

[Dashboard](#) / [My courses](#) / [COSC367-2020S2](#) / [Weekly quizzes](#) / [2. Cost-sensitive search, pruning, and heuristics](#)

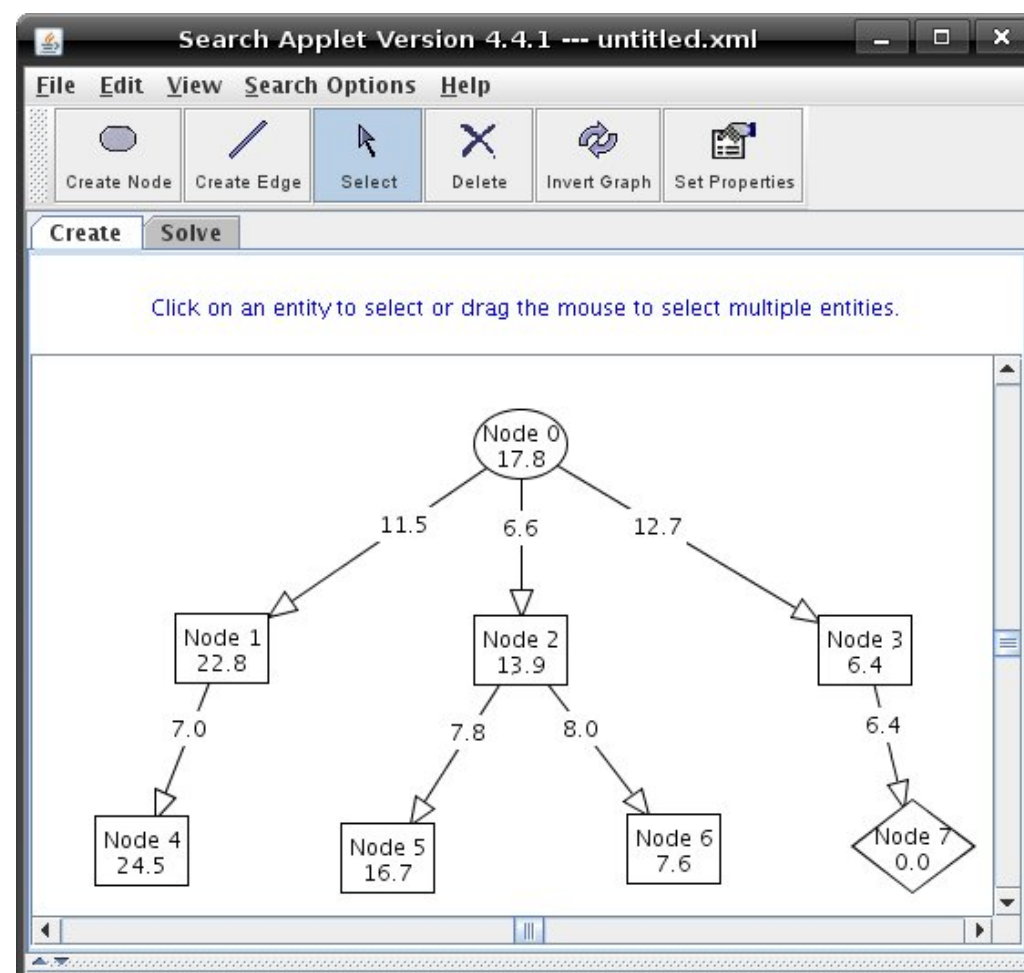
Started on	Thursday, 23 July 2020, 2:57 PM
State	Finished
Completed on	Friday, 31 July 2020, 5:06 PM
Time taken	8 days 2 hours
Marks	21.03/22.00
Grade	95.61 out of 100.00

Optional activity: using the search applet to explore cost-sensitive and heuristic algorithms

The search applet is not directly used in any of the questions in this quiz (or future quizzes) but you may find it useful in understanding cost-based and heuristic algorithms. Remember to study the relevant sections of the textbook and the lecture notes first. The jar file of the applet is available at <http://www.aispace.org/downloads.shtml> and the documentation and tutorials are available at <http://www.aispace.org/search/index.shtml>.

In order to create your own graphs follow the same procedure as that in the previous quiz. Node heuristics are specified when you create new nodes. Arc costs are specified when you create new arcs. If you want to change the node heuristics or arc costs in a graph, go to "Create" mode and depress "Set Properties" then select the node or the arc you want to change. Two viewing options that you may find helpful are "Show Node Heuristics" and "Show Edge Costs" which are available from the "View" menu.

