Dashboard / My courses / COSC367-2020S2 / Assignment super-quizzes / A1. A* Routing

Started on	Monday, 3 August 2020, 1:32 PM
State	Finished
Completed on	Sunday, 23 August 2020, 5:06 PM
Time taken	20 days 3 hours
Marks	3.00/3.00
Grade	100.00 out of 100.00

Information

Introduction

In this assignment you apply the search techniques you have learnt in the course to a routing problem. There are a number of mobile agents (think of self-driving taxis) scattered across a flat rectangular grid environment. There are also a number of call points (think of customers) waiting to be served. The objective is to, if possible, navigate an agent to a call point that are time-wise closest to each other.

Technical notes

The following points apply to all the questions in this super quiz.

- 1. In each instance of this problem we want to navigate one agent to one call point (not all agent and not all call points).
- 2. In all the questions you can assume that the search module (the file <u>search.py</u>) is available on the quiz server. This means that you can safely import the search module (in addition to all the standard Python modules) in your program. Do **not** repeat the code of search module in your answers.
- 3. Your solution must contain all the import statements that it requires, even for the search module.
- 4. You can have your entire program for all the three questions in one file and every time you want to submit your answer you can simply paste the content of the entire file in the answer box. Just make sure that you don't have any function calls in the global section of your file that may interfere with auto-grading. You can put all of your own test cases and calls to the print function in a main function and then have a statement like the following at the global level:

```
if __name__ == "__main__":
    main()
```

- 5. Answer the questions in order. The answer to some questions require your answer to earlier questions (to use or to build on).
- 6. Your answer to a question may pass all the test cases for the question (and receive full mark) but it may not have all the functionalities that are needed for later questions or it may a bug that is not detected by our test cases!

Question **1**Correct
Mark 1.00 out of 1.00

Writing the RoutingGraph class

In the first step of the assignment you have to write a subclass of Graph for a routing problem in an environment. The map of the environment is given in the form of a multi-line string of characters. The following shows an example map.

```
map_str = """\
+----+
| G G|
| XXX |
| S X |
| X 2 |
+-----+
```

Map description

- The map is always rectangular. We refer to positions in the map by a pair of numbers (row, col) which correspond to the row and column number of the position. Row numbers start from 0 for the first (topmost) line, 1 for the second line, and so on. The column numbers start from 0 for the first (leftmost) position (character) in the line, 1 for the second position, and so on.
- The environment is always surrounded by walls which are represented with characters '+' or '-' or '|'. For example the position (0,0) is always a '+' (i.e. wall) and so are all other three corners of the map. The first and last rows and the first and last columns are always '-' and '|' respectively (except for the corners).
- The obstacles are marked by 'x'.
- There may be zero or more agents on the map. The location of agents are marked by 's' or digits 0..9.
 - Agents indicated by s have solar panels and never require fueling. We can think of their fuel capacity to be infinity.
 - Agents indicated by digits have fuel tanks. The capacity of the tank is 9. The digit used to indicate the agent shows how much fuel is initially available in the tank.
- There may be zero or more call points (customers) on the map. The location of call points (potential destinations) are marked by 'G'. To simplify textual representation, we assume that an agent is never initially at a call point.
- An agent can move in four directions, N, E, S, W, as long as it has fuel and there is no obstacle or wall in the way. This means that agents can also go to cells where other agents are present. The agent loses one unit fuel for each move. The order of actions is clockwise starting from N. For example, if from a position all four directional moves are possible, then the first arc in the sequence of arcs returned by outgoing_arcs is for going north, then east, and so on until the last arc which goes to west. All single directional moves take 5 units of time.
- If an agent is in a cell marked as F and its current fuel amount is less than 9 it can take the action of "Fuel up" which fills the tank to its maximum capacity of 9. In the sequence of arcs, the "Fuel up" action (if available) should appear after any other directional actions. The action costs 15 units of time (regardless of how much fuel is obtained).

Task

Write a class RoutingGraph which is initialised by the map string and represents the map in the form of a graph by implementing all the required methods in the Graph class including the method estimated_cost_to_goal. Represent the state of the agent by a tuple of the form (row, column, fuel)

Notes

- 1. It is recommended that you write your class as a subclass of Graph.
- 2. The test cases do not test the method <code>estimated_cost_to_goal</code> in this question.
- 3. Try to avoid using indices to refer to elements of a tuple. For example instead of using position[0] and position[1], use readable names. For example use row, col = position instead.
- 4. You may find math.inf useful.
- 5. If you copy a multi-line string from the examples given in the questions into a function in your code, some leading spaces will be introduced. Use the method strip to get rid of these leading/trailing spaces. Also be mindful of the difference between the following two strings:

```
str1 = """This string splits into one line.
"""

def main():
    str2 = """This string splits into TWO lines.
    """
```

- 6. It is recommended (but not required) that your answer for RoutingGraph is shorter than 70 lines of code. If your code is much longer, you might be doing something wrong.
- 7. Avoid repetitive code. If you wish you can use the following list when implementing outgoing_arcs. You have to decipher it yourself.

```
[('N' , -1, 0),
('E' , 0, 1),
('S' , 1, 0),
('W' , 0, -1),]
```

For example:

