College of Engineering & Applied Sciences

CSPB 3202

Introduction To Artifical Intelligence

Assignment 1 - Intro To AI, Search Problems

UNIVERSITY OF COLORADO

2024

Introduction To Artifical Intelligence - Assignment 1 - Intro To AI, Search Problems

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

```python
from collections import deque
from collections import OrderedDict
map_distances = dict(
    chi=OrderedDict([("det",283), ("cle",345), ("ind",182)]),
    cle=OrderedDict([("chi",345), ("det",169), ("col",144), ("pit",134), ("buf",189)]),
    ind=OrderedDict([("chi",182), ("col",176)]),
    col=OrderedDict([("ind",176), ("cle",144), ("pit",185)]),
    det=OrderedDict([("chi",283), ("cle",169), ("buf",256)]),
    buf=OrderedDict([("det",256), ("cle",189), ("pit",215), ("syr",150)]),
    pit=OrderedDict([("col",185), ("cle",134), ("buf",215), ("phi",305), ("bal",247)]),
    syr=OrderedDict([("buf",150), ("phi",253), ("new",254), ("bos",312)]),
    bal=OrderedDict([("phi",101), ("pit",247)]),
    phi=OrderedDict([("pit",305), ("bal",101), ("syr",253), ("new",97)]),
    new=OrderedDict([("syr",254), ("phi",97), ("bos",215), ("pro",181)]),
    pro=OrderedDict([("bos",50), ("new",181)]),
    bos=OrderedDict([("pro",50), ("new",215), ("syr",312), ("por",107)]),
    por=OrderedDict([("bos",107)]))

map_times = dict(
    chi=dict(det=280, cle=345, ind=200),
    cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
    ind=dict(chi=200, col=175),
    col=dict(ind=175, cle=155, pit=185),
    det=dict(chi=280, cle=170, buf=270),
    buf=dict(det=270, cle=185, pit=215, syr=145),
    pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
    syr=dict(buf=145, phi=245, new=260, bos=290),
    bal=dict(phi=145, pit=255),
    phi=dict(pit=305, bal=145, syr=245, new=150),
    new=dict(syr=260, phi=150, bos=270, pro=260),
    pro=dict(bos=90, new=260),
    bos=dict(pro=90, new=270, syr=290, por=120),
    por=dict(bos=120))

def path(previous, s):
    '''
    `previous` is a dictionary chaining together the predecessor state that led to each state
    `s` will be None for the initial state
    otherwise, start from the last state `s` and recursively trace `previous` back to the initial state,
    constructing a list of states visited as we go
    '''
    if s is None:
        return []
    else:
        return path(previous, previous[s])+[s]

def pathcost(path, step_costs):
    '''
    add up the step costs along a path, which is assumed to be a list output from the `path` function above
    '''
    cost = 0
    for s in range(len(path)-1):
        cost += step_costs[path[s]][path[s+1]]
    return cost


# Solution:

""" breadth_first - Performs a breadth first search on cities
    Input:
        start - Node that represents the start of the path
        goal - Node that represents the desired end point of the path
        state_graph - Represents the graph that is being searched in
        return_cost - Boolean value that indicates the cost of the path
    Algorithm:
        * Create a queue with the start node as the first node
        * Create a visited set where the first node is visited
        * Create a dictionary for the previous nodes that have been visited
        * While the queue is not empty:
```

```
            * Pop the current node from the queue
            * If we reach the goal
                * Create a path with the previous nodes and the goal
                * Return the path and the cost if return_cost is set to true, otherwise just the path
            * Iterate over the neighbors of the current node
            * Add the neighbor to the visited set if it isn't visited
            * Update the previous node with the current node
            * Add the neighbor to the queue
        * Return the cost of the traversal
    Output:
        Returns the path in the search as well as the cost in the path
"""
def breadth_first(start, goal, state_graph, return_cost=False):
    queue = deque([start])
    visited = set([start])
    previous = {start: None}
    while (queue):
        current = queue.popleft()
        if (current == goal):
            path_to_goal = path(previous, goal)
            if (return_cost):
                cost = pathcost(path_to_goal, state_graph)
                return path_to_goal, cost
            else:
                return path_to_goal
        for neighbor in state_graph[current]:
            if (neighbor not in visited):
                visited.add(neighbor)
                previous[neighbor] = current
                queue.append(neighbor)
    return None if not return_cost else (None, 0)
```