If you would like to set the nodes heuristics to be the distance from the node to the goal node (on the canvas), go to the 'Search Options' menu and click on 'Set Node Heuristics Automatically'. If you would like to set the edge costs to be the distance (on the canvas) from the two nodes the edge is on, go to the 'Search Options' menu and click on 'Set Edge Costs Automatically'. After creating a few nodes and edges, and after enabling the "Show Node Heuristics" or "Show Edge Costs" check boxes under the view menu, your graph might look something like this:



Here are some sample problems (graphs) related to cost-sensitive and heuristic search that can be loaded by selecting "Load Sample Problem" from the "File" menu:

1. **Bicycle Courier Problem (cyclic or acyclic):** Try solving this problem with lowest-cost-first search (LCFS) and A* search with various pruning options. Pruning options can be set through the "Search Options" menu.
2. **Misleading Heuristic Demo:** Try solving this problem using best-first search (with no loop detection or pruning) and observe the behaviour of the algorithm. Observe how loop detection or multiple-path pruning can help best-first search solve the problem. What search algorithms can solve this problem without pruning?
3. **Delivery Robot (cyclic):** First try finding a solution using depth-first search (DFS). Can DFS find a solution? Does cycle-checking (or multiple-path pruning) help solve the problem? How about breadth-first search (BFS)? Does BFS need pruning in order to find an optimal solution? Does pruning (cycle-checking or multiple path pruning) reduce the number of nodes need to be expanded by BFS?
4. **Multiple-Path Pruning Demo:** First try solving this problem without pruning. Does it make any difference if you used lowest-cost-first search (LCFS) or breadth-first search (BFS)? How many nodes are expanded until a solution is found? Can cycle (loop) checking reduce the number of nodes that need to be expanded before a solution is found? How about multiple-path pruning?

Frontier trace format

This information box appears in any quiz that includes questions on tracing the frontier of a graph search problem.

Trace format

- Starting with an empty frontier we record all the calls to the frontier: to add or to get a path. We dedicate one line per call.
- When we ask the frontier to add a path, we start the line with a + followed by the path that is being added.
- When we ask for a path from the frontier, we start the line with a - followed by the path that is being removed.
- When using a priority queue, the path is followed by a comma and then the key (e.g. cost, heuristic, f-value, ...).
- The lines of the trace should match the following regular expression (case and space insensitive): `^[+-][a-z]+(,\d+)?!?$`
- The symbol ! is used at the end of a line to indicate a pruning operation. It must be used when the path that is being added to, or is removed from the frontier is to a node that is already expanded. Note that if a path that is removed from the frontier is pruned, it is not returned to the search algorithm; the frontier has to remove more paths until one can be returned.

Precheck

You can check the format of your answer by clicking the "Precheck" button which checks the format of each line against a regular expression. Precheck only looks at the syntax of your answer and the frontier that must be used. It does not use the information in the graph object. So for example, + a would pass the format check for a BFS or DFS question even if the graph does not have a node called a.

If your answer fails the precheck, it will fail the check. If it passes the precheck, it may pass the main check, it may not. You will not lose or gain any marks by using "Precheck".

Notes

- Frontier traces are not case sensitive - you don't have to type capital letters.
- There can be any number of white space characters between characters in a trace line and in between lines. When evaluating your answer, white spaces will be removed even if they are between characters.
- You can have comments in a trace. Comments start with a hash sign, #. They can take an entire line or just appear at the end of a line. Any character string appearing after # in a line will be ignored by the grader. You can use comments to keep track of things such as the visited set (when pruning).
- In questions where the search strategy is cost-sensitive (e.g. LCFS) you must include the cost information on all lines. The 'Precheck' button checks this for you.
- In questions where a priority queue is needed, the queue must be stable.

Question **1**

Correct


Mark 2.00 out of 2.00

Given the following graph, trace the frontier of a lowest-cost-first search (LCFS).

```
ExplicitGraph(  
  nodes={'A', 'B', 'C', 'D', 'G'},  
  edge_list=[('A', 'B', 2), ('A', 'C', 3), ('B', 'D', 5),  
              ('C', 'D', 5), ('D', 'G', 3)],  
  starting_nodes=['A'],  
  goal_nodes = {'G'},  
)
```

Answer: (penalty regime: 20, 40, ... %)

```
1  +A, 0  
2  -A, 0  
3  +AB, 2  
4  +AC, 3  
5  -AB, 2  
6  +ABD, 7  
7  -AC, 3  
8  +ACD, 8  
9  -ABD, 7  
10 +ABDG, 10  
11 -ACD, 8  
12 +ACDG, 11  
13 -ABDG, 10  
14
```

- +A,0 (OK)
 - A,0 (OK)
 - +AB,2 (OK)
 - +AC,3 (OK)
 - AB,2 (OK)
 - +ABD,7 (OK)
 - AC,3 (OK)
 - +ACD,8 (OK)
 - ABD,7 (OK)
 - +ABDG,10 (OK)
 - ACD,8 (OK)
 - +ACDG,11 (OK)
 - ABDG,10 (OK)
- Passed all tests! 

