CISS450: Artificial Intelligence
Assignment 6

Objectives

1. The goal is to implement a backtracking strategy for graph search.

Note that you must implement the late graph search algorithm for this assignment (i.e., not the early graph search algorithm).

Q1. Maze problem (again)

Modify the maze program by adding the `bts` (backtracking search) option where `bts` is implemented not with recursion but uses a graph search and with fringe and closed list instead. Note that the fringe for the general `dfs` can contain multiple successors of a node. For `bts` fringe, only one successor of a node is in the `bts` fringe; siblings will not be in the fringe at the same time. See notes for description of `bts`. (Spoiler hints below.)

The input/output console screen shot is the same as a05. Here's an example to fix input/output:

```
enter random seed: 5
0-random maze or 1-stored maze: 0
initial row: 1
initial column: 2
goal row: 3
goal column: 4
bfs or dfs or ucs or iddfs or bts: bts
solution: ['S', 'N', 'E', 'N']
len(solution): 4
len(closed_list): 0
len(fringe): 0
```

SPOILERS, HINTS, AND IDEAS

The ideas on the non-recursive version of `bts` were already mentioned in class. Here's a repeat of that with more details.

An item in the fringe is of the form `(node, actions left)`. For instance `(n1,[S,E,W])` means that node `n1` has to expand successors for actions `S,E,W` in that order. Suppose the fringe contains just one item right now:

`(n1,[S,E,W])`

When you call get on the fringe, suppose the expansion of node `n1` with action `S` gives node `n2`. Then `n2` is returned and furthermore if `n2` has action `N,S,E`, then `(n2,[N,S,E])` is placed in the fringe so that the fringe is

`(n2,[N,S,E]), (n1,[E,W])`

where I'm drawing the above with the top of stack on the left. Note that the item `(n1,[S,E,W])` is changed to `(n1,[E,W])` since `n1` has already expanded the successor with action `S`.

If you call get again, suppose the expansion of `n2` with action `N` is `n3`, then `n3` is returned. Furthermore if `n3` has actions `N,W`, then `(n3,[N,W])` is placed in the fringe and the fringe becomes

`(n3,[N,W]), (n2,[N,S,E]), (n1,[E,W])`

On calling get, if the successor of `n3` with action `N` is `n4`, but `n4` has no actions, then `n4` is returned and the fringe becomes

`(n3,[W]), (n2,[N,S,E]), (n1,[E,W])`

In other words, the point is to keep the parent around and also remember for each parent what actions need to be executed to get the successors of this parent.

Be careful!!! Make sure you do some traces on the above idea on a small state graph before writing the `BTSFringe` class.

For instance what do you do if you have a search node that is a leaf with no successor? And in the example above after `(n3,[W])` has produced a successor this item becomes `(n3,[])` – what do you do with this? What about the initial node? Notice that all nodes returned by get of the fringe is due to a successor computation. But the initial node is not a successor! How would you handle the initial node? (Spoiler hint: Make it a special case. For instance for the initial node, say it's `n1`, let the actions left `None`, i.e., initially put `(n1,None)` into the fringe and modify the code for get to handle the case separately. If `(n1,None)` is the top of the fringe, remove it, return `n1` and push `(n1, actions of n1)` onto the fringe.)

Now once you understood the strategy above, implement it this way: instead of using `(n1,[S,E,W])`, add the actions `[S,E,W]` as an instance to `n1`. In other words modify the `SearchNode` class so that each `SearchNode` object has an `actions_left` member so that in the above example, `n1.actions_left` is `[S,E,W]`. Of course this means that the `SearchNode` constructor must also have an `actions_left` parameter.

NOTES.

- One thing to note is that in `dfs`, all children are generated and then placed on the stack. This means that the last child is at the top of the stack and is processed sooner than the first child. But for `bts` first child is processed *before* the last child.

- Note that the backtracking strategy can be applied to any form of depth first search. For instance the `bts` search idea can be used in `iddfs` or `ildfs` too.

- If you really want to have a clean implementation of all the graph search strategies, the best design is to have a collection of classes of each strategy. For instance for the ucs, you should have UCSState, UCSProblem, UCSSearchNode, UCSFringe, etc. For instance notice that we started with a simple `SearchNode` class and we have been adding information to it (depth, path cost, etc.) A better design is to have `SearchNode`, `UCSSearchNode`, `IDDFSSearchNode`, etc.

  You can read up more about design ideas if you google design patterns. Do not try the above in the assignments for this class. You can do this as a personal project, including writing your own graphical animation and post your project on github.