Test	Result
------	--------

```
Test
from student_answer import RoutingGraph
import math
map_str = """\
| 9 XG|
X XXX
| S 0FG|
+----+
11 11 11
graph = RoutingGraph(map_str)
print("Starting nodes:",
sorted(graph.starting_nodes()))
print("Outgoing arcs (available actions) at
starting states:")
for s in sorted(graph.starting_nodes()):
   print(s)
    for arc in graph.outgoing_arcs(s):
        print (" " + str(arc))
node = (1,1,5)
print("\nIs {} goal?".format(node),
graph.is_goal(node))
print("Outgoing arcs (available actions) at
{}:".format(node))
for arc in graph.outgoing_arcs(node):
    print (" " + str(arc))
node = (1,7,2)
print("\nIs {} goal?".format(node),
graph.is_goal(node))
print("Outgoing arcs (available actions) at
{}:".format(node))
for arc in graph.outgoing_arcs(node):
    print (" " + str(arc))
node = (3, 7, 0)
print("\nIs {} goal?".format(node),
graph.is_goal(node))
node = (3, 7, math.inf)
print("\nIs {} goal?".format(node),
graph.is_goal(node))
node = (3,6,5)
print("\nIs {} goal?".format(node),
graph.is_goal(node))
print("Outgoing arcs (available actions) at
{}:".format(node))
for arc in graph.outgoing_arcs(node):
    print (" " + str(arc))
node = (3, 6, 9)
print("\nIs {} goal?".format(node),
graph.is_goal(node))
print("Outgoing arcs (available actions) at
{}:".format(node))
for arc in graph.outgoing_arcs(node):
    print (" " + str(arc))
```

```
Result
Starting nodes: [(1, 3, 9), (3, 2, inf), (3, 5,
Outgoing arcs (available actions) at starting
states:
(1, 3, 9)
 Arc(tail=(1, 3, 9), head=(1, 4, 8), action='E',
 Arc(tail=(1, 3, 9), head=(1, 2, 8), action='W',
cost=5)
(3, 2, inf)
 Arc(tail=(3, 2, inf), head=(2, 2, inf),
action='N', cost=5)
 Arc(tail=(3, 2, inf), head=(3, 3, inf),
action='E', cost=5)
 Arc(tail=(3, 2, inf), head=(3, 1, inf),
action='W', cost=5)
(3, 5, 0)
Is (1, 1, 5) goal? False
Outgoing arcs (available actions) at (1, 1, 5):
 Arc(tail=(1, 1, 5), head=(1, 2, 4), action='E',
cost=5)
Is (1, 7, 2) goal? True
Outgoing arcs (available actions) at (1, 7, 2):
 Arc(tail=(1, 7, 2), head=(2, 7, 1), action='S',
cost=5)
Is (3, 7, 0) goal? True
Is (3, 7, inf) goal? True
Is (3, 6, 5) goal? False
Outgoing arcs (available actions) at (3, 6, 5):
 Arc(tail=(3, 6, 5), head=(2, 6, 4), action='N',
cost=5)
 Arc(tail=(3, 6, 5), head=(3, 7, 4), action='E',
cost=5)
 Arc(tail=(3, 6, 5), head=(3, 5, 4), action='W',
cost=5)
 Arc(tail=(3, 6, 5), head=(3, 6, 9), action='Fuel
up', cost=15)
Is (3, 6, 9) goal? False
Outgoing arcs (available actions) at (3, 6, 9):
 Arc(tail=(3, 6, 9), head=(2, 6, 8), action='N',
cost=5)
 Arc(tail=(3, 6, 9), head=(3, 7, 8), action='E',
 Arc(tail=(3, 6, 9), head=(3, 5, 8), action='W',
cost=5)
```

```
Test
                                                   Result
from student_answer import RoutingGraph
                                                   Starting nodes: [(1, 2, inf)]
                                                   Outgoing arcs (available actions) at the start:
map_str = """\
                                                     Arc(tail=(1, 2, inf), head=(1, 1, inf),
+--+
                                                   action='W', cost=5)
|GS|
+--+
                                                   Is (1, 1, 1) goal? True
11 11 11
                                                   Outgoing arcs (available actions) at (1, 1, 1):
                                                     Arc(tail=(1, 1, 1), head=(1, 2, 0), action='E',
graph = RoutingGraph(map_str)
                                                   cost=5)
print("Starting nodes:",
sorted(graph.starting_nodes()))
print("Outgoing arcs (available actions) at the
start:")
for start in graph.starting_nodes():
   for arc in graph.outgoing_arcs(start):
        print (" " + str(arc))
node = (1,1,1)
print("\nIs {} goal?".format(node),
graph.is_goal(node))
print("Outgoing arcs (available actions) at
{}:".format(node))
for arc in graph.outgoing_arcs(node):
   print (" " + str(arc))
from student_answer import RoutingGraph
                                                   Starting nodes: [(1, 1, inf), (1, 6, inf), (3, 1,
                                                   inf)]
map_str = """\
+---+
|S S|
| GXXX|
|S
+---+
11 11 11
graph = RoutingGraph(map_str)
print("Starting nodes:",
sorted(graph.starting_nodes()))
```

Answer: (penalty regime: 0, 15, ... %)

```
1
2
    import math
3
    from search import *
4
5 ▼
    class RoutingGraph(Graph):
6
        """This is a concrete subclass of Graph where vertices and edges
7
         are explicitly enumerated. Objects of this type are useful for
8
         testing graph algorithms."""
9
        def __init__(self, map_str):
10 ▼
             """Initialises an Routing graph."""
11
12
13
            self.goal_nodes = []
14
            self.start_nodes = []
15
            self.fuel_points = []
16
            self.obstacle = []
            self.estimated_cost_to_goal = 0
17
18
            map_str.strip()
19
            map_array = map_str.split('\n')
20
            #print(map_array)
21
            for row in range(len(map_array)):
22 ▼
                for col in range(len(map_array[row])):
23 🔻
24
                     #blank space so no need save anything
                     if (map_array[row][col] == ' '):
25 🔻
26
27
                    #its an obstacle so cannot traverse this direction
                    elif (map_array[row][col] == '+'):
28 ▼
                         self.obstacle.append((row, col))
29
                    #its an obstacle so cannot traverse this direction
30
                    elif (map_array[row][col] == 'X'):
31 ▼
                         self.obstacle.append((row, col))
32
33
                    #its an obstacle so cannot traverse this direction
                    elif (map_array[row][col] == '-'):
34
35
                         self.obstacle.append((row, col))
```

```
36
                      #its an obstacle so cannot traverse this direction
 37 ▼
                      elif (map_array[row][col] == '|'):
 38
                          self.obstacle.append((row, col))
                      #its a goal node so append to goal node set
 39
 40 ▼
                      elif (map_array[row][col] == 'G'):
 41
                          self.goal_nodes.append((row, col))
 42
                      #its a solar powered agent so append it with infinite cost
                      elif (map_array[row][col] == 'S'):
 43
                          self.start nodes.append((row, col, math.inf))
 44
                      #its a fuel point does not need cost just +15 if choose to top up
 45
 46 ▼
                      elif (map_array[row][col] == 'F'):
                          self.fuel_points.append((row, col))
 47
 48
                      #only other case is that its from 0-9
 49
                          self.start_nodes.append((row, col, int(map_array[row][col])))
 50
 51
             #print("obstacles are \n")
 52
             #print(self.obstacle)
 53
             #print("STARTING NODES ARE are \n")
             #print(self.start_nodes)
 54
 55
 56
 57
         #def estimated_cost_to_goal(self,
 58
 59
 60 ▼
         def starting_nodes(self):
              """Returns a sequence of starting nodes."""
 61
 62
             return self.start_nodes
 63
 64 ▼
         def is_goal(self, node):
              """Returns true if the given node is a goal node."""
 65
 66
             #print("IS GOAL?")
             node_x, node_y, _ = node
 67
             return (node_x, node_y) in self.goal_nodes
 68
 69
 70 ▼
         def estimated_cost_to_goal(self):
 71
             return 0
 72
         def outgoing arcs(self, node):
 73 •
              """Returns a sequence of Arc objects that go out from the given
 74
 75
             node. The action string is automatically generated.
 76
             0.000
 77
 78
             arcs = []
 79
             movement_grid = [('N', -1, 0),
 80
                               ('E', 0, 1),
                               ('S', 1, 0),
 81
                               ('W', 0, -1),]
 82
 83
             #for every edge in list of edges
 84 •
             for directional_movement in movement_grid:
 85
                  #get all possible movements
                  #eleminate those arcs which move into obstacles
 86
                  #append allowed movement arcs
 87
                 #print("Start node is:")
 88
                  #print(node)
 89
                  for i in range(len(movement_grid)):
 90
                      direction, horizontal, vertical = movement_grid[i]
 91
                      new_node_location = (node[0] + horizontal, node[1]+ vertical, node[2] - 1)
 92
 93
                      #if fuel is below 1 cant do action, cancel this iteration
                      if (new node location[2] < 0):</pre>
 94 ▼
 95
                          continue
                      #extract x and y coordinates
 96
                      new_node_x, new_node_y, _ = new_node_location
 97
                      #if i am crashing into obstacle, cancel this iteration
 98
                      if (new_node_x, new_node_y) in self.obstacle:
 99
100
                          continue
101
                          #print("direction, horizontal, vertical is\n")
102
                          #print(direction, horizontal, vertical)
103
                          arcs.append(Arc(node, new_node_location, str(direction), 5))
104
105
                 if ((node[0], node[1]) in self.fuel_points):
106
107
                      #i can only stay and top up fuel if i am not on a full tank
108 •
                      if (node[2] < 9):</pre>
                          arcs.append(Arc(node, (node[0], node[1], 9), "Fuel up", 15))
109
                          #print(arcs)
110
111
                  return arcs
```