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

```python
from collections import deque
from collections import OrderedDict

map_distances = dict(
    chi=OrderedDict([("det",283), ("cle",345), ("ind",182)]),
    cle=OrderedDict([("chi",345), ("det",169), ("col",144), ("pit",134), ("buf",189)]),
    ind=OrderedDict([("chi",182), ("col",176)]),
    col=OrderedDict([("ind",176), ("cle",144), ("pit",185)]),
    det=OrderedDict([("chi",283), ("cle",169), ("buf",256)]),
    buf=OrderedDict([("det",189), ("pit",215), ("syr",150)]),
    pit=OrderedDict([("col",185), ("cle",134), ("buf",215), ("phi",305), ("bal",247)]),
    syr=OrderedDict([("buf",150), ("phi",253), ("new",254), ("bos",312)]),
    bal=OrderedDict([("phi",101), ("pit",247)]),
    phi=OrderedDict([("pit",305), ("bal",101), ("syr",253), ("new",97)]),
    new=OrderedDict([("syr",254), ("phi",97), ("bos",215), ("pro",181)]),
    pro=OrderedDict([("bos",50), ("new",181)]),
    bos=OrderedDict([("pro",50), ("new",215), ("syr",312), ("por",107)]),
    por=OrderedDict([("bos",107)]))


map_times = dict(
    chi=dict(det=280, cle=345, ind=200),
    cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
    ind=dict(chi=200, col=175),
    col=dict(ind=175, cle=155, pit=185),
    det=dict(chi=280, cle=170, buf=270),
    buf=dict(det=270, cle=185, pit=215, syr=145),
    pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
    syr=dict(buf=145, phi=245, new=260, bos=290),
    bal=dict(phi=145, pit=255),
    phi=dict(pit=305, bal=145, syr=245, new=150),
    new=dict(syr=260, phi=150, bos=270, pro=260),
    pro=dict(bos=90, new=260),
    bos=dict(pro=90, new=270, syr=290, por=120),
    por=dict(bos=120))

def path(previous, s):
    '''
    `previous` is a dictionary chaining together the predecessor state that led to each state
    `s` will be None for the initial state
    otherwise, start from the last state `s` and recursively trace `previous` back to the initial
 state,
    constructing a list of states visited as we go
    '''
    if s is None:
        return []
    else:
        return path(previous, previous[s])+[s]

def pathcost(path, step_costs):
    '''
    add up the step costs along a path, which is assumed to be a list output from the `path` function
  above
    '''
    cost = 0
    for s in range(len(path)-1):
        cost += step_costs[path[s]][path[s+1]]
    return cost

""" depth_first - Performs a DFS on the state graph for the path between a start and goal
    Input:
        start - Node that represents the start of the path
        goal - Node that represents the desired end point of the path
        state_graph - Represents the graph that is being searched in
        return_cost - Boolean value that indicates the cost of the path
    Algorithm:
        * Create a stack of nodes with the start node as the original element in it
        * Create a set of visited nodes with the start node as the original element in it
        * Create a dictionary of previous nodes that is empty
        * While the stack is not empty
            * Pop the top most node from the stack
```

```
                * Check if that node is the goal
                * If it is
                    * Update the path to the goal with the previous nodes and the goal
                    * Return the path to goal and the cost if return_cost is set to true
                    * Otherwise, just return the path
                * If it is not
                    * Iterate over the neighbors of the current node
                    * Add the neighbor to the visited set if it is not visited
                    * Update the previous nodes with the current node
                    * Add the neighbor to the stack
            * Return the path and cost, or just the path if designated
        Output:
            Returns the path in the search as well as the cost in the path
    """
    def depth_first(start, goal, state_graph, return_cost=False):
        stack = [start]
        visited = set([start])
        previous = {start: None}
        while (stack):
            current = stack.pop()
            if (current == goal):
                path_to_goal = path(previous, goal)
                if (return_cost):
                    cost = pathcost(path_to_goal, state_graph)
                    return path_to_goal, cost
                else:
                    return path_to_goal
            for neighbor in state_graph[current]:
                if (neighbor not in visited):
                    visited.add(neighbor)
                    previous[neighbor] = current
                    stack.append(neighbor)
        return None if not return_cost else (None, 0)

```

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

```python
from collections import deque
import heapq

map_distances = dict(
    chi=dict(det=283, cle=345, ind=182),
    cle=dict(chi=345, det=169, col=144, pit=134, buf=189),
    ind=dict(chi=182, col=176),
    col=dict(ind=176, cle=144, pit=185),
    det=dict(chi=283, cle=169, buf=256),
    buf=dict(det=256, cle=189, pit=215, syr=150),
    pit=dict(col=185, cle=134, buf=215, phi=305, bal=247),
    syr=dict(buf=150, phi=253, new=254, bos=312),
    bal=dict(phi=101, pit=247),
    phi=dict(pit=305, bal=101, syr=253, new=97),
    new=dict(syr=254, phi=97, bos=215, pro=181),
    pro=dict(bos=50, new=181),
    bos=dict(pro=50, new=215, syr=312, por=107),
    por=dict(bos=107))


