problem.py: Implement Problem Class

Your task is to implement a base class for search problems that we will use later to define other problems via inheritance. Specifically, implement a class with the following properties:

Class name	Inherits from
`Problem`	`object`

Problem Constructor	
Constructor arguments	Constructor body
`initial_state, goal_state=None`	Add arguments to self.

Problem Functions			
Name	Arguments	Returns	implementation
`actions`	`state`	Returns actions available in state	Since Problem is a base class, this function should just raise NotImplementedError.
`result`	`state, action`	Returns the transition-state that results from executing action in state.	Since Problem is a base class, this function should just raise NotImplementedError.
`is_goal`	`state`	Returns true if the given state is equal to the goal, otherwise false. Assume == is enough to check for equality.	Returns true if the given state is equal to the goal, otherwise false. Use = = for equality.
`action_cost`	`state1, action, state2`	Returns the cost of transition from state1 to state2 using action.	For this base class, should return 1.
`h`	`node`	Returns the heuristic value of the given node.	For this base class, should return 0.

node.py: Implement Node Class

Your task is to implement a Node class that we will use as the building blocks for the search tree. Implement as follows:

Class name	Inherits from
`Node`	`object`

Node Constructor	
Constructor arguments	Constructor body
`state, parent_node=None, action_from_parent=None, path_cost=0`	Add arguments to self. In addition, add self.depth to be 0 if no parent-node or depth of the parent-node + 1.

Node class has only a single function:

```
    def __lt__(self, other):
        return self.state < other.state
```

This is just a safe-guard to break ties in the priority-queue using the state. See next page for priority queue class.

search_algorithms.py: Implement search helper functions

Firstly, add the following implementation of PriorityQueue that we will use later for search algorithms:

```
import heapq

class PriorityQueue:
    def_init_(self, items=(), priority_function=(lambda x: x)):
        self.priority_function = priority_function
        self.pqueue = []
        # add the items to the PQ
        for item in items:
            self.add(item)

    """
    Add item to PQ with priority-value given by call to priority_function
    """
    def add(self, item):
        pair = (self.priority_function(item), item)
        heapq.heappush(self.pqueue, pair)

    """
    pop and return item from PQ with min priority-value
    """
    def pop(self):
        return heapq.heappop(self.pqueue)[1]

    """
    gets number of items in PQ
    """
    def___len__(self):
        return len(self. pqueue)
```

Now implement the following search helper functions in search_algorithms.py. Be sure to add imports for problem.py and node.py. Implement the following functions (note these functions are not part of a class, just regular python functions):

Name	Arguments	Returns	implementation
`expand`	`problem, node`	Returns the children nodes of the given node.	See figure 3.7 pg 91 in AIMI 4/E.
`get_path_actions`	`node`	Returns a list of actions to get to the given node. If node is None return empty list []. If parent of node is None return an empty list [].	Hint: follow the parent pointers to the root. Be sure to return in the correct order (not reverse order).
`get_path_states`	`node`	Returns a list of states to get to the given node. If node is None return empty list []	Hint: follow the parent pointers to the root. Be sure to return in the correct order (not reverse order).

search_algorithms.py: Implement search algorithms

Your task is to implement search algorithms in search_algorithms.py. To make your job easier, first implement the best-first-search template. Then implement the main search algorithms as functions that call the best-first-search template with appropriate evaluation functions. Be sure to add imports for problem.py and node.py.

The following are the functions you need to implement:

Name	Arguments	Returns	implementation
`best_first_search`	`problem, f`	Returns the goal-node if found, otherwise returns `None` in case of failure.	See figure 3.7 pg 91 in AIMI 4/E. You can implement *reached* as a python dictionary with states as keys and nodes as values. Note that the argument `f` is a function (in python you can pass functions as arguments).
`best_first_search_treelike`	`problem, f`	Same as above.	Implement tree-like version of best_first_search. See slides
`breadth_first_search`	`problem, treelike=false`	Same as above.	Hint: implement as a call to best_first_search (or treelike version) with appropriate f function. Use the `treelike` flag to decide which version to call.
`depth_first_search`	`problem, treelike=false`	Same as above.	Same as above.
`uniform_cost_search`	`problem, treelike=false`	Same as above.	Same as above.
`greedy_search`	`problem, h, treelike=false`	Same as above.	Same as above. Note that the argument `h` is a heuristic function.
`astar_search`	`problem, h, treelike=false`	Same as above.	Same as above.

