Assignment 3

Anytime Search Algorithms

Important!

The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone else for help. Under no circumstances should you work together with another student on any code ***you or they*** write for the assignment. You are expected to modify the Prolog code that is provided.

However, it is allowable and even encouraged for you to discuss what the provided Prolog code does and how it does it. The class forum would be the most appropriate place for this discussion to take place so that all the class can participate and learn :^)

It is allowable and even encouraged to discuss, on the class forum, exactly what is being expected for this assignment.

Due: *11pm Friday 5th June*

Worth: *10% of final grade*

Introduction

A* needs to store all of the created but unexpanded nodes of the search space, in order to be able to select the nodes to expand in ascending ***f*** value order. These unexpanded nodes are usually stored in the *open list*. In many search spaces, these unexpanded nodes constitute a majority of the nodes created so far. So, storing all the nodes created usually does not cost significantly more than just storing the open list.

If all the created nodes are stored then we can avoid expanding nodes which have already been expanded and for which an expansion will not yield any more useful information. In some spaces (e.g., sliding tiles problem spaces), avoiding this redundant search can result in significant savings. When ***A**** checks for duplicate states, it is called a graph search and the states which have already been expanded are stored in the *closed list*.

The closed list is also used to store all the back pointers (i.e., a pointer to the parent state of a node), which enables ***A**** to cheaply extract the solution path when a goal state is found. If the heuristic is consistent (in addition to being admissible) then the first time a state has been created, its optimal path (from the initial state) was found. Thus, when a state is visited a subsequent time, it is not expanded again. This means that closed list need only store the state description and its back pointer (i.e., parent state).

If, however, the heuristic is not consistent then there is no guarantee that the first time a state has been created that its optimal path has been found. Thus, later re-creations of that state may discover better paths. Since we want ***A**** to always find the optimal solution path, it must always check whether a later visit to that state has found a shorter path. If it has then that state must now have a back pointer to its new parent state. In addition to changing the back pointer, the new lower cost path to the state may result in there now being new lower cost paths to all of that state's descendants. This means that the change in the cost to that state needs to be propagated forward, so that the ***f*** values associated with all that node's descendants can be updated. Thus, if the heuristic is inconsistent, not only must the ***A**** algorithm become slightly more complicated, but, also the lowest discovered ***f*** value for each state must also be stored with the state so that the algorithm can detect when a new shorter path to it has been found.

This assignment looks at part of a new variant of ***A****. This new variant combines both data structures into one, a *node list*. This new structure causes one more field to be added to the node information, namely, whether it needs to be expanded. The part this assignment explores is to define in Prolog the *updated* relationship between a node list, NL, along with a node, N, and a node list, UNL. This relationship is defined formally as follows:

Given that a node list never has two nodes that have identical states, that N is never an element of NL, that NL and UNL are sets, and that *state* and *fValue* are functions that take a node as an argument and return (respectively) the state description or the *f* value of that node, then

$$
\begin{aligned}
&\mathit{updatedNodeList}(NL, N, UNL) \Leftrightarrow \\
&\quad ((\forall N1 \in NL\ \mathit{state}(N) \neq \mathit{state}(N1)) \\
&\qquad \rightarrow UNL = NL \cup \{N\}) \\
&\vee\ ((\exists N1 \in NL\ \mathit{state}(N) = \mathit{state}(N1)\ \&\ \mathit{fValue}(N) \geq \mathit{fValue}(N1)) \\
&\qquad \rightarrow UNL = NL) \\
&\vee\ ((\exists N1 \in NL\ \mathit{state}(N) = \mathit{state}(N1)\ \&\ \mathit{fValue}(N) < \mathit{fValue}(N1)) \\
&\qquad \rightarrow UNL = \{N\} \cup NL - \{N1\})
\end{aligned}
$$

Note that the three disjuncts form mutually disjoint and exhaustive cases (i.e., one and only one of those three disjuncts can be true and one of them must be true for the relationship to hold). These three disjuncts form the basis for the three Prolog clauses you will write for the ***updatedNodeList/3*** predicate. The first disjunct says that if no node in ***NL*** has the same state description as ***N*** then ***UNL*** contains the node ***N*** in addition to containing the exact same nodes as ***UNL.*** The second disjunct states that if there does exist a node in ***NL*** that has the same state description as ***N*** and has an ***f*** value that is less than or equal to ***N***'s then ***UNL*** contains exactly the same nodes as ***NL***. The last disjunct states that if there does exist a node, ***N1***, in ***NL*** that has the same state description as ***N*** and has an ***f*** value that is greater than ***N***'s then ***UNL*** has exactly the same nodes as ***NL*** except that it doesn't have ***N1*** but does have ***N***.

Resources

In my assignment directory on the course web site, there is a file called ***updatedNodeList.pl***, which is what you must modify for your assignment. This file is self-contained. You do not need any other files. Besides containing the auxillary predicates you will need to do the assignment, it also contains some example test predicates to perform some unit testing of your code.

Marking Schedule

You should write 3 clauses, one for each of the disjuncts that define the ***updatedNodeList/3*** predicate in this document. There are 3 marks for a correct Prolog implementation of the first disjunct. There are 3 marks for a correct Prolog implementation of the second disjunct. There are 4 marks for a correct Prolog implementation of the last disjunct.

Submission

You submit your modified version of ***updatedNodeList.pl*** to the ADB.