UNIVERSIDADE DA CORUÑA

Raúl Fernández del Blanco
Armando Martínez Noya
Grupo: 2.4

Práctica 1: Estrategias de búsqueda

Problema 1:

Para el primer ejercicio se nos solicitó hacer los siguientes cambios en el código:

- Crea una clase Nodo según lo descrito en teoría.
- Adaptar la clase "Estrategia4.java" para que implemente los nodos.
- Cambiar el método soluciona de la clase "EstrategiaBusqueda.java" para que devuelva un array de Nodos.
- Implementar un método "reconstruye_sol como el descrito en teoría.
- Implementar una clase EstrategiaBusquedaGrafo.

Solución:

Para este problema, creamos una clase "Nodo.java" la cual contiene tres atributos: "estado" de tipo Estado para indicar el estado de la aspiradora, "padre" de tipo Nodo para indicar cual es el nodo padre del nodo en cuestión y por último "accion" de tipo Accion para indicar la acción que se va a realizar.

En la clase "Estrategia4.java" modificamos las listas que almacenaban los estados explorados y a explorar para que estas guardasen objetos de tipo "Nodo" descritos anteriormente, además de esto, modificamos también la definición de nuestras clases para que devolviesen los tipos de datos que buscábamos, como es el caso de "EstrategiaBusqueda.java" que la modificamos para que devolviese un array de Nodos "Nodo[]".

Implementamos también en nuestro programa los métodos: "reconstruye_sol" que se encarga de devolver un array de nodos de la solución, reconstruyéndola a partir de los padres del nodo que aporta la solución, el método "Sucesores" que devuelve un array con los diferentes nodos a los que se han aplicado todas las acciones posibles, todos estos métodos fueron implementados tal y como se explican en teoría.

Por último, dado que "Estrategia4.java" falla cuando llega a un estado sin sucesores, creamos la clase "EstrategiaBusquedaGrafo.java" que se encarga de hacer una búsqueda en el grafo creado por los nodos y solucionando el problema anteriormente nombrado.

Raúl Fernández del Blanco
Armando Martínez Noya
Grupo: 2.4

Problema 2:

Queremos desarrollar un agente capaz de, a partir de un cuadrado parcialmente relleno, completar el cuadrado mágico de NxN (si es que es posible), respetando los valores suministrados originalmente.

Para este ejercicio se nos solicitó hacer lo siguiente:

- Formalizar el problema para poder resolverlo con estrategias de búsqueda e implementar la clase "ProblemaCuadradoMagico" que sea subclase de "ProblemaBusqueda" y define las subclases de Estado y Accion necesarias para representar el problema.
- Implementar las estrategias búsqueda en profundidad y en anchura.
- ¿Cuál de las dos estrategias es la más adecuada? Compruébalo mediante experimentos. ¿Cuál es la causa?
- Indica una heurística apropiada para el Cuadrado Mágico. ¿Es tu heurística admisible? ¿Y consistente? Justificar.
- Implementa el método de búsqueda A*.
- ¿Puede tu programa resolver el tercer ejemplo en un tiempo razonable? ¿Qué mejoras harías para conseguir resolverlo más rápidamente? Descríbelas en detalle e impleméntalas.

Solución:

A la hora de formalizar el problema y pensar una implementación para el mismo nos encontramos con lo siguiente a tener en cuenta:

- Los estados del problema están representados por cuadrados con valores distintos en sus casillas, respetando en todo caso los valores del cuadrado inicial.
- Los valores de cada una de las casillas del cuadrado no pueden repetirse y tienen que estar entre [1, n^2], en caso de estar vacía se representa con el número 0.
- La única accion posible que tenemos en el problema es insertar un valor en una determinada casilla, para insertar esta casilla ha de estar vacía y el valor cumplir la condición anterior.
- A la hora de recorrer el cuadrado e insertar un valor, se insertará en la casilla vacía más próxima del recorrido, este recorrido se hace de izquierda a derecha y de arriba para abajo.

