Intelligent Systems Practical 1
Implementing Basic Search Strategies

Introduction

This practical investigates the implementation of some basic search strategies in a generic search library. You should ensure that the code of your solution is well documented. The deadline for submitting solutions to this practical is week 4.

A Description of the Code Archive

Before starting your work on this practical, download the code archive from the course Web site and import its contents into an Integrated Development Environment. (Use of an IDE is not mandatory; however, you are likely to find working on this practical significant easier if you use an IDE, such as Eclipse[1]).

The code in the archive that is relevant to the first two practicals is organized in three packages. **You should familiarize yourself thoroughly with the code in these packages, and make sure you understand how it works, before you start working on these practicals.** This will save you a lot of time later. A brief description of the classes in the three relevant packages is given here:

The *search* Package

The *search* package contains

- The *BreadthFirstTreeSearch* class implements the breadth first tree search algorithm. Its *findSolution()* method is given a root node, and it returns a node that contains a solution, or *null* if no solution can be found.
- The *Node* class represents a search node. It stores a reference to the parent node; a reference to the current state; and a reference to the action that, when applied to the state stored in the parent node, produces the state stored in this node. For the root node, parent and action are set to *null*.
- The *Printing* class is an abstract class that sets up the framework for solution printing. It contains the *printSolution()* method that, given a solution node, prints the path to the solution by calling two abstract *print()* methods to print each *Action* and *State* in this path.
- The *Action*, *GoalTest* and *State* interfaces specify how the *BreadthFirstTreeSearch* class interacts with the particular problem to be solved.

The *npuzzle* Package

The *npuzzle* package describes the n-puzzle problem and provides suitable implementations of the *Action*, *GoalTest* and *State* interfaces.

[1] In order to ensure that the Java Virtual Machine has sufficient heap space available, it may be helpful to specify the *-Xmx700M* option on the command line when running this code; in Eclipse, this option can be placed into the "VM Arguments" field of the "Arguments" tab of the "Run Configurations" menu option.

- The *Tiles* class implements the *State* interface and represents a configuration of tiles. The *width* member specifies the width of the puzzle; thus, each puzzle consists of tiles arranged in an $width \times width$ grid. The layout of the tiles is stored in the *tiles* array, where $tiles[i \times width + j]$ determines the number of the tile located in row i and column j of the grid; the empty tile is identified by the tile number 0. Finally, *emptyTileRow* and *emptyTileColumn* contain the row and the column of the empty tile, respectively; although these values can be obtained from the *tiles* array, they are stored explicitly for convenience.

- The *Movement* class implements the *Action* interface and enumerates the four directions in which the empty tile can be moved.

- The *TilesGoalTest* class implements the *GoalTest* interface and checks whether all tiles are in the correct positions.

- The *NPuzzlePrinting* class extends the abstract *Printing* class by providing two *print()* methods that print a movement and a tile configuration, respectively.

- The *BFTS_Demo* class demonstrates the breadth-first tree search (this is the meaning of the *BFTS* prefix). It contains a *main()* method and can thus be executed. When started, the *main()* method creates an initial tile configuration, invokes the search process, and prints a solution (if one was found).

The *tour* Package

The *tour* package should be used when solving the optional task of this practical, which is to find a tour which starts in Bucharest, visits all the cities in the map of Romania used in the AIMA book, and returns to Bucharest.

- The *SetUpRomania* class contains two static methods that instantiate maps of Romania. The *getRomaniaMap()* method returns the entire map of Romania as given in the AIMA book, and the *getRomaniaMapSmall()* method returns a smaller version of the map.

- The *City* class represents a city on the map. Each city is uniquely identified by its name, and one can retrieve for each city the set of roads leaving the city. Furthermore, for each city, one can also retrieve the shortest distance to an arbitrary other city on the map; if the cities are not connected, *Integer.MAX_VALUE* is returned. (The distances will be used only in the optional part of Practical 2.)

- The *Cities* class contains details of a set of cities indexed by their name.

- The *TourState* class implements the *State* interface and represents a current *City* and a collection of visited cities.

