Instituto Tecnológico de Buenos Aires

Sistemas de Inteligencia Artificial

Ingeniería en Informática

Métodos de búsqueda

Segundo Trabajo Práctico Especial

Titular: Parpaglione, Cristina

Semestre: 2018A

Grupo: 4

Repositorio: https://bitbucket.org/itba/sia-2018-04

Fecha: 2 de mayo de 2018

Entrega: 2 de mayo de 2018 a las 15:00hs

Autores: Buscaglia, Matías Alejandro (#53551)

Costesich, Pablo Alejandro (#50109)

Delgado, Francisco (#57101)

Gonzalez, Lautaro Nicolás (#54315)

1. Introducción

Durante este trabajo se implementa un motor de inferencia con el fin de resolver problemas de búsqueda tanto informados como no informados. En particular, se trata de resolver el juego Rolling Cubes. La implementación de este trabajo se divide en motor de inferencia y lógica del juego. Esto nos permite que el motor de inferencia pueda resolver cualquier problemática de búsqueda.

2. Motor

2.1. Estructura

El motor consta de una clase Solver<E>, que es responsable de verificar que los nodos obtenidos sean solución del problema. El mismo hace uso de las distintas estrategias implementadas de búsqueda informada y desinformada que serán descriptas a continuación.

Para conocer el estado del motor se utilizan spies que están suscriptos a determinados eventos de interés.

2.2. Búsqueda desinformada

2.2.1. Breadth First Search

El BFSStrategy<E>implementa la interfaz SearchSolver<E>en sus métodos offer, getNextNode y reset. El backend de los mismos es una cola basada en un ArrayDeque<E>, y utiliza los métodos push para offer, poll para getNextNode y clear para reset.

2.2.2. Depth First Search

El DFSStrategy<E>implementa la interfaz SearchSolver<E>en sus métodos offer, getNextNode y reset. El backend de los mismos es una cola basada en un ArrayDeque<E>, y utiliza los métodos add para offer, poll para getNextNode y clear para reset.

2.2.3. Profundización Iterativa

El IDDFSStrategy<E>extiende DFSStrategy<E>y reimplementa el método isIterative para que devuelva true.

2.3. Búsqueda informada

2.3.1. Greedy Search

El GreedyStrategy<E>reimplementa explodeNode creando child nodes con costo cero, a diferencia de A*.

2.3.2. A*

La búsqueda con A* se realiza con AStarStrategy<E>, al igual que Greedy Search, reimplementa explodeNode, pero crea child nodes tanto con el costo normal como el de la heurística.

2.4. Costo

En un principio definimos el costo de hacer cada movimiento como 1. Esto resultó en que la búsqueda con A* sin heurística terminara alcanzando una profundidad de 35 niveles encontrando la solución óptima (35 pasos) en 202.4 segundos.

Finalmente el costo se definió de manera tal que los movimientos que alejaran a un bloque del blanco tuvieran un costo mayor que los que hacen el movimiento contrario. Con esta nueva configuración de costos, la misma búsqueda alcanza una profundidad de 39 niveles y encuentra la solución óptima en 31.12 segundos.

2.5. Heurísticas

2.5.1. Máxima cantidad de blancos

En esta heurística se busca favorecer a los tableros con mayor cantidad de bloques blancos. Para calcular el valor que toma cada tablero se recorre el mismo y se acumula un puntaje por cada bloque. Este puntaje se define como 0 si el bloque es blanco, 2 si es negro y 1 si es un color intermedio. Es interesante notar que el puntaje para cada bloque es la cantidad mínima de movimientos que necesita para convertirse en un bloque blanco. De esta manera podemos asegurar que el valor final es menor o igual a la cantidad total de movimientos necesarios para resolver el tablero, por lo que esta heurística es admisible.

2.5.2. Posición del espacio vacío

Esta heurística se basa en una propiedad del juego que determina que la mínima cantidad de movimientos para ganar dejando el espacio vacío en el medio es 38, mientras que dejandolo en los bordes o esquinas es 35. Lo que se hace, entonces, es revisar la posición del espacio en blanco y favorecer a los que no están en el centro del tablero. Es una heurística admisible porque los posibles valores de un tablero son 1 y 0, por lo que la cantidad de movimientos necesarios para resolver el tablero es mayor o igual.