map_times = dict(
    chi=dict(det=280, cle=345, ind=200),
    cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
    ind=dict(chi=200, col=175),
    col=dict(ind=175, cle=155, pit=185),
    det=dict(chi=280, cle=170, buf=270),
    buf=dict(det=270, cle=185, pit=215, syr=145),
    pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
    syr=dict(buf=145, phi=245, new=260, bos=290),
    bal=dict(phi=145, pit=255),
    phi=dict(pit=305, bal=145, syr=245, new=150),
    new=dict(syr=260, phi=150, bos=270, pro=260),
    pro=dict(bos=90, new=260),
    bos=dict(pro=90, new=270, syr=290, por=120),
    por=dict(bos=120))

def path(previous, s):
    '''
    `previous` is a dictionary chaining together the predecessor state that led to each state
    `s` will be None for the initial state
    otherwise, start from the last state `s` and recursively trace `previous` back to the initial state,
    constructing a list of states visited as we go
    '''
    if s is None:
        return []
    else:
        return path(previous, previous[s])+[s]

def pathcost(path, step_costs):
    '''
    add up the step costs along a path, which is assumed to be a list output from the `path` function above
    '''
    cost = 0
    for s in range(len(path)-1):
        cost += step_costs[path[s]][path[s+1]]
    return cost

# Solution:
""" Frontier_PQ - Implements a priority queue ordered by path cost for uniform cost search
    Methods:
        __init__ - Initializes an empty priority queue
        is_empty - Checks if the priority queue is empty
        put - Adds an item with a specified priority to the priority queue
        get - Removes and returns the item with the lowest priority from the priority queue
    Algorithm:
        * __init__ initializes an empty list to represent the priority queue
```

```
            * is_empty returns True if the list is empty, otherwise False
            * put uses heapq.heappush to add an item to the priority queue with the given priority
            * get uses heapq.heappop to remove and return the item with the lowest priority
        Output:
            * is_empty returns a boolean indicating whether the priority queue is empty
            * put does not return a value
            * get returns the item with the lowest priority
"""
class Frontier_PQ:
    ''' frontier class for uniform search, ordered by path cost '''
    # add your code here
    def __init__(self):
        self.elements = []
    def is_empty(self):
        return len(self.elements) == 0
    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))
    def get(self):
        return heapq.heappop(self.elements)

# Solution:
""" uniform_cost - Performs a Uniform Cost Search (UCS) on the state graph for the path between a
 start and goal
    Input:
        start - Node that represents the start of the path
        goal - Node that represents the desired end point of the path
        state_graph - Dictionary representing the graph being searched, with costs for each edge
        return_cost - Boolean value that indicates whether to return the cost of the path
    Algorithm:
        * Initialize a Frontier_PQ instance and add the start node with a priority of 0
        * Initialize a dictionary of previous nodes with the start node set to None
        * Initialize a dictionary to keep track of the cost to reach each node with the start node
 set to 0
        * While the priority queue is not empty
            * Get the node with the lowest cost from the priority queue
            * Check if that node is the goal
            * If it is
                * Update the path to the goal using the previous nodes and the goal
                * Calculate the cost if return_cost is True
                * Return the path to the goal and the cost if return_cost is set to True
                * Otherwise, just return the path
            * If it is not
                * Iterate over the neighbors of the current node
                * Calculate the new cost to reach each neighbor
                * If the neighbor has not been visited or the new cost is lower than the recorded
 cost
                    * Update the cost to reach the neighbor
                    * Add the neighbor to the priority queue with the new cost as priority
                    * Update the previous nodes with the current node
        * Return None if the goal is not reachable or return (None, 0) if return_cost is True
    Output:
        Returns the path in the search as well as the cost in the path if return_cost is True
"""
def uniform_cost(start, goal, state_graph, return_cost=False):
    frontier = Frontier_PQ()
    frontier.put(start, 0)
    previous = {start: None}
    cost_so_far = {start: 0}
    while not frontier.is_empty():
        current_priority, current = frontier.get()
        if current == goal:
            path_to_goal = path(previous, goal)
            if return_cost:
                cost = pathcost(path_to_goal, state_graph)
                return path_to_goal, cost
            else:
                return path_to_goal
        for neighbor in state_graph[current]:
            new_cost = cost_so_far[current] + state_graph[current][neighbor]
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost
                frontier.put(neighbor, priority)
                previous[neighbor] = current
    return None if not return_cost else (None, 0)
```

