Assignment 5: Trains in the Storm
Programming Report

s5343798 & s5460484

Algorithms and Data Structures 2024

1 Problem Description

This assignment aims to identify the most efficient path between two specified train stations in the Netherlands utilizing Dijkstra's Algorithm. It requires accounting for the network of station connections and their associated costs. The solution must be robust against potential service disruptions, taking into account any reported disruptions, the stations impacted, and the origin and destination stations for which the shortest path is sought. The system outputs include the fastest travel time and the optimal path for connecting the two stations.

2 Problem Analysis

This assignment is completely open to approach, excluding the necessity of Dijkstra's Algorithm - any data structure is open to use, and importing the data is done however fit. As such, a big challenge this assignment brings is deciding on an approach: What data structure(s) should we use? How should we format the data - raw or tabulated?

Furthermore, an additional challenge involves employing an algorithm that explores all potential paths to discover the shortest one, declaring the destination unreachable when no path exists. A significant hurdle was to trace the optimal path for presentation, which involved maintaining a record of each station and the previous station leading to it. This complexity was compounded by the necessity to ascertain when the destination could not be reached due to unexpected service interruptions or similar issues.

3 Program Design

In our design, we first check all disruptions, then import the data for the station connections and their weights from a CSV file. This makes handling the data non-hardcoded - the file can be further expanded without having to touch the code - and scalable. When we import the data and add it to our UnweightedGraph, we check if any connections match those "banned" - the disruptions. That way, we do not need to go backwards and remove the connections later. In the end, we have a complete UnweightedGraph.

Next, we utilise Dijkstra's algorithm to calculate the shortest path from the user-selected start station to every other station in the network. We compile a list of the most direct connections by recording each station's predecessor (parent). This setup allows us to display the sequence of stations forming the shortest path to the destination.

4 Evaluation of the Program

We tested the program using the provided test cases, primarily the two given in the lab text: The one with no disruptions, and the one with two disruptions. In addition, we tested with the cases given on Themis. Output-wise, given the efficiency and accuracy of Dijkstra's Algorithm, the output was correct and speedy, and Themis accepted all test cases.

5 Extension of the Program: Going International

We additionally completed one of the bonus assignments, which was adapting our program to read in an arbitrary train network. While our process was quite similar to the original assignment, one problem remained distinguished: Setting up the networks. Because we have no initial data, we would need to change the program to create a system that allows for setting up a new network during runtime. In the end, we developed a new TrainNetwork class, which on initialising would run through input steps to create an UnweightedGraph. Effiency-wise, the program can handle the load, but at extremely large networks (such as test case 3 on Themis), the output would take around a second. Naturally, for a modification of the original assignment, there is a lot of improvement and consideration (such as utilising the A* Algorithm instead of Dijkstra's Algorithm?).

6 Process Description

Because we were provided the pseudo-code for Dijkstra's Algorithm, the process of writing the code to determine the shortest path was straightforward. Arguably, the biggest hurdle in the assignment was deciding which data structure(s) to use - Dijkstra's Algorithm can be modified to use different data structures. In the end, an UndirectedGraph to plot out the points and weights and a MinHeap to create a priority queue proved most efficient.

An interesting problem that came up during the process was tracking the actual pathway, i.e. the order of nodes. Initially, we tracked it in a list, appending a node each time the node with the least pseudo-distance is chosen. This brought up a problem though where an extra node would be included in the list, creating a conflicting path. Our solution was to create a list where each element represented a node's parent node so that by the end of the algorithm, a pathway is formed by looking at the end node and tracing back via the parent node.

This assignment demonstrated the effectiveness of using multiple data structures to solve complex problems. And the case of logging the pathway taught us to think beyond approaching the problem linearly, i.e. going straightforwardly.

Distribution of labour is split: We worked on our implementations, reviewed which of ours was more effective, and divided the work equally on the assignment report.