Correct
Marks for this submission: 2.00/2.00.

Question **2**

Correct

Mark 1.83 out of 2.00

Given the following graph (the same graph as that in the previous question) trace the frontier of a lowest-cost-first search (LCFS) with (multiple-path) pruning.

If you wish, adapt the answer from the previous question to have pruning.

```
ExplicitGraph(  
  nodes={'A', 'B', 'C', 'D', 'G'},  
  edge_list=[('A', 'B', 2), ('A', 'C', 3), ('B', 'D', 5),  
              ('C', 'D', 5), ('D', 'G', 3)],  
  starting_nodes=['A'],  
  goal_nodes = {'G'},  
)
```

Answer: (penalty regime: 20, 40, ... %)

```
1  +A, 0  
2  -A, 0 #expended = {A}  
3  +AB, 2  
4  +AC, 3  
5  -AB, 2 #expended = {A,B}  
6  +ABD, 7  
7  -AC, 3 #expended = {A,B,C}  
8  +ACD, 8  
9  -ABD, 7 #expended = {A,B,C,D}  
10 +ABDG, 10  
11 -ACD, 8! #!not added  
12 -ABDG, 10  
13
```

+A,0 (OK)
-A,0 (OK)
+AB,2 (OK)
+AC,3 (OK)
-AB,2 (OK)
+ABD,7 (OK)
-AC,3 (OK)
+ACD,8 (OK)
-ABD,7 (OK)
+ABDG,10 (OK)
-ACD,8! (OK)
-ABDG,10 (OK)
Passed all tests! ✓

Correct
Marks for this submission: 2.00/2.00. Accounting for previous tries, this gives **1.83/2.00**.

Question **3**

Correct

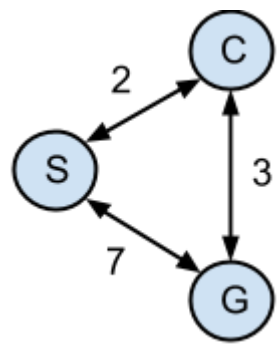
Mark 1.60 out of 2.00

Given the following graph, trace the frontier of a lowest-cost-first search (LCFS) without pruning.

Assume an alphabetic ordering over nodes such that if for example, a path leading to "S" is being expanded and "A" and "B" are neighbours of "S", then the edge (S,A) comes before (S, B); in other words, (S,A) is added to the frontier before (S,B).

Note that all the edges in this graph are bidirectional (i.e. they represent two directed edges with the same cost).

The starting state is 'S' and the goal state is 'G'.



Answer: (penalty regime: 20, 40, ... %)

1

+S, 0

2

-S, 0

3

+SC, 2

4

+SG, 7

5

-SC, 2

6

+SCG, 5

7

+SCS, 4

8

-SCS, 4

9

+SCSC, 6

10

+SCSG, 11

11

-SCG, 5

- +S,0 (OK)
- S,0 (OK)
- +SC,2 (OK)
- +SG,7 (OK)
- SC,2 (OK)
- +SCG,5 (OK)
- +SCS,4 (OK)
- SCS,4 (OK)
- +SCSC,6 (OK)
- +SCSG,11 (OK)
- SCG,5 (OK)

Passed all tests! ✓

Correct

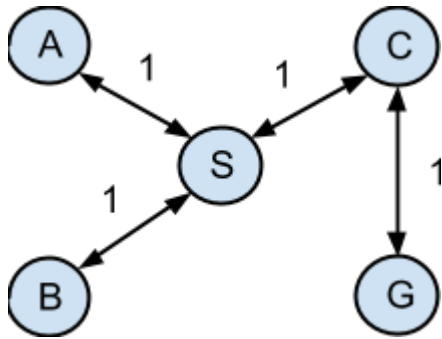
Marks for this submission: 2.00/2.00. Accounting for previous tries, this gives **1.60/2.00**.

Question **4**
Correct
Mark 1.60 out of 2.00

Given the following graph, trace the frontier of a lowest-cost-first search (LCFS) with multiple-path pruning. The starting state is 'S' and the goal state is 'G'.

Assume an alphabetic ordering over nodes such that if, for example, a path leading to "S" is being expanded and "A" and "B" are neighbours of "S", then the edge (S,A) comes before (S, B); in other words, (S,A) is added to the frontier before (S,B).

Note that all the edges in this graph are bidirectional (i.e. each one represents two directed edges).