3. Juego

3.1. Descripción

El juego consiste en una grilla de nueve espacios con ocho cubos pintados de blanco y negro, que deben ser rotados para que todos muestren la cara blanca hacia arriba. En cada rotación los cubos se van pintando por la mitad. En caso de que ya estuvieran pintados por la mitad, solo se seguirán pintando si se los rota en la misma direccion. En estado inicial del juego todos los cubos están pintados de negro y el espacio vacío se encuentra en el medio del mismo.

3.2. Implementación

La implementación del juego consiste principalmente de las clases Board y Cube. La clase Board, que representa el estado del juego, cuenta con una matriz de 3x3 donde se ubican los 8 Cubes. Además guarda la posición de la celda vacía y de la cantidad de bloques blancos, para facilitar ciertos cálculos. Cube, por su lado, cuenta con un FaceColor que se encarga de calcular el resultado de rotar el bloque en cierta dirección.

Para obtener las posibles reglas válidas dado un Board, la clase RollingCubeGame tiene guardadas todas las direcciones válidas en base a la posición del espacio vacío. Es decir, si el espacio vacío se encontrara en la esquina superior izquierda, solo se buscarán rotar los cubos que estén debajo y a la derecha.

4. Conclusiones

4.1. Análisis de resultados

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

- Definir un buen costo es esencial, ya que permite llegar a una solución mucho más rápidamente con poco esfuerzo.
- Iterative Deepening logra soluciones mucho más rápidas que BFS dado que consume mucha menos memoria.
- Utilizar una cache de estados para evitar reencolar nodos no sólo evita potenciales loops sino que acelera la ejecución de los algoritmos de forma sustancial.
- Dentro de las búsquedas desinformadas, DFS es particularmente rápido. Esto se debe a que el ancho del árbol es amplio y dada la naturaleza del juego se puede encontrar una solución en nodos profundos.
- Dada la gran cantidad de tableros y posiciones posibles, Greedy con una buena heurística fue el más eficiente en cuanto a recursos y tiempo.
- La heurística que busca maximizar los bloques blancos fue la mejor. Esto se debe a que trata directamente sobre el objetivo del juego, mientras que la otra heurística lo hace más indirectamente.
- Dado que el objetivo del juego es único, y que el mayor limitante o aspecto es ese solo, pudimos encontrar una única heurística de valor. El resto de las heurísticas son formas parciales de esta o lo tratan indirectamente.

4.2. Mejoras posibles

Una característica interesante del juego es la simetría del tablero. Dada la misma, un tablero A es en esencia igual a un tablero A rotado una cantidad arbitraria de veces o espejado. Si bien los pasos para llegar a ambos son distintos, el estado final del juego es el mismo y podemos asegurar que ya ha sido explorado y que el camino entre ambos es equivalente. Esta mejora se implementa como un chequeo adicional en el equals y una modificación en el hashCode para que dos tableros retornen el mismo hashCode.

Se puede agregar como funcionalidad un corte por tiempo de ejecución usando un thread distinto. El mismo puede matar al thread una vez alcanzado el tiempo de ejecución, o con IDDFS puede

Apéndice

Algoritmo	Pasos	Depth	Promedio [s]	Costo	Expandidos	Encolados
DFS	384385	384385	4,711667	$3{,}841840 \times 10^{05}$	386824	686053
BFS	35	35	196,59	$2{,}5 \times 10^{01}$	10209834	11365242
IDDFS	37	37	77,835	$2{,}7 \times 10^{01}$	1307660	1307670
A* MaxWhite	35	42	13,989	$2{,}5 \times 10^{01}$	1023855	1596441
A* emptySlot	35	41	37,076	$2{,}5 \times 10^{01}$	2627615	3723804
Greedy MaxWhite	272	305	0,206	N/A	386824	9495
Greedy emptySlot	1005	3248	5,229	N/A	386824	760736

Cuadro 1: Tabla de Comparaciones entre Algoritmos