Test		Expected	Got	
from student_answer	import	Starting nodes: [(1, 3, 9),	Starting nodes: [(1, 3, 9),	~
RoutingGraph		(3, 2, inf), (3, 5, 0)]	(3, 2, inf), (3, 5, 0)]	
import math		Outgoing arcs (available	Outgoing arcs (available	
		actions) at starting	actions) at starting	
map_str = """\		states:	states:	
++		(1, 3, 9)	(1, 3, 9)	
9 XG		Arc(tail=(1, 3, 9), head=	Arc(tail=(1, 3, 9), head=	
X XXX		(1, 4, 8), action='E',	(1, 4, 8), action='E',	
S OFG		cost=5)	cost=5)	
++		Arc(tail=(1, 3, 9), head=	Arc(tail=(1, 3, 9), head=	
"""		(1, 2, 8), action='W',	(1, 2, 8), action='W',	
		cost=5)	cost=5)	
graph = RoutingGrap	oh(map_str)	(3, 2, inf)	(3, 2, inf)	
		Arc(tail=(3, 2, inf),	Arc(tail=(3, 2, inf),	
print("Starting noo	· ·	head=(2, 2, inf),	head=(2, 2, inf),	
sorted(graph.start		action='N', cost=5)	action='N', cost=5)	
print("Outgoing ard	-	Arc(tail=(3, 2, inf),	Arc(tail=(3, 2, inf),	
actions) at starting	ng states:")	head=(3, 3, inf),	head=(3, 3, inf),	
for s in		action='E', cost=5)	action='E', cost=5)	
sorted(graph.start)	ing_nodes()):	Arc(tail=(3, 2, inf),	Arc(tail=(3, 2, inf),	
print(s)		head=(3, 1, inf),	head=(3, 1, inf),	
for arc in	() (action='W', cost=5)	action='W', cost=5)	
graph.outgoing_arcs		(3, 5, 0)	(3, 5, 0)	
print (" '	' + str(arc))	To (1 1 E) cools Falsa	To (1 1 E) 20010 501-	
nodo = (4 4 5)		Is (1, 1, 5) goal? False	Is (1, 1, 5) goal? False	
node = (1,1,5)		Outgoing arcs (available	Outgoing arcs (available	
<pre>print("\nIs {} goal2" format(node)</pre>	,	actions) at (1, 1, 5):	actions) at (1, 1, 5):	
<pre>goal?".format(node) graph.is_goal(node)</pre>	-	Arc(tail=(1, 1, 5), head= (1, 2, 4), action='E',	Arc(tail=(1, 1, 5), head= (1, 2, 4), action='E',	
print("Outgoing are	-		cost=5)	
actions) at {}:".fo	,	COST-3)	(031-3)	
for arc in	or mac(node))	Is (1, 7, 2) goal? True	Is (1, 7, 2) goal? True	
graph.outgoing_arcs	s(node):	Outgoing arcs (available	Outgoing arcs (available	
print (" " + s		actions) at (1, 7, 2):	actions) at (1, 7, 2):	
print (ser (a. 6))	Arc(tail=(1, 7, 2), head=	Arc(tail=(1, 7, 2), head=	
node = $(1,7,2)$		(2, 7, 1), action='S',	(2, 7, 1), action='S',	
<pre>print("\nIs {}</pre>		cost=5)	cost=5)	
goal?".format(node)),	,	,	
graph.is_goal(node)	•	Is (3, 7, 0) goal? True	Is (3, 7, 0) goal? True	
print("Outgoing are	-			
actions) at {}:".fo	•	Is (3, 7, inf) goal? True	Is (3, 7, inf) goal? True	
for arc in	. ,,	, , , , , , , , , , , , , , , , , , ,		
graph.outgoing_arcs	s(node):	Is (3, 6, 5) goal? False	Is (3, 6, 5) goal? False	
print (" " + s	str(arc))	Outgoing arcs (available	Outgoing arcs (available	
		actions) at (3, 6, 5):	actions) at (3, 6, 5):	
node = (3, 7, 0)		Arc(tail=(3, 6, 5), head=	Arc(tail=(3, 6, 5), head=	
<pre>print("\nIs {}</pre>		(2, 6, 4), action='N',	(2, 6, 4), action='N',	
goal?".format(node)	· ·	cost=5)	cost=5)	
graph.is_goal(node)))	Arc(tail=(3, 6, 5), head=	Arc(tail=(3, 6, 5), head=	
		(3, 7, 4), action='E',	(3, 7, 4), action='E',	
node = $(3, 7, math)$.inf)	cost=5)	cost=5)	
<pre>print("\nIs {}</pre>		Arc(tail=(3, 6, 5), head=	Arc(tail=(3, 6, 5), head=	
<pre>goal?".format(node)</pre>	•	(3, 5, 4), action='W',	(3, 5, 4), action='W',	
<pre>graph.is_goal(node)</pre>))	cost=5)	cost=5)	
		Arc(tail=(3, 6, 5), head=	Arc(tail=(3, 6, 5), head=	
node = $(3,6,5)$		(3, 6, 9), action='Fuel	(3, 6, 9), action='Fuel	
<pre>print("\nIs {}</pre>	_	up', cost=15)	up', cost=15)	
goal?".format(node)	· ·	T- (0 0 0)	7- (0 0 0)	
graph.is_goal(node)	•	Is (3, 6, 9) goal? False	Is (3, 6, 9) goal? False	
		Outgoing arcs (available	Outgoing arcs (available	
actions) at {}:".fo	יוומד(node))	actions) at (3, 6, 9):	actions) at (3, 6, 9):	
for arc in	c(node):	Arc(tail=(3, 6, 9), head=	Arc(tail=(3, 6, 9), head=	
graph.outgoing_arcs	, ,	(2, 6, 8), action='N',	(2, 6, 8), action='N',	
print (" " + s	oci (ai C))	COST=5)	Cost=5)	
node = (3,6,9)		Arc(tail=(3, 6, 9), head= (3, 7, 8), action='E',	Arc(tail=(3, 6, 9), head= (3, 7, 8), action='E',	
print("\nIs {}		(3, 7, 8), action= E, cost=5)	cost=5)	
goal?".format(node)).	Arc(tail=(3, 6, 9), head=	Arc(tail=(3, 6, 9), head=	
graph.is_goal(node)	•	(3, 5, 8), action='W',	(3, 5, 8), action='W',	
print("Outgoing are	•	(3, 5, 8), action w ,	cost=5)	
actions) at {}:".fo	`	335-37	0000-07	
for arc in	Ji mac (110de))			
	c(node):			
I graph outgoing area				
graph.outgoing_arcs print (" " + s	•			

