Assignment 2

This assignment is to build a planning module for your agent that will allow it to move from one location to another in as few actions as possible.

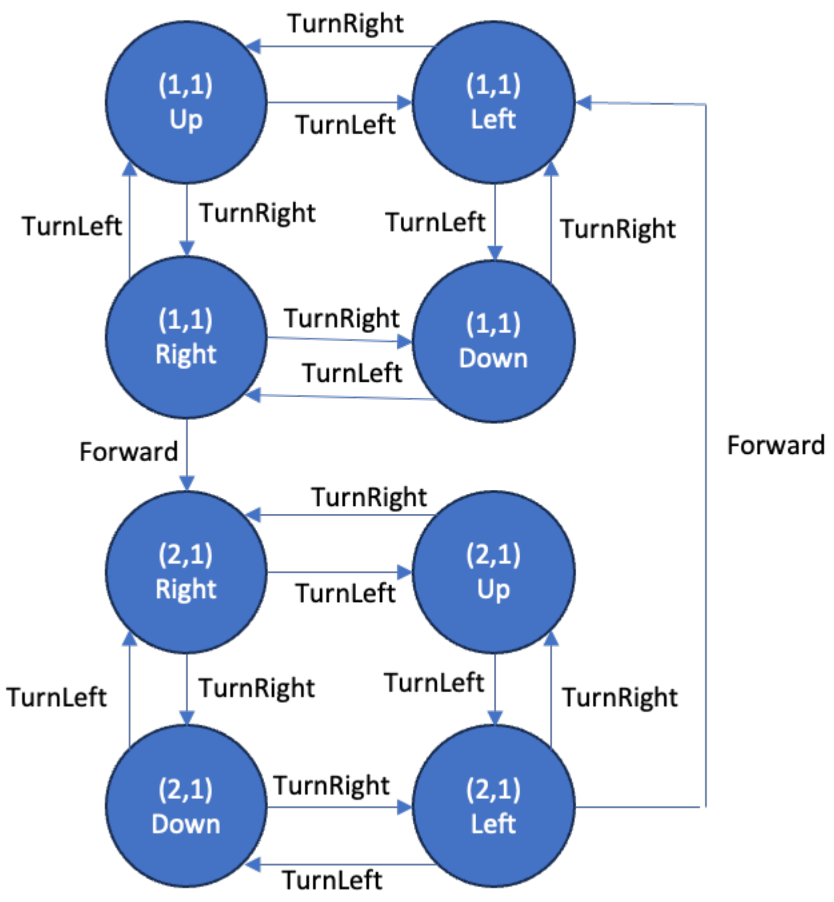
We'll add this capability to our naïve agent to create a new one we will call the MovePlanningAgent. Our agent won't yet know how to avoid dangers in cells it hasn't visited, but will stick to using cells it knows are safe to the extent it can when planning a move. Any cell the agent has visited previously and hasn't died can be considered safe.

We can describe the problem as a state-space graph which we can search for solutions. For the agent to get from one cell to another, sticking to cells it knows are safe, we will need a graph with a node for each safe state. It isn't sufficient to have a node just for each safe cell because the agent may have to turn to be able to move to the next cell in the chain and each turn also costs one point.

So we want a graph where each node is a valid safe state and it needs to include not just the location of the agent, but also its orientation. If a cell is safe, all four orientations of the agent within that cell are safe. We'll need four nodes in our graph for each cell our agent has successfully visited during its travels.

Each arc will be an action, of which there are three that involve moving around: Move, TurnLeft and TurnRight. We can label the arcs with these three labels so that when we execute the plan we can look at the arc labels to tell us what actions to take.

Each time the agent reaches a cell it hasn't been in previously we will need to extend our state-space graph with four nodes for the four possible agent orientations and all the arcs necessary to describe the possible additional safe moves. If the agent's first action where a Move and the agent has not died, the graph should then look like:



Note that we can leave off move actions that don't change the state e.g. moving down from (1, 1).

Now, if we want to get from (1, 1) to (2, 1) as quickly as possible and we start out facing left, any shortest path in this graph from (1, 1) Left to any (2, 1) node is a lowest cost plan i.e. [TurnLeft, TurnLeft, Forward] or [TurnRIght, TurnRight, Forward].

Instructions

1. Create a copy of the NaïveAgent called. Remove the Grab and Climb actions from the list of random actions and add code to the agent so it only does a Grab when it first senses a glitter and a Climb if it is in (1,1) and has the gold. You'll need to add state to the agent so that it knows it's holding the gold when it has it. Change the main loop of the program to use your new agent.
2. Add state to your agent so it keeps track of where it's been and therefore which cells it knows to be safe. (We won't have our agent try to infer yet whether cells it hasn't visited are safe).
3. Use the Python NetworkX library to update the graph of safe locations each time the agent enters a cell it hasn't visited previously. We would like to use NetworkX's implementation of the A* algorithm and Manhattan distance as the admissible heuristic to find the shortest path, but it does not allow multiple target nodes for a search. (You can only specify a single goal node, not a function that recognizes any of the four alternative states for a cell as a goal node). We can

work around this by choosing any of the four goal nodes to use for the search target, then remove any turns that occur at the end of the returned list as they are superfluous. Unfortunately, we also need to reconstruct the actions from the node names because NetworkX returns a list of nodes traversed but not arcs taken.

Alternatively, you can write your own search, and use your knowledge of the problem to make it more efficient. Breadth-first would work well here because it is guaranteed to find the shortest path and is easier to implement from scratch than A^*. To keep the breadth-first search fast you want to avoid looping around in any of the many cycles in the graph, which can be done by not adding any new paths to the frontier where the path's last node already occurs anywhere in any of the paths already on the frontier. There is also no point in expanding with an action that just reverses the last action taken because it just takes us back to a state already on the path. Keep a list of the actions taken for each path because that is what you want to return from the search. Terminate the search as soon as any of the four goal nodes is reached.

4. Use your search to allow your agent to take the shortest known safe path out of the maze when if finds the gold. Modify your agent so that if it senses a Glitter at any time, it does a Grab to get the gold, then generates an escape plan to get to (1, 1) with the least number of actions. Perform the list of actions from the plan, then do a Climb and win the game.