```python
# search.py

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

    from util import Stack
    fringe = Stack()                # Fringe to manage which states to expand
    fringe.push(problem.getStartState())
    visited = []                    # List to check whether state has already been visited
    path=[]                         # Final direction list
    pathToCurrent=Stack()           # Stack to maintaing path from start to a state
    currState = fringe.pop()
    while not problem.isGoalState(currState):
        if currState not in visited:
            visited.append(currState)
            successors = problem.getSuccessors(currState)
            for child,direction,cost in successors:
                fringe.push(child)
                tempPath = path + [direction]
                pathToCurrent.push(tempPath)
        currState = fringe.pop()
        path = pathToCurrent.pop()
    return path

    #util.raiseNotDefined()

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

    from util import Queue
    fringe = Queue()                        # Fringe to manage which states to expand
```

```
    fringe.push(problem.getStartState())
    visited = []                            # List to check whether state has already been visited
    tempPath=[]                             # Temp variable to get intermediate paths
    path=[]                                 # List to store final sequence of directions
    pathToCurrent=Queue()                   # Queue to store direction to children (currState and
pathToCurrent go hand in hand)
    currState = fringe.pop()
    while not problem.isGoalState(currState):
        if currState not in visited:
            visited.append(currState)
            successors = problem.getSuccessors(currState)
            for child,direction,cost in successors:
                fringe.push(child)
                tempPath = path + [direction]
                pathToCurrent.push(tempPath)
        currState = fringe.pop()
        path = pathToCurrent.pop()

    return path

    #util.raiseNotDefined()

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

    from util import Queue,PriorityQueue
    fringe = PriorityQueue()                    # Fringe to manage which states to expand
    fringe.push(problem.getStartState(),0)
    visited = []                                # List to check whether state has already been
visited
    tempPath=[]                                 # Temp variable to get intermediate paths
    path=[]                                     # List to store final sequence of directions
    pathToCurrent=PriorityQueue()               # Queue to store direction to children (currState
and pathToCurrent go hand in hand)
```

```
    currState = fringe.pop()
    while not problem.isGoalState(currState):
        if currState not in visited:
            visited.append(currState)
            successors = problem.getSuccessors(currState)
            for child,direction,cost in successors:
                tempPath = path + [direction]
                costToGo = problem.getCostOfActions(tempPath)
                if child not in visited:
                    fringe.push(child,costToGo)
                    pathToCurrent.push(tempPath,costToGo)
        currState = fringe.pop()
        path = pathToCurrent.pop()
    return path
    #util.raiseNotDefined()

def nullHeuristic(state, problem=None):
    """
    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 ***"

    from util import Queue,PriorityQueue
    fringe = PriorityQueue()                    # Fringe to manage which states to expand
    fringe.push(problem.getStartState(),0)
    currState = fringe.pop()
    visited = []                                # List to check whether state has already been
visited
    tempPath=[]                                 # Temp variable to get intermediate paths
    path=[]                                     # List to store final sequence of directions
```

```
    pathToCurrent=PriorityQueue()              # Queue to store direction to children (currState
and pathToCurrent go hand in hand)
    while not problem.isGoalState(currState):
        if currState not in visited:
            visited.append(currState)
            successors = problem.getSuccessors(currState)
            for child,direction,cost in successors:
                tempPath = path + [direction]
                costToGo = problem.getCostOfActions(tempPath) + heuristic(child,problem)
                if child not in visited:
                    fringe.push(child,costToGo)
                    pathToCurrent.push(tempPath,costToGo)
        currState = fringe.pop()
        path = pathToCurrent.pop()
    return path

    #util.raiseNotDefined()


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