7 Conclusions

Did our program solve the problem? Yes, to an extent. The program handled the majority of test cases, except one. So, for the most part, it is functional but still needs some work.

How efficient is our program? Very much so - it runs very fast due to the efficiency of Dijkstra's Algorithm.

Is our program optimal? Given that we failed one test case, there is room for improvement - perhaps in the logic of how we wrote the algorithm, the flow of the program, and so forth. A specific note for improvement is how we parse the connections into the UnweightedGraph; instead of taking in a CSV file and reading line-by-line, maybe there is a more efficient system of doing that, i.e. a separate class? Regardless, the data should not be hardcoded in the program.

A Program Text

A.1 a_lab5trains.py

```python
"""
File: a_lab5trains.py
Authors:
    Marcus Persson (m.h.o.persson@student.rug.nl)
    Marinus van den Ende (m.van.den.ende.1@student.rug.nl)

Description:
    This program uses Dijkstras Algorithm to find the fastest connection
    and output the list of all stations along that route, including the
    starting and ending station, as well as the total time the connection
    will take.
"""
from graph import UndirectedGraph
from minheap import MinHeap
import csv


def create_network(banned: list[tuple]) -> (UndirectedGraph, list):
```

```python
    """
    Returns an UndirectedGraph of all connected stations and their distances,
    in which each node is a dictionary with a key tuple (station1, station2)
    and value distance.
    :param banned: A list of tuples of banned connections.
    :return: A tuple of the network and list of all stations.
    """
    connections = []
    cities = []

    with open("connections.csv") as f:
        reader = csv.reader(f)
        for line in reader:
            if (line[0], line[1]) not in banned and (line[1], line[0]) not in banned:
                if line[0] not in cities:
                    cities.append(line[0])
                if line[1] not in cities:
                    cities.append(line[1])

                # (Origin, Destination, Distance)
                connections.append((cities.index(line[0]),
                                    cities.index(line[1]),
                                    int(line[2])))

    network = UndirectedGraph(len(cities))

    for item in connections:
        network.add_edge(item[0], item[1], item[2])

    return network, cities


# Input: n current disruptions.
disruptions = int(input())

# Input: n many disruptions, in which each disruption consists of a tuple
#        regarding a direct connection.
banned_connections = []
for i in range(disruptions):
    connection = (input(), input())
    banned_connections.append(connection)

network, cities = create_network(banned_connections)


def find_shortest_path(graph: UndirectedGraph, start: int, end: int) -> \
        (list[int], int):
    """
    Returns the shortest path in an undirected graph from a start node and
    end node, using Dijkstra's algorithm.
    :param graph: UndirectedGraph class.
    :param start: Starting node.
    :param end: Ending node.
    :return: Tuple of the shortest possible path, represented as a list of
             points, and the minimum distance.
    """
    # Accounts for if we put in the same two nodes.
    if start == end:
        return [start, end], 0

    # Implements a priority queue as a MinHeap.
```

```python
    p_queue = MinHeap()
    p_queue.enqueue((0, start))

    # We create a list of (minimum) distances from the start node.
    dist = [float("inf")] * graph.size()
    dist[start] = 0

    # We create a reference list of nodes' parent nodes
    parent_of = [None] * graph.size()

    while p_queue.size():
        min_dist, node = p_queue.remove_min()

        for e in graph._neighbours[node]:
            # In our undirected graph, we do not add two pathways twice, so
            # destination/origin are interchangeable.
            vertex = e._destination if e._destination != node else e._origin
            weight = e._weight
            if dist[vertex] > dist[node] + weight:
                dist[vertex] = dist[node] + weight
                p_queue.enqueue((dist[vertex], vertex))
                parent_of[vertex] = node

    if parent_of[end] is None:
        # If there is no parent, that means there is no possible path.
        return None, None

    pathway = [end]
    parent = parent_of[end]

    while parent is not None:
        pathway.append(parent)
        parent = parent_of[parent]

    pathway.reverse()

    return pathway, dist[end]


# Input: Queries, in which each query consists of a tuple regarding a start
#        and end.
start = input()

while start != "!":
    end = input()
    if start not in cities or end not in cities:
        print("UNREACHABLE")
    else:
        journey, distance = find_shortest_path(network,
                                               cities.index(start),
                                               cities.index(end))
        if distance is None:
            print("UNREACHABLE")
        else:
            for j in journey:
                print(cities[j])
            print(distance)
    start = input()
```