	Test	Expected Got		
_	from student_answer import	Starting nodes: [(1, 2,	Starting nodes: [(1, 2,	•
	<pre>RoutingGraph map_str = """\ ++ GS ++ """ graph = RoutingGraph(map_str) print("Starting nodes:", sorted(graph.starting_nodes()))</pre>	<pre>inf)] Outgoing arcs (available actions) at the start: Arc(tail=(1, 2, inf), head=(1, 1, inf), action='W', cost=5) Is (1, 1, 1) goal? True Outgoing arcs (available actions) at (1, 1, 1): Arc(tail=(1, 1, 1), head= (1, 2, 0), action='E',</pre>	inf)] Outgoing arcs (available actions) at the start: Arc(tail=(1, 2, inf), head=(1, 1, inf), action='W', cost=5) Is (1, 1, 1) goal? True Outgoing arcs (available actions) at (1, 1, 1): Arc(tail=(1, 1, 1), head=(1, 2, 0), action='E',	
	<pre>print("Outgoing arcs (available actions) at the start:") for start in graph.starting_nodes(): for arc in graph.outgoing_arcs(start): print (" " + str(arc))</pre>	cost=5)	cost=5)	
	<pre>node = (1,1,1) print("\nIs {} goal?".format(node), graph.is_goal(node)) print("Outgoing arcs (available actions) at {}:".format(node)) for arc in graph.outgoing_arcs(node): print (" " + str(arc))</pre>			
~	<pre>from student_answer import RoutingGraph map_str = """\ ++ X</pre>	Starting nodes: [(2, 2, inf)] Available actions at the start:	Starting nodes: [(2, 2, inf)] Available actions at the start:	*
	<pre>graph = RoutingGraph(map_str) print("Starting nodes:", sorted(graph.starting_nodes())) print("Available actions at the</pre>			
•	<pre>from student_answer import RoutingGraph map_str = """\ ++ S S GXXX S ++ """ graph = RoutingGraph(map_str) print("Starting nodes:", sorted(graph.starting_nodes()))</pre>	Starting nodes: [(1, 1, inf), (1, 6, inf), (3, 1, inf)]	Starting nodes: [(1, 1, inf), (1, 6, inf), (3, 1, inf)]	~

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **2**Correct
Mark 1.00 out of 1.00

Writing the AStarFrontier class

Write a class AStarFrontier for performing A* search on graphs. An instance of AStarFrontier together with an instance of RoutingGraph that you wrote in the previous question will be passed to the generic search procedure in order to find the lowest cost solution (if one exists) from one of the agents to the goal node.

In this question, the heuristic value returned by a graph object for any node (state) must be zero.

Notes:

- 1. Your solution must contain the definitions of both AStarFrontier and RoutingGraph classes.
- 2. Unlike other frontier objects, the AStarFrontier objects are initialised with an instance of a graph. This is because AStarFrontier needs to access the estimated_cost_to_goal method of the graph object.
- 3. Remember that in this course priority queues must be <u>stable</u>. See priority queue implementation notes in <u>heapq</u> <u>documentation</u> for a suggestion on how this can be achieved.
- 4. The algorithm must halt on all valid maps even when there is no solution.
- 5. It is recommended that before you submit your solution, you test it against the given test cases and some new examples (maps) designed by yourself with different positioning of agents and building blocks. You should think whether or not the frontier requires pruning and implement it accordingly.
- 6. Note the distinction between time and fuel consumption. We are interested in a solution with the shortest time.
- 7. The requirement of having a zero heuristic implies that the generic graph search algorithm will behave as an LCFS (lowest-cost-first search). In the next question, we will more thoroughly test your A* frontier and graph class.
- 8. When there is no solution, the generic graph search automatically returns None instead of a path which causes the print_actions procedure to print "There is no solution." This happens when the frontier becomes empty and no solution has been reached.
- 9. It is recommended (but not required) that your answer for AStarFrontier is shorter than 40 lines of code. If your code is much longer, you might be doing something wrong.

For example:

Test	Result
<pre>from student_answer import RoutingGraph, AStarFrontier from search import * map_str = """\</pre>	Actions: N, N. Total cost: 10
G	Total Cost: 10
<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	
<pre>from student_answer import RoutingGraph, AStarFrontier from search import *</pre>	Actions: N, N.
map_str = """\ ++ GG S G S ++	Total cost: 10
<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	

Test	Result
<pre>from student_answer import RoutingGraph, AStarFrontier from search import *</pre>	Actions: E, E,
map_str = """\ ++	E, E,
XG	N, E, N.
++ 	Total cost: 35
<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	
<pre>from student_answer import RoutingGraph, AStarFrontier from search import *</pre>	Actions: N,
map_str = """\	N, E,
++ F X	Fuel up, W,
[X XXXXG]	S,
3	S, E,
ппп	Ε,
<pre>map_graph = RoutingGraph(map_str)</pre>	E, E,
<pre>frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None)</pre>	E, N.
<pre>print_actions(solution)</pre>	Total cost: 75
<pre>from student_answer import RoutingGraph, AStarFrontier from search import *</pre>	Actions: W.
map_str = """\ ++	Total cost: 5
GS ++	
<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	
<pre>from student_answer import RoutingGraph, AStarFrontier from search import *</pre>	Actions: W,
map_str = """\	W. Total cost: 10
GF2 +	
<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	
<pre>from student_answer import RoutingGraph, AStarFrontier from search import *</pre>	Actions: W,
map_str = """\ ++	S. Total cost: 10
S SX GX G	
<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	

Test	Result
<pre>from student_answer import RoutingGraph, AStarFrontier from search import *</pre>	There is no solution!
map_str = """\ ++	
<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	