Answer: (penalty regime: 20, 40, ... %)

```
1  +S, 0
2  -S, 0 #expended = {S}
3  +SA, 1
4  +SB, 1
5  +SC, 1
6  -SA, 1 #expended = {S, A}
7  +SAS, 2!
8  -SB, 1 #expended = {S, A, B}
9  +SBS, 2!
10 -SC, 1 #expended = {S, A, B, C}
11 +SCG, 2
12 +SCS, 2!
13 -SCG, 2 #expended = {S, A, B, C, G}
14
```

- +S,0 (OK)
- S,0 (OK)
- +SA,1 (OK)
- +SB,1 (OK)
- +SC,1 (OK)
- SA,1 (OK)
- +SAS,2! (OK)
- SB,1 (OK)
- +SBS,2! (OK)
- SC,1 (OK)
- +SCG,2 (OK)
- +SCS,2! (OK)
- SCG,2 (OK)

Passed all tests! ✓

Correct
Marks for this submission: 2.00/2.00. Accounting for previous tries, this gives **1.60/2.00**.

Question 5

Correct

Mark 2.00 out of 2.00

Given the following graph, trace the frontier of a A-star search without pruning. The h-value of nodes are given in estimates.

```
ExplicitGraph(  
  nodes={'A', 'B', 'C', 'D', 'G'},  
  estimates={'A':5, 'B':5, 'C':9, 'D':1, 'G':0},  
  edge_list=[('A','B',2), ('A','C',3), ('B','D',5), ('C','D',10), ('D','G',3)],  
  starting_nodes=['A'],  
  goal_nodes={'G'},  
)
```

Note: we suggest that you solve this type of problems without drawing the graph, however, if drawing it makes it more understandable to you, draw this graph by putting the nodes along the same line in this order: CABDG. This way the length of the arcs would be more proportional to their costs and the heuristic values may make more sense.

Answer: (penalty regime: 20, 40, ... %)

```
1  +A, 5  
2  -A, 5  
3  +AB, 7 #2dst 5h  
4  +AC, 12 #3dst 9h  
5  -AB, 7  
6  +ABD, 8 #7dst 1h  
7  -ABD, 8  
8  +ABDG, 10 #10dst 0h  
9  -ABDG, 10  
10  
11 ##with pruning  
12 # +A, 5  
13 # -A, 5 #expended = {A}  
14 # +AB, 7  
15 # +AC, 12  
16 # -AB, 7 #expended = {A,B}  
17 # +ABD, 8  
18 # -ABD, 8 #expended = {A, B, D}  
19 # +ABDG, 10  
20 # -ABDG, 10 #expended = {A, B, D, G}
```

- +A,5 (OK)
 - A,5 (OK)
 - +AB,7 (OK)
 - +AC,12 (OK)
 - AB,7 (OK)
 - +ABD,8 (OK)
 - ABD,8 (OK)
 - +ABDG,10 (OK)
 - ABDG,10 (OK)
- Passed all tests! ✓

Correct
Marks for this submission: 2.00/2.00.

Question 6

Correct

Mark 2.00 out of 2.00

Consider the explicit graph in the previous question and select the correct statement(s).

- Select one or more:
- ☒ The heuristic function is admissible. ✓
 - ☒ LCFS would have produced an optimal solution for this graph. ✓
 - ☒ The heuristic function is consistent. ✓
 - ☐ A-star expanded as many paths on this graph as LCFS would have.
 - ☐ A-star with pruning would have produced a non-optimal solution (path to goal).

Your answer is correct.

Correct
Marks for this submission: 2.00/2.00.

Question **7**

Correct

Mark 2.00 out of 2.00

Given the following graph, trace the frontier of a A-star search with pruning. The h-value of nodes are given in estimates. Note multiple start nodes and the order elements.

```
ExplicitGraph(  
  nodes={'A', 'B', 'C', 'D', 'G'},  
  estimates={'A':6, 'B':3, 'C':2, 'D':1, 'G':0},  
  edge_list=[('C','D',3), ('B','D',4), ('A','D',5), ('D','G', 2)],  
  starting_nodes=['B', 'C', 'A'],  
  goal_nodes={'G'},  
)
```

Answer: (penalty regime: 20, 40, ... %)

```
1  +B, 3  
2  +C, 2  
3  +A, 6  
4  -C, 2 #expended nodes = {C}  
5  +CD, 4  
6  -B, 3 #expended nodes = {C, B}  
7  +BD, 5  
8  -CD, 4 #expended nodes = {C, B, D}  
9  +CDG, 5  
10 -BD, 5!  
11 -CDG, 5 #expended nodes = {C, B, D, G}
```

- +B,3 (OK)
- +C,2 (OK)
- +A,6 (OK)
- C,2 (OK)
- +CD,4 (OK)
- B,3 (OK)
- +BD,5 (OK)
- CD,4 (OK)
- +CDG,5 (OK)
- BD,5! (OK)
- CDG,5 (OK)

Passed all tests! ✓

Correct

Marks for this submission: 2.00/2.00.

