



College of Engineering & Applied Sciences

# CSPB 3202

*Introduction To Artificial Intelligence*

*Assignment 1 - Intro To AI, Search Problems*

UNIVERSITY OF COLORADO

2024

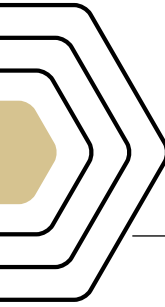
# Introduction To Artificial Intelligence - Assignment 1 - Intro To AI, Search Problems

<b>1 Assignment 1 - Intro To AI, Search Problems</b>	<b>3</b>
Assignment 1 - Intro To AI, Search Problems. ....	3
Problem 1. ....	4
Problem 2. ....	6
Problem 3. ....	8
Problem 4. ....	10
Problem 5. ....	13
Problem 6. ....	14
Problem 7. ....	16
Problem 8. ....	18
Problem 9. ....	19
Problem 10 . ....	21



---

## Assignment 1 - Intro To AI, Search Problems



---

## Assignment 1 - Intro To AI, Search Problems

---

I have neither given nor received unauthorized assistance.

---

Taylor Larrechea

---

- The instructions for this assignment can be found here [here](#)
- The submission for this assignment can be found here [here](#)



# Problem 1

## Problem Statement

Implement a function `breadth_first(start, goal, state_graph, return_cost)` to search the state space (and step costs) defined by `state_graph` using breadth-first search:

## Solution

```

1  from collections import deque
2  from collections import OrderedDict
3  map_distances = dict(
4      chi=OrderedDict([("det",283), ("cle",345), ("ind",182)]),
5      cle=OrderedDict([("chi",345), ("det",169), ("col",144), ("pit",134), ("buf",189)]),
6      ind=OrderedDict([("chi",182), ("col",176)]),
7      col=OrderedDict([("ind",176), ("cle",144), ("pit",185)]),
8      det=OrderedDict([("chi",283), ("cle",169), ("buf",256)]),
9      buf=OrderedDict([("det",256), ("cle",189), ("pit",215), ("syr",150)]),
10     pit=OrderedDict([("col",185), ("cle",134), ("buf",215), ("phi",305), ("bal",247)]),
11     syr=OrderedDict([("buf",150), ("phi",253), ("new",254), ("bos",312)]),
12     bal=OrderedDict([("phi",101), ("pit",247)]),
13     phi=OrderedDict([("pit",305), ("bal",101), ("syr",253), ("new",97)]),
14     new=OrderedDict([("syr",254), ("phi",97), ("bos",215), ("pro",181)]),
15     pro=OrderedDict([("bos",50), ("new",181)]),
16     bos=OrderedDict([("pro",50), ("new",215), ("syr",312), ("por",107)]),
17     por=OrderedDict([("bos",107)])
18
19  map_times = dict(
20     chi=dict(det=280, cle=345, ind=200),
21     cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
22     ind=dict(chi=200, col=175),
23     col=dict(ind=175, cle=155, pit=185),
24     det=dict(chi=280, cle=170, buf=270),
25     buf=dict(det=270, cle=185, pit=215, syr=145),
26     pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
27     syr=dict(buf=145, phi=245, new=260, bos=290),
28     bal=dict(phi=145, pit=255),
29     phi=dict(pit=305, bal=145, syr=245, new=150),
30     new=dict(syr=260, phi=150, bos=270, pro=260),
31     pro=dict(bos=90, new=260),
32     bos=dict(pro=90, new=270, syr=290, por=120),
33     por=dict(bos=120))
34
35  def path(previous, s):
36      '''
37      'previous' is a dictionary chaining together the predecessor state that led to each state
38      's' will be None for the initial state
39      otherwise, start from the last state 's' and recursively trace 'previous' back to the initial
40      state,
41      constructing a list of states visited as we go
42      '''
43      if s is None:
44          return []
45      else:
46          return path(previous, previous[s])+[s]
47
48  def pathcost(path, step_costs):
49      '''
50      add up the step costs along a path, which is assumed to be a list output from the 'path' function
51      above
52      '''
53      cost = 0
54      for s in range(len(path)-1):
55          cost += step_costs[path[s]][path[s+1]]
56      return cost
57
58  # Solution:
59  """ breadth_first - Performs a breadth first search on cities
60  Input:
61      start - Node that represents the start of the path
62      goal - Node that represents the desired end point of the path
63      state_graph - Represents the graph that is being searched in
64      return_cost - Boolean value that indicates the cost of the path
65  Algorithm:
66      * Create a queue with the start node as the first node
67      * Create a visited set where the first node is visited
68      * Create a dictionary for the previous nodes that have been visited
69      * While the queue is not empty:

```

```
70         * Pop the current node from the queue
71         * If we reach the goal
72             * Create a path with the previous nodes and the goal
73             * Return the path and the cost if return_cost is set to true, otherwise just the path
74         * Iterate over the neighbors of the current node
75         * Add the neighbor to the visited set if it isn't visited
76         * Update the previous node with the current node
77         * Add the neighbor to the queue
78         * Return the cost of the traversal
79     Output:
80         Returns the path in the search as well as the cost in the path
81 """
82 def breadth_first(start, goal, state_graph, return_cost=False):
83     queue = deque([start])
84     visited = set([start])
85     previous = {start: None}
86     while (queue):
87         current = queue.popleft()
88         if (current == goal):
89             path_to_goal = path(previous, goal)
90             if (return_cost):
91                 cost = pathcost(path_to_goal, state_graph)
92                 return path_to_goal, cost
93             else:
94                 return path_to_goal
95     for neighbor in state_graph[current]:
96         if (neighbor not in visited):
97             visited.add(neighbor)
98             previous[neighbor] = current
99             queue.append(neighbor)
100     return None if not return_cost else (None, 0)
101
```



# Problem 2

## Problem Statement

Implement a function `depth_first(start, goal, state_graph, return_cost)` to search the state space (and step costs) defined by `state_graph` using depth-first search:

