Determining Shortest Path Across Most Mentioned Real Locations in *The Adventures of Sherlock Holmes* Using Brute Force Algorithm

Ray Owen Martin - 13524033
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: rayowenmartin2018@gmail.com , 13524033@std.stei.itb.ac.id

***Abstract*—Travelling has always been time-well-spent needed during breaks. *Sherlockians* might find usual travelling boring, hence the need for something new. One example is to travel across most mentioned real locations in *The Adventures of Sherlock Holmes*, of course, with least amount of money spent. This problem can be identified as finding a Hamiltonian path, which is a common tricky problem in Computer Science to determine the shortest route across all nodes exactly once without having to return to the starting point. To solve this problem, Natural Language Processing (NLP) is used to firstly determine all the locations mentioned in the novel. Subsequently, travelling cost between one location and another can be computed, resulting in a weighted-directed graph. Lastly, an suitable algorithm—in this case, Brute Force Algorithm— is applied to find the answer. This paper will present how this algorithm can be neatly applied for the problem.**

Keywords—Natural Language Processing, Brute Force Algorithm, Weighted-Directed Graph, Shortest Path Algorithm, Sherlock Holmes

I. Introduction

Travelling has always been a part of humans' daily life, whether going far like the other side of the Earth we are living on, or simply going to a nearby park. It has always been refreshing to find a different scenery in the midst of hustle and bustle of life.

Sherlockians—fans of the novel series *Sherlock Holmes*—might find travelling, while being calming, boring and inadequate. However, *Sherlockians* would still be intrigued knowing it is possible to travel across locations mentioned in Sherlock Holmes, whether it is the famous Baker Street 221B where Holmes and his assistant, dr. John Watson, lives, or other random places mentioned such as China. It would be like travelling across half of the world while reliving moments in the novel leading up to the places being mentioned.

Even when it is possible to travel like so, it is still rather tricky to choose the right path to go through each location efficiently, meaning with least possible cost. The locations are possible to illustrate with a weighted-directed graph, hence, the problem of finding the shortest possible path across these locations can be solved by applying Brute Force algorithm to the graph.

II. Theoretical Framework

A. Natural Language Processing

Natural Language Processing (NLP) is one part of Computer Science that has deep connection with Artificial Intelligence. It enables computer to communicate with human language by combining multiple aspects, including computational linguistics, human language rules, statistical modeling, machine learning and deep learning.

From linguistics point of view, NLP can be used to determine parts of a sentence, including geographical locations. One possible way to extract these locations is by using Named Entity Recognition (NER), an information extraction that locates named entities in unstructured texts, in this case Geopolitical Entity (GPE) and Non-GPE locations (LOC). GPE is described as countries, cities, or states, while LOC is decribed as mountain ranges or bodies of water.

B. Graph Theory Basics

A graph is a representation of points (vertices) and lines, where each line connects a pair of points (edges). Graph G can be defined as G(V, E) where V is a nonempty set of vertices $\{v_1, v_2, \ldots, v_n\}$ and E is a set of edges $\{e_1, e_2, \ldots, e_m\}$. An edge $e_m(u_1, u_2)$ means edge e_m connects vertex u_1 to vertex u_2. Graphs are excellent ways to represent real-life situations, such as road maps and electrical networks. Figure 1 shows a simple graph with 5 vertices {0, 1, 2, 3, 4} with a total of 4 edges.

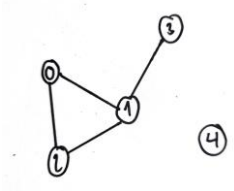

Fig. 1. Example of a graph. (Source: Author's document)

A graph can also consist of multiple edges, which is when more than 1 edge connects the same pair of vertices.

Additionally, a loop can also be created when an edge connects a vertice to itself. Figure 2 shows the example of loop and multiple edge on a graph.

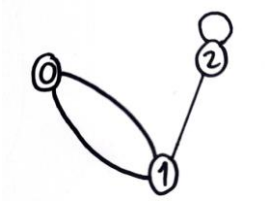

Fig. 2. Example of an unsimple graph. (Source: Author's document)

It is clear that there are 2 edges connected vertices 0 and 1. This shows that the graph has multiple edges. There is also an edge connecting vertex 2 to itself. This is a loop.