Question **8**

Correct

Mark 4.00 out of 4.00

In this question you are to write a class `LocationGraph` which describes a set of nodes and the connection between them on a 2D plane. The class shares a lot of similarities with `ExplicitGraph` but it is not as explicit—you have to compute some information such as the cost of arcs based on other information provided. The following are the differences:

1. Each node in the graph has an associated location on a 2D plane which is represented with a pair of numbers. For instance each node is a city and has a location in the form of *(longitude, latitude)* on a 2D map. Therefore when initialising the graph object, in addition to the usual parameters of `ExplicitGraph`, a dictionary called `locations` specifies the location of each node (i.e. the dictionary maps a node to a pair of numbers). See the test cases below for some examples.
2. The edges are specified by the `set` edges. All edges in the graph are bidirectional; that is, if there is an edge ('A', 'B') in the set then it implies that there is an arc from 'A' to 'B' and also an arc from 'B' to 'A', even if the edge ('B', 'A') is not explicitly specified in the edges parameter. The cost of each arc is equal to the Euclidean distance between the tail and head nodes (you can think of arcs as straight lines connecting nodes). Note that the costs are not specified at initialisation time; they must be calculated automatically by the graph object.

Notes:

1. The order of arcs (Arc objects) returned by the `outgoing_arcs` must be alphabetical (see the test cases).
2. Consider subclassing `ExplicitGraph`.
3. Remember to include your import statements in the solution (or just paste your entire file if it does not produce unwanted output when it is imported by another module).

For example:

Test	Result
<pre>from student_answer import LocationGraph graph = LocationGraph(nodes=set('ABC'), locations={'A': (0, 0), 'B': (3, 0), 'C': (3, 4)}, edges=({'A', 'B'}, ('B', 'C'), ('C', 'A'))}, starting_nodes=['A'], goal_nodes={'C'}) for arc in graph.outgoing_arcs('A'): print(arc) for arc in graph.outgoing_arcs('B'): print(arc) for arc in graph.outgoing_arcs('C'): print(arc)</pre>	<pre>Arc(tail='A', head='B', action='A->B', cost=3.0) Arc(tail='A', head='C', action='A->C', cost=5.0) Arc(tail='B', head='A', action='B->A', cost=3.0) Arc(tail='B', head='C', action='B->C', cost=4.0) Arc(tail='C', head='A', action='C->A', cost=5.0) Arc(tail='C', head='B', action='C->B', cost=4.0)</pre>
<pre>from student_answer import LocationGraph pythagorean_graph = LocationGraph(nodes=set("abc"), locations={'a': (5, 6), 'b': (10,6), 'c': (10,18)}, edges={tuple(s) for s in {'ab', 'ac', 'bc'}}}, starting_nodes=['a'], goal_nodes={'c'}) for arc in pythagorean_graph.outgoing_arcs('a'): print(arc)</pre>	<pre>Arc(tail='a', head='b', action='a->b', cost=5.0) Arc(tail='a', head='c', action='a->c', cost=13.0)</pre>

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

```
1 from search import *
2 from collections import deque
3 import numpy as np
4 class LocationGraph(Graph):
5
6
7
8
9
10
11     """This is a concrete subclass of Graph where vertices and edges
12     are explicitly enumerated. Objects of this type are useful for
13     testing graph algorithms."""
14
```

```

15 ▼ def __init__(self, nodes, locations, edges, starting_nodes, goal_nodes):
16     """Initialises an LocationGraph graph.
17     Keyword arguments:
18     nodes -- a set of nodes
19     locations -- specifies the location of each node (i.e. the dictionary maps a node to a pair of
20     edges -- set edges. All edges in the graph are bidirectional; that is, if there is an edge (
21         then it implies that there is an arc from 'A' to 'B' and also an arc from 'B' to 'A'
22         is not explicitly specified in the edges parameter.
23
24     starting_nodes -- the list of starting nodes. We use a list
25         to remind you that the order can influence
26         the search behaviour.
27     goal_node -- the set of goal nodes. It's better if you use a set
28         here to remind yourself that the order does not matter
29         here. This is used only by the is_goal method.
30
31
32     # A few assertions to detect possible errors in
33     # instantiation. These assertions are not essential to the
34     # class functionality.
35     #assert all(tail in nodes and head in nodes for tail, head, *_ in edge_list)\
36     #, "An edge must link two existing nodes!"
37     #assert all(node in nodes for node in starting_nodes),\
38     #, "The starting_states must be in nodes."
39     #assert all(node in nodes for node in goal_nodes),\
40     #, "The goal states must be in nodes."
41
42     self.nodes = nodes
43     self.locations = locations
44     self.edges = edges
45     self._starting_nodes = starting_nodes
46     self.goal_nodes = goal_nodes
47
48 ▼ def starting_nodes(self):
49     """Returns a sequence of starting nodes."""
50     return self._starting_nodes
51
52 ▼ def is_goal(self, node):
53     """Returns true if the given node is a goal node."""
54     print("IS GOAL")
55     print(node, self.goal_nodes)
56     return node in self.goal_nodes
57
58 ▼ def outgoing_arcs(self, node):
59     """Returns a sequence of Arc objects that go out from the given
60     node. The action string is automatically generated.
61
62     """
63     arcs = []
64     #for every edge in list of edges
65 ▼ for edge in self.edges:
66 ▼     if node in edge:
67         node_locations = []
68         #append the locations of the nodes in the edges to list
69 ▼ for letter in edge:
70         node_location = self.locations[letter]
71         node_locations.append(node_location)
72         point_a = np.array(node_locations[0])
73         point_b = np.array(node_locations[1])
74         distance = np.linalg.norm(point_a - point_b)
75         #print("node locations are")
76         #print(node_locations)
77         #print("distance is")
78         #print(distance)
79
80 ▼         if len(edge) == 2: # if no cost is specified
81             tail, head = edge
82             cost = distance # assume unit cost
83 ▼         else:
84             tail, head, cost = edge
85 ▼         if tail == node:
86             arcs.append(Arc(tail, head, str(tail) + '->' + str(head), cost))
87 ▼         else:
88             arcs.append(Arc(head, tail, str(head) + '->' + str(tail), cost))
89
90     arcs = list(dict.fromkeys(arcs))
91     return sorted(arcs)