## Solution

```

1  from collections import deque
2  from collections import OrderedDict
3
4  map_distances = dict(
5      chi=OrderedDict([("det",283), ("cle",345), ("ind",182)]),
6      cle=OrderedDict([("chi",345), ("det",169), ("col",144), ("pit",134), ("buf",189)]),
7      ind=OrderedDict([("chi",182), ("col",176)]),
8      col=OrderedDict([("ind",176), ("cle",144), ("pit",185)]),
9      det=OrderedDict([("chi",283), ("cle",169), ("buf",256)]),
10     buf=OrderedDict([("det",256), ("cle",189), ("pit",215), ("syr",150)]),
11     pit=OrderedDict([("col",185), ("cle",134), ("buf",215), ("phi",305), ("bal",247)]),
12     syr=OrderedDict([("buf",150), ("phi",253), ("new",254), ("bos",312)]),
13     bal=OrderedDict([("phi",101), ("pit",247)]),
14     phi=OrderedDict([("pit",305), ("bal",101), ("syr",253), ("new",97)]),
15     new=OrderedDict([("syr",254), ("phi",97), ("bos",215), ("pro",181)]),
16     pro=OrderedDict([("bos",50), ("new",181)]),
17     bos=OrderedDict([("pro",50), ("new",215), ("syr",312), ("por",107)]),
18     por=OrderedDict([("bos",107)])
19
20
21  map_times = dict(
22      chi=dict(det=280, cle=345, ind=200),
23      cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
24      ind=dict(chi=200, col=175),
25      col=dict(ind=175, cle=155, pit=185),
26      det=dict(chi=280, cle=170, buf=270),
27      buf=dict(det=270, cle=185, pit=215, syr=145),
28      pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
29      syr=dict(buf=145, phi=245, new=260, bos=290),
30      bal=dict(phi=145, pit=255),
31      phi=dict(pit=305, bal=145, syr=245, new=150),
32      new=dict(syr=260, phi=150, bos=270, pro=260),
33      pro=dict(bos=90, new=260),
34      bos=dict(pro=90, new=270, syr=290, por=120),
35      por=dict(bos=120)
36
37  def path(previous, s):
38      '''
39      'previous' is a dictionary chaining together the predecessor state that led to each state
40      's' will be None for the initial state
41      otherwise, start from the last state 's' and recursively trace 'previous' back to the initial
42      state,
43      constructing a list of states visited as we go
44      '''
45      if s is None:
46          return []
47      else:
48          return path(previous, previous[s])+[s]
49
50  def pathcost(path, step_costs):
51      '''
52      add up the step costs along a path, which is assumed to be a list output from the 'path' function
53      above
54      '''
55      cost = 0
56      for s in range(len(path)-1):
57          cost += step_costs[path[s]][path[s+1]]
58      return cost
59
60  """ depth_first - Performs a DFS on the state graph for the path between a start and goal
61  Input:
62      start - Node that represents the start of the path
63      goal - Node that represents the desired end point of the path
64      state_graph - Represents the graph that is being searched in
65      return_cost - Boolean value that indicates the cost of the path
66  Algorithm:
67      * Create a stack of nodes with the start node as the original element in it
68      * Create a set of visited nodes with the start node as the original element in it
69      * Create a dictionary of previous nodes that is empty
70      * While the stack is not empty
71      * Pop the top most node from the stack

```

```
70     * Check if that node is the goal
71     * If it is
72         * Update the path to the goal with the previous nodes and the goal
73         * Return the path to goal and the cost if return_cost is set to true
74         * Otherwise, just return the path
75     * If it is not
76         * Iterate over the neighbors of the current node
77         * Add the neighbor to the visited set if it is not visited
78         * Update the previous nodes with the current node
79         * Add the neighbor to the stack
80     * Return the path and cost, or just the path if designated
81 Output:
82     Returns the path in the search as well as the cost in the path
83 """
84 def depth_first(start, goal, state_graph, return_cost=False):
85     stack = [start]
86     visited = set([start])
87     previous = {start: None}
88     while (stack):
89         current = stack.pop()
90         if (current == goal):
91             path_to_goal = path(previous, goal)
92             if (return_cost):
93                 cost = pathcost(path_to_goal, state_graph)
94                 return path_to_goal, cost
95             else:
96                 return path_to_goal
97         for neighbor in state_graph[current]:
98             if (neighbor not in visited):
99                 visited.add(neighbor)
100                 previous[neighbor] = current
101                 stack.append(neighbor)
102     return None if not return_cost else (None, 0)
103
```



# Problem 3

## Problem Statement

First, let's create our own `Frontier_PQ` class to represent the frontier (priority queue) for uniformcost search. Note that the `heapq` package is imported in the helpers at the bottom of this assignment; you may find that package useful. You could also use the `Queue` package. Your implementation of the uniform-cost search frontier should adhere to these specifications

## Solution

```

1  from collections import deque
2  import heapq
3
4  map_distances = dict(
5      chi=dict(det=283, cle=345, ind=182),
6      cle=dict(chi=345, det=169, col=144, pit=134, buf=189),
7      ind=dict(chi=182, col=176),
8      col=dict(ind=176, cle=144, pit=185),
9      det=dict(chi=283, cle=169, buf=256),
10     buf=dict(det=256, cle=189, pit=215, syr=150),
11     pit=dict(col=185, cle=134, buf=215, phi=305, bal=247),
12     syr=dict(buf=150, phi=253, new=254, bos=312),
13     bal=dict(phi=101, pit=247),
14     phi=dict(pit=305, bal=101, syr=253, new=97),
15     new=dict(syr=254, phi=97, bos=215, pro=181),
16     pro=dict(bos=50, new=181),
17     bos=dict(pro=50, new=215, syr=312, por=107),
18     por=dict(bos=107))
19
20
21 map_times = dict(
22     chi=dict(det=280, cle=345, ind=200),
23     cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
24     ind=dict(chi=200, col=175),
25     col=dict(ind=175, cle=155, pit=185),
26     det=dict(chi=280, cle=170, buf=270),
27     buf=dict(det=270, cle=185, pit=215, syr=145),
28     pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
29     syr=dict(buf=145, phi=245, new=260, bos=290),
30     bal=dict(phi=145, pit=255),
31     phi=dict(pit=305, bal=145, syr=245, new=150),
32     new=dict(syr=260, phi=150, bos=270, pro=260),
33     pro=dict(bos=90, new=260),
34     bos=dict(pro=90, new=270, syr=290, por=120),
35     por=dict(bos=120))
36
37 def path(previous, s):
38     '''
39     'previous' is a dictionary chaining together the predecessor state that led to each state
40     's' will be None for the initial state
41     otherwise, start from the last state 's' and recursively trace 'previous' back to the initial
42     state,
43     constructing a list of states visited as we go
44     '''
45     if s is None:
46         return []
47     else:
48         return path(previous, previous[s])+[s]
49
50 def pathcost(path, step_costs):
51     '''
52     add up the step costs along a path, which is assumed to be a list output from the 'path' function
53     above
54     '''
55     cost = 0
56     for s in range(len(path)-1):
57         cost += step_costs[path[s]][path[s+1]]
58     return cost
59
60 # Solution:
61 """ Frontier_PQ - Implements a priority queue ordered by path cost for uniform cost search
62 Methods:
63     __init__ - Initializes an empty priority queue
64     is_empty - Checks if the priority queue is empty
65     put - Adds an item with a specified priority to the priority queue
66     get - Removes and returns the item with the lowest priority from the priority queue
67 Algorithm:
68     * __init__ initializes an empty list to represent the priority queue