A.2 graph.py

```
"""
File: graph.py
Authors:
    Marcus Persson (m.h.o.persson@student.rug.nl)
    Marinus van den Ende (m.van.den.ende.1@student.rug.nl)

Description:
    This program creates an UndirectedGraph class, based off the one taught
    in the lecture and reader.
"""


class GraphEdge:
    def __init__(self, origin, destination, weight: float = 1.0):
        self._origin = origin
        self._destination = destination
        self._weight = weight

    def is_incident(self, node: int) -> bool:
        return node == self._origin or node == self._destination

    def other_node(self, node: int) -> int:
        if self.is_incident(node):
            return self._origin + self._destination - node - node

        return -1


class UndirectedGraph:
    def __init__(self, node_count: int) -> None:
        self._neighbours = [[] for _ in range(node_count)]

    def __getitem__(self, node: int):
        return self._neighbours[node]

    def add_edge(self, node1: int, node2: int, weight: int = 1):
        new_edge = GraphEdge(node1, node2, weight)
        self._neighbours[node1].append(new_edge)
        self._neighbours[node2].append(new_edge)

    def size(self) -> int:
        return len(self._neighbours)
```

A.3 minheap.py

```
"""
File: minheap.py
Authors:
    Marcus Persson (m.h.o.persson@student.rug.nl)
    Marinus van den Ende (m.van.den.ende.1@student.rug.nl)

Description:
    This program is a class implementation of a MinHeap, in which the root
    node is the lowest value, and lower nodes/leaf nodes are higher values.
"""


class MinHeap:
```

```
    def __init__(self):
        self._heap = [0]

    def size(self) -> int:
        return len(self._heap) - 1

    def _heap_empty_error(self):
        print("Heap empty")

    def _upheap(self, index: int) -> None:
        if index > 1:
            parent_index = index // 2

            if self._heap[parent_index] > self._heap[index]:
                self._heap[parent_index], self._heap[index] = (
                    self._heap[index], self._heap[parent_index])
                self._upheap(parent_index)

    def enqueue(self, value) -> None:
        self._heap.append(value)
        self._upheap(self.size())

    def _downheap(self, index: int) -> None:
        lc = index * 2
        rc = lc + 1

        if lc < self.size():
            value = self._heap[index]
            left_child = self._heap[lc]

            if self._heap[lc] and self._heap[rc]:
                right_child = self._heap[rc]
            else:
                right_child = left_child

            if left_child < value and left_child <= right_child:
                self._heap[lc], self._heap[index] = (
                    self._heap[index], self._heap[lc])
                self._downheap(lc)
            elif right_child < value:
                self._heap[rc], self._heap[index] = (
                    self._heap[index], self._heap[rc])
                self._downheap(rc)

    def remove_min(self):
        return_value = self._heap[1]

        if self.size() > 1:
            self._heap[1] = self._heap.pop()
            self._downheap(1)
        else:
            self._heap.pop()

        return return_value
```

A.4 connections.csv

```
Amsterdam,Den Haag,46
Amsterdam,Den Helder,77
Amsterdam,Utrecht,26
```

```
Den Haag,Eindhoven,89
Eindhoven,Maastricht,63
Eindhoven,Nijmegen,55
Eindhoven,Utrecht,47
Enschede,Zwolle,50
Groningen,Leeuwarden,34
Groningen,Meppel,49
Leeuwarden,Meppel,40
Maastricht,Nijmegen,111
Meppel,Zwolle,15
Nijmegen,Zwolle,77
Utrecht,Zwolle,51
```