- The *Road* class implements the *Action* interface and represents a (directed) road between two cities. Each road consists of a source city, a target city, and the road length.

- The *TourGoalTest* class implements the *GoalTest* interface and checks whether all desired cities have been visited and desired destination has been reached.
- The *TourPrinting* class extends the abstract *Printing* class by providing two *print()* methods that print the road taken and the current city.
- The *BFTS_Demo* class demonstrates the breadth-first tree search When started, the *main()* method creates a small map of Romania, invokes the search process, and prints a solution (if one was found).

Task 1: Encapsulate the Notion of a Frontier

Whether a search algorithm is depth- or breadth-first depends on the way in which the frontier is managed during the search process. Your task is to encapsulate this behaviour in a separate interface and provide the implementations of the interface that implement depth-first and breadth-first frontiers. Doing this will allow you to implement tree and graph search in a generic way, without hard-coding the frontier behaviour into the search algorithm.

To this end, add to the *search* package an interface called *Frontier* that contains methods providing the following functionality.

- It should be possible to add a node to the frontier.
- It should be possible to clear the contents of a frontier.
- It should be possible to test whether the frontier is empty.
- If the frontier is not empty, it should be possible to remove a node from the frontier.

Furthermore, create two implementations of the *Frontier* interface called *DepthFirstFrontier* and *BreadthFirstFrontier*, each implementing the required behaviour.

Task 2: State Equality

In order to implement graph search, you will need a way to determine whether two states are equal to each other; in other words, classes implementing the *State* interface are required to support a proprietary notion of equality that can be used to check whether two states are equal. In Java, this can be achieved by implementing the following two methods:

- *boolean equals(Object that)*
- *int hashCode()*

Add these two methods to the *State* interface. Furthermore, implement these methods in the *Tiles* and the *TourState* classes in the appropriate way.

The general principles for implementing a proprietary notion of equality in Java are available in Java textbooks and online.[2]

[2] e.g.,http://technologiquepanorama.wordpress.com/2009/02/12/use-of-hashcode-and-equals/

Task 3: Encapsulate Search Algorithms

Add to the *search* package an interface called *Search* that encapsulates the notion of a search algorithm. The interface should provide a method that, given a root node and a goal test, returns a node containing a solution or *null* if no solution can be found.

Provide two implementations of the *Search* interface called *TreeSearch* and *GraphSearch*. The constructors of both classes should take an instance of the *Frontier* class, which will parameterize the search algorithm with the appropriate frontier behaviour. In this way it should be possible to obtain the four combinations of depth-first vs. breadth-first search and graph vs. tree search without any code duplication; for example, to obtain depth-first graph search, one should instantiate the *GraphSearch* class parameterized with an instance of the *DepthFirstFrontier* class.

To provide information about the performance of each kind of search, extend the *Search* interface with a method that returns the number of nodes generated during the last search, extend the *Frontier* interface with a method that returns the maximum number of nodes stored on the frontier since the frontier was created, and implement these methods in the relevant classes.

Task 4: Implement Iterative Deepening (Optional)

Provide an implementation of the *Search* interface that implements iterative deepening; the implementation class should be called *IterativeDeepeningTreeSearch*. The constructor of the class should take no arguments (this is because iterative deepening always performs depth-first search, so it is not necessary to parameterize it with a frontier). Instead of using recursion to implement depth-first depth-limited search, you should use the *DepthFirstFrontier* and a non-recursive algorithm. In order to cut off the search at a particular depth, you will need to extend the *Node* class with an integer member called *depth* which will keep track of the node's depth; this member should be set in the constructor of the *Node* class.

Task 5: Compare Efficiency

For the n-puzzle and optionally for the Romania tour, implement code that will execute the following types of search:

- breadth-first tree search,
- breadth-first graph search,
- depth-first tree search,
- depth-first graph search, and
- iterative deepening tree search (optional).

In addition to running the search and printing the results, in each case your code should should print

- the total number of nodes generated during the search, and
- the maximum number of nodes stored on the frontier at any point during the search.

Discuss in a few sentences the results obtained by running the above mentioned search classes on the n-puzzle and optionally the Romania tour problem.