```

	Test	Expected	Got	
✓	<pre> from student_answer import LocationGraph graph = LocationGraph(nodes=set('ABC'), locations={'A': (0, 0), 'B': (3, 0), 'C': (3, 4)}}, edges=({'A', 'B'), ('B', 'C'), ('C', 'A')}}, starting_nodes= ['A'], goal_nodes= {'C'}) for arc in graph.outgoing_arcs('A'): print(arc) for arc in graph.outgoing_arcs('B'): print(arc) for arc in graph.outgoing_arcs('C'): print(arc) </pre>	<pre> Arc(tail='A', head='B', action='A->B', cost=3.0) Arc(tail='A', head='C', action='A->C', cost=5.0) Arc(tail='B', head='A', action='B->A', cost=3.0) Arc(tail='B', head='C', action='B->C', cost=4.0) Arc(tail='C', head='A', action='C->A', cost=5.0) Arc(tail='C', head='B', action='C->B', cost=4.0) </pre>	<pre> Arc(tail='A', head='B', action='A->B', cost=3.0) Arc(tail='A', head='C', action='A->C', cost=5.0) Arc(tail='B', head='A', action='B->A', cost=3.0) Arc(tail='B', head='C', action='B->C', cost=4.0) Arc(tail='C', head='A', action='C->A', cost=5.0) Arc(tail='C', head='B', action='C->B', cost=4.0) </pre>	✓
✓	<pre> from student_answer import LocationGraph graph = LocationGraph(nodes=set('ABC'), locations={'A': (0, 0), 'B': (3, 0), 'C': (3, 4)}}, edges=({'A', 'B'), ('B', 'C'), ('B', 'A'), ('C', 'A')}}, starting_nodes= ['A'], goal_nodes= {'C'}) for arc in graph.outgoing_arcs('A'): print(arc) for arc in graph.outgoing_arcs('B'): print(arc) for arc in graph.outgoing_arcs('C'): print(arc) </pre>	<pre> Arc(tail='A', head='B', action='A->B', cost=3.0) Arc(tail='A', head='C', action='A->C', cost=5.0) Arc(tail='B', head='A', action='B->A', cost=3.0) Arc(tail='B', head='C', action='B->C', cost=4.0) Arc(tail='C', head='A', action='C->A', cost=5.0) Arc(tail='C', head='B', action='C->B', cost=4.0) </pre>	<pre> Arc(tail='A', head='B', action='A->B', cost=3.0) Arc(tail='A', head='C', action='A->C', cost=5.0) Arc(tail='B', head='A', action='B->A', cost=3.0) Arc(tail='B', head='C', action='B->C', cost=4.0) Arc(tail='C', head='A', action='C->A', cost=5.0) Arc(tail='C', head='B', action='C->B', cost=4.0) </pre>	✓
✓	<pre> from student_answer import LocationGraph pythagorean_graph = LocationGraph(nodes=set("abc"), locations={'a': (5, 6), 'b': (10,6), 'c': (10,18)}, edges={tuple(s) for s in {'ab', 'ac', 'bc'}}}, starting_nodes=['a'], goal_nodes={'c'}) for arc in pythagorean_graph.outgoing_arcs('a'): print(arc) </pre>	<pre> Arc(tail='a', head='b', action='a->b', cost=5.0) Arc(tail='a', head='c', action='a->c', cost=13.0) </pre>	<pre> Arc(tail='a', head='b', action='a->b', cost=5.0) Arc(tail='a', head='c', action='a->c', cost=13.0) </pre>	✓

Passed all tests! ✓

Correct

Marks for this submission: 4.00/4.00.