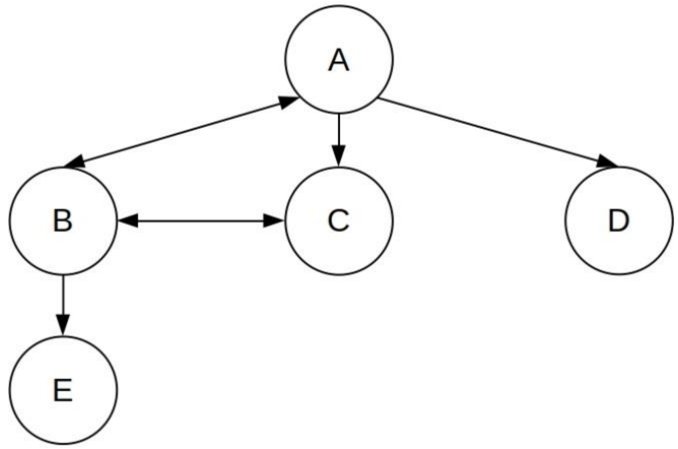
Feel free to add other functions to help you implement these algorithms. Hint: you can implement various functions to use as the `f` argument for certain search algorithms.

problem.py: Implement RouteProblem Class

Your task is to implement a class for route problems in problem.py. To ease your implementation, be sure to inherit the Problem class and override its functions.

<table>
<tr><th colspan="2">Class name</th><th colspan="2">Inherits from</th></tr>
<tr><td colspan="2">RouteProblem</td><td colspan="2">Problem</td></tr>
<tr><th colspan="4">RouteProblem Constructor</th></tr>
<tr><th colspan="2">Constructor arguments</th><th colspan="2">Constructor body</th></tr>
<tr><td colspan="2">initial_state,
goal_state=None,
map_graph=None,
map_coords=None</td><td colspan="2">Add arguments to self.
Assume map_graph is a dictionary where keys are (v1, v2) tuple indicating a directed edge between v1 and v2 states, values give the cost between v1 and v2.
Assume map_coords is a dictionary where keys are states, and values are (x, y) tuple indicating the xy-coordinates in 2D.</td></tr>
<tr><th colspan="4">RouteProblem Functions to Override</th></tr>
<tr><th>Name</th><th>Arguments</th><th>Returns</th><th>implementation</th></tr>
<tr><td>actions</td><td>state</td><td>Returns the states that can be reached by state If no state is reachable, then return an empty list []</td><td>Hint: although not necessary, consider making a self.neighbors dictionary in the constructor to help you.</td></tr>
<tr><td>result</td><td>state, action</td><td>Returns the transition-state that results from executing action in state. If no transition is possible, then return state (i.e. do not move)</td><td>Assume action will be a state. For example, result('A', 'B') is the action of going from A to B.</td></tr>
<tr><td>action_cost</td><td>state1, action, state2</td><td>Returns the cost of transition from state1 to state2 using action.</td><td>Hint: use map_graph Assume action == state2.</td></tr>
<tr><td>h</td><td>node</td><td>Returns the Euclidean distance to the goal.</td><td>Hint: use map_coords
Note: make sure to return 0 if node is a goal-node.</td></tr>
</table>

To clarify the usage of RouteProblem, consider the following map of cities.