Answer: (penalty regime: 0, 15, ... %)

```
1
2
    import math
3
    from search import *
4
    import heapq
5
    import numpy as np
6
7 ▼
    class AStarFrontier(Frontier):
        """A* Frontier class but all heuristics node value is zero
8
9
10
11 ▼
        def __init__(self, graph_map):
12
            self.graph_map = graph_map
13
            self.pruned = []
            self.heap = []
14
15
            self.entry = 0
16
17
18 ▼
        def add(self, path):
            """Adds a new path to the frontier. A path is a sequence (tuple) of
19
20
            Arc objects.
21
22
            #if it is a starting node, it should have no cost
            if path[-1].tail == None:
23 ▼
                totalCost = 0 + 0
24
25
            else:
26 ▼
27
                #totalCost = 0
                #print("path is")
28
29
                #print(path)
                #for arc in path:
30 ▼
31
                    #totalCost += arc.cost
                totalCost = sum(arc.cost for arc in path)
32
33
            if path[-1].head not in self.pruned:
34 ▼
                #print(path[-1].head)
35
36
                #print("path[-1].head here")
                self.entry += 1
37
38
                heapq.heappush(self.heap,(totalCost, ) + (self.entry, ) + path)
                #print(self.heap)
39
40
                #print("heap here")
            \#print("path to be appended is\n")
41
42
            #print(path)
43
            #print("\n")
44
            #self.heap.append(path)
45
46
        def __iter__(self):
47 ▼
             """We don't need a separate iterator object. Just return self. You
48
49
            don't need to change this method."""
            return self
50
51
52
        def __next__(self):
53 ▼
54
            """Selects, removes, and returns a path on the frontier if there is
55
56
            any.Recall that a path is a sequence (tuple) of Arc
            objects. Override this method to achieve a desired search
57
58
            strategy. If there nothing to return this should raise a
59
            StopIteration exception.
60
61
62 ▼
            while len(self.heap) > 0:
63
                nath = heard.hearnon(self.hear)
```

```
64
                 if path[-1].head not in self.pruned:
 65
                      self.pruned.append(path[-1].head)
                      return path[2:]
 66
             raise StopIteration
 67
 68
 69
 70
 71 ▼
     def print_actions(path):
          """Given a path (a sequence of Arc objects), prints the actions that
 72
 73
         need to be taken and the total cost of those actions. The path is
 74
         usually a solution (a path from the starting node to a goal
 75
         node."""
 76
         if path:
 77
 78
             print("Actions:")
 79
             print(",\n".join(" {}".format(arc.action) for arc in path[1:]) + ".")
             print("Total cost:", sum(arc.cost for arc in path))
 80
 81 •
             print("There is no solution!")
 82
 83
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99 ▼
     class RoutingGraph(Graph):
         """This is a concrete subclass of Graph where vertices and edges
100
          are explicitly enumerated. Objects of this type are useful for
101
          testing graph algorithms."""
102
103
104
         def __init__(self, map_str):
             """Initialises an Routing graph."""
105
106
107
             self.goal_nodes = []
108
             self.start_nodes = []
             self.fuel_points = []
109
110
             self.obstacle = []
111
             self.estimated_cost_to_goal = 0
112
113
             map_str.strip()
114
             map_array = map_str.split('\n')
115
             #print(map_array)
116 ▼
             for row in range(len(map_array)):
117 ▼
                  for col in range(len(map_array[row])):
                      #blank space so no need save anything
118
                      if (map_array[row][col] == ' '):
119 ▼
120
                          pass
                      #its an obstacle so cannot traverse this direction
121
                      elif (map_array[row][col] == '+'):
122 ▼
                          self.obstacle.append((row, col))
123
124
                      #its an obstacle so cannot traverse this direction
125 ▼
                      elif (map_array[row][col] == 'X'):
                          self.obstacle.append((row, col))
126
127
                      #its an obstacle so cannot traverse this direction
128 •
                      elif (map_array[row][col] == '-'):
                          self.obstacle.append((row, col))
129
                      #its an obstacle so cannot traverse this direction
130
                      elif (map_array[row][col] == '|'):
131
                          self.obstacle.append((row, col))
132
133
                      #its a goal node so append to goal node set
134 ▼
                      elif (map_array[row][col] == 'G'):
135
                          self.goal_nodes.append((row, col))
136
                      #its a solar powered agent so append it with infinite cost
137 ▼
                      elif (map_array[row][col] == 'S'):
                          self.start_nodes.append((row, col, math.inf))
138
                      #its a fuel point does not need cost just +15 if choose to top up
139
                      elif (map_array[row][col] == 'F'):
140 ▼
                          self.fuel points.append((row, col))
141
                      #only other case is that its from 0-9
142
143 v
                      else:
                          self.start_nodes.append((row, col, int(map_array[row][col])))
144
             #print("obstacles are \n")
145
             #print(self.obstacle)
146
             #print("STARTING NODES ARE are \n")
147
148
             #nrint(self.start nodes)
```

```
A1. A* Routing: Attempt review
149
150
151
         #def estimated_cost_to_goal(self,
152
153
154 ▼
         def starting_nodes(self):
155
              """Returns a sequence of starting nodes."""
              return self.start_nodes
156
157
158 ▼
         def is_goal(self, node):
              """Returns true if the given node is a goal node."""
159
160
              #print("IS GOAL?")
161
              node_x, node_y, _ = node
162
              return (node_x, node_y) in self.goal_nodes
163
         def estimated_cost_to_goal(self):
164 ▼
              return 0
165
166
         def outgoing_arcs(self, node):
167 ▼
              """Returns a sequence of Arc objects that go out from the given
168
169
              node. The action string is automatically generated.
170
171
172
              arcs = []
173
              movement_grid = [('N', -1, 0),
174
                               ('E', 0, 1),
                               ('S' , 1, 0),
175
                               ('W', 0, -1),]
176
              #for every edge in list of edges
177
178 ▼
              for directional_movement in movement_grid:
179
                  #get all possible movements
                  #eleminate those arcs which move into obstacles
180
                  #append allowed movement arcs
181
182
                  #print("Start node is:")
183
                  #print(node)
                  for i in range(len(movement_grid)):
184 ▼
                      direction, horizontal, vertical = movement_grid[i]
185
                      new_node_location = (node[0] + horizontal, node[1]+ vertical, node[2] - 1)
186
                      #if fuel is below 1 cant do action, cancel this iteration
187
188 •
                      if (new_node_location[2] < 0):</pre>
189
                          continue
190
                      #extract x and y coordinates
191
                      new_node_x, new_node_y, _ = new_node_location
192
                      #if i am crashing into obstacle, cancel this iteration
193 ▼
                      if (new_node_x, new_node_y) in self.obstacle:
                          continue
194
195 ▼
                      else:
                          #print("direction, horizontal, vertical is\n")
196
                          #print(direction, horizontal, vertical)
197
198
                          arcs.append(Arc(node, new_node_location, str(direction), 5))
199
200 ▼
                  if ((node[0], node[1]) in self.fuel_points):
201
                      #i can only stay and top up fuel if i am not on a full tank
202 ▼
                      if (node[2] < 9):</pre>
                          arcs.append(Arc(node, (node[0], node[1], 9), "Fuel up", 15))
203
204
                          #print(arcs)
205
                  return arcs
```