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

```
from collections import deque
from collections import OrderedDict
import heapq
map_distances = dict(
    chi=OrderedDict([("det",283), ("cle",345), ("ind",182)]),
    cle=OrderedDict([("chi",345), ("det",169), ("col",144), ("pit",134), ("buf",189)]),
    ind=OrderedDict([("chi",182), ("col",176)]),
    col=OrderedDict([("ind",176), ("cle",144), ("pit",185)]),
    det=OrderedDict([("chi",283), ("cle",169), ("buf",256)]),
    buf=OrderedDict([("det",189), ("pit",215), ("syr",150)]),
    pit=OrderedDict([("col",185), ("cle",134), ("buf",215), ("phi",305), ("bal",247)]),
    syr=OrderedDict([("buf",150), ("phi",253), ("new",254), ("bos",312)]),
    bal=OrderedDict([("phi",101), ("pit",247)]),
    phi=OrderedDict([("pit",305), ("bal",101), ("syr",253), ("new",97)]),
    new=OrderedDict([("syr",254), ("phi",97), ("bos",215), ("pro",181)]),
    pro=OrderedDict([("bos",50), ("new",181)]),
    bos=OrderedDict([("pro",50), ("new",215), ("syr",312), ("por",107)]),
    por=OrderedDict([("bos",107)]))

map_times = dict(
    chi=dict(det=280, cle=345, ind=200),
    cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
    ind=dict(chi=200, col=175),
    col=dict(ind=175, cle=155, pit=185),
    det=dict(chi=280, cle=170, buf=270),
    buf=dict(det=270, cle=185, pit=215, syr=145),
    pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
    syr=dict(buf=145, phi=245, new=260, bos=290),
    bal=dict(phi=145, pit=255),
    phi=dict(pit=305, bal=145, syr=245, new=150),
    new=dict(syr=260, phi=150, bos=270, pro=260),
    pro=dict(bos=90, new=260),
    bos=dict(pro=90, new=270, syr=290, por=120),
    por=dict(bos=120))

def path(previous, s):
    '''
    `previous` is a dictionary chaining together the predecessor state that led to each state
    `s` will be None for the initial state
    otherwise, start from the last state `s` and recursively trace `previous` back to the initial
 state,
    constructing a list of states visited as we go
    '''
    if s is None:
        return []
    else:
        return path(previous, previous[s])+[s]

def pathcost(path, step_costs):
    '''
    add up the step costs along a path, which is assumed to be a list output from the `path` function
  above
    '''
    cost = 0
    for s in range(len(path)-1):
        cost += step_costs[path[s]][path[s+1]]
    return cost


# Solution:
""" Frontier_PQ - Implements a priority queue ordered by path cost for uniform cost search
    Methods:
        __init__ - Initializes an empty priority queue
        is_empty - Checks if the priority queue is empty
        put - Adds an item with a specified priority to the priority queue
        get - Removes and returns the item with the lowest priority from the priority queue
    Algorithm:
        * __init__ initializes an empty list to represent the priority queue
        * is_empty returns True if the list is empty, otherwise False
        * put uses heapq.heappush to add an item to the priority queue with the given priority
        * get uses heapq.heappop to remove and return the item with the lowest priority
```

```
    Output:
        * is_empty returns a boolean indicating whether the priority queue is empty
        * put does not return a value
        * get returns the item with the lowest priority
"""
class Frontier_PQ:
    ''' frontier class for uniform search, ordered by path cost '''
    # add your code here
    def __init__(self):
        self.elements = []
    def is_empty(self):
        return len(self.elements) == 0
    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))
    def get(self):
        return heapq.heappop(self.elements)


""" breadth_first - Performs a breadth first search on cities
    Input:
        start - Node that represents the start of the path
        goal - Node that represents the desired end point of the path
        state_graph - Represents the graph that is being searched in
        return_cost - Boolean value that indicates the cost of the path
    Algorithm:
        * Create a queue with the start node as the first node
        * Create a visited set where the first node is visited
        * Create a dictionary for the previous nodes that have been visited
        * While the queue is not empty:
            * Pop the current node from the queue
            * If we reach the goal
                * Create a path with the previous nodes and the goal
                * Return the path and the cost if return_cost is set to true, otherwise just the path
            * Iterate over the neighbors of the current node
            * Add the neighbor to the visited set if it isn't visited
            * Update the previous node with the current node
            * Add the neighbor to the queue
        * Return the cost of the traversal
    Output:
        Returns the path in the search as well as the cost in the path
"""
def breadth_first(start, goal, state_graph, return_cost=False):
    queue = deque([start])
    visited = set([start])
    previous = {start: None}
    while (queue):
        current = queue.popleft()
        if (current == goal):
            path_to_goal = path(previous, goal)
            if (return_cost):
                cost = pathcost(path_to_goal, state_graph)
                return path_to_goal, cost
            else:
                return path_to_goal
        for neighbor in state_graph[current]:
            if (neighbor not in visited):
                visited.add(neighbor)
                previous[neighbor] = current
                queue.append(neighbor)
    return None if not return_cost else (None, 0)

""" depth_first - Performs a DFS on the state graph for the path between a start and goal
    Input:
        start - Node that represents the start of the path
        goal - Node that represents the desired end point of the path
        state_graph - Represents the graph that is being searched in
        return_cost - Boolean value that indicates the cost of the path
    Algorithm:
        * Create a stack of nodes with the start node as the original element in it
        * Create a set of visited nodes with the start node as the original element in it
        * Create a dictionary of previous nodes that is empty
        * While the stack is not empty
            * Pop the top most node from the stack
            * Check if that node is the goal
            * If it is
                * Update the path to the goal with the previous nodes and the goal
                * Return the path to goal and the cost if return_cost is set to true
                * Otherwise, just return the path
            * If it is not
                * Iterate over the neighbors of the current node
                * Add the neighbor to the visited set if it is not visited
                * Update the previous nodes with the current node
                * Add the neighbor to the stack
        * Return the path and cost, or just the path if designated
    Output:
        Returns the path in the search as well as the cost in the path
"""
def depth_first(start, goal, state_graph, return_cost=False):
    stack = [start]
    visited = set([start])
    previous = {start: None}
    while (stack):
```

```
        current = stack.pop()
        if (current == goal):
            path_to_goal = path(previous, goal)
            if (return_cost):
                cost = pathcost(path_to_goal, state_graph)
                return path_to_goal, cost
            else:
                return path_to_goal
        for neighbor in state_graph[current]:
            if (neighbor not in visited):
                visited.add(neighbor)
                previous[neighbor] = current
                stack.append(neighbor)
    return None if not return_cost else (None, 0)

# Solution:
""" uniform_cost - Performs a Uniform Cost Search (UCS) on the state graph for the path between a
 start and goal
    Input:
        start - Node that represents the start of the path
        goal - Node that represents the desired end point of the path
        state_graph - Dictionary representing the graph being searched, with costs for each edge
        return_cost - Boolean value that indicates whether to return the cost of the path
    Algorithm:
        * Initialize a Frontier_PQ instance and add the start node with a priority of 0
        * Initialize a dictionary of previous nodes with the start node set to None
        * Initialize a dictionary to keep track of the cost to reach each node with the start node
 set to 0
        * While the priority queue is not empty
            * Get the node with the lowest cost from the priority queue
            * Check if that node is the goal
            * If it is
                * Update the path to the goal using the previous nodes and the goal
                * Calculate the cost if return_cost is True
                * Return the path to the goal and the cost if return_cost is set to True
                * Otherwise, just return the path
            * If it is not
                * Iterate over the neighbors of the current node
                * Calculate the new cost to reach each neighbor
                * If the neighbor has not been visited or the new cost is lower than the recorded
 cost
                    * Update the cost to reach the neighbor
                    * Add the neighbor to the priority queue with the new cost as priority
                    * Update the previous nodes with the current node
        * Return None if the goal is not reachable or return (None, 0) if return_cost is True
    Output:
        Returns the path in the search as well as the cost in the path if return_cost is True
"""
def uniform_cost(start, goal, state_graph, return_cost=False):
    frontier = Frontier_PQ()
    frontier.put(start, 0)
    previous = {start: None}
    cost_so_far = {start: 0}
    while not frontier.is_empty():
        current_priority, current = frontier.get()
        if current == goal:
            path_to_goal = path(previous, goal)
            if return_cost:
                cost = pathcost(path_to_goal, state_graph)
                return path_to_goal, cost
            else:
                return path_to_goal
        for neighbor in state_graph[current]:
            new_cost = cost_so_far[current] + state_graph[current][neighbor]
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost
                frontier.put(neighbor, priority)
                previous[neighbor] = current
    return None if not return_cost else (None, 0)
```

