Sistemas de inteligencia artificial

(1er cuatrimestre 2018)

Trabajo Práctico 2

Algoritmos de Búsqueda

Julián Benítez 56283

Nicolás Marcantonio 56288

Eliseo Parodi Almaraz 56399

Índice

0hh1

El juego consiste en completar un tablero de NxN, siendo N un número par, con fichas de 2 colores, rojo y azul. Este tablero comienza con cierta cantidad de fichas de colores y el resto de los casilleros se encuentran vacíos. Para completar este tablero se deben seguir las reglas del juego que se describen a continuación.

Reglas

- No pueden haber 3 o más casilleros contiguos con el mismo color en una fila o columna.
- Debe haber la misma cantidad de casilleros de color rojo y azul en cada fila y columna.
- No puede haber dos filas o columnas iguales.

Estas reglas cobran real importancia a la hora de realizar las heurísticas.

Implementación

Juego

Para representar el problema creamos 3 clases que implementan las interfaces provistas por la cátedra en *GPS.jar* para que los juegos puedan ser intercambiados entre los diferentes grupos.

1. **Ohh1Problem:** Esta clase representa el problema que vamos a resolver. Su constructor recibe el path a un archivo de texto que contiene la información necesaria para representar el estado inicial del problema. También es el encargado de generar las reglas que se aplican a los estados. También realiza una poda para que no llegue a estados sin sentido, es decir, que no cumplan las reglas.
2. **Ohh1State:** Esta clase contiene la información de los estados. La información que consideramos fue el tablero, representado por una matriz de int y la cantidad de espacios vacíos. Para representar el tablero decidimos representar los colores azul, con el valor -1 y el rojo con el valor 1. Para los casilleros vacíos utilizamos el valor 0.
3. **Ohh1Rule:** Esta clase representa la transición de un estado a otro. Contiene una función para aplicarse a sí misma en un estado para transicionar a otro y una función para poder comprobar si la regla es aplicable a dicho estado.

Motor

Para la implementación del motor utilizamos las siguientes clases e interfaces:

1. **Node**: Es la que contiene la información de los estados, el costo de transición de un estado a otro, una referencia a su nodo padre para poder reconstruir el camino de resolución o aplicar algún tipo de algoritmo, la profundidad en el árbol que se está recorriendo dentro del grafo de transiciones y además contiene el valor de la heurística para dicho nodo para no tener que volver a calcular el valor del mismo.
2. **SearchStrategy**: Es una interfaz que representa los diferentes tipos de algoritmos de búsqueda que se pueden correr. Contiene 4 métodos:
    a. *getNext*: devuelve el siguiente *Node* a analizar por el algoritmo y lo elimina de la frontera.
    b. *addToFrontier*: agrega un nodo a la frontera.
    c. *peekNext*: devuelve el siguiente *Node* a analizar sin eliminarlo de la frontera.

d. *getNumOfNodesInFrontier*: devuelve la cantidad de *Node* que hay en la frontera.

De esta interfaz heredan nuestras implementaciones de algoritmos de búsqueda, y así se puede aplicar el patrón de software *Strategy*. Los que implementamos fueron los siguientes:

- *Depth-first search* (DFS)
- *Breadth-first search* (BFS)
- *A** (AStarStrategy)
- *Greedy search*
- *Iterative Deepening Search*

Para el caso del algoritmo *Greedy Search* y el *A** se les agregó a la implementación *HashMap* de todos los nodos recorridos hasta el momento para evitar volver a recorrer y guardar estados ya analizados y así ahorrar memoria y tiempo.

3. **Motor**: Es la clase que se encarga de correr los algoritmos. Para poder correr recibe una *SearchStrategy* en su constructor. Con el método *run* ejecuta el algoritmo. Primero crea el nodo inicial que es el primer nodo que se va a expandir, luego expande el nodo utilizando todas las reglas posibles para ese estado, creando nuevos nodos con nuevos estados dentro que se agregaran a la frontera. Luego le pide al algoritmo cuál va a ser el próximo nodo a expandir, y así sucesivamente hasta llegar a la solución. Nuestra condición de completado es que el tablero esté completo.