Dicho esto, en nuestro programa podemos encontrar:

La clase “ProblemaCuadradoMagico.java” que extiende a “ProblemaBusqueda”, en esta clase es donde formalizamos el problema per se y donde definimos las clases AcciónCuadrado que extiende a la clase Acción y EstadoCuadrado que define a la clase Estado.

En esta clase también podremos encontrar el constructor propio de ProblemaCuadradoMágico y una serie de métodos auxiliares que se encargan de comprobar el total de la suma de las columnas, filas y diagonales, pudiendo comprobar así en caso de tener un cuadrado mágico completo si este es meta o no.

Creamos tres clases llamadas “EstrategiaBusquedaAnchura.java”, “EstrategiaBusquedaProfundidad.java” y “EstrategiaBusquedaA.java”, las cuales extienden la interfaz “EstrategiaBusqueda”. En ellas implementamos las búsquedas en anchura, profundidad y A* respectivamente vistas en teoría.

Por último, la clase “HeurísticaImplementada.java” , que extiende al interfaz “Heurística.java”, es la clase que contiene la implementación de nuestra heurística (explicada con detalle en la próxima página) y que se encarga de asignarle un valor numérico a un estado.

Durante la implementación de este ejercicio, nos hemos encontrado con distintos problemas con los que hemos tenido que lidiar:

- En un principio, la intención del método “acciones” era devolver una lista con todas las diferentes acciones posibles del estado inicial. Debido a esto, el programa realizaba un bucle casi infinito. Este problema nos llevó a decidir que este método devolviese todas las diferentes acciones posibles del estado pasado a la función, pero volvimos a tener el mismo problema anteriormente nombrado, por lo que finalmente decidimos que el método “acciones” tendría que devolver todas las acciones posibles de la primera casilla vacía.

¿Cuál de las dos estrategias es la más adecuada? Compruébalo mediante experimentos. ¿Cuál es la causa?

Para dar respuesta a esta pregunta ejecutamos nuestro programa para dar solución al problema que se nos expone en el PDF, siendo el estado inicial: [[4, 9, 2], [3, 5, 0], [0, 1, 0]], al momento de ejecutar la solución es evidente la respuesta a la pregunta, esto se debe a que la estrategia de Búsqueda en Amplitud genera y recorre de media casi el doble de nodos que genera la Búsqueda en Profundidad siendo esta ultima la más adecuada y óptima para trabajar este problema en sí.

La razón de que esto suceda es debido a que en nuestra implementación la matriz solo se llena del todo al final del problema, donde ya se puede evaluar si dicho estado es o no meta, por lo que gracias a esto la búsqueda en profundidad va a alcanzar este nivel para evaluar la meta mucho antes que el método por amplitud.

Raúl Fernández del Blanco
Armando Martínez Noya
Grupo: 2.4

Heurística

A la hora de aplicar la heurística al problema de nuestro cuadrado mágico decidimos representar los valores de la siguiente manera: el número de casillas en blanco que faltan para llegar a la meta son representadas por h(n), en caso de que la matriz actual ya no pueda ser meta, el valor heurístico será un número muy grande para que siempre se descarte esa opción.

La implementación funciona de manera que, siempre que el estado de la matriz sea una posible solución o vaya camino a serlo (comprobando la suma de las filas, columnas y diagonales que sean S=(N(N^2+1))/2), en ese caso nuestra heurística devuelve el número de casillas que están vacías (con valor a 0).

En caso contrario (que la matriz actual no pueda ser solución, por la comprobación de S) nuestra heurística devolverá un número suficientemente alto como para indicar que dicho estado no podrá ser solución.

Consideramos que la heurística implementada es bastante eficiente, ya que con ella podemos representar de manera bastante acertada la situación actual de la resolución del problema. Además, consideramos que es consistente, ya que cumple la condición de que para cada nodo el coste para conseguir la meta es siempre menor que el coste de alcanzar la meta desde su sucesor.