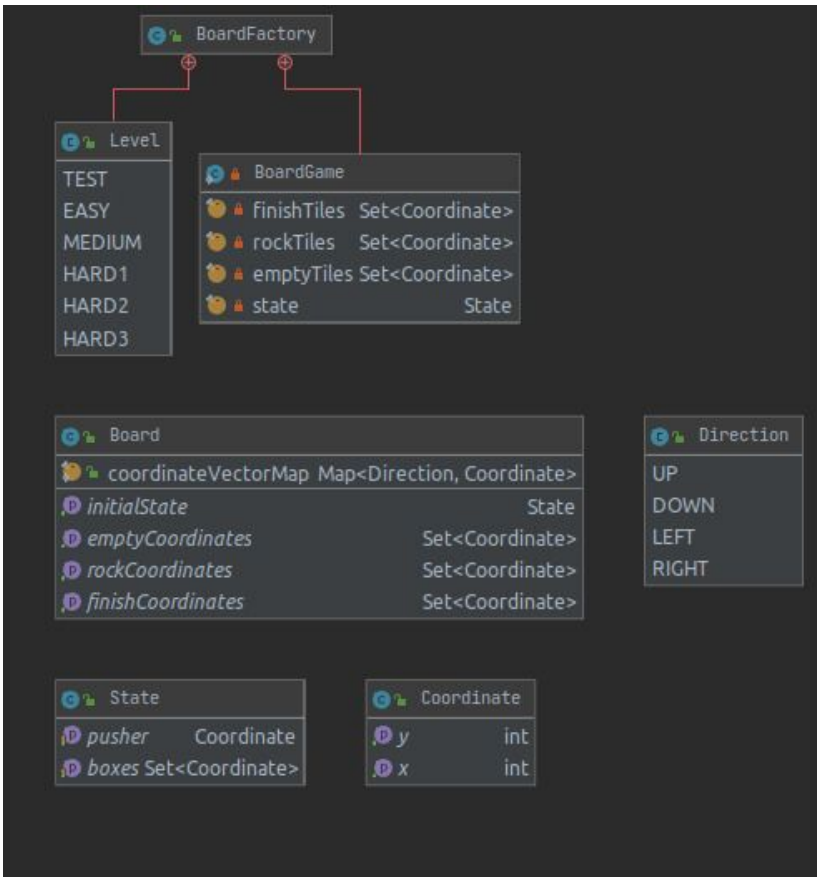
Se sabe que en los problemas que son NP-hard, la fuerza bruta sirve para casos no tan complejos, sería ideal encontrar soluciones para minimizar estos errores de falta de memoria, de ser necesario, cómo generar procesamiento paralelo, o podas más grandes sobre las soluciones, o bajar a disco niveles anteriores del árbol que se van a visitar dentro de mucho tiempo.

Otra solución, luego de investigar un poco al respecto, sería la utilización de redes neuronales cómo heurísticas, cómo por ejemplo hace alusión este paper (https://arxiv.org/pdf/1807.00049.pdf). Con ese **aprendizaje supervisado** podríamos ganar tanto tiempo de ejecución durante la búsqueda de la solución, ya que estaríamos utilizando una red neuronal previamente entrenada, cómo también encontraríamos los mejores movimientos a realizar en cada estado del juego.

Luego de probar varias heurísticas que no implementamos en la entrega final, nos dimos cuenta del valor de una buena heurística. Tanto desde el punto de vista de buscar una que sea admisible, cómo también viendo que a veces ganando tiempo de ejecución, encontramos una solución quizás un poco más larga en cuanto a movimientos realizados, pero visitamos menos nodos, y hacemos menos cálculo sobre cada estado. Por lo tanto, a fin de cuentas tenemos que ver cual es el fin que perseguimos; si nos preocupa más buscar la menor cantidad de movimientos, o si a costas de resignar algunos movimientos de más, buscamos un mejor tiempo de ejecución. Ajustar ese trade-off depende en su gran mayoría de la heurística a utilizar.

Anexo

UML de Sokoban



Test coverage

Element	Class, %	Method, %	Line, %
Class50			
com			
game	100% (13/13)	84% (42/50)	84% (169/200)
images			
java			
javax			
jdk			
lombok			
META-INF			
netscape			
org			
strategies	100% (17/17)	74% (56/75)	85% (304/356)
sun			
toolbarButtonGraphics			
Main	0% (0/1)	0% (0/2)	0% (0/88)