```



```

67     * is_empty returns True if the list is empty, otherwise False
68     * put uses heapq.heappush to add an item to the priority queue with the given priority
69     * get uses heapq.heappop to remove and return the item with the lowest priority
70 Output:
71     * is_empty returns a boolean indicating whether the priority queue is empty
72     * put does not return a value
73     * get returns the item with the lowest priority
74 """
75 class Frontier_PQ:
76     ''' frontier class for uniform search, ordered by path cost '''
77     # add your code here
78     def __init__(self):
79         self.elements = []
80     def is_empty(self):
81         return len(self.elements) == 0
82     def put(self, item, priority):
83         heapq.heappush(self.elements, (priority, item))
84     def get(self):
85         return heapq.heappop(self.elements)
86
87 # Solution:
88 """ uniform_cost - Performs a Uniform Cost Search (UCS) on the state graph for the path between a
89 start and goal
90 Input:
91     start - Node that represents the start of the path
92     goal - Node that represents the desired end point of the path
93     state_graph - Dictionary representing the graph being searched, with costs for each edge
94     return_cost - Boolean value that indicates whether to return the cost of the path
95 Algorithm:
96     * Initialize a Frontier_PQ instance and add the start node with a priority of 0
97     * Initialize a dictionary of previous nodes with the start node set to None
98     * Initialize a dictionary to keep track of the cost to reach each node with the start node
99 set to 0
100     * While the priority queue is not empty
101         * Get the node with the lowest cost from the priority queue
102         * Check if that node is the goal
103         * If it is
104             * Update the path to the goal using the previous nodes and the goal
105             * Calculate the cost if return_cost is True
106             * Return the path to the goal and the cost if return_cost is set to True
107             * Otherwise, just return the path
108         * If it is not
109             * Iterate over the neighbors of the current node
110             * Calculate the new cost to reach each neighbor
111             * If the neighbor has not been visited or the new cost is lower than the recorded
112 cost
113             * Update the cost to reach the neighbor
114             * Add the neighbor to the priority queue with the new cost as priority
115             * Update the previous nodes with the current node
116         * Return None if the goal is not reachable or return (None, 0) if return_cost is True
117 Output:
118     Returns the path in the search as well as the cost in the path if return_cost is True
119 """
120 def uniform_cost(start, goal, state_graph, return_cost=False):
121     frontier = Frontier_PQ()
122     frontier.put(start, 0)
123     previous = {start: None}
124     cost_so_far = {start: 0}
125     while not frontier.is_empty():
126         current_priority, current = frontier.get()
127         if current == goal:
128             path_to_goal = path(previous, goal)
129             if return_cost:
130                 cost = pathcost(path_to_goal, state_graph)
131                 return path_to_goal, cost
132             else:
133                 return path_to_goal
134         for neighbor in state_graph[current]:
135             new_cost = cost_so_far[current] + state_graph[current][neighbor]
136             if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
137                 cost_so_far[neighbor] = new_cost
138                 priority = new_cost
139                 frontier.put(neighbor, priority)
140                 previous[neighbor] = current
141     return None if not return_cost else (None, 0)
142

```

# Problem 4

## Problem Statement

Use each of your search functions to find routes for Neal to travel from New York to Chicago, with path costs defined by the distance between cities.

## Solution

```

1  from collections import deque
2  from collections import OrderedDict
3  import heapq
4  map_distances = dict(
5      chi=OrderedDict([("det",283), ("cle",345), ("ind",182)]),
6      cle=OrderedDict([("chi",345), ("det",169), ("col",144), ("pit",134), ("buf",189)]),
7      ind=OrderedDict([("chi",182), ("col",176)]),
8      col=OrderedDict([("ind",176), ("cle",144), ("pit",185)]),
9      det=OrderedDict([("chi",283), ("cle",169), ("buf",256)]),
10     buf=OrderedDict([("det",256), ("cle",189), ("pit",215), ("syr",150)]),
11     pit=OrderedDict([("col",185), ("cle",134), ("buf",215), ("phi",305), ("bal",247)]),
12     syr=OrderedDict([("buf",150), ("phi",253), ("new",254), ("bos",312)]),
13     bal=OrderedDict([("phi",101), ("pit",247)]),
14     phi=OrderedDict([("pit",305), ("bal",101), ("syr",253), ("new",97)]),
15     new=OrderedDict([("syr",254), ("phi",97), ("bos",215), ("pro",181)]),
16     pro=OrderedDict([("bos",50), ("new",181)]),
17     bos=OrderedDict([("pro",50), ("new",215), ("syr",312), ("por",107)]),
18     por=OrderedDict([("bos",107)])
19
20 map_times = dict(
21     chi=dict(det=280, cle=345, ind=200),
22     cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
23     ind=dict(chi=200, col=175),
24     col=dict(ind=175, cle=155, pit=185),
25     det=dict(chi=280, cle=170, buf=270),
26     buf=dict(det=270, cle=185, pit=215, syr=145),
27     pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
28     syr=dict(buf=145, phi=245, new=260, bos=290),
29     bal=dict(phi=145, pit=255),
30     phi=dict(pit=305, bal=145, syr=245, new=150),
31     new=dict(syr=260, phi=150, bos=270, pro=260),
32     pro=dict(bos=90, new=260),
33     bos=dict(pro=90, new=270, syr=290, por=120),
34     por=dict(bos=120))
35
36 def path(previous, s):
37     '''
38     'previous' is a dictionary chaining together the predecessor state that led to each state
39     's' will be None for the initial state
40     otherwise, start from the last state 's' and recursively trace 'previous' back to the initial
41     state,
42     constructing a list of states visited as we go
43     '''
44     if s is None:
45         return []
46     else:
47         return path(previous, previous[s])+[s]
48
49 def pathcost(path, step_costs):
50     '''
51     add up the step costs along a path, which is assumed to be a list output from the 'path' function
52     above
53     '''
54     cost = 0
55     for s in range(len(path)-1):
56         cost += step_costs[path[s]][path[s+1]]
57     return cost
58
59 # Solution:
60 """ Frontier_PQ - Implements a priority queue ordered by path cost for uniform cost search
61 Methods:
62     __init__ - Initializes an empty priority queue
63     is_empty - Checks if the priority queue is empty
64     put - Adds an item with a specified priority to the priority queue
65     get - Removes and returns the item with the lowest priority from the priority queue
66 Algorithm:
67     * __init__ initializes an empty list to represent the priority queue
68     * is_empty returns True if the list is empty, otherwise False
69     * put uses heapq.heappush to add an item to the priority queue with the given priority
70     * get uses heapq.heappop to remove and return the item with the lowest priority