Question 9

Correct

Mark 4.00 out of 4.00

Write a class `LCFSFrontier` such that when an instance of this class along with a graph object is passed to `generic_search`, lowest-cost-first search (LCFS) is performed. It is strongly recommended that you write `LCFSFrontier` as a subclass of `Frontier` class (instead of writing it from scratch); although the `Frontier` class does not provide any functionality to reuse, subclassing makes it easier for you to check whether the new frontier has all the methods required by the abstract base class. Your answer must include the code for both `LCFSFrontier` class and `LocationGraph` class (from the previous question). The test cases check whether these two together produce a valid lowest-cost-first search.

Hint: Read the documentation of the `heapq` module in standard Python library.

For example:

Test	Result
<pre>from search import * from student_answer import LocationGraph, LCFSFrontier graph = LocationGraph(nodes=set('ABC'), locations={'A': (0, 0), 'B': (3, 0), 'C': (3, 4)}, edges=({'A', 'B'), ('B', 'C'), ('B', 'A'), ('C', 'A')}), starting_nodes=['A'], goal_nodes={'C'}) solution = next(generic_search(graph, LCFSFrontier())) print_actions(solution)</pre>	Actions: A->C. Total cost: 5.0
<pre>from search import * from student_answer import LocationGraph, LCFSFrontier graph = LocationGraph(nodes=set('ABC'), locations={'A': (0, 0), 'B': (3, 0), 'C': (3, 4)}, edges=({'A', 'B'), ('B', 'C'), ('B', 'A')}), starting_nodes=['A'], goal_nodes={'C'}) solution = next(generic_search(graph, LCFSFrontier())) print_actions(solution)</pre>	Actions: A->B, B->C. Total cost: 7.0
<pre>from search import * from student_answer import LocationGraph, LCFSFrontier pythagorean_graph = LocationGraph(nodes=set("abc"), locations={'a': (5, 6), 'b': (10,6), 'c': (10,18)}, edges={tuple(s) for s in {'ab', 'ac', 'bc'}}, starting_nodes=['a'], goal_nodes={'c'}) solution = next(generic_search(pythagorean_graph, LCFSFrontier())) print_actions(solution)</pre>	Actions: a->c. Total cost: 13.0

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

```
1 from search import *
2 from collections import deque
3 import numpy as np
4 import heapq
5
6 class LCFSFrontier(Frontier):
7     """At each stage, lowest-cost-first search selects a path on the
8     frontier with lowest cost.
9     The frontier is a priority queue ordered by path cost.
10    When all arc costs are equal, LCFS is equivalent to BFS.
11    """
12    def __init__(self):
13        self.Arclist = []
14
15    def add(self, path):
16        """Adds a new path(arc)? to the frontier. A path is a sequence (tuple) of
17        Arc objects.
18        """
19        #assuming that i am creating a heap of objects to be the frontier
```

```

20     #objects in this case are paths which are a list of arcs
21     #all starting arcs are added to heap at start(from generic_search) then expand one by one
22     #push starting node paths(arc?) onto the heap?
23     #print(path[0].cost)
24     ##need to calculate cost of path and pass it into the heap
25     #cost is calculated already in LocationGraph so i just need to extract it and append
26
27     total_cost = 0
28     #sum the total costs of the path to be appeneded not the euclidean cost only
29     for i in path:
30         #print(i)
31         total_cost += i.cost
32         #print("current cost is of path")
33         #print(path)
34         #print(total_cost)
35     heapq.heappush(self.Arclist, (total_cost, path))
36
37
38     def __iter__(self):
39         """We don't need a separate iterator object. Just return self. You
40         don't need to change this method."""
41         return self
42
43
44     def __next__(self):
45         """Selects, removes, and returns a path on the frontier if there is
46         any. Recall that a path is a sequence (tuple) of Arc
47         objects. Override this method to achieve a desired search
48         strategy. If there nothing to return this should raise a
49         StopIteration exception.
50         """
51         #who calles next? add?
52         #print(next_path[-1])
53         #remove the cost and only pass the path at the end
54         if len(self.Arclist) == 0:
55
56             raise StopIteration
57         else:
58             next_path = heapq.heappop(self.Arclist)
59
60             return next_path[1]
61
62
63
64
65
66
67     ###locationgraph
68
69
70     class LocationGraph(Graph):
71         """This is a concrete subclass of Graph where vertices and edges
72         are explicitly enumerated. Objects of this type are useful for
73         testing graph algorithms."""
74
75         def __init__(self, nodes, locations, edges, starting_nodes, goal_nodes):
76             """Initialises an LocationGraph graph.
77             Keyword arguments:
78             nodes -- a set of nodes
79             locations -- specifies the location of each node (i.e. the dictionary maps a node to a pair
80             edges -- set edges. All edges in the graph are bidirectional; that is, if there is an edge
81                     then it implies that there is an arc from 'A' to 'B' and also an arc from 'B' to
82                     is not explicitly specified in the edges parameter.
83
84             starting_nodes -- the list of starting nodes. We use a list
85                             to remind you that the order can influence
86                             the search behaviour.
87             goal_node -- the set of goal nodes. It's better if you use a set
88                        here to remind yourself that the order does not matter
89                        here. This is used only by the is_goal method.
90
91             """
92
93             # A few assertions to detect possible errors in
94             # instantiation. These assertions are not essential to the
95             # class functionality.
96             #assert all(tail in nodes and head in nodes for tail, head, *_ in edge_list)\
97             #    #, "An edge must link two existing nodes!")
98             #assert all(node in nodes for node in starting_nodes),\
99             #    # "The starting_states must be in nodes."
100            #assert all(node in nodes for node in goal_nodes),\
101            #    # "The goal states must be in nodes."
102
103            self.nodes = nodes
104            self.locations = locations
105            self.edges = edges