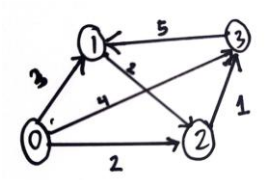

Fig. 3. Example of a weighted-directed graph. (Source: Author's document)

C. Types of Graph

Different types of graph presents different purposes. There are a few types of graph that are commonly found.

1. Simple graph, which has no multiple edges nor loops. Example is shown in Figure 1.
2. Unsimple graph, which has multiple edges or loops. Example is shown in Figure 2.
3. Undirected graph, which has no arrows so that an edge $e_m(u_1, u_2)$ is the same as $e_m(u_2, u_1)$. Figure 1 and 2 are examples of undirected graph.
4. Directed graph, which has arrows so that an edge $e_m(u_1, u_2)$ is not the same as $e_m(u_2, u_1)$. Figure 3 is an example of directed graph.
5. Weighted graph, which edges has weights, usually to represent the cost needed to travel from the starting vertex to the ending vertex. Figure 3 is an example of weighted graph.

D. Hamiltonian Path and Shortest Path Algorithm

If it is possible to visit each vertex exactly once for all vertices in a graph, then the graph is said to have a Hamiltonian path. If there is an edge connecting the starting vertex to the ending vertex, then the graph is said to have a Hamiltonian cycle. This is the foundation of the Shortest Path Algorithm, marking its importance in daily life, for instance, for determining how to get from Bandung to Medan with the lowest possible cost and how to traverse locations mentioned in *The Adventures of Sherlock Holmes*.

There are a few known algortihms to solve such problems, however, in this paper only Brute Force Algorithm will be discussed. This algorithm traverses all possible paths and saves the best path, which has the lowest cost.

E. Sherlock Holmes

Sherlock Holmes is a fictional character created by Arthur Conan Doyle. As the world's first and only "consulting detective", he pursued criminals and solved the most complicated, unthinkable crime mysteries during the late 1800s. He is well-known for his keen observation and deductive reasoning shown during his works investigating crime mysteries. *The Adventures of Sherlock Holmes* is a collection of short stories written by Doyle, which showcases some of the detective's stories. Centering in London, the collection also mentions tons of other well-known places, such as France and Florida.

III. Proposed Method

The proposed method to determine the shortest path across the most mentioned locations in *The Adventures of Sherlock Holmes* will use Jupyter Notebook service to run algorithms in Python 3. A few libraries are used, such as spaCy. The method can be divided into few parts as follows:

A. Location Extraction

Initially, the text document of *The Adventures of Sherlock Holmes* is downloaded from [2] The .txt file will then be uploaded to the Jupyter Notebook before running the algorithm as shown in Figure 4. The algorithm is from [1] modified.

```
import spacy
# from spacy import displacy
from collections import Counter

nlp = spacy.load('en_core_web_sm')

# Open full text
f = open('./taosh.txt', 'r')
taosh_content = f.read()

# Process full text with NLP
doc2 = nlp(taosh_content)
# Show ent labeling in text if needed
# displacy.render(doc2, style="ent")

#initialization of sets to store unique gpe and loc labelled words
unique_gpe = set()
unique_loc = set()
gpe_counter = Counter()
loc_counter = Counter()

for ent in doc2.ents:
    if ent.label_ in ["GPE", "LOC"]:
        text = ent.text
        # print to show results and labeling process
        # print(f"{ent.text} → {ent.label_}")
        if ent.label_ == "GPE":
            unique_gpe.add(text)
            gpe_counter[text] += 1
        else:
            unique_loc.add(ent.text)
            loc_counter[text] += 1

# Print results
print("Unique GPE:", len(unique_gpe))
print("Unique LOC:", len(unique_loc))

# Most mentioned is defined as showing more than 2 times
for element in unique_gpe:
  if gpe_counter[element] > 2:
    print('Element gpe: ', element, 'counter: ', gpe_counter[element])

for element in unique_loc:
  if loc_counter[element] > 2:
    print('element loc: ', element, 'counter: ', loc_counter[element])

f.close()
```

Fig. 4. Algorithm to find most mentioned locations in *The Adventures of Sherlock Holmes*. (Source: Author's Document)

Firstly, the library spaCy and tool Counter are imported for multiple reasons which will be explained later. Then, the small English NLP model en_core_web_sm is loaded to process the text file (line 5). Subsequently, the .txt file is opened and read (line 8-9) before being processed with NLP (line 12). It must be noted that the NLP can only run with spaCy library already imported. Line 2 and 14 is commented, but can be uncommented if the reader wants to try the algorithm to show how the NER labelling is done.