```

```

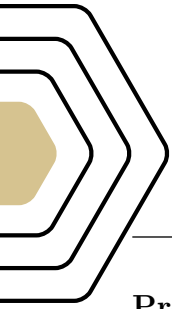
70     Output:
71         * is_empty returns a boolean indicating whether the priority queue is empty
72         * put does not return a value
73         * get returns the item with the lowest priority
74     """
75     class Frontier_PQ:
76         ''' frontier class for uniform search, ordered by path cost '''
77         # add your code here
78         def __init__(self):
79             self.elements = []
80         def is_empty(self):
81             return len(self.elements) == 0
82         def put(self, item, priority):
83             heapq.heappush(self.elements, (priority, item))
84         def get(self):
85             return heapq.heappop(self.elements)
86
87     """
88     breadth_first - Performs a breadth first search on cities
89     Input:
90         start - Node that represents the start of the path
91         goal - Node that represents the desired end point of the path
92         state_graph - Represents the graph that is being searched in
93         return_cost - Boolean value that indicates the cost of the path
94     Algorithm:
95         * Create a queue with the start node as the first node
96         * Create a visited set where the first node is visited
97         * Create a dictionary for the previous nodes that have been visited
98         * While the queue is not empty:
99             * Pop the current node from the queue
100             * If we reach the goal
101                 * Create a path with the previous nodes and the goal
102                 * Return the path and the cost if return_cost is set to true, otherwise just the path
103             * Iterate over the neighbors of the current node
104             * Add the neighbor to the visited set if it isn't visited
105             * Update the previous node with the current node
106             * Add the neighbor to the queue
107         * Return the cost of the traversal
108     Output:
109         Returns the path in the search as well as the cost in the path
110     """
111     def breadth_first(start, goal, state_graph, return_cost=False):
112         queue = deque([start])
113         visited = set([start])
114         previous = {start: None}
115         while (queue):
116             current = queue.popleft()
117             if (current == goal):
118                 path_to_goal = path(previous, goal)
119                 if (return_cost):
120                     cost = pathcost(path_to_goal, state_graph)
121                     return path_to_goal, cost
122                 else:
123                     return path_to_goal
124             for neighbor in state_graph[current]:
125                 if (neighbor not in visited):
126                     visited.add(neighbor)
127                     previous[neighbor] = current
128                     queue.append(neighbor)
129         return None if not return_cost else (None, 0)
130
131     """
132     depth_first - Performs a DFS on the state graph for the path between a start and goal
133     Input:
134         start - Node that represents the start of the path
135         goal - Node that represents the desired end point of the path
136         state_graph - Represents the graph that is being searched in
137         return_cost - Boolean value that indicates the cost of the path
138     Algorithm:
139         * Create a stack of nodes with the start node as the original element in it
140         * Create a set of visited nodes with the start node as the original element in it
141         * Create a dictionary of previous nodes that is empty
142         * While the stack is not empty
143             * Pop the top most node from the stack
144             * Check if that node is the goal
145             * If it is
146                 * Update the path to the goal with the previous nodes and the goal
147                 * Return the path to goal and the cost if return_cost is set to true
148                 * Otherwise, just return the path
149             * If it is not
150                 * Iterate over the neighbors of the current node
151                 * Add the neighbor to the visited set if it is not visited
152                 * Update the previous nodes with the current node
153                 * Add the neighbor to the stack
154         * Return the path and cost, or just the path if designated
155     Output:
156         Returns the path in the search as well as the cost in the path
157     """
158     def depth_first(start, goal, state_graph, return_cost=False):
159         stack = [start]
160         visited = set([start])
161         previous = {start: None}
162         while (stack):

```

```

162     current = stack.pop()
163     if (current == goal):
164         path_to_goal = path(previous, goal)
165         if (return_cost):
166             cost = pathcost(path_to_goal, state_graph)
167             return path_to_goal, cost
168         else:
169             return path_to_goal
170     for neighbor in state_graph[current]:
171         if (neighbor not in visited):
172             visited.add(neighbor)
173             previous[neighbor] = current
174             stack.append(neighbor)
175     return None if not return_cost else (None, 0)
176
177 # Solution:
178 """ uniform_cost - Performs a Uniform Cost Search (UCS) on the state graph for the path between a
179 start and goal
180     Input:
181         start - Node that represents the start of the path
182         goal - Node that represents the desired end point of the path
183         state_graph - Dictionary representing the graph being searched, with costs for each edge
184         return_cost - Boolean value that indicates whether to return the cost of the path
185     Algorithm:
186         * Initialize a Frontier_PQ instance and add the start node with a priority of 0
187         * Initialize a dictionary of previous nodes with the start node set to None
188         * Initialize a dictionary to keep track of the cost to reach each node with the start node
189         set to 0
190         * While the priority queue is not empty
191             * Get the node with the lowest cost from the priority queue
192             * Check if that node is the goal
193             * If it is
194                 * Update the path to the goal using the previous nodes and the goal
195                 * Calculate the cost if return_cost is True
196                 * Return the path to the goal and the cost if return_cost is set to True
197                 * Otherwise, just return the path
198             * If it is not
199                 * Iterate over the neighbors of the current node
200                 * Calculate the new cost to reach each neighbor
201                 * If the neighbor has not been visited or the new cost is lower than the recorded
202                     cost
203                     * Update the cost to reach the neighbor
204                     * Add the neighbor to the priority queue with the new cost as priority
205                     * Update the previous nodes with the current node
206             * Return None if the goal is not reachable or return (None, 0) if return_cost is True
207     Output:
208         Returns the path in the search as well as the cost in the path if return_cost is True
209 """
210 def uniform_cost(start, goal, state_graph, return_cost=False):
211     frontier = Frontier_PQ()
212     frontier.put(start, 0)
213     previous = {start: None}
214     cost_so_far = {start: 0}
215     while not frontier.is_empty():
216         current_priority, current = frontier.get()
217         if current == goal:
218             path_to_goal = path(previous, goal)
219             if return_cost:
220                 cost = pathcost(path_to_goal, state_graph)
221                 return path_to_goal, cost
222             else:
223                 return path_to_goal
224         for neighbor in state_graph[current]:
225             new_cost = cost_so_far[current] + state_graph[current][neighbor]
226             if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
227                 cost_so_far[neighbor] = new_cost
228                 priority = new_cost
229                 frontier.put(neighbor, priority)
230                 previous[neighbor] = current
231     return None if not return_cost else (None, 0)
232

```



## Problem 5

### Problem Statement

Which algorithm yields the shortest path?

