Lab *1*

Tasnimul Hasnat

190041113

Thursday, Aug 31 2023

Depth First Search

```
def depthFirstSearch(problem):
     fringe = util.Stack()
     closedSet = []

     startNode = (problem.getStartState(), [])
     fringe.push(startNode)
     while True:
          if fringe.isEmpty():
               return None
          currNode = fringe.pop()
          currState, currActionSequence = currNode
          if problem.isGoalState(currState):
               return currActionSequence
          if currState not in closedSet:
               for successor in
problem.getSuccessors(currState):
                    nextState, nextAction, nextCost =
successor
                    nextNode = (nextState,
currActionSequence + [nextAction])
                    fringe.push(nextNode)

               closedSet.append(currState)
```

The code was given by the teacher. At first a fringe was created using the LIFO-structured stack. This fringe stores tuples that include the state and the action list needed to get to the current state. Also a list is kept that stores the states already visited. The problem's initial state is first pushed to the empty action list. Then we began looping and kept doing so until the fringe was empty. Inside the loop the last entered state was popped and checked whether it's our goal state or not. If it is then we return the action list. Otherwise we check if the node is in the visited list or not. If it is not, then we fetch the successors of the current state by using getSucessors() function, push them into the fringe with their updated action list and update the visited array. If even after expanding all nodes, the goal state is not reached, we return none.

Breadth First Search

```
    def breadthFirstSearch(problem):
        fringe = util.Queue()
        closedSet = []

        startNode = (problem.getStartState(), [])
        fringe.push(startNode)
        while True:
          if fringe.isEmpty():
              return None
          currNode = fringe.pop()
          currState, currActionSequence = currNode
          if problem.isGoalState(currState):
              return currActionSequence
          if currState not in closedSet:
              for successor in
problem.getSuccessors(currState):
                  nextState, nextAction, nextCost =
successor
                  nextNode = (nextState,
currActionSequence + [nextAction])
                  fringe.push(nextNode)
              closedSet.append(currState)
```

The code for the BFS search algorithm is quite similar to the DFS search algorithm. The only difference is that instead of using stack here, I use FIFO structured Queue.

Uniform Cost Search

```
    def uniformCostSearch(problem):
        """Search the node of least total cost
first."""
        "*** YOUR CODE HERE ***"
        # util.raiseNotDefined()
        fringe = util.PriorityQueue()
        closedSet = []

        startNode = (problem.getStartState(), [])
        fringe.push(startNode,0)
        while True:
            if fringe.isEmpty():
                return None
            currNode = fringe.pop()
            currState, currActionSequence = currNode
            currNodeCost =
    problem.getCostOfActions(currActionSequence)
            if problem.isGoalState(currState):
                return currActionSequence
            if currState not in closedSet:
                for successor in
    problem.getSuccessors(currState):
                    nextState, nextAction, nextCost =
    successor
                    nextNode = (nextState,
    currActionSequence + [nextAction])

    fringe.push(nextNode,currNodeCost+nextCost)
                closedSet.append(currState)
```

The implementation is quite similar to that of DFS and BFS, but here we used priority queue instead of stack and queue. The total cost to reach a state is considered as the priority for any state. For the initial

state this cost is set to 0. When we push the successor state to the fringe, the cost is calculated as the sum of the current total cost of the current action list and the cost to reach the next state. As the total or cumulative cost to reach a state is used as the priority of that state in the fringe, the UCS will return a path from source to goal state that has the least cost associated with it. If no path is found, it will simply return none.

—xxx—