Heurísticas

Para nuestro juego planteamos 2 heurísticas distintas, ambas admisibles, tomando en cuenta las reglas del juego y teniendo en cuenta que el valor de las heurísticas cuando se llega a la solución es 0.

- *Ohh1HeuristicMust*: Esta heurística se basa en las reglas de que no puede haber más de dos casilleros contiguos y en que debe haber la misma cantidad de casilleros azules y rojos en la misma columna o fila. Debido a estas reglas quedan lugares donde el jugador está obligado a colocar una ficha de cierto color, esto se da en los casos en los que en una fila o columna la mitad de los casilleros están ocupados por fichas de un color en particular, obligando al jugador a completar con fichas de un color; y cuando en una fila o columna se encuentran 2 fichas contiguas del mismo color, obligando a colocar en los extremos fichas de otro color, o cuando hay dos fichas del mismo color separadas por un casillero vacío. Para esta heurística entonces se toma en cuenta la siguiente fórmula:
  $h_1(n) = cantidadEspaciosVaciosNoObligados + 3 * cantidadEspaciosObligados$

- *Ohh1HeuristicMustHalf*: Esta heurística es similar a la anterior en el sentido de que aplica las mismas reglas, pero tomando en cuenta de que si se divide el tablero en 2 mitades, horizontalmente y/o verticalmente cada mitad tiene la misma cantidad de fichas rojas y azules cuando el tablero está terminado. Entonces, tomando en cuenta que tan completo esté el tablero, ya que no tiene sentido aplicar esta heurística cuando el tablero está con pocas piezas, el tablero con mayor similitud entre ambas mitades tiene el valor heurístico más bajo.

Costo

El costo de aplicar una regla es constante para cualquier estado, en nuestro caso decidimos arbitrariamente asignarle 5 al costo. Esto se debe al hecho de que agregar una ficha a un casillero no tiene ningún costo.

Resultados

A continuación se detallan una serie de pruebas realizadas a tableros de 6x6 obtenidos de la url provista por la cátedra, con un número pequeño de jugadas realizadas, con el fin de poder comparar las dos heurísticas en un tiempo razonable.

Aclaraciónes:

1- La idea era comparar también con los algoritmos de búsqueda desinformados, pero dado que se tomó un timeout de 30 minutos y ninguno terminó antes de dicho tiempo no se incluirán estas comparaciones en los resultados.
2- En cuanto a los algoritmos informados, Greedy Search resultó superior a A* para nuestro problema, y para mostrar eso se corrió el siguiente tablero de 4x4, con sus resultados debajo de la imagen:

Greedy Search:
Heurística 1 : 9ms 16 nodos expandidos
Heurística 2 : 11ms 18 nodos expandidos

A* :
Heurística 1 : 187ms 3390 nodos expandidos
Heurística 2 : 194ms 2022 nodos expandidos

Como se observa , el algoritmo A* tarda casi 21 veces más que Greedy Search, por lo que no se incluirá en las comparaciones de tableros 6x6.

Ahora sí, estos son los tableros probados de 6x6:

1)

Tablero base:

Tablero solución:

Heurística 1 : 550ms 10877 nodos expandidos

Heurística 2 : 21291ms 468907 nodos expandidos

2)

Tablero base:

Tablero solución:

Heurística 1 : 18269ms 351210 nodos expandidos

Heurística 2 : 46801ms 886203 nodos expandidos

3)
Tablero base: Tablero solución:

Heurística 1 : 11816ms 255691 nodos expandidos
Heurística 2 : 8744ms 251590 nodos expandidos

Conclusiones

Los algoritmos informados resultaron ser mucho mejor que los desinformados teniendo en cuenta nuestras heurísticas. Ahora, teniendo en cuenta el algoritmo Greedy y A*, el primero es mucho mejor que A* para nuestras heurísticas, en tiempo y en resultado, ya que cualquier solución que se encuentre que cumpla con las reglas del juego, ya es óptima. Por lo tanto era de esperar que el algoritmo Greedy haya sido el más efectivo de entre los dos.
En cuanto a la comparación de los resultados entre heurísticas, la Ohh1HeuristicMust resultó ser la que mejor resultado dio, a pesar de que la segunda pretendía ser una mejora de esta.