```



```

105     self._starting_nodes = starting_nodes
106     self.goal_nodes = goal_nodes
107
108     def starting_nodes(self):
109         """Returns a sequence of starting nodes."""
110         return self._starting_nodes
111
112     def is_goal(self, node):
113         """Returns true if the given node is a goal node."""
114         #print("IS GOAL")
115         #print(node, self.goal_nodes)
116         return node in self.goal_nodes
117
118     def outgoing_arcs(self, node):
119         """Returns a sequence of Arc objects that go out from the given
120         node. The action string is automatically generated.
121
122         """
123         arcs = []
124         #for every edge in list of edges
125         for edge in self.edges:
126             if node in edge:
127                 node_locations = []
128                 #append the locations of the nodes in the edges to list
129                 for letter in edge:
130                     node_location = self.locations[letter]
131                     node_locations.append(node_location)
132                 point_a = np.array(node_locations[0])
133                 point_b = np.array(node_locations[1])
134                 distance = np.linalg.norm(point_a - point_b)
135                 #print("node locations are")
136                 #print(node_locations)
137                 #print("distance is")
138                 #print(distance)
139
140                 if len(edge) == 2: # if no cost is specified
141                     tail, head = edge
142                     cost = distance # assume unit cost
143                 else:
144                     tail, head, cost = edge
145                 if tail == node:
146                     arcs.append(Arc(tail, head, str(tail) + '->' + str(head), cost))
147                 else:
148                     arcs.append(Arc(head, tail, str(head) + '->' + str(tail), cost))
149
150         arcs = list(dict.fromkeys(arcs))
151         return sorted(arcs)
152

```

	Test	Expected	Got	
✓	<pre> from search import * from student_answer import LocationGraph, LCFSFrontier graph = LocationGraph(nodes=set('ABC'), locations={'A': (0, 0), 'B': (3, 0), 'C': (3, 4)}, edges=({'A', 'B'}, {'B', 'C'}, {'B', 'A'}, {'C', 'A'}), starting_nodes=['A'], goal_nodes={'C'}) solution = next(generic_search(graph, LCFSFrontier())) print_actions(solution) </pre>	Actions: A->C. Total cost: 5.0	Actions: A->C. Total cost: 5.0	✓

	Test	Expected	Got	
✓	<pre>from search import * from student_answer import LocationGraph, LCFSFrontier graph = LocationGraph(nodes=set('ABC'), locations={'A': (0, 0), 'B': (3, 0), 'C': (3, 4)}, edges=({'A', 'B'}, ('B', 'C'), ('B', 'A'))}, starting_nodes=['A'], goal_nodes={'C'}) solution = next(generic_search(graph, LCFSFrontier())) print_actions(solution)</pre>	Actions: A->B, B->C. Total cost: 7.0	Actions: A->B, B->C. Total cost: 7.0	✓
✓	<pre>from search import * from student_answer import LocationGraph, LCFSFrontier pythagorean_graph = LocationGraph(nodes=set("abc"), locations={'a': (5, 6), 'b': (10,6), 'c': (10,18)}, edges={tuple(s) for s in {'ab', 'ac', 'bc'}}}, starting_nodes=['a'], goal_nodes={'c'}) solution = next(generic_search(pythagorean_graph, LCFSFrontier())) print_actions(solution)</pre>	Actions: a->c. Total cost: 13.0	Actions: a->c. Total cost: 13.0	✓

Passed all tests! ✓

Correct
Marks for this submission: 4.00/4.00.