B Extended Program Text

B.1 a_lab5trains_extra2.py

```python
"""
File: a_lab5trains_extra2.py
Authors:
    Marcus Persson (m.h.o.persson@student.rug.nl)
    Marinus van den Ende (m.van.den.ende.1@student.rug.nl)

Description:
    This program is a bonus assignment, regarding going international.
"""
from graph import UndirectedGraph
from minheap import MinHeap


def find_shortest_path(graph: UndirectedGraph, start: int, end: int) -> \
        (list[int], int):
    """
    Returns the shortest path in an undirected graph from a start node and
    end node, using Dijkstra's algorithm.
    :param graph: UndirectedGraph class.
    :param start: Starting node.
    :param end: Ending node.
    :return: Tuple of the shortest possible path, represented as a list of
             points, and the minimum distance.
    """
    # Accounts for if we put in the same two nodes.
    if start == end:
        return [start, end], 0

    # Implements a priority queue as a MinHeap.
    p_queue = MinHeap()
    p_queue.enqueue((0, start))

    # We create a list of (minimum) distances from the start node.
    dist = [float("inf")] * graph.size()
    dist[start] = 0

    # We create a reference list of nodes' parent nodes
    parent_of = [None] * graph.size()

    while p_queue.size():
        min_dist, node = p_queue.remove_min()

```

```python
        for e in graph._neighbours[node]:
            # In our undirected graph, we do not add two pathways twice, so
            # destination/origin are interchangeable.
            vertex = e._destination if e._destination != node else e._origin
            weight = e._weight
            if dist[vertex] > dist[node] + weight:
                dist[vertex] = dist[node] + weight
                p_queue.enqueue((dist[vertex], vertex))
                parent_of[vertex] = node

    if parent_of[end] is None:
        # If there is no parent, that means there is no possible path.
        return None, None

    pathway = [end]
    parent = parent_of[end]

    while parent is not None:
        pathway.append(parent)
        parent = parent_of[parent]

    pathway.reverse()

    return pathway, dist[end]


class TrainNetwork:
    """
    This class represents a train network, represented as an UndirectedGraph.
    """

    def __init__(self):
        _station_count = int(input())
        self.stations = [None] * _station_count
        for i in range(_station_count):
            pos, item = input().split(" ", 1)
            self.stations[int(pos)] = item

        _connection_count = int(input())
        self._connections = []
        while len(self._connections) < _connection_count:
            self._connections.append(tuple(input().split()))

        _disruption_count = int(input())
        self._disruptions = []
        while len(self._disruptions) < _disruption_count:
            self._disruptions.append((input(), input()))

        self.network = self._create_network(self.stations,
                                            self._connections,
                                            self._disruptions)

    def _create_network(self, stations: list, connections: list,
                        disruptions: list) -> (UndirectedGraph, list):
        network = UndirectedGraph(len(stations))

        for i in range(len(connections)):
            if (stations[int(connections[i][0])],
                stations[int(connections[i][1])]) not in disruptions:
                network.add_edge(int(connections[i][0]),
                                 int(connections[i][1]),
```

```python
                                        int(connections[i][2]))

        return network


network_count = int(input())
networks = [None] * network_count

for i in range(network_count):
    networks[i] = TrainNetwork()
    start = input()
    while start != "!":
        end = input()
        if start not in networks[i].stations or end not in networks[i].stations:
            print("UNREACHABLE")
        else:
            journey, distance = (
                find_shortest_path(networks[i].network,
                                   networks[i].stations.index(start),
                                   networks[i].stations.index(end)))

            if distance is None:
                print("UNREACHABLE")
            else:
                for j in journey:
                    print(networks[i].stations[j])
                print(distance)
        start = input()
```