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

```
from collections import deque
import heapq

map_distances = dict(
    chi=dict(det=283, cle=345, ind=182),
    cle=dict(chi=345, det=169, col=144, pit=134, buf=189),
    ind=dict(chi=182, col=176),
    col=dict(ind=176, cle=144, pit=185),
    det=dict(chi=283, cle=169, buf=256),
    buf=dict(det=256, cle=189, pit=215, syr=150),
    pit=dict(col=185, cle=134, buf=215, phi=305, bal=247),
    syr=dict(buf=150, phi=253, new=254, bos=312),
    bal=dict(phi=101, pit=247),
    phi=dict(pit=305, bal=101, syr=253, new=97),
    new=dict(syr=254, phi=97, bos=215, pro=181),
    pro=dict(bos=50, new=181),
    bos=dict(pro=50, new=215, syr=312, por=107),
    por=dict(bos=107))


map_times = dict(
    chi=dict(det=280, cle=345, ind=200),
    cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
    ind=dict(chi=200, col=175),
    col=dict(ind=175, cle=155, pit=185),
    det=dict(chi=280, cle=170, buf=270),
    buf=dict(det=270, cle=185, pit=215, syr=145),
    pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
    syr=dict(buf=145, phi=245, new=260, bos=290),
    bal=dict(phi=145, pit=255),
    phi=dict(pit=305, bal=145, syr=245, new=150),
    new=dict(syr=260, phi=150, bos=270, pro=260),
    pro=dict(bos=90, new=260),
    bos=dict(pro=90, new=270, syr=290, por=120),
    por=dict(bos=120))

def path(previous, s):
    '''
    `previous` is a dictionary chaining together the predecessor state that led to each state
    `s` will be None for the initial state
    otherwise, start from the last state `s` and recursively trace `previous` back to the initial state,
    constructing a list of states visited as we go
    '''
    if s is None:
        return []
    else:
        return path(previous, previous[s])+[s]

def pathcost(path, step_costs):
    '''
    add up the step costs along a path, which is assumed to be a list output from the `path` function above
    '''
    cost = 0
    for s in range(len(path)-1):
        cost += step_costs[path[s]][path[s+1]]
    return cost

# Solution:
""" Frontier_PQ - Implements a priority queue ordered by path cost for uniform cost search
    Methods:
        __init__ - Initializes an empty priority queue
        is_empty - Checks if the priority queue is empty
        put - Adds an item with a specified priority to the priority queue
        get - Removes and returns the item with the lowest priority from the priority queue
    Algorithm:
        * __init__ initializes an empty list to represent the priority queue
        * is_empty returns True if the list is empty, otherwise False
        * put uses heapq.heappush to add an item to the priority queue with the given priority
        * get uses heapq.heappop to remove and return the item with the lowest priority
```

```python
    Output:
        * is_empty returns a boolean indicating whether the priority queue is empty
        * put does not return a value
        * get returns the item with the lowest priority
"""
class Frontier_PQ:
    ''' frontier class for uniform search, ordered by path cost '''
    # add your code here
    def __init__(self):
        self.elements = []
    def is_empty(self):
        return len(self.elements) == 0
    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))
    def get(self):
        return heapq.heappop(self.elements)


# Solution:
""" uniform_cost - Performs a Uniform Cost Search (UCS) on the state graph for the path between a start and goal
    Input:
        start - Node that represents the start of the path
        goal - Node that represents the desired end point of the path
        state_graph - Dictionary representing the graph being searched, with costs for each edge
        return_cost - Boolean value that indicates whether to return the cost of the path
    Algorithm:
        * Initialize a Frontier_PQ instance and add the start node with a priority of 0
        * Initialize a dictionary of previous nodes with the start node set to None
        * Initialize a dictionary to keep track of the cost to reach each node with the start node set to 0
        * While the priority queue is not empty
            * Get the node with the lowest cost from the priority queue
            * Check if that node is the goal
            * If it is
                * Update the path to the goal using the previous nodes and the goal
                * Calculate the cost if return_cost is True
                * Return the path to the goal and the cost if return_cost is set to True
                * Otherwise, just return the path
            * If it is not
                * Iterate over the neighbors of the current node
                * Calculate the new cost to reach each neighbor
                * If the neighbor has not been visited or the new cost is lower than the recorded cost
                    * Update the cost to reach the neighbor
                    * Add the neighbor to the priority queue with the new cost as priority
                    * Update the previous nodes with the current node
        * Return None if the goal is not reachable or return (None, 0) if return_cost is True
    Output:
        Returns the path in the search as well as the cost in the path if return_cost is True
"""
def uniform_cost(start, goal, state_graph, return_cost=False):
    frontier = Frontier_PQ()
    frontier.put(start, 0)
    previous = {start: None}
    cost_so_far = {start: 0}
    while not frontier.is_empty():
        current_priority, current = frontier.get()
        if current == goal:
            path_to_goal = path(previous, goal)
            if return_cost:
                cost = pathcost(path_to_goal, state_graph)
                return path_to_goal, cost
            else:
                return path_to_goal
        for neighbor in state_graph[current]:
            new_cost = cost_so_far[current] + state_graph[current][neighbor]
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost
                frontier.put(neighbor, priority)
                previous[neighbor] = current
    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