Sets for unique gpe and unique loc is then initialized along with the a counter for each set. Then the whole .txt file will be scanned for any entities found, if the word or phrase scanned has an entity type of GPE or LOC, then it will be added to corresponding sets. If the word or phrase has ever been scanned, then it will just increase the counter of the word or phrase. The number of elements for each set will then be printed to validate whether the algorithm works well.

Finally, most mentioned GPE or LOC will be printed. The author chooses a minimum of 2 appearances in the text file to ensure that the number of GPE and LOC combined are enough to be analyzed. The output for the algorithm can be seen as follows:

```
Unique GPE: 110
Unique LOC: 24
Element gpe:  the United
States counter:  3
Element gpe:  Bohemia counter:  3
Element gpe:  Eyford counter:  6
Element gpe:  California counter:  3
Element gpe:  England counter:  20
Element gpe:  Scarlet counter:  3
Element gpe:  China counter:  5
Element gpe:  the United States counter:  10
Element gpe:  India counter:  4
Element gpe:  Florida counter:  3
Element gpe:  Streatham counter:  3
Element gpe:  France counter:  6
Element gpe:  Savannah counter:  3
Element gpe:  America counter:  10
Element gpe:  London counter:  36
Element gpe:  U.S. counter:  7
element loc:  Europe counter:  6
```

Fig. 5. Output of the algorithm to find most mentioned locations in *The Adventures of Sherlock Holmes*. (Source: Author's Document)

B. Data Cleansing

The data will then be cleansed manually through the process as follows:

1. "The United <newline> States", "the United States" and "U.S." are united as "United States".
2. "Eyford" and "Scarlet" are also deleted because they do not represent locations.
3. "Europe" is deleted because there are already countries and states representing Europe (England and Bohemia).
4. "United States" is also deleted because there are already cities representing the United States (California, Florida and Savannah).
5. "England" is also deleted because there are already cities representing England (London and Streatham).

At the end of this process, we now have California, Bohemia, Florida, Savannah, India, France, London, Streatham and China as our data. It must be noted that Savannah references to the city in Georgia due to the context from the short story *The Five Orange Pips* in the collection, while Bohemia references to the present-day Czech Republic (Bohemia is used in older English) [4].

C. Data Transformation

In order to improve the accuracy of the latter results, countries and cities will be transformed into its capitol as of the date of the paper being written, which correspondingly results in Sacramento, Prague, Tallahassee, Savannah, New Delhi, Paris, London, Streatham and Beijing.

D. Data Visualization

Data can then be visualized for better understanding of the path being searched. The algorithm and output for data visualization can be seen in Figure 6 and Figure 7. The algorithm is from [5] modified.

The library folium and tools Nominatim and FastMarkerCluster are imported. The cleansed and transformed data will then be initialized accordingly, hence the index of each city will be 0 for Sacramento, 1 for Prague, 2 for Tallahassee, 3 for Savannah, 4 for New Delhi, 5 for Paris, 6 for London, 7 for Streatham and 8 for Beijing. Each location's latitude and longitude will firstly be stored in a 2-Dimensional Array with all elements initialized as 0.

Baker Street is chosen as the center of the map due to the importance and fame of the location, along with the strategic position. Subsequently the Nominatim tool will be called to determine every place's, including Baker Street's, latitude and longitude, which will be printed out. Each city's latitude and longitude will then be stored in the array mentioned before. Using the known positions, a marker will be placed on the map with colours according to the array *color*. After everything is done, the map will be displayed on the screen. Streatham is shown right above London but due to the overlapping colours, it is not much visible.

E. Graph Making

To find the weights and possible edges for all vertices, plane ticket fees are used. As for the date, July 2nd 2025 and August 2nd will both be used to make a comparison. Travelling duration, departing and arriving time will not be considered, hence only one date will be used for each mode (July 2nd or August 2nd but not both). The date July 2nd is chosen for being the middle of the year 2025, while August 2nd is chosen

because it is the exact one month after the middle of the year, hence suitable for stabilizing the fluctuations of the ticket fees.

