```python
# search.py
# ---------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.
#
# Attribution Information: The Pacman AI projects were developed at UC Berkeley.
# The core projects and autograders were primarily created by John DeNero
# (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# Student side autograding was added by Brad Miller, Nick Hay, and
# Pieter Abbeel (pabbeel@cs.berkeley.edu).


"""
In search.py, you will implement generic search algorithms which are called by
Pacman agents (in searchAgents.py).
"""

import util

class SearchProblem:
    """
    This class outlines the structure of a search problem, but doesn't implement
    any of the methods (in object-oriented terminology: an abstract class).

    You do not need to change anything in this class, ever.
    """

    def getStartState(self):
        """
        Returns the start state for the search problem.
        """
        util.raiseNotDefined()

    def isGoalState(self, state):
        """
          state: Search state

        Returns True if and only if the state is a valid goal state.
        """
        util.raiseNotDefined()

    def getSuccessors(self, state):
        """
          state: Search state

        For a given state, this should return a list of triples, (successor,
        action, stepCost), where 'successor' is a successor to the current
        state, 'action' is the action required to get there, and 'stepCost' is
        the incremental cost of expanding to that successor.
        """
        util.raiseNotDefined()

    def getCostOfActions(self, actions):
        """
         actions: A list of actions to take

        This method returns the total cost of a particular sequence of actions.
        The sequence must be composed of legal moves.
        """
        util.raiseNotDefined()


def tinyMazeSearch(problem):
    """
    Returns a sequence of moves that solves tinyMaze.  For any other maze, the
    sequence of moves will be incorrect, so only use this for tinyMaze.
    """
    from game import Directions
    s = Directions.SOUTH
    w = Directions.WEST
    return  [s, s, w, s, w, w, s, w]

def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
```

```python
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    "*** YOUR CODE HERE ***"




    fringe = util.Stack()

    visited=[] # To keep the track of the visited points
    directions=[] # To provide the direction to reach the goal node.


    #Check if the start state is equal to the goal state
    if problem.isGoalState(problem.getStartState()):
        return []

    fringe.push((problem.getStartState(),[]))
    current_point, directions = fringe.pop()
    visited.append(current_point)

    while not problem.isGoalState(current_point):

        #If the fringe is Empty means to elements to explore hence goal sate is not possible


        #pop the data from the fringe

        #Get the successor value
        successor = problem.getSuccessors(current_point)

        if successor:

            for child in successor:

                if child[0] not in visited:
                    #If the successor point not in the stack then push into the fringe
                    fringe.push((child[0], directions+[child[1]]))

        #pop the data from the fringe
        current_point, directions = fringe.pop()
        visited.append(current_point)


    return directions
    util.raiseNotDefined()

def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"

    fringe= util.Queue()
    directions=[]
    visited=[]

    #Pushing the starting point into the Queue and also the visited list
    fringe.push((problem.getStartState(), []))
    (current_point, directions) = fringe.pop()

    #Mark the current point as visited
    visited.append(current_point)

    while not problem.isGoalState(current_point):

        #We will seek for the next point ie. Successor
        succ= problem.getSuccessors(current_point)
        #Push the successor into the Queue

        for child in succ:
            if child[0] not in visited:
                fringe.push((child[0],directions+[child[1]]) )
                # We will also mark this visited
                visited.append(child[0])
        (current_point, directions)= fringe.pop()

    return directions



    util.raiseNotDefined()

def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()

def nullHeuristic(state, problem=None):
```

```python
    """
    A heuristic function estimates the cost from the current state to the nearest
    goal in the provided SearchProblem.  This heuristic is trivial.
    """
    return 0

def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    #Initialization
    opened= util.PriorityQueue()
    visited=[]

    #Pushing the very first point into the fringe
    #Also we will prioritize based on the huristic whenever we push into the stack

    opened.push((problem.getStartState(),[], 0), 0 + heuristic(problem.getStartState(),problem))
    #Pop the value out of the fringe
    (current_node,path,cost) = opened.pop()
    #Add the point to the visited list
    visited.append((current_node,cost + heuristic(current_node, problem) ))


    while not problem.isGoalState(current_node):

        #Get the successor nodes
        successor = problem.getSuccessors(current_node)

        for child in successor:
            newcost= cost+ child[2]
            visitedExist = False

            for (VisitedNode,VisitedCost ) in visited:

                if (VisitedNode==child[0]) and (VisitedCost <= newcost):
                    visitedExist= True
                    break

            if not visitedExist:

                opened.push((child[0],path + [child[1]], newcost), newcost+ heuristic(child[0], problem))
                #Now we will add this node to the visited Node

                visited.append((child[0], newcost))

        (current_node,path,cost)= opened.pop()

    return path




    util.raiseNotDefined()


# Abbreviations
bfs = breadthFirstSearch
dfs = depthFirstSearch
astar = aStarSearch
ucs = uniformCostSearch
```