```
from collections import deque
import heapq

map_distances = dict(
    chi=dict(det=283, cle=345, ind=182),
    cle=dict(chi=345, det=169, col=144, pit=134, buf=189),
    ind=dict(chi=182, col=176),
    col=dict(ind=176, cle=144, pit=185),
    det=dict(chi=283, cle=169, buf=256),
    buf=dict(det=256, cle=189, pit=215, syr=150),
    pit=dict(col=185, cle=134, buf=215, phi=305, bal=247),
    syr=dict(buf=150, phi=253, new=254, bos=312),
    bal=dict(phi=101, pit=247),
    phi=dict(pit=305, bal=101, syr=253, new=97),
    new=dict(syr=254, phi=97, bos=215, pro=181),
    pro=dict(bos=50, new=181),
    bos=dict(pro=50, new=215, syr=312, por=107),
    por=dict(bos=107))


map_times = dict(
    chi=dict(det=280, cle=345, ind=200),
    cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
    ind=dict(chi=200, col=175),
    col=dict(ind=175, cle=155, pit=185),
    det=dict(chi=280, cle=170, buf=270),
    buf=dict(det=270, cle=185, pit=215, syr=145),
    pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
    syr=dict(buf=145, phi=245, new=260, bos=290),
    bal=dict(phi=145, pit=255),
    phi=dict(pit=305, bal=145, syr=245, new=150),
    new=dict(syr=260, phi=150, bos=270, pro=260),
    pro=dict(bos=90, new=260),
    bos=dict(pro=90, new=270, syr=290, por=120),
    por=dict(bos=120))

def path(previous, s):
    '''
    ‘previous‘ is a dictionary chaining together the predecessor state that led to each state
    ‘s‘ will be None for the initial state
    otherwise, start from the last state ‘s‘ and recursively trace ‘previous‘ back to the initial state,
    constructing a list of states visited as we go
    '''
    if s is None:
        return []
    else:
        return path(previous, previous[s])+[s]

def pathcost(path, step_costs):
    '''
    add up the step costs along a path, which is assumed to be a list output from the ‘path‘ function above
    '''
    cost = 0
    for s in range(len(path)-1):
        cost += step_costs[path[s]][path[s+1]]
    return cost

# Solution:
""" Frontier_PQ - Implements a priority queue ordered by path cost for uniform cost search
    Methods:
        __init__ - Initializes an empty priority queue
        is_empty - Checks if the priority queue is empty
        put - Adds an item with a specified priority to the priority queue
        get - Removes and returns the item with the lowest priority from the priority queue
    Algorithm:
        * __init__ initializes an empty list to represent the priority queue
        * is_empty returns True if the list is empty, otherwise False
        * put uses heapq.heappush to add an item to the priority queue with the given priority
        * get uses heapq.heappop to remove and return the item with the lowest priority
```

```python
    Output:
        * is_empty returns a boolean indicating whether the priority queue is empty
        * put does not return a value
        * get returns the item with the lowest priority
"""
class Frontier_PQ:
    ''' frontier class for uniform search, ordered by path cost '''
    # add your code here
    def __init__(self):
        self.elements = []
    def is_empty(self):
        return len(self.elements) == 0
    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))
    def get(self):
        return heapq.heappop(self.elements)

# Solution:
""" uniform_cost - Performs a Uniform Cost Search (UCS) on the state graph for the path between a start and goal
    Input:
        start - Node that represents the start of the path
        goal - Node that represents the desired end point of the path
        state_graph - Dictionary representing the graph being searched, with costs for each edge
        return_cost - Boolean value that indicates whether to return the cost of the path
    Algorithm:
        * Initialize a Frontier_PQ instance and add the start node with a priority of 0
        * Initialize a dictionary of previous nodes with the start node set to None
        * Initialize a dictionary to keep track of the cost to reach each node with the start node set to 0
        * While the priority queue is not empty
            * Get the node with the lowest cost from the priority queue
            * Check if that node is the goal
            * If it is
                * Update the path to the goal using the previous nodes and the goal
                * Calculate the cost if return_cost is True
                * Return the path to the goal and the cost if return_cost is set to True
                * Otherwise, just return the path
            * If it is not
                * Iterate over the neighbors of the current node
                * Calculate the new cost to reach each neighbor
                * If the neighbor has not been visited or the new cost is lower than the recorded cost
                    * Update the cost to reach the neighbor
                    * Add the neighbor to the priority queue with the new cost as priority
                    * Update the previous nodes with the current node
        * Return None if the goal is not reachable or return (None, 0) if return_cost is True
    Output:
        Returns the path in the search as well as the cost in the path if return_cost is True
"""
def uniform_cost(start, goal, state_graph, return_cost=False):
    frontier = Frontier_PQ()
    frontier.put(start, 0)
    previous = {start: None}
    cost_so_far = {start: 0}
    while not frontier.is_empty():
        current_priority, current = frontier.get()
        if current == goal:
            path_to_goal = path(previous, goal)
            if return_cost:
                cost = pathcost(path_to_goal, state_graph)
                return path_to_goal, cost
            else:
                return path_to_goal
        for neighbor in state_graph[current]:
            new_cost = cost_so_far[current] + state_graph[current][neighbor]
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost
                frontier.put(neighbor, priority)
                previous[neighbor] = current
    return None if not return_cost else (None, 0)
```

Problem 8

Problem Statement

Pass the maze-to-graph unit test.

Solution

```python
import numpy as np

""" maze_to_graph - Converts a maze represented as a numpy array into a graph
    Input:
        maze - 2D numpy array where 0 represents walkable cells and 1 represents walls
    Algorithm:
        * Initialize an empty dictionary to represent the graph
        * Define the directions for North, South, East, and West movements
        * Iterate over each cell in the maze
            * Initialize an empty dictionary for each cell in the graph
            * For each direction, calculate the neighboring cell's coordinates
            * Check if the neighboring cell is within the maze bounds and is walkable (contains 0)
                * If it is, add the neighbor to the current cell's dictionary in the graph with the direction as the value
    Output:
        Returns a dictionary representing the graph where keys are coordinates of cells and values are dictionaries
        of neighboring cells with directions
"""
def maze_to_graph(maze):
    ''' takes in a maze as a numpy array, converts to a graph '''
    graph = {}
    rows, cols = maze.shape
    directions = {
        'N': (1, 0),  # North
        'S': (-1, 0),   # South
        'E': (0, 1),   # East
        'W': (0, -1)   # West
    }
    for r in range(rows):
        for c in range(cols):
            graph[(c, r)] = {}
            for direction, (dr, dc) in directions.items():
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols and maze[nr, nc] == 0:
                    graph[(c, r)][(nc, nr)] = direction
    return graph

```

Problem 9

Problem Statement

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

Solution