### Solution

Uniform Cost Search



# Problem 6

## Problem Statement

Use your choice of search function to show the list of cities that Neal will traverse to get to Chicago on time, should such a path exist.

## Solution

```

1  from collections import deque
2  import heapq
3
4  map_distances = dict(
5      chi=dict(det=283, cle=345, ind=182),
6      cle=dict(chi=345, det=169, col=144, pit=134, buf=189),
7      ind=dict(chi=182, col=176),
8      col=dict(ind=176, cle=144, pit=185),
9      det=dict(chi=283, cle=169, buf=256),
10     buf=dict(det=256, cle=189, pit=215, syr=150),
11     pit=dict(col=185, cle=134, buf=215, phi=305, bal=247),
12     syr=dict(buf=150, phi=253, new=254, bos=312),
13     bal=dict(phi=101, pit=247),
14     phi=dict(pit=305, bal=101, syr=253, new=97),
15     new=dict(syr=254, phi=97, bos=215, pro=181),
16     pro=dict(bos=50, new=181),
17     bos=dict(pro=50, new=215, syr=312, por=107),
18     por=dict(bos=107))
19
20
21  map_times = dict(
22     chi=dict(det=280, cle=345, ind=200),
23     cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
24     ind=dict(chi=200, col=175),
25     col=dict(ind=175, cle=155, pit=185),
26     det=dict(chi=280, cle=170, buf=270),
27     buf=dict(det=270, cle=185, pit=215, syr=145),
28     pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
29     syr=dict(buf=145, phi=245, new=260, bos=290),
30     bal=dict(phi=145, pit=255),
31     phi=dict(pit=305, bal=145, syr=245, new=150),
32     new=dict(syr=260, phi=150, bos=270, pro=260),
33     pro=dict(bos=90, new=260),
34     bos=dict(pro=90, new=270, syr=290, por=120),
35     por=dict(bos=120))
36
37  def path(previous, s):
38      '''
39      'previous' is a dictionary chaining together the predecessor state that led to each state
40      's' will be None for the initial state
41      otherwise, start from the last state 's' and recursively trace 'previous' back to the initial
42      state,
43      constructing a list of states visited as we go
44      '''
45      if s is None:
46          return []
47      else:
48          return path(previous, previous[s])+[s]
49
50  def pathcost(path, step_costs):
51      '''
52      add up the step costs along a path, which is assumed to be a list output from the 'path' function
53      above
54      '''
55      cost = 0
56      for s in range(len(path)-1):
57          cost += step_costs[path[s]][path[s+1]]
58      return cost
59
60  # Solution:
61  """ Frontier_PQ - Implements a priority queue ordered by path cost for uniform cost search
62  Methods:
63      __init__ - Initializes an empty priority queue
64      is_empty - Checks if the priority queue is empty
65      put - Adds an item with a specified priority to the priority queue
66      get - Removes and returns the item with the lowest priority from the priority queue
67  Algorithm:
68      * __init__ initializes an empty list to represent the priority queue
69      * is_empty returns True if the list is empty, otherwise False
70      * put uses heapq.heappush to add an item to the priority queue with the given priority
71      * get uses heapq.heappop to remove and return the item with the lowest priority

```

```

70     Output:
71     * is_empty returns a boolean indicating whether the priority queue is empty
72     * put does not return a value
73     * get returns the item with the lowest priority
74     """
75     class Frontier_PQ:
76     ''' frontier class for uniform search, ordered by path cost '''
77     # add your code here
78     def __init__(self):
79         self.elements = []
80     def is_empty(self):
81         return len(self.elements) == 0
82     def put(self, item, priority):
83         heapq.heappush(self.elements, (priority, item))
84     def get(self):
85         return heapq.heappop(self.elements)
86
87
88 # Solution:
89 """ uniform_cost - Performs a Uniform Cost Search (UCS) on the state graph for the path between a
90 start and goal
91 Input:
92     start - Node that represents the start of the path
93     goal - Node that represents the desired end point of the path
94     state_graph - Dictionary representing the graph being searched, with costs for each edge
95     return_cost - Boolean value that indicates whether to return the cost of the path
96 Algorithm:
97     * Initialize a Frontier_PQ instance and add the start node with a priority of 0
98     * Initialize a dictionary of previous nodes with the start node set to None
99     * Initialize a dictionary to keep track of the cost to reach each node with the start node
100 set to 0
101     * While the priority queue is not empty
102     * Get the node with the lowest cost from the priority queue
103     * Check if that node is the goal
104     * If it is
105     * Update the path to the goal using the previous nodes and the goal
106     * Calculate the cost if return_cost is True
107     * Return the path to the goal and the cost if return_cost is set to True
108     * Otherwise, just return the path
109     * If it is not
110     * Iterate over the neighbors of the current node
111     * Calculate the new cost to reach each neighbor
112     * If the neighbor has not been visited or the new cost is lower than the recorded
113 cost
114     * Update the cost to reach the neighbor
115     * Add the neighbor to the priority queue with the new cost as priority
116     * Update the previous nodes with the current node
117     * Return None if the goal is not reachable or return (None, 0) if return_cost is True
118 Output:
119     Returns the path in the search as well as the cost in the path if return_cost is True
120 """
121 def uniform_cost(start, goal, state_graph, return_cost=False):
122     frontier = Frontier_PQ()
123     frontier.put(start, 0)
124     previous = {start: None}
125     cost_so_far = {start: 0}
126     while not frontier.is_empty():
127         current_priority, current = frontier.get()
128         if current == goal:
129             path_to_goal = path(previous, goal)
130             if return_cost:
131                 cost = pathcost(path_to_goal, state_graph)
132                 return path_to_goal, cost
133             else:
134                 return path_to_goal
135         for neighbor in state_graph[current]:
136             new_cost = cost_so_far[current] + state_graph[current][neighbor]
137             if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
138                 cost_so_far[neighbor] = new_cost
139                 priority = new_cost
140                 frontier.put(neighbor, priority)
141                 previous[neighbor] = current
142     return None if not return_cost else (None, 0)

```

# Problem 7

## Problem Statement

Use your choice of search function to show the list of cities that Neal would traverse to get to Chicago as quickly as possible.