Test	Expected	Got	
	-		 ı

	Test	Expected	Got	
~	<pre>from student_answer import RoutingGraph, AStarFrontier from search import *</pre>	Actions: N, N.	Actions: N, N.	~
	map_str = """\ ++ G S ++	Total cost: 10	Total cost: 10	
	<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>			
~	<pre>from student_answer import RoutingGraph, AStarFrontier from search import * map_str = """\ ++</pre>	Actions: N, N. Total cost: 10	Actions: N, N. Total cost: 10	•
	GG			
	<pre>frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	Astions	Antique	
•	<pre>from student_answer import RoutingGraph, AStarFrontier from search import * map_str = """\ ++ XG X XXX S ++ """</pre>	Actions: E, E, E, N, E, N. Total cost: 35	Actions: E, E, E, N, E, N. Total cost: 35	•
	<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>			
~	<pre>from student_answer import RoutingGraph, AStarFrontier from search import * map_str = """\ ++ F X X XXXXG 3 </pre>	Actions: N, N, E, Fuel up, W, S, S, E,	Actions: N, N, E, Fuel up, W, S, S,	*
	<pre>#+ """ map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	E, E, E, E, N. Total cost: 75	E, E, E, N. Total cost: 75	

	Test	Expected	Got	
~	<pre>from student_answer import RoutingGraph, AStarFrontier from search import * map_str = """\ ++ GS ++ """ map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	Actions: W. Total cost: 5	Actions: W. Total cost: 5	•
•	<pre>from student_answer import RoutingGraph, AStarFrontier from search import * map_str = """\ ++ GF2 ++ """ map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_actions(solution)</pre>	Actions: W, W. Total cost: 10	Actions: W, W. Total cost: 10	*
	<pre>from student_answer import RoutingGraph, AStarFrontier from search import * map_str = """\ ++ S</pre>	Actions: W, S. Total cost: 10	Actions: W, S. Total cost: 10	•
~	<pre>from student_answer import RoutingGraph, AStarFrontier from search import * map_str = """\ ++ </pre>	There is no solution!	There is no solution!	*

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Question **3**Correct
Mark 1.00 out of 1.00

The print_map procedure

Write a procedure print_map that takes three parameters: an instance of RoutingGraph, a frontier object, and a solution (which is a sequence of Arc objects that make up a path from a starting position to the goal position) and then prints a map such that:

- the position of the walls, obstacles, agents, and the goal points are all unchanged and they are marked by the same set of characters as in the original map string; and
- those free spaces (space characters) that have been expanded during the search are marked with a '.' (a period character); and
- those free spaces (spaces characters) that are part of the solution (best path to the goal) are marked with '*' (an asterisk character).

Further assumptions and requirements

- 1. For this question, the graph class must have a proper heuristic function named estimated_cost_to_goal. You have to design the most dominant (highest value) function that can be computed very efficiently. See the signature of the method in the Graph class in search.py.
- 2. In this question, we are only concerned with agents of type s—agents that have infinite amount of fuel and do not require to fuel up. The test cases do <u>not</u> include any fuel-based agents or fuel stations. Do not consider anything fuel-related when devising the heuristic function.
- 3. Only the first solution returned by the generic search procedure (if there is one) is used to test your procedure.
- 4. In addition to print_map, your solution must include the code for RoutingGraph and AStarFrontier.

Notes

- 1. A node is said to have been *expanded* if a path leading to that node is removed from the frontier and then if the node has neighbours, corresponding extended paths are added to the frontier.
- 2. This question puts your code for the graph and A* frontier into real test. Previous questions did not test the heuristic function and as long as the frontier class could provide the functionality of the LCFS, it would pass the test cases. In this question, however, your code needs to produce the correct A* behaviour. Therefore, even if your code has passed previous tests, you may still need to modify it in order to meet the required spec in this question.
- 3. Note that you only need to consider movement actions when designing the heuristic function and that all movement actions have the same cost.
- 4. If your algorithm expands more nodes than the expected output, you might be using a heuristic that is not good enough; you need to find a better heuristic. Refer to the specification of the heuristic function stated above.
- 5. If for any reason you decide to answer this question before Question 2, please remember that in Question 2 the heuristic function is required to always return zero.
- 6. The generic graph search algorithm returns None when no solution if found.
- 7. It is recommended (but not required) that your answer for print_map is shorter than 20 lines of code. If your code is much longer, you might be doing something wrong.

For example:

Test	Result
<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import *</pre>	+
<pre>map_str = """\ ++ </pre>	
<pre>print_map(map_graph, frontier, solution)</pre>	

Total	Dani II
Test	Result
<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import *</pre>	
map_str = """\	
++	
i i	*
	* G***
ļ! <u>!</u>	
i	
<pre>map_graph = RoutingGraph(map_str) # changing the heuristic so the search behaves like LCFS</pre>	
<pre>map_graph.estimated_cost_to_goal = lambda node: 0</pre>	
<pre>frontier = AStarFrontier(map_graph)</pre>	
<pre>solution = next(generic_search(map_graph, frontier), None) print_map(map_graph, frontier, solution)</pre>	
<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import *</pre>	G****G
map_str = """\	S GG
++	++
G	
G	
++ 	
<pre>map_graph = RoutingGraph(map_str) frontior = AstarErontion(map_graph)</pre>	
<pre>frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None)</pre>	
<pre>print_map(map_graph, frontier, solution)</pre>	
from student_answer import RoutingGraph, AStarFrontier, print_map	++
from search import *	XG
map_str = """\	X XXX** S***.
++	++
XG X XXX	
S	
++ """	
map_graph = RoutingGraph(map_str)	
<pre>frontier = AStarFrontier(map_graph)</pre>	
<pre>solution = next(generic_search(map_graph, frontier), None) print_map(map_graph, frontier, solution)</pre>	
<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import *</pre>	++ GS ++
map_str = """\ ++	
GS	
++ nnn	
<pre>map_graph = RoutingGraph(map_str)</pre>	
<pre>frontier = AStarFrontier(map_graph)</pre>	
<pre>solution = next(generic_search(map_graph, frontier), None) print_map(map_graph, frontier, solution)</pre>	
, , , , , , , , , , , , , , , , , , , ,	

A1. A Routing, Attempt review	T
Test	Result
<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import * map_str = """\ ++ </pre>	++ *** .SX* .X G ++
<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_map(map_graph, frontier, solution)</pre>	
<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import * map_str = """\ ++ G</pre>	++ G****** XXXXXXXXXXXX .*******X* .*XXXXXXXX.X* .*X.S**X*X* .*X**X* .*XXXXXXXXXX .********* +
<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_map(map_graph, frontier, solution)</pre>	
<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import * map_str = """\ ++ </pre>	++
<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_map(map_graph, frontier, solution)</pre>	