Assume that start state is A, and the goal is E, assume all costs are 1. Then you can construct this graph as a dictionary:

```
example_map_graph = {
('A', 'B'): 1,
('A', 'C'): 1,
('A', 'D'): 1,
('B', 'A'): 1,
('B', 'C'): 1,
('B', 'E'): 1,
('C', 'B'): 1
}

example_coords = {
'A': (1,2),
'B': (0,1),
'C': (1,1),
'D': (2,1),
'E': (0,0),
}
```

Then create a RouteProblem object as:

```
example_route_problem = RouteProblem(initial_state='A', goal_state='E',
                                     map_graph=example_map_graph,
                                     map_coords=example_coords)

goal_node = breadth_first_search(example_route_problem, tree_like=False)
```

You can try testing your search algorithms by passing a route-problem. Try testing the practice problem from the slides.

problem.py: Implement GridProblem Class

Your task is to implement a class for grid problems in problem.py. To ease your implementation, be sure to inherit the Problem class and override its functions. Unlike route-problems where we pass the entire graph in the constructor, in a grid problem, we just need certain static information about the environment.

A grid problem instance consists of an N-by-M grid (N rows and M columns). Some grid locations contain walls that the agent cannot move into; we will use a list of locations `wall_coords` to store these wall locations. Some grid locations contain food that the agent can consume `food_coords`. The agent starts at some grid location (xA, yA). The agent can move left, right, up, and down one grid. The agent cannot move outside the bounds of the N-by-M grid. The goal is to consume all food on the grid.

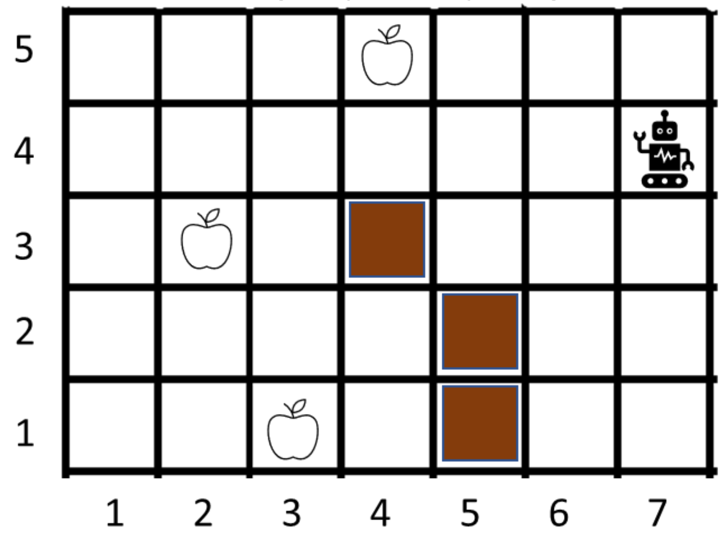
With this formulation, we need to keep track of which food was eaten. So, a `state` is a tuple of two elements: the current location (xA, yA) of the agent, and a Boolean **tuple** `food_eaten` of the same size as `food_coords` where `food_eaten[i]` indicates if `food_coords[i]` was eaten (true) or not (false). Initially, this tuple should be set to false (all food not eaten at start).

Note: the reason we represent `food_eaten` as a tuple instead of a list is because python dictionaries do not allow keys to be lists. Recall that we want to use a reached dictionary for our graph-like search algorithms. So, our states cannot have list objects. Moreover, you cannot modify or change tuples, they are immutable. However, you can convert a tuple X to list as list(X), perform modifications, then convert back to tuple as tuple(mylist). This may be convenient when you create the transition-states in the `result` function.