## Solution

```

1  from collections import deque
2  import heapq
3
4  map_distances = dict(
5      chi=dict(det=283, cle=345, ind=182),
6      cle=dict(chi=345, det=169, col=144, pit=134, buf=189),
7      ind=dict(chi=182, col=176),
8      col=dict(ind=176, cle=144, pit=185),
9      det=dict(chi=283, cle=169, buf=256),
10     buf=dict(det=256, cle=189, pit=215, syr=150),
11     pit=dict(col=185, cle=134, buf=215, phi=305, bal=247),
12     syr=dict(buf=150, phi=253, new=254, bos=312),
13     bal=dict(phi=101, pit=247),
14     phi=dict(pit=305, bal=101, syr=253, new=97),
15     new=dict(syr=254, phi=97, bos=215, pro=181),
16     pro=dict(bos=50, new=181),
17     bos=dict(pro=50, new=215, syr=312, por=107),
18     por=dict(bos=107))
19
20
21  map_times = dict(
22     chi=dict(det=280, cle=345, ind=200),
23     cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
24     ind=dict(chi=200, col=175),
25     col=dict(ind=175, cle=155, pit=185),
26     det=dict(chi=280, cle=170, buf=270),
27     buf=dict(det=270, cle=185, pit=215, syr=145),
28     pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
29     syr=dict(buf=145, phi=245, new=260, bos=290),
30     bal=dict(phi=145, pit=255),
31     phi=dict(pit=305, bal=145, syr=245, new=150),
32     new=dict(syr=260, phi=150, bos=270, pro=260),
33     pro=dict(bos=90, new=260),
34     bos=dict(pro=90, new=270, syr=290, por=120),
35     por=dict(bos=120))
36
37  def path(previous, s):
38      '''
39      'previous' is a dictionary chaining together the predecessor state that led to each state
40      's' will be None for the initial state
41      otherwise, start from the last state 's' and recursively trace 'previous' back to the initial
42      state,
43      constructing a list of states visited as we go
44      '''
45      if s is None:
46          return []
47      else:
48          return path(previous, previous[s])+[s]
49
50  def pathcost(path, step_costs):
51      '''
52      add up the step costs along a path, which is assumed to be a list output from the 'path' function
53      above
54      '''
55      cost = 0
56      for s in range(len(path)-1):
57          cost += step_costs[path[s]][path[s+1]]
58      return cost
59
60  # Solution:
61  """ Frontier_PQ - Implements a priority queue ordered by path cost for uniform cost search
62  Methods:
63      __init__ - Initializes an empty priority queue
64      is_empty - Checks if the priority queue is empty
65      put - Adds an item with a specified priority to the priority queue
66      get - Removes and returns the item with the lowest priority from the priority queue
67  Algorithm:
68      * __init__ initializes an empty list to represent the priority queue
69      * is_empty returns True if the list is empty, otherwise False
70      * put uses heapq.heappush to add an item to the priority queue with the given priority
71      * get uses heapq.heappop to remove and return the item with the lowest priority

```



```

70     Output:
71     * is_empty returns a boolean indicating whether the priority queue is empty
72     * put does not return a value
73     * get returns the item with the lowest priority
74     """
75     class Frontier_PQ:
76     ''' frontier class for uniform search, ordered by path cost '''
77     # add your code here
78     def __init__(self):
79         self.elements = []
80     def is_empty(self):
81         return len(self.elements) == 0
82     def put(self, item, priority):
83         heapq.heappush(self.elements, (priority, item))
84     def get(self):
85         return heapq.heappop(self.elements)
86
87     # Solution:
88     """ uniform_cost - Performs a Uniform Cost Search (UCS) on the state graph for the path between a
89     start and goal
90     Input:
91     start - Node that represents the start of the path
92     goal - Node that represents the desired end point of the path
93     state_graph - Dictionary representing the graph being searched, with costs for each edge
94     return_cost - Boolean value that indicates whether to return the cost of the path
95     Algorithm:
96     * Initialize a Frontier_PQ instance and add the start node with a priority of 0
97     * Initialize a dictionary of previous nodes with the start node set to None
98     * Initialize a dictionary to keep track of the cost to reach each node with the start node
99     set to 0
100     * While the priority queue is not empty
101     * Get the node with the lowest cost from the priority queue
102     * Check if that node is the goal
103     * If it is
104     * Update the path to the goal using the previous nodes and the goal
105     * Calculate the cost if return_cost is True
106     * Return the path to the goal and the cost if return_cost is set to True
107     * Otherwise, just return the path
108     * If it is not
109     * Iterate over the neighbors of the current node
110     * Calculate the new cost to reach each neighbor
111     * If the neighbor has not been visited or the new cost is lower than the recorded
112     cost
113     * Update the cost to reach the neighbor
114     * Add the neighbor to the priority queue with the new cost as priority
115     * Update the previous nodes with the current node
116     * Return None if the goal is not reachable or return (None, 0) if return_cost is True
117     Output:
118     Returns the path in the search as well as the cost in the path if return_cost is True
119     """
120     def uniform_cost(start, goal, state_graph, return_cost=False):
121         frontier = Frontier_PQ()
122         frontier.put(start, 0)
123         previous = {start: None}
124         cost_so_far = {start: 0}
125         while not frontier.is_empty():
126             current_priority, current = frontier.get()
127             if current == goal:
128                 path_to_goal = path(previous, goal)
129                 if return_cost:
130                     cost = pathcost(path_to_goal, state_graph)
131                     return path_to_goal, cost
132                 else:
133                     return path_to_goal
134             for neighbor in state_graph[current]:
135                 new_cost = cost_so_far[current] + state_graph[current][neighbor]
136                 if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
137                     cost_so_far[neighbor] = new_cost
138                     priority = new_cost
139                     frontier.put(neighbor, priority)
140                     previous[neighbor] = current
141         return None if not return_cost else (None, 0)

```

# Problem 8

## Problem Statement

Pass the maze-to-graph unit test.