Answer: (penalty regime: 0, 15, ... %)

```
import math
    from search import *
3
 4
    import heapq
5
    import numpy as np
    #i tried my best but i had help with the printing map portion...
6
7 ▼ class AStarFrontier(Frontier):
        """A* Frontier class but all heuristics node value is zero
8
9
10
        def __init__(self, graph_map):
11 ▼
            self.graph_map = graph_map
12
            self.pruned = []
13
14
            self.heap = []
            self.entry = 0
15
16
17
18 ▼
        def add(self, path):
            """Adds a new path to the frontier. A path is a sequence (tuple) of
19
            Arc objects.
20
21
            #if it is a starting node, it should have no cost
22
23 ▼
            if path[-1].tail == None:
24
                #print("head is")
```

```
#print(path[-1].head)
 25
                 #print(self.graph_map.estimated_cost_to_goal(path[-1].head) )
 26
 27
                  totalCost = 0 + self.graph_map.estimated_cost_to_goal(path[-1].head)
 28 ▼
             else:
                 #totalCost = 0
 29
 30
                 #print("path is")
 31
                 #print(path)
 32 ▼
                 #for arc in path:
 33
                      #totalCost += arc.cost
                 totalCost = sum(arc.cost for arc in path) + self.graph_map.estimated_cost_to_goal(path[
 34
 35
 36 ▼
             if path[-1].head not in self.pruned:
                 #print(path[-1].head)
 37
 38
                 #print("path[-1].head here")
                 self.entry += 1
 39
 40
                 heapq.heappush(self.heap,(totalCost, ) + (self.entry, ) + path)
                 #print(self.heap)
 41
                 #print("heap here")
 42
             #print("path to be appended is\n")
 43
             #print(path)
 44
             #print("\n")
 45
             #self.heap.append(path)
 46
 47
 48
 49 ▼
         def __iter__(self):
              """We don't need a separate iterator object. Just return self. You
 50
             don't need to change this method."""
 51
             return self
 52
 53
 54
         def __next__(self):
 55 ▼
 56
             """Selects, removes, and returns a path on the frontier if there is
 57
 58
             any.Recall that a path is a sequence (tuple) of Arc
 59
             objects. Override this method to achieve a desired search
             strategy. If there nothing to return this should raise a
 60
 61
             StopIteration exception.
 62
             0.000
 63
             while len(self.heap) > 0:
 64 ▼
 65
                  path = heapq.heappop(self.heap)
                 if path[-1].head not in self.pruned:
 66 ₹
 67
                      self.pruned.append(path[-1].head)
 68
                      return path[2:]
             raise StopIteration
 69
 70
 71
 72
     def print_actions(path):
 73 ▼
 74
         """Given a path (a sequence of Arc objects), prints the actions that
 75
         need to be taken and the total cost of those actions. The path is
 76
         usually a solution (a path from the starting node to a goal
         node."""
 77
 78
 79 ▼
         if path:
 80
             print("Actions:")
             print(",\n".join(" {}".format(arc.action) for arc in path[1:]) + ".")
 81
             print("Total cost:", sum(arc.cost for arc in path))
 82
 83 ▼
             print("There is no solution!")
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99
100
101 ▼
     class RoutingGraph(Graph):
          """This is a concrete subclass of Graph where vertices and edges
102
103
          are explicitly enumerated. Objects of this type are useful for
          testing graph algorithms."""
104
105
         def __init__(self, map_str):
106 ▼
              """Initialises an Routing graph."""
107
108
109
             self.goal nodes = []
```

```
self.start_nodes = []
110
111
             self.fuel_points = []
             self.obstacle = []
112
113
             self.map_array = []
             #self.estimated_cost_to_goal = 0
114
115
116
             map_str.strip()
117
             self.mapy = self.create_map(map_str)
118
             map array = map str.split('\n')
119
             self.map_array = map_array
120
121
             #print(map_array)
122 ▼
             for row in range(len(map_array)):
                 for col in range(len(map_array[row])):
123 ▼
                      #blank space so no need save anything
124
                      if (map_array[row][col] == ' '):
125 v
126
                          pass
                      #its an obstacle so cannot traverse this direction
127
128 ▼
                      elif (map_array[row][col] == '+'):
129
                          self.obstacle.append((row, col))
130
                      #its an obstacle so cannot traverse this direction
131 •
                      elif (map_array[row][col] == 'X'):
                          self.obstacle.append((row, col))
132
133
                      #its an obstacle so cannot traverse this direction
134
                      elif (map_array[row][col] == '-'):
                          self.obstacle.append((row, col))
135
136
                      #its an obstacle so cannot traverse this direction
                      elif (map array[row][col] == '|'):
137 ▼
138
                          self.obstacle.append((row, col))
139
                      #its a goal node so append to goal node set
140 ▼
                      elif (map_array[row][col] == 'G'):
                          self.goal_nodes.append((row, col))
141
142
                      #its a solar powered agent so append it with infinite cost
                      elif (map_array[row][col] == 'S'):
143 •
144
                          self.start_nodes.append((row, col, math.inf))
                      #its a fuel point does not need cost just +15 if choose to top up
145
                      elif (map array[row][col] == 'F'):
146 ▼
147
                          self.fuel_points.append((row, col))
                      #only other case is that its from 0-9
148
149
                      else:
                          self.start_nodes.append((row, col, int(map_array[row][col])))
150
151
             #print("obstacles are \n")
152
             #print(self.obstacle)
153
             #print("STARTING NODES ARE are \n")
154
             #print(self.start_nodes)
155
156
157
         #def estimated_cost_to_goal(self,
158 ▼
         def create_map(self, map_str):
              . . . . . . . . . .
159
160
             map_list = []
161
             for i in map_str.split('\n'):
162
                 i = i.strip()
                 map_list.append(i)
163
164
165
             return map_list
166
167
168 ▼
         def starting_nodes(self):
              """Returns a sequence of starting nodes."""
169
170
             return self.start_nodes
171
172 ▼
         def is goal(self, node):
              """Returns true if the given node is a goal node."""
173
             #print("IS GOAL?")
174
175
             node_x, node_y, _ = node
176
             return (node_x, node_y) in self.goal_nodes
177
         def estimated cost to goal(self):
178 ▼
179
             return 0
180
181
         def outgoing_arcs(self, node):
              """Returns a sequence of Arc objects that go out from the given
182
183
             node. The action string is automatically generated.
184
             ....
185
186
             arcs = []
             movement_grid = [('N', -1, 0),
187
                               ('E', 0, 1),
188
                               ('S' , 1, 0),
189
                               ('W', 0, -1),]
190
             #for every edge in list of edges
191
             for directional movement in movement grid:
192 ▼
                  #get all possible movements
193
194
                  #eleminate those arcs which move into obstacles
```

```
195
                 #append allowed movement arcs
196
                 #print("Start node is:")
                 #print(node)
197
198
                 for i in range(len(movement_grid)):
199
                     direction, horizontal, vertical = movement_grid[i]
200
                      new_node_location = (node[0] + horizontal, node[1]+ vertical, node[2] - 1)
                      #if fuel is below 1 cant do action, cancel this iteration
201
202 ▼
                      if (new_node_location[2] < 0):</pre>
203
                          continue
204
                      #extract x and y coordinates
205
                     new_node_x, new_node_y, _ = new_node_location
                      #if i am crashing into obstacle, cancel this iteration
206
207
                      if (new_node_x, new_node_y) in self.obstacle:
208
                          continue
209
                     else:
210
                          #print("direction, horizontal, vertical is\n")
211
                          #print(direction, horizontal, vertical)
                          arcs.append(Arc(node, new_node_location, str(direction), 5))
212
213
                 if ((node[0], node[1]) in self.fuel_points):
214 ▼
215
                      #i can only stay and top up fuel if i am not on a full tank
216
                     if (node[2] < 9):</pre>
                          arcs.append(Arc(node, (node[0], node[1], 9), "Fuel up", 15))
217
218
                          #print(arcs)
219
                 return arcs
220
221
222
         def estimated_cost_to_goal(self, node):
             """Return the estimated cost to a goal node from the given
223
             state. This function is usually implemented when there is a
224
             single goal state. The function is used as a heuristic in
225
226
             search. The implementation should make sure that the heuristic
227
             meets the required criteria for heuristics."""
228
             #print("node_coord is")
229
             node_coord = (node[0], node[1])
230
             #print(node_coord)
231
232 ▼
             if node_coord in self.goal_nodes:
233
                 return 0
234
             if node_coord is None:
235
                 return 0
236
             heuristic = lambda x1,x2,y1,y2 : abs(x1-x2) + abs(y1-y2) #manhattan distance
237
             pool = []
             nrow = node_coord[0]
238
239
             ncol = node_coord[1]
             #print(self.goal_nodes)
240
241 •
             for row,col in self.goal_nodes:
                  pool.append(heuristic(ncol,col,nrow,row))
242
243
             #print("pool is")
244
             #print(pool)
             return min(pool)*5
245
246
247 ▼
         print_map(map_graph, frontier, solution):
248
         graph_rows = []
249
         agentCoord = []
250
         #converted graph into rows
251
         for row in map_graph.map_array:
             graph_rows.append(row.strip())
252
         #converted startnodes into x, y
253
         for coord in map_graph.start_nodes:
254 ▼
255
             agentCoord.append((coord[0], coord[1]))
256
         #for every node that got visited, should be .
257 ▼
         for node in frontier.pruned:
258
             node = (node[0], node[1]) #shorten from len 3 tuple to len 2
259
             #if the pruned node is not start node and goal node
260 •
             if node not in agentCoord and node not in map_graph.goal_nodes:
261
                 row, col = node[0], node[1]
                 #replacing the item in the list with the .
262
                 graph rows[row] = graph rows[row][:col] + '.' + graph rows[row][col+1:]
263
             if solution is not None:
264 ▼
265 ▼
                 for arcs in solution:
                      node = (arcs.head[0], arcs.head[1])
266
                      if node not in agentCoord and node not in map_graph.goal_nodes:
267 ▼
                          row, col = node[0], node[1]
268
                          graph rows[row] = graph rows[row][:col] + '*' + graph rows[row][col+1:]
269
270
         graphMap = '\n'.join(graph_rows)
271
         print(graphMap)
272
273
```