<table>
<tr><th>Class name</th><th>Inherits from</th></tr>
<tr><td><code>GridProblem</code></td><td><code>Problem</code></td></tr>
<tr><th colspan="2">GridProblem Constructor</th></tr>
<tr><th>Constructor arguments</th><th>Constructor body</th></tr>
<tr><td><code>initial_state,</code>
<code>N,</code>
<code>M,</code>
<code>wall_coords,</code>
<code>food_coords</code></td><td>Add arguments to self. You can assume the following:
<code>initial_state</code> is a pair (xA, yA) indicating the agent’s starting grid location.
For this problem, <code>goal_state=None</code>.
Assume <code>N</code> is a positive integer for row-size of the grid problem.
Assume <code>M</code> is a positive integer for column-size of the grid problem.
Assume <code>wall_coords</code> is a list of (x, y) tuple grid locations indicating the position of walls.
Assume <code>food_coords</code> is a list of (x, y) tuple grid locations indicating the position of food.
Create a tuple of booleans <code>food_eaten</code> of the same size as <code>food_coords</code> where <code>food_eaten[i]</code> indicates if <code>food_coords[i]</code> was eaten (true) or not (false). Initially, this tuple should be set to false (all food not eaten at start).
Set <code>self.initial_state = (initial_state, food_eaten)</code></td></tr>
</table>

GridProblem Functions to Override			
Name	Arguments	Returns	implementation
`actions`	`state`	Returns a list of actions that can be reached by `state.` If no action is legal, then return an empty list []. **Note**: you can use whatever format (strings, integers, etc) for the actions, but make sure to return the list of legal actions in up, down, right, left order.	Hint: based on walls and grid limits, you should return only legal actions. It is up to you decide on the format of the list of actions. I used strings 'up', 'down', 'right, 'left'. **Note**: you can use whatever format (strings, integers, etc) for the actions, but make sure to return the list of legal actions in up, down, right, left order.
`result`	`state, action`	Returns the transition-state that results from executing action. Recall that a state is a 2-element list.	Hint: if the action is illegal, then agent should remain in `state.` If the agent moves to a grid containing food, then the transition state should reflect this through the tuple of Booleans.
`action_cost`	`state1, action, state2`	Returns the cost of transition from state1 to state2 using action.	Any action costs 1.
`is_goal`	`state`	Returns true if all food items in state consumed.	Hint: recall that a state contains a tuple of Booleans.
`h`	`node`	Returns the heuristic distance to the goal.	Implement a heuristic that computes the **Manhattan** distance to the **nearest uneaten** food item. **Note**: make sure to return 0 if node is a goal-node.

To clarify the usage of GridProblem, consider the following 5-by-7 (N-by-M) grid instance:



The agent is in (7, 4). The walls are at [(4,3), (5,1), (5,2)]. The food are at [(3,1), (2,3), (4,5)].
We can create this GridProblem as follows:

```
example_walls = [(4,3), (5,1), (5,2)]

example_food = [(3,1), (2,3), (4,5)]

example_grid_problem = GridProblem(initial_state=(7,4),
                                   N=5, M=7,
                                   wall_coords=example_walls,
                                   food_coords=example_food)

goal_node = breadth_first_search(example_grid_problem, tree_like=False)
```

You can try testing your search algorithms by passing this grid-problem.

<u>Test your code on runner.py</u>

You can test your code by running test_runner.py which will run the search algorithms on toy problems. You can inspect the code and add your own problem cases.

