Generic Search Algorithm

CSCI 2400 “Part 3b”
Instructor: David Byrd

Here is *one* version of a generic search algorithm. This version sacrifices memory efficiency in two ways: (1) for code simplicity and (2) to enable UCS and A* with minimal changes.

The Algorithm

```
GSA(problem, open):
    open.push( (S_0, []) )
    closed <- []

    while open is not empty:
        curr, path <- open.pop()
        if curr not in closed:
            if curr is goal: return path
            closed.append(curr)

            for s, act in successors(curr):
                if s not in closed:
                    open.push(s, path + [act])
    return []
```

This pseudocode obviously assumes that you have start state S_0, an is_goal function, and a successors function already defined somewhere, perhaps in the `problem` parameter. (Hint: in the first several Pacman questions, this is done for you.)

What’s nice about this generic algorithm? Now implement breadth-first search and depth-first search...

```
BFS(problem):
   GSA (problem, Queue())
```

```
DFS(problem):
   GSA (problem, Stack())
```

That’s it! With a few small changes to `GSA`, you could also allow UCS and A* to be one-line calls to `GSA()`.

Example Walkthrough

Look at the map of Romania in Figure 3.1 of your textbook. (You can also legally find all the figures here, straight from the authors: `http://aima.cs.berkeley.edu/figures.pdf`).

Here I walk through how the above algorithm handles depth-first search while finding a path from Arad to Bucharest. For the problem, I assume that each edge (action) in the graph has a label combining the origin and destination node. That is, the action (edge) that takes you from A to Z is called $A \rightarrow Z$.

We will assume that the successors for each state are returned in such an order that we will *actually explore them* in alphabetical order. (That is, if the successors are X, Y, and Z, they will be pushed on the open stack such that X will come off first. This only matters for successors discovered at the same time.)

Initialization

```
open: [ (A, []) ]
closed: []
```

While loop 1

```
curr: A
path: []

closed: [A]
open: [ (S, [ A->S ]), (T, [ A->T ]), (Z, [ A->Z ]) ]
```

We first pop a tuple from the open stack into the curr and path variables. (This time, it is simply the start node with an empty path.). We test to see if it was already in the closed list. It was not. We test if it is a goal. It is not. (Note that if A, the start, were also a goal, we would correctly stop here and return an empty path!). So we add A to the closed list, fetch its successors, and add them to the open stack. The path added for each successor in the open stack is the path we had to the current node (A) plus the single action that gets us from A to the successor.

Remember that we have no control over the order the successors are returned in. We just pick an order for class purposes so we can all get the same answers.

While loop 2

```
curr: S
path: [ A->S ]

closed: [A, S]
open: [ (F, [ A->S, S->F ]), (O, [ A->S, S->O ]), (R, [ A->S, S->R ]),
        (T, [ A->T ]), (Z, [ A->Z ]) ]
```

For visual purposes, I am pushing/popping at the left side of the stack. The second time through, we pop current node S with a path-so-far of $A \rightarrow S$. S is neither closed nor a goal, so we add it to the closed list. Then we get the successors and add them to the open list. Note that they *all* go to the *left* of all existing nodes in the open list, meaning they will be searched first. It is critical for depth first search that F, O, and R all be searched before T and Z, but remember the order of F, O, R with respect to each other is *arbitrary.* The path so far to S was $A \rightarrow S$, so the path successor F is $A \rightarrow S \rightarrow F$, and similarly for the others.

Important: In this version of GSA, I am not adding successors that are already in the closed list. This is safe; we have already expanded its successors, and it is not possible to find a shorter path to a node in the closed list. It would *not* be safe to skip adding a successor just because it is already in the open list!

While loop 3

```
curr: F
path: [ A->S, S->F ]

closed: [A, S, F]
open: [ (B, [ A->S, S->F, F->B ]), (O, [ A->S, S->O ]), (R, [ A->S, S->R ]),
        (T, [ A->T ]), (Z, [ A->Z ]) ]
```

The current node is F and path-so-far is $(A \rightarrow S \rightarrow F)$. F is not closed and not a goal, so we add it to the closed list. The successors are B and S. Node S is already in our closed list, so we only add B, reachable by action $F \rightarrow B$, to the open list.

While loop 4

```
curr: B
path: [ A->S, S->F, F->B ]

closed: [A, S, F]
open: [ (O, [ A->S, S->O ]), (R, [ A->S, S->R ]), (T, [ A->T ]), (Z, [ A->Z ]) ]
```

The current node is B and path-so-far is $(A \rightarrow S \rightarrow F \rightarrow B)$. B is not closed, and it *is* in the goal set, so we stop and return the current path. Successors are not generated. It is implementation-depended whether B ends up in the closed list or now. (You could check for goal status before or after adding it.)

Result

The algorithm returns path $[A \rightarrow S, S \rightarrow F, F \rightarrow B]$. Note that the path is of length 3, *not* 4, because it is the actions (transitions or edges) taken, not the nodes visited. In this case, depth first search happened to find precisely the shortest path, and found it with zero "wasted" searching of incorrect paths, but obviously this will not be true in most cases.

Acknowledgements and thanks to Professors Mark Riedl and Jim Rehg of the Georgia Tech School of Interactive Computing.