The fees are taken from [6]. One thing to note is that Streatham to London and London to Streatham will be using bus ticket fee for there is no plane travelling in the route. The bus ticket fees are taken from [7]. There are few conditions for the plane ticket fees to be considered valid, hence chosen: the flight must be the cheapest with 2 stops or fewer, with no layover more than 6 hours and no self-transfer between airports. The fees will first be recorded in IDR (Indonesian Rupiah) then be converted to USD (United States Dollar) in the algorithm to ease graph-making process. The complete fee can be seen in Fee. The completed dataset will then be converted into a 3-Dimensional Array.

The algorithm can be seen in Figure 8, 9 and 10. The algorithm is from [8] modified. When running the program for the first time, it is necessary to first install the needed library from [11].

Then, libraries networkx and matplotlib.pyplot, along wit[h tool MultiGraph from the recently installed library are imported. Subsequently, function to create a July 2nd graph is defined with weight according to the 3-Dimensional Array as mentioned above (named as *all_costs* in the algorithm). A similar function is created for August 2nd.

```python
# importing geopy library
from geopy.geocoders import Nominatim
import folium
from folium.plugins import FastMarkerCluster

cleansed_data = ['Sacramento', 'Prague', 'Tallahassee', 'Savannah', 'New Delhi', 'Paris', 'London', 'Streatham', 'Beijing']
color = ['red', 'blue', 'green', 'purple', 'orange', 'darkred', 'lightred', 'beige', 'darkblue']
data_lat_long = [[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0]]

# Setting the center of the map
loc = Nominatim(user_agent="GetLoc")
getLoc = loc.geocode('Baker Street', timeout = 2)
map2=folium.Map(location=[getLoc.latitude, getLoc.longitude],zoom_start=3)

# calling the Nominatim tool
loc = Nominatim(user_agent="GetLoc")
for i in range(9):
  loc_text = cleansed_data[i]
  getLoc = loc.geocode(loc_text, timeout = 10)

  # printing latitude and longitude
  print("City: ", loc_text)
  print("Latitude = ", getLoc.latitude, "\t Longitude = ", getLoc.longitude)

  #save data into a 2D matrix
  data_lat_long[i] = (getLoc.latitude, getLoc.longitude)
  map2.add_child(folium.Marker(location=[getLoc.latitude, getLoc.longitude],popup=getLoc,icon=folium.Icon(color=color[i])))

display(map2)
```

Fig. 6. The algorithm to visualize map. (Source: Author's Document)