The following are my implementation results:

```
BFS
      7 generated nodes |        5 popped |    2 solution cost |       2 solution depth | <problem.RouteProblem object a
    358 generated nodes |      118 popped |   11 solution cost |      11 solution depth | <problem.GridProblem object at
    365 generated nodes |      123 popped |   13 solution cost |      13 solution depth | TOTAL

-----------------------------------------------------------------
DFS
      6 generated nodes |        3 popped |    2 solution cost |       2 solution depth | <problem.RouteProblem object a
    169 generated nodes |       58 popped |   26 solution cost |      26 solution depth | <problem.GridProblem object at
    175 generated nodes |       61 popped |   28 solution cost |      28 solution depth | TOTAL

-----------------------------------------------------------------
UCS
      7 generated nodes |        5 popped |    2 solution cost |       2 solution depth | <problem.RouteProblem object a
    358 generated nodes |      118 popped |   11 solution cost |      11 solution depth | <problem.GridProblem object at
    365 generated nodes |      123 popped |   13 solution cost |      13 solution depth | TOTAL

-----------------------------------------------------------------
astar_treelike
      6 generated nodes |        3 popped |    2 solution cost |       2 solution depth | <problem.RouteProblem object a
 40,276 generated nodes |   12,426 popped |   11 solution cost |      11 solution depth | <problem.GridProblem object at
 40,282 generated nodes |   12,429 popped |   13 solution cost |      13 solution depth | TOTAL

-----------------------------------------------------------------
astar
      6 generated nodes |        3 popped |    2 solution cost |       2 solution depth | <problem.RouteProblem object a
    210 generated nodes |       68 popped |   11 solution cost |      11 solution depth | <problem.GridProblem object at
    216 generated nodes |       71 popped |   13 solution cost |      13 solution depth | TOTAL

-----------------------------------------------------------------
greedy
      6 generated nodes |        3 popped |    2 solution cost |       2 solution depth | <problem.RouteProblem object a
    266 generated nodes |       88 popped |   14 solution cost |      14 solution depth | <problem.GridProblem object at
    272 generated nodes |       91 popped |   16 solution cost |      16 solution depth | TOTAL

-----------------------------------------------------------------
```

Notice how the treelike version of astar generated a large number of nodes, whereas graph-like astar produced much fewer and finds the same solution.

[Optional Bonus] search_algorithms.py: Visualizing solutions

Your task is to implement some helper functions in search_algorithms.py to help us visualize the solution paths of RouteProblem and GridProblem. We will use the **matplotlib** packages to ease this implementation, so add these imports to search_algorithms.py as follows:

```
import matplotlib.pyplot as plt
```

First let us define a function that will visualize the solution of a RouteProblem instance.

Name: `visualize_route_problem_solution`
Arguments: `problem, goal_node, file_name`
Returns: does not return anything.

Implementation: You may want to make use of `get_path_states` to get the states-path to `goal_node`. Using problem and the path of states, you can plot each state using its (x, y) coordinate which will depict the graph vertices. When plotting the states, use scatter and set `marker` to make the plot points have a square shape (see documentation). Also, be sure to set the `color` of initial state to red, the goal state to green, and the transition states to blue (see color documentation). Next, plot the possible actions (directed edges between states) as black arrows using arrow. Next, plot the solution path as magenta directed arrows using arrow. Finally, save the resulting figure as a png using savefig. Do **not** define a figure size at the beginning of plotting, just start using plt to plot right away. Also, make sure at the end to call plt.close().

[Optional Bonus] search_algorithms.py: Visualizing solutions

Next let us define a function that will visualize the solution of a GridProblem instance.

Name: `visualize_grid_problem_solution`
Arguments: `problem, goal_node, file_name`
Returns: does not return anything.

Implementation: You may want to make use of `get_path_states` to get the states-path to `goal_node`. Using problem, you can plot the location of walls, food, and agent.

- Use scatter to plot the walls as black squares. Set the size of marker to `s=2500`.
- Use scatter to plot the food as green hexagons. Set the size of marker to `s=1000`.
- Use scatter to plot the initial location of the agent as red triangle. Set the size of marker to `s=1000`.
- Plot the agent movement as magenta directed arrows using arrow.
- Now, we want to adjust the limits of the plot to be more visually pleasing. So call these two commands which will set the x-axis and y-axis limits of the plot:
    - plt.ylim([0.5, problem.N + 0.5])
    - plt.xlim([0.5, problem.M + 0.5])

Finally, save the resulting figure as a png using savefig. Do **not** define a figure size at the beginning of plotting, just start using plt to plot right away. Also, make sure at the end to call plt.close().

You can test your function implementations on example RouteProblem and GridProblem to see the resulting figures.

If you run visualize_runner.py on your code, you can check your resulting images against the reference images: [ref_grid.png, ref_route.png]. You should reproduce the same images.