## Solution

```

1  import numpy as np
2
3  """ maze_to_graph - Converts a maze represented as a numpy array into a graph
4  Input:
5      maze - 2D numpy array where 0 represents walkable cells and 1 represents walls
6  Algorithm:
7      * Initialize an empty dictionary to represent the graph
8      * Define the directions for North, South, East, and West movements
9      * Iterate over each cell in the maze
10         * Initialize an empty dictionary for each cell in the graph
11         * For each direction, calculate the neighboring cell's coordinates
12         * Check if the neighboring cell is within the maze bounds and is walkable (contains 0)
13         * If it is, add the neighbor to the current cell's dictionary in the graph with the
14           direction as the value
15  Output:
16      Returns a dictionary representing the graph where keys are coordinates of cells and values
17      are dictionaries
18      of neighboring cells with directions
19  """
20  def maze_to_graph(maze):
21      """ takes in a maze as a numpy array, converts to a graph """
22      graph = {}
23      rows, cols = maze.shape
24      directions = {
25          'N': (1, 0), # North
26          'S': (-1, 0), # South
27          'E': (0, 1), # East
28          'W': (0, -1) # West
29      }
30      for r in range(rows):
31          for c in range(cols):
32              graph[(c, r)] = {}
33              for direction, (dr, dc) in directions.items():
34                  nr, nc = r + dr, c + dc
35                  if 0 <= nr < rows and 0 <= nc < cols and maze[nr, nc] == 0:
36                      graph[(c, r)][(nc, nr)] = direction
37      return graph

```

# Problem 9

## Problem Statement

Use your depth-first search function to solve the maze and provide the solution path.

## Solution

```

1  import numpy as np
2  from collections import OrderedDict
3  from collections import deque
4  maze = np.array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
5                  [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
6                  [1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
7                  [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
8                  [1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1],
9                  [1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1],
10                 [1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1],
11                 [1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1],
12                 [1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1],
13                 [1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1],
14                 [1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1],
15                 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
16
17  map_distances = dict(
18      chi=OrderedDict([("det",283), ("cle",345), ("ind",182)]),
19      cle=OrderedDict([("chi",345), ("det",169), ("col",144), ("pit",134), ("buf",189)]),
20      ind=OrderedDict([("chi",182), ("col",176)]),
21      col=OrderedDict([("ind",176), ("cle",144), ("pit",185)]),
22      det=OrderedDict([("chi",283), ("cle",169), ("buf",256)]),
23      buf=OrderedDict([("det",256), ("cle",189), ("pit",215), ("syr",150)]),
24      pit=OrderedDict([("col",185), ("cle",134), ("buf",215), ("phi",305), ("bal",247)]),
25      syr=OrderedDict([("buf",150), ("phi",253), ("new",254), ("bos",312)]),
26      bal=OrderedDict([("phi",101), ("pit",247)]),
27      phi=OrderedDict([("pit",305), ("bal",101), ("syr",253), ("new",97)]),
28      new=OrderedDict([("syr",254), ("phi",97), ("bos",215), ("pro",181)]),
29      pro=OrderedDict([("bos",50), ("new",181)]),
30      bos=OrderedDict([("pro",50), ("new",215), ("syr",312), ("por",107)]),
31      por=OrderedDict([("bos",107)]))
32
33  map_times = dict(
34      chi=dict(det=280, cle=345, ind=200),
35      cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
36      ind=dict(chi=200, col=175),
37      col=dict(ind=175, cle=155, pit=185),
38      det=dict(chi=280, cle=170, buf=270),
39      buf=dict(det=270, cle=185, pit=215, syr=145),
40      pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
41      syr=dict(buf=145, phi=245, new=260, bos=290),
42      bal=dict(phi=145, pit=255),
43      phi=dict(pit=305, bal=145, syr=245, new=150),
44      new=dict(syr=260, phi=150, bos=270, pro=260),
45      pro=dict(bos=90, new=260),
46      bos=dict(pro=90, new=270, syr=290, por=120),
47      por=dict(bos=120))
48
49  def path(previous, s):
50      '''
51      'previous' is a dictionary chaining together the predecessor state that led to each state
52      's' will be None for the initial state
53      otherwise, start from the last state 's' and recursively trace 'previous' back to the initial
54      state,
55      constructing a list of states visited as we go
56      '''
57      if s is None:
58          return []
59      else:
60          return path(previous, previous[s])+[s]
61
62  def pathcost(path, step_costs):
63      '''
64      add up the step costs along a path, which is assumed to be a list output from the 'path' function
65      above
66      '''
67      cost = 0
68      for s in range(len(path)-1):
69          cost += step_costs[path[s]][path[s+1]]
70      return cost

```

```

71 """ depth_first - Performs a DFS on the state graph for the path between a start and goal
72 Input:
73     start - Node that represents the start of the path
74     goal - Node that represents the desired end point of the path
75     state_graph - Represents the graph that is being searched in
76     return_cost - Boolean value that indicates the cost of the path
77 Algorithm:
78     * Create a stack of nodes with the start node as the original element in it
79     * Create a set of visited nodes with the start node as the original element in it
80     * Create a dictionary of previous nodes that is empty
81     * While the stack is not empty
82         * Pop the top most node from the stack
83         * Check if that node is the goal
84         * If it is
85             * Update the path to the goal with the previous nodes and the goal
86             * Return the path to goal and the cost if return_cost is set to true
87             * Otherwise, just return the path
88         * If it is not
89             * Iterate over the neighbors of the current node
90             * Add the neighbor to the visited set if it is not visited
91             * Update the previous nodes with the current node
92             * Add the neighbor to the stack
93     * Return the path and cost, or just the path if designated
94 Output:
95     Returns the path in the search as well as the cost in the path
96 """
97 def depth_first(start, goal, state_graph, return_cost=False):
98     stack = [start]
99     visited = set([start])
100     previous = {start: None}
101     while (stack):
102         current = stack.pop()
103         if (current == goal):
104             path_to_goal = path(previous, goal)
105             if (return_cost):
106                 cost = pathcost(path_to_goal, state_graph)
107                 return path_to_goal, cost
108             else:
109                 return path_to_goal
110         for neighbor in state_graph[current]:
111             if (neighbor not in visited):
112                 visited.add(neighbor)
113                 previous[neighbor] = current
114                 stack.append(neighbor)
115     return None if not return_cost else (None, 0)
116
117 # Solution:
118
119 """ maze_to_graph - Converts a maze represented as a numpy array into a graph
120 Input:
121     maze - 2D numpy array where 0 represents walkable cells and 1 represents walls
122 Algorithm:
123     * Initialize an empty dictionary to represent the graph
124     * Define the directions for North, South, East, and West movements
125     * Iterate over each cell in the maze
126         * Initialize an empty dictionary for each cell in the graph
127         * For each direction, calculate the neighboring cell's coordinates
128         * Check if the neighboring cell is within the maze bounds and is walkable (contains 0)
129         * If it is, add the neighbor to the current cell's dictionary in the graph with the
130 direction as the value
131 Output:
132     Returns a dictionary representing the graph where keys are coordinates of cells and values
133 are dictionaries
134     of neighboring cells with directions
135 """
136 def maze_to_graph(maze):
137     ''' takes in a maze as a numpy array, converts to a graph '''
138     graph = {}
139     rows, cols = maze.shape
140     directions = {
141         'N': (1, 0), # North
142         'S': (-1, 0), # South
143         'E': (0, 1), # East
144         'W': (0, -1) # West
145     }
146     for r in range(rows):
147         for c in range(cols):
148             graph[(c, r)] = {}
149             for direction, (dr, dc) in directions.items():
150                 nr, nc = r + dr, c + dc
151                 if 0 <= nr < rows and 0 <= nc < cols and maze[nr, nc] == 0:
152                     graph[(c, r)][(nc, nr)] = direction
153     return graph

```

# Problem 10

## Problem Statement

Use your breadth-first search function to solve the maze and provide the solution path and its length.