```python
import numpy as np
from collections import OrderedDict
from collections import deque
maze = np.array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                 [1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
                 [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                 [1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1],
                 [1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1],
                 [1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1],
                 [1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1],
                 [1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1],
                 [1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1],
                 [1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1],
                 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

map_distances = dict(
    chi=OrderedDict([("det",283), ("cle",345), ("ind",182)]),
    cle=OrderedDict([("chi",345), ("det",169), ("col",144), ("pit",134), ("buf",189)]),
    ind=OrderedDict([("chi",182), ("col",176)]),
    col=OrderedDict([("ind",176), ("cle",144), ("pit",185)]),
    det=OrderedDict([("chi",283), ("cle",169), ("buf",256)]),
    buf=OrderedDict([("det",256), ("cle",189), ("pit",215), ("syr",150)]),
    pit=OrderedDict([("col",185), ("cle",134), ("buf",215), ("phi",305), ("bal",247)]),
    syr=OrderedDict([("buf",150), ("phi",253), ("new",254), ("bos",312)]),
    bal=OrderedDict([("phi",101), ("pit",247)]),
    phi=OrderedDict([("pit",305), ("bal",101), ("syr",253), ("new",97)]),
    new=OrderedDict([("syr",254), ("phi",97), ("bos",215), ("pro",181)]),
    pro=OrderedDict([("bos",50), ("new",181)]),
    bos=OrderedDict([("pro",50), ("new",215), ("syr",312), ("por",107)]),
    por=OrderedDict([("bos",107)]))

map_times = dict(
    chi=dict(det=280, cle=345, ind=200),
    cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
    ind=dict(chi=200, col=175),
    col=dict(ind=175, cle=155, pit=185),
    det=dict(chi=280, cle=170, buf=270),
    buf=dict(det=270, cle=185, pit=215, syr=145),
    pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
    syr=dict(buf=145, phi=245, new=260, bos=290),
    bal=dict(phi=145, pit=255),
    phi=dict(pit=305, bal=145, syr=245, new=150),
    new=dict(syr=260, phi=150, bos=270, pro=260),
    pro=dict(bos=90, new=260),
    bos=dict(pro=90, new=270, syr=290, por=120),
    por=dict(bos=120))

def path(previous, s):
    '''
    `previous` is a dictionary chaining together the predecessor state that led to each state
    `s` will be None for the initial state
    otherwise, start from the last state `s` and recursively trace `previous` back to the initial
 state,
    constructing a list of states visited as we go
    '''
    if s is None:
        return []
    else:
        return path(previous, previous[s])+[s]

def pathcost(path, step_costs):
    '''
    add up the step costs along a path, which is assumed to be a list output from the `path` function
  above
    '''
    cost = 0
    for s in range(len(path)-1):
        cost += step_costs[path[s]][path[s+1]]
    return cost
```

```
""" depth_first - Performs a DFS on the state graph for the path between a start and goal
    Input:
        start - Node that represents the start of the path
        goal - Node that represents the desired end point of the path
        state_graph - Represents the graph that is being searched in
        return_cost - Boolean value that indicates the cost of the path
    Algorithm:
        * Create a stack of nodes with the start node as the original element in it
        * Create a set of visited nodes with the start node as the original element in it
        * Create a dictionary of previous nodes that is empty
        * While the stack is not empty
            * Pop the top most node from the stack
            * Check if that node is the goal
            * If it is
                * Update the path to the goal with the previous nodes and the goal
                * Return the path to goal and the cost if return_cost is set to true
                * Otherwise, just return the path
            * If it is not
                * Iterate over the neighbors of the current node
                * Add the neighbor to the visited set if it is not visited
                * Update the previous nodes with the current node
                * Add the neighbor to the stack
        * Return the path and cost, or just the path if designated
    Output:
        Returns the path in the search as well as the cost in the path
"""
def depth_first(start, goal, state_graph, return_cost=False):
    stack = [start]
    visited = set([start])
    previous = {start: None}
    while (stack):
        current = stack.pop()
        if (current == goal):
            path_to_goal = path(previous, goal)
            if (return_cost):
                cost = pathcost(path_to_goal, state_graph)
                return path_to_goal, cost
            else:
                return path_to_goal
        for neighbor in state_graph[current]:
            if (neighbor not in visited):
                visited.add(neighbor)
                previous[neighbor] = current
                stack.append(neighbor)
    return None if not return_cost else (None, 0)


