Intelligent Systems Practical 2
Implementing Best-First and A* Search

Introduction

This practical illustrates the problems involved in implementing best-first and A* search. 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 uninformed search strategies that can explore large portions of the search space unnecessarily. In order to improve search performance, you will now implement the best-first strategy, in which nodes are sorted on the frontier according to a node evaluation function. In addition, you will instantiate your best-first search implementation to carry out an A* search, using appropriate heuristics.

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: Implement Best-First Search

Add to the *search* package an interface called *NodeFunction*. This interface should contain a single method that is given a node and that produces an integer value. Implementations of this interface will provide the values that are denoted as $f(n)$ and $h(n)$ in the lectures.

In best-first search, nodes are sorted on the frontier according to a *node value*; in lectures, this value is denoted as $f(n)$. To this end, extend the *Node* class with a member that keeps track of the node's value. This member should be set when adding a node to the best-first frontier, as described next.

Provide an implementation of the *Frontier* interface called *BestFirstFrontier*. The constructor of this class should accept an instance of the *NodeFunction* interface that will be used to compute $f(n)$. When a node is added to a best-first frontier, the frontier should invoke the evaluation function on the node to determine the node's value, and it should store the value in the node. (In this way, the node's value is cached and does not need to be recomputed over and over.) When removing a node from a best-first frontier, a node with the lowest value should be returned; if there are several such nodes, then one of them should be selected arbitrarily. A simple way to implement this behaviour is to keep the nodes on the frontier sorted by their value; you can use Java's *PriorityQueue* class to obtain the required behaviour.

Task 2: Implement the A* Node Function

Extend the *Action* interface with a method that returns the action's cost; in lectures, this is denoted as $c(n', a, n)$. You should assume that the cost is a non-negative integer. Implement this method in the n-puzzle problem and optionally in the Romania tour problem.

Extend the *Node* class with a member that keeps track of the cost of the path from the root node; in lectures, this is denoted as $g(n)$. This value should be set in the constructor of the *Node* class. For a root node n, set $g(n) = 0$; and for a node n with parent n', set $g(n) = g(n') + c(n', a, n)$.

The A* node value is given by $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic function that estimates the cost of reaching a goal from node n. Provide an implementation of the *NodeFunction* called *AStarFunction*. The constructor of *AStarFunction* should accept an instance of *NodeFunction* called *heuristicFunction*; the latter should provide the *AStarFunction* with $h(n)$. When an instance of *AStarFunction* is asked to provide the value for a node n, it should invoke *heuristicFunction* in order to obtain the value of $h(n)$, and it should then return $g(n) + h(n)$.

Task 2: Implement a Heuristic for the n-Puzzle

Provide an implementation of the *NodeFunction* interface that, given a node n, returns the number of tiles that are not in the required positions; the implementation class should be called *MisplacedTilesHeuristicFunction*.

Task 3 (Optional): Implement a Heuristic for the Romania Tour

Provide an implementation of the *NodeFunction* interface that, given a node n in a search tree for the Romania tour problem, provides an admissible and consistent heuristic function for that node.

A suitable value for the heuristic function can be obtained using the following observation: to complete the tour it will be necessary to travel from the current city c to the city c' in the set of cities not yet visited that is furthest away from c, and then from c' to the goal city. Note that the heuristic function obtained in this way depends on the set of cities not yet visited and the goal city, so sufficient information needs to be supplied in the constructor to enable these to be calculated.

Task 4: Compare Efficiency

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

- A* tree search, and
- A* graph search.

Just as in Practical 1, your implementation 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. Compare and discuss these values for A* search and the values obtained in Practical 1.