## Solution

```

1  import numpy as np
2  maze = np.array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
3                  [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
4                  [1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
5                  [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
6                  [1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1],
7                  [1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1],
8                  [1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1],
9                  [1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1],
10                 [1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1],
11                 [1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1],
12                 [1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1],
13                 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
14
15 #####
16 from collections import deque
17
18 map_distances = dict(
19     chi=dict(det=283, cle=345, ind=182),
20     cle=dict(chi=345, det=169, col=144, pit=134, buf=189),
21     ind=dict(chi=182, col=176),
22     col=dict(ind=176, cle=144, pit=185),
23     det=dict(chi=283, cle=169, buf=256),
24     buf=dict(det=256, cle=189, pit=215, syr=150),
25     pit=dict(col=185, cle=134, buf=215, phi=305, bal=247),
26     syr=dict(buf=150, phi=253, new=254, bos=312),
27     bal=dict(phi=101, pit=247),
28     phi=dict(pit=305, bal=101, syr=253, new=97),
29     new=dict(syr=254, phi=97, bos=215, pro=181),
30     pro=dict(bos=50, new=181),
31     bos=dict(pro=50, new=215, syr=312, por=107),
32     por=dict(bos=107))
33
34
35 map_times = dict(
36     chi=dict(det=280, cle=345, ind=200),
37     cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
38     ind=dict(chi=200, col=175),
39     col=dict(ind=175, cle=155, pit=185),
40     det=dict(chi=280, cle=170, buf=270),
41     buf=dict(det=270, cle=185, pit=215, syr=145),
42     pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
43     syr=dict(buf=145, phi=245, new=260, bos=290),
44     bal=dict(phi=145, pit=255),
45     phi=dict(pit=305, bal=145, syr=245, new=150),
46     new=dict(syr=260, phi=150, bos=270, pro=260),
47     pro=dict(bos=90, new=260),
48     bos=dict(pro=90, new=270, syr=290, por=120),
49     por=dict(bos=120))
50
51 def path(previous, s):
52     '''
53     'previous' is a dictionary chaining together the predecessor state that led to each state
54     's' will be None for the initial state
55     otherwise, start from the last state 's' and recursively trace 'previous' back to the initial
56     state,
57     constructing a list of states visited as we go
58     '''
59     if s is None:
60         return []
61     else:
62         return path(previous, previous[s])+[s]
63
64 def pathcost(path, step_costs):
65     '''
66     add up the step costs along a path, which is assumed to be a list output from the 'path' function
67     above
68     '''
69     cost = 0
70     for s in range(len(path)-1):
71         cost += step_costs[path[s]][path[s+1]]
72     return cost

```

```

71
72
73 """ breadth_first - Performs a breadth first search on cities
74 Input:
75     start - Node that represents the start of the path
76     goal - Node that represents the desired end point of the path
77     state_graph - Represents the graph that is being searched in
78     return_cost - Boolean value that indicates the cost of the path
79 Algorithm:
80     * Create a queue with the start node as the first node
81     * Create a visited set where the first node is visited
82     * Create a dictionary for the previous nodes that have been visited
83     * While the queue is not empty:
84         * Pop the current node from the queue
85         * If we reach the goal
86             * Create a path with the previous nodes and the goal
87             * Return the path and the cost if return_cost is set to true, otherwise just the path
88         * Iterate over the neighbors of the current node
89         * Add the neighbor to the visited set if it isn't visited
90         * Update the previous node with the current node
91         * Add the neighbor to the queue
92     * Return the cost of the traversal
93 Output:
94     Returns the path in the search as well as the cost in the path
95 """
96 def breadth_first(start, goal, state_graph, return_cost=False):
97     queue = deque([start])
98     visited = set([start])
99     previous = {start: None}
100     while (queue):
101         current = queue.popleft()
102         if (current == goal):
103             path_to_goal = path(previous, goal)
104             if (return_cost):
105                 cost = pathcost(path_to_goal, state_graph)
106                 return path_to_goal, cost
107             else:
108                 return path_to_goal
109         for neighbor in state_graph[current]:
110             if (neighbor not in visited):
111                 visited.add(neighbor)
112                 previous[neighbor] = current
113                 queue.append(neighbor)
114     return None if not return_cost else (None, 0)
115
116 # Solution:
117
118 """ maze_to_graph - Converts a maze represented as a numpy array into a graph
119 Input:
120     maze - 2D numpy array where 0 represents walkable cells and 1 represents walls
121 Algorithm:
122     * Initialize an empty dictionary to represent the graph
123     * Define the directions for North, South, East, and West movements
124     * Iterate over each cell in the maze
125         * Initialize an empty dictionary for each cell in the graph
126         * For each direction, calculate the neighboring cell's coordinates
127         * Check if the neighboring cell is within the maze bounds and is walkable (contains 0)
128         * If it is, add the neighbor to the current cell's dictionary in the graph with the
129 direction as the value
130 Output:
131     Returns a dictionary representing the graph where keys are coordinates of cells and values
132 are dictionaries
133     of neighboring cells with directions
134 """
135 def maze_to_graph(maze):
136     """ takes in a maze as a numpy array, converts to a graph """
137     graph = {}
138     rows, cols = maze.shape
139     directions = {
140         'N': (1, 0), # North
141         'S': (-1, 0), # South
142         'E': (0, 1), # East
143         'W': (0, -1) # West
144     }
145     for r in range(rows):
146         for c in range(cols):
147             graph[(c, r)] = {}
148             for direction, (dr, dc) in directions.items():
149                 nr, nc = r + dr, c + dc
150                 if 0 <= nr < rows and 0 <= nc < cols and maze[nr, nc] == 0:
151                     graph[(c, r)][(nc, nr)] = direction
152     return graph

```