# Solution:

""" maze_to_graph - Converts a maze represented as a numpy array into a graph
    Input:
        maze - 2D numpy array where 0 represents walkable cells and 1 represents walls
    Algorithm:
        * Initialize an empty dictionary to represent the graph
        * Define the directions for North, South, East, and West movements
        * Iterate over each cell in the maze
            * Initialize an empty dictionary for each cell in the graph
            * For each direction, calculate the neighboring cell's coordinates
            * Check if the neighboring cell is within the maze bounds and is walkable (contains 0)
                * If it is, add the neighbor to the current cell's dictionary in the graph with the
 direction as the value
    Output:
        Returns a dictionary representing the graph where keys are coordinates of cells and values
 are dictionaries
        of neighboring cells with directions
"""
def maze_to_graph(maze):
    ''' takes in a maze as a numpy array, converts to a graph '''
    graph = {}
    rows, cols = maze.shape
    directions = {
        'N': (1, 0),  # North
        'S': (-1, 0),   # South
        'E': (0, 1),   # East
        'W': (0, -1)   # West
    }
    for r in range(rows):
        for c in range(cols):
            graph[(c, r)] = {}
            for direction, (dr, dc) in directions.items():
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols and maze[nr, nc] == 0:
                    graph[(c, r)][(nc, nr)] = direction
    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

```python
import numpy as np
maze = np.array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                 [1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
                 [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                 [1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1],
                 [1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1],
                 [1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1],
                 [1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1],
                 [1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1],
                 [1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1],
                 [1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1],
                 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

##################
from collections import deque

map_distances = dict(
    chi=dict(det=283, cle=345, ind=182),
    cle=dict(chi=345, det=169, col=144, pit=134, buf=189),
    ind=dict(chi=182, col=176),
    col=dict(ind=176, cle=144, pit=185),
    det=dict(chi=283, cle=169, buf=256),
    buf=dict(det=256, cle=189, pit=215, syr=150),
    pit=dict(col=185, cle=134, buf=215, phi=305, bal=247),
    syr=dict(buf=150, phi=253, new=254, bos=312),
    bal=dict(phi=101, pit=247),
    phi=dict(pit=305, bal=101, syr=253, new=97),
    new=dict(syr=254, phi=97, bos=215, pro=181),
    pro=dict(bos=50, new=181),
    bos=dict(pro=50, new=215, syr=312, por=107),
    por=dict(bos=107))


map_times = dict(
    chi=dict(det=280, cle=345, ind=200),
    cle=dict(chi=345, det=170, col=155, pit=145, buf=185),
    ind=dict(chi=200, col=175),
    col=dict(ind=175, cle=155, pit=185),
    det=dict(chi=280, cle=170, buf=270),
    buf=dict(det=270, cle=185, pit=215, syr=145),
    pit=dict(col=185, cle=145, buf=215, phi=305, bal=255),
    syr=dict(buf=145, phi=245, new=260, bos=290),
    bal=dict(phi=145, pit=255),
    phi=dict(pit=305, bal=145, syr=245, new=150),
    new=dict(syr=260, phi=150, bos=270, pro=260),
    pro=dict(bos=90, new=260),
    bos=dict(pro=90, new=270, syr=290, por=120),
    por=dict(bos=120))

def path(previous, s):
    '''
    `previous` is a dictionary chaining together the predecessor state that led to each state
    `s` will be None for the initial state
    otherwise, start from the last state `s` and recursively trace `previous` back to the initial state,
    constructing a list of states visited as we go
    '''
    if s is None:
        return []
    else:
        return path(previous, previous[s])+[s]

def pathcost(path, step_costs):
    '''
    add up the step costs along a path, which is assumed to be a list output from the `path` function above
    '''
    cost = 0
    for s in range(len(path)-1):
        cost += step_costs[path[s]][path[s+1]]
    return cost
```

```python


""" breadth_first - Performs a breadth first search on cities
    Input:
        start - Node that represents the start of the path
        goal - Node that represents the desired end point of the path
        state_graph - Represents the graph that is being searched in
        return_cost - Boolean value that indicates the cost of the path
    Algorithm:
        * Create a queue with the start node as the first node
        * Create a visited set where the first node is visited
        * Create a dictionary for the previous nodes that have been visited
        * While the queue is not empty:
            * Pop the current node from the queue
            * If we reach the goal
                * Create a path with the previous nodes and the goal
                * Return the path and the cost if return_cost is set to true, otherwise just the path
            * Iterate over the neighbors of the current node
            * Add the neighbor to the visited set if it isn't visited
            * Update the previous node with the current node
            * Add the neighbor to the queue
        * Return the cost of the traversal
    Output:
        Returns the path in the search as well as the cost in the path
"""
def breadth_first(start, goal, state_graph, return_cost=False):
    queue = deque([start])
    visited = set([start])
    previous = {start: None}
    while (queue):
        current = queue.popleft()
        if (current == goal):
            path_to_goal = path(previous, goal)
            if (return_cost):
                cost = pathcost(path_to_goal, state_graph)
                return path_to_goal, cost
            else:
                return path_to_goal
        for neighbor in state_graph[current]:
            if (neighbor not in visited):
                visited.add(neighbor)
                previous[neighbor] = current
                queue.append(neighbor)
    return None if not return_cost else (None, 0)

# Solution:

""" maze_to_graph - Converts a maze represented as a numpy array into a graph
    Input:
        maze - 2D numpy array where 0 represents walkable cells and 1 represents walls
    Algorithm:
        * Initialize an empty dictionary to represent the graph
        * Define the directions for North, South, East, and West movements
        * Iterate over each cell in the maze
            * Initialize an empty dictionary for each cell in the graph
            * For each direction, calculate the neighboring cell's coordinates
            * Check if the neighboring cell is within the maze bounds and is walkable (contains 0)
                * If it is, add the neighbor to the current cell's dictionary in the graph with the
 direction as the value
    Output:
        Returns a dictionary representing the graph where keys are coordinates of cells and values
 are dictionaries
        of neighboring cells with directions
"""
def maze_to_graph(maze):
    ''' takes in a maze as a numpy array, converts to a graph '''
    graph = {}
    rows, cols = maze.shape
    directions = {
        'N': (1, 0),  # North
        'S': (-1, 0),   # South
        'E': (0, 1),   # East
        'W': (0, -1)   # West
    }
    for r in range(rows):
        for c in range(cols):
            graph[(c, r)] = {}
            for direction, (dr, dc) in directions.items():
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols and maze[nr, nc] == 0:
                    graph[(c, r)][(nc, nr)] = direction
    return graph

```