Intelligent Systems Practical 2
Implementing Basic Search Strategies

Introduction

This practical illustrates the problems involved in implementing basic search strategies. The tasks in this practical build on the result of Practical 1; you can use your own solution to Practical 1 or you can start from the model answers; these can be obtained from staff running the practicals.

The search library produced in Practical 1 implements only breadth-first tree search. Your goal is to extend the library with the following search strategies:

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

This goal can be achieved in a straightforward way by providing five classes each implementing a different search algorithm from the above list. Note, however, that there is significant commonality between, say, depth-first tree search and breadth-first tree search. Your implementation should therefore be factored in a way that minimizes code duplication.

You should ensure that the code of your solution is well documented. The deadline for submitting solutions to this practical is week 6.

Task 1: 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()*

The general principles for implementing a proprietary notion of equality in Java are described in Lecture 6 of the Object Oriented Programming course; furthermore, tutorials on this topic are available online.[1]

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

[1] http://technologiquepanorama.wordpress.com/2009/02/12/use-of-hashcode-and-equals/

Task 2: 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 behavior 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 behavior 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 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.
- It should be possible to add a node to the frontier.

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

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 behavior. 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.

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 the following classes that invoke the particular type of search:

- *BFTS_Demo* should invoke the breadth-first tree search,
- *BFGS_Demo* should invoke the breadth-first graph search,
- *DFTS_Demo* should invoke the depth-first tree search,
- *DFGS_Demo* should invoke the depth-first graph search, and
- *IDTS_Demo* should invoke the iterative deepening tree search (optional).

In addition to running the search and printing the results, each of these classes should print

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

To retrieve this information in a modular way, 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 all these methods in the relevant classes.

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.