Test	Expected	Got	
from student_answer import RoutingGraph, AStarFrontier,	+	+	~
print_map	+	+	
from search import *			
map_str = """\ ++			
 			
i i	i	l i	
	1	1	
		1	
S			
	S	S	
	*		
	*		
++	li	li	
нии	G***	G***	
	1		
<pre>map_graph = RoutingGraph(map_str)</pre>			
frontier = AStarFrontier(map_graph)			
<pre>solution = next(generic_search(map_graph, frontier),</pre>			
<pre>None) print_map(map_graph, frontier, solution)</pre>			
print_map(map_graph, rrontier, solution)			
	+	' +	
	+	+	
from student engues import DoutingCraph ASterFronties	+	+	١,
<pre>from student_answer import RoutingGraph, AStarFrontier, print_map</pre>	+	+	
from search import *			
	li ·	l i	
map_str = """\	i	i	
++	1	1	
	1	1	
	1		
i i	li	l į	
G	S	S	
	*	*	
	*	*	
++			
] 6 ' ' ' ' ' ' ' '	
<pre>map_graph = RoutingGraph(map_str)</pre>		li	
# changing the heuristic so the search behaves like	i	¦	
 LCFS			
<pre>map_graph.estimated_cost_to_goal = lambda node: 0</pre>	i .	i	
	Li	1	
	1	'	
frontier = AStarFrontier(map_graph)	+	+	
<pre>frontier = AStarFrontier(map_graph)</pre>	+	+	
	+	+	

	Test	Expected	Got	
~	from student_answer import RoutingGraph, AStarFrontier,		++	~
	<pre>print_map from search import *</pre>	1 '	G****G S	
		GG	G	
	map_str = """\ ++	++	++	
	G			
	i G G i			
	++ 			
	<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_map(map_graph, frontier, solution)</pre>			
_	from student_answer import RoutingGraph, AStarFrontier,	++	++	_
ľ	print_map	XG	XG	ľ
	from search import *	X XXX**	X XXX** S***.	
	map_str = """\ ++	++	++	
	XG			
	X XXX			
	++ 			
	<pre>map_graph = RoutingGraph(map_str)</pre>			
	<pre>frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier),</pre>			
	None) print_map(map_graph, frontier, solution)			
~	from student_answer import RoutingGraph, AStarFrontier,	++	++	~
	<pre>print_map from search import *</pre>	GS ++	GS ++	
	·			
	map_str = """\ ++			
	GS ++			
	11111			
	<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph)</pre>			
	<pre>solution = next(generic_search(map_graph, frontier), None)</pre>			
	<pre>print_map(map_graph, frontier, solution)</pre>			
~	from student_answer import RoutingGraph, AStarFrontier,	++	++	~
	<pre>print_map from search import *</pre>	*** .SX*	*** .SX*	
	map_str = """\	.X G	.X G ++	
			. ,	
	X G			
	ппп			
	map_graph = RoutingGraph(map_str)			
	frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier),			
	None)			
	<pre>print_map(map_graph, frontier, solution)</pre>			

	A1. A* Routing: Attempt review			
	Test	Expected	Got	
*	<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import * map_str = """\ ++ G</pre>	+	 .*XXXXXX*X* .*X.S**X*X*	*
•	<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import * map_str = """\ ++ </pre>	++ X .SX G X X X ++	X	*
~	<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import * map_str = """\ ++ </pre>	++ G G ++	++ G G ++	*

	A1. A* Routing: Attempt review			
	Test	Expected	Got	
	<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import * map_str = """\ ++ </pre>	++ G S * S ++	++ G S * S ++	*
•	<pre>print_map(map_graph, frontier, solution) from student_answer import RoutingGraph, AStarFrontier, print_map from search import * map_str = """\ ++ </pre>	++ **** S.X*** XXXXX* G*X*** .*** ++	++ **** S.X*** XXXXX* G*X*** .*** ++	~
	<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_map(map_graph, frontier, solution)</pre>			
•	<pre>from student_answer import RoutingGraph, AStarFrontier, print_map from search import * map_str = """\ ++ S</pre>	+	+	
	<pre>map_graph = RoutingGraph(map_str) frontier = AStarFrontier(map_graph) solution = next(generic_search(map_graph, frontier), None) print_map(map_graph, frontier, solution)</pre>			

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

◀ 11. Games

Jump to...

A2. Predicting number sequences ▶