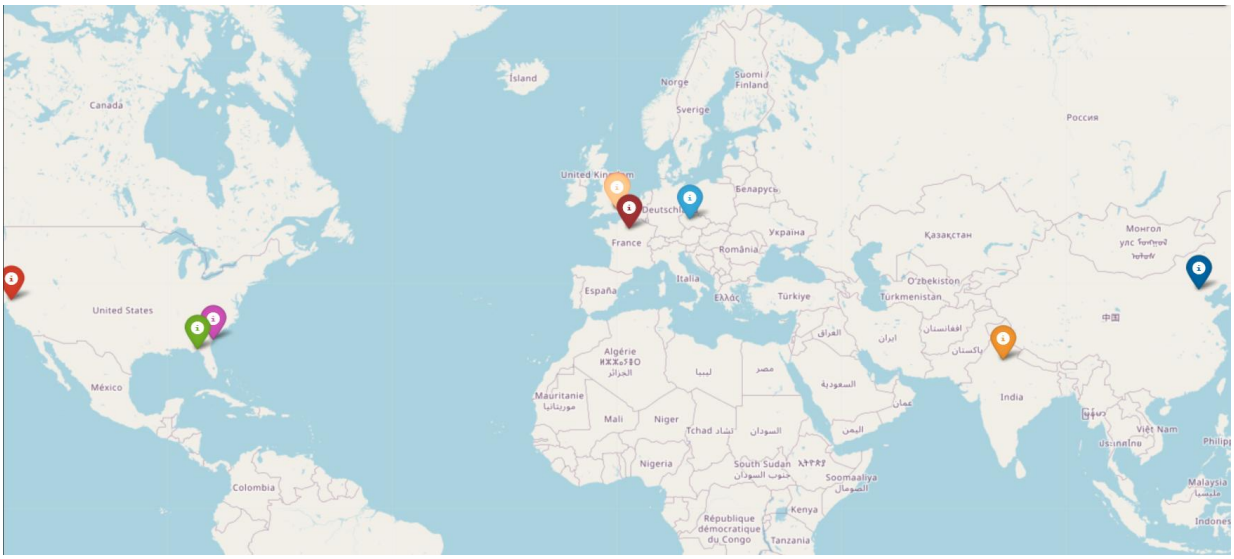

Fig. 7. Map visualization. (Source: Author's Document)

```python
# for running code for the first time
# !pip install https://github.com/paulbrodersen/netgraph/archive/dev.zip

import networkx as nx
import matplotlib.pyplot as plt

from netgraph import MultiGraph

# For July cost
def create_graph1(n_nodes):
    G = nx.MultiDiGraph()
    for i in range(n_nodes):
        lat, lon = data_lat_long[i]

    for i in range(n_nodes):
        for j in range(n_nodes):
          weight = all_costs[i][j][0]*0.000061
          if weight != 0:
            G.add_edge(i, j, weight=round(weight, 2))
    return G

# For August cost
def create_graph2(n_nodes):
    G = nx.MultiDiGraph()
    for i in range(n_nodes):
        lat, lon = data_lat_long[i]

    for i in range(n_nodes):
        for j in range(n_nodes):
          weight = all_costs[i][j][1]*0.000061
          if weight != 0:
            G.add_edge(i, j, weight=round(weight, 2))
    return G

# The format for each city cost is [[CostToVertex0inJuly, CostToVertex0inAugust], [CostToVertex1inJuly, CostToVertex1inAugust], ...]
sacramento_costs = [[0,0], [10144177, 8802097], [4221789, 4816037], [2925113, 2924949], [14527773, 8818712], [11979136, 8492175], [13497225, 5199974], [10838322, 5199974], [13841600, 13160809]]
prague_costs = [[18875100, 15204064], [0, 0], [24318718, 15261062], [18581420, 15735539], [4981190, 5816373], [480508, 1260387], [773760, 577850], [2595806, 2168251], [7722000, 10433306]]
tallhassee_costs = [[4221789,4816037], [13393842, 8656275], [0, 0], [3695368, 3908534], [12542163, 12627495], [14432310, 6293235], [8984475, 6129135], [8987103, 6130938], [24617276, 21436087]]
savannah_costs = [[2925113,2925113], [12236822, 7978542], [3780700, 3498284], [0, 0], [12956202, 11002840], [12289761, 6295076], [8987103, 5638484], [8987103, 5638484], [17247030, 17590100]]
newdelhi_costs = [[10806936,16098085], [5238173, 5177449], [12451062, 21176498], [12814019, 16362551], [0, 0], [4018539, 5729678], [4268607, 4456720], [4268607, 4456720], [4338165, 4852306]]
paris_costs = [[14927835, 21428342], [654558, 1942415], [21103348, 19901851], [16966345, 18835247], [6473462, 11347290], [0, 0], [933087, 914267], [1539727, 1870929], [9778855, 11685022]]
london_costs = [[18358350, 20118119], [544649, 1291562], [29992623, 29708359], [18159614, 20274693], [6438838, 7971845], [683881, 797892], [0, 0], [65783, 65783], [8120811, 11704957]]
streatham_costs = [[19047740,20807509], [2092586, 2238646], [39214794, 39214794], [18159614, 20274693], [6844855, 9158529], [779081, 1559151], [65783, 65783], [0, 0], [8120811, 15022297]]
```

Fig. 8. The algorithm to create graphs and calculate shortest path (part 1).
(Source: Author's Document)

```python
beijing_costs = [[14701757,15882784], [6794971, 7272232], [21144671, 20541908], [16371693, 19592099], [5373920, 4333569], [6650367, 9502421], [6620999, 7821521], [6620999, 7821521], [0, 0]]

all_costs = [sacramento_costs, prague_costs, tallhassee_costs, savannah_costs, newdelhi_costs, paris_costs, london_costs, streatham_costs, beijing_costs]

G = create_graph1(9)
G2 = create_graph2(9)

# Graph visualization example
plt.figure(figsize=(15, 10))
MultiGraph(
    G,
    node_labels=True,
    edge_label_fontdict=dict(fontsize=10),
    edge_labels_pos=5,
    arrows=True,
    node_color='skyblue',
    edge_color='red',
    node_edge_color='black',
    edge_width = 0.3,
)
plt.show()

# Create adjacency matrix
adj_mtrx_1 = [[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0]]
adj_mtrx_2 = [[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0]]
# Scale July cost matrix
adj_mtrx_1 = [[round(all_costs[i][j][0]*0.000061, 2) for j in range(9)] for i in range(9)]
# Scale August cost matrix
adj_mtrx_2 = [[round(all_costs[i][j][1]*0.000061, 2) for j in range(9)] for i in range(9)]

import itertools
from itertools import permutations

def brute_force_tsp(adj_matrix, start):
    num_nodes = 9
    nodes = list(range(num_nodes))
    nodes.remove(start)
    all_paths = itertools.permutations(nodes)
    best_path = []
    min_cost = float('inf')
    for perm in all_paths:
        # start = perm[0]
        current_path = [start] + list(perm)
        current_cost = 0
```

