```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
from game import Directions
from typing import List


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: SearchProblem) -> List[Directions]:
    """
    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.
    """
    s = Directions.SOUTH
    w = Directions.WEST
    return  [s, s, w, s, w, w, s, w]

def depthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """
    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
```

```python
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    visits = set()
    start = problem.getStartState()
    stack = util.Stack()
    stack.push((start, []))

    while not stack.isEmpty():
        current, path = stack.pop() # Tuple (state, path) - pop the last element
        if problem.isGoalState(current):   # Check if the current state is the goal state
            break
        if current not in visits:  # Check if the current state is visited
            visits.add(current)
            for adj, direction, _ in problem.getSuccessors(current): # Get the successors of the current state
                if adj not in visits:
                    stack.push((adj, path + [direction]))
                    #print(stack)


    return path
    #util.raiseNotDefined()

def breadthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """Search the shallowest nodes in the search tree first."""
    visits = set()
    start = problem.getStartState()
    queue = util.Queue()
    queue.push((start, []))

    while not queue.isEmpty():
        current, path = queue.pop() # Tuple (state, path) - remove the first element
        if problem.isGoalState(current):   # Check if the current state is the goal state
            break
        if current not in visits:  # Check if the current state is visited
            visits.add(current)
            for adj, direction, _ in problem.getSuccessors(current): # Get the successors of the current state
                if adj not in visits:
                    queue.push((adj, path + [direction]))
                    #print(queue)

    return path
    #util.raiseNotDefined()

def uniformCostSearch(problem: SearchProblem) -> List[Directions]:
    """Search the node of least total cost first."""
    visits = set()
    start = problem.getStartState()
    priqueue = util.PriorityQueue()
    priqueue.push((start, []), 0)

    while not priqueue.isEmpty():
        current, path = priqueue.pop() # Tuple (state, path, priority) - remove the element with the lowest cost
        if problem.isGoalState(current):   # Check if the current state is the goal state
            break
        if current not in visits:  # Check if the current state is visited
            visits.add(current)
            for adj, direction, _ in problem.getSuccessors(current): # Get the successors of the current state
                if adj not in visits:
                    priqueue.push((adj, path + [direction]), problem.getCostOfActions(path + [direction]))
                    #print(priqueue.heap)

    return path
    #util.raiseNotDefined()

def nullHeuristic(state, problem=None) -> float:
    """
    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: SearchProblem, heuristic=nullHeuristic) -> List[Directions]:
    """Search the node that has the lowest combined cost and heuristic first."""
    start = problem.getStartState()
    priqueue = util.PriorityQueue()
    priqueue.push((start, [], 0), 0)  # (state, path, g-cost)
    best_path_cost = {}

    while not priqueue.isEmpty():
        current, path, g_cost = priqueue.pop()  # Pop the state with the lowest f = g + h

        if problem.isGoalState(current):
            break
```

```
        if current not in best_path_cost or g_cost < best_path_cost[current][1]:
            best_path_cost[current] = (path, g_cost)
            for adj, direction, step_cost in problem.getSuccessors(current):
                new_g = g_cost + step_cost
                f = new_g + heuristic(adj, problem)  # f = g + h
                if adj not in best_path_cost or new_g < best_path_cost[adj][1]:
                    priqueue.push((adj, path + [direction], new_g), f)

    return path
    #util.raiseNotDefined()

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