Fig. 9. The algorithm to create graphs and calculate shortest path (part 2).
(Source: Author's Document)

```
        current_cost = 0
        for i in range(len(current_path) - 1):
            cost = adj_matrix[current_path[i]][current_path[i+1]]
            current_cost += cost
        if current_cost < min_cost:
            min_cost = current_cost
            best_path = current_path
    return min_cost, best_path

min_all_cost = float('inf')
min_path = []
for i in range(9):
    min_cost, best_path = brute_force_tsp(adj_mtrx_1, i)
    if min_cost < min_all_cost:
        min_all_cost = min_cost
        min_path = best_path
print("For 2 July: ")
print("Minimum cost:", min_all_cost)
print("Best path:", min_path)

min_all_cost = float('inf')
min_path = []
for i in range(9):
    min_cost, best_path = brute_force_tsp(adj_mtrx_2, i)
    if min_cost < min_all_cost:
        min_all_cost = min_cost
        min_path = best_path
print("For 2 August: ")
print("Minimum cost:", min_all_cost)
print("Best path:", min_path)
```

Fig. 10. The algorithm to create graphs and calculate shortest path (part 3). (Source: Author's Document)

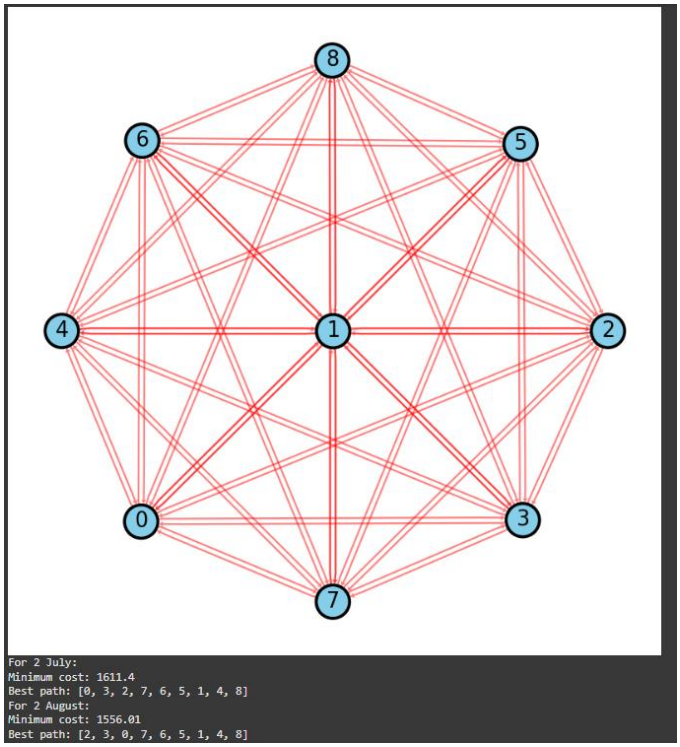

Fig. 11. Output of the algorithm to create graphs and calculate shortest path. (Source: Author's Document)

F. Shortest Path Algorithm

Two additional 2-Dimensional Arrays are added for the scaled cost, one for July 2nd and one for August 2nd. The cost is scaled by multiplying the original value by 0.000061, the exchange rate of IDR to USD at the time of this paper written. After that, the library itertools and tool permutations are imported. Subsequently, a function to call brute force algorithm is defined. The algorithm will list every possible paths with the starting point as the second parameter/argument, and then a minimum cost will be returned to the main function along with its best path.

Outside the function, a float type variable is created as a temporary slot for the minimum cost for all possible starting vertex and paths. An array of integers is also created to store the shortest path. Then best path and minimum cost of both July 2nd and August 2nd will be calculated by calling the shortest path algorithm for each date and the results will be displayed on the screen.

The result can be seen as in Figure 11, showing that the shortest path can be traversed through vertices [0,3,2,7,6,5,1,4,8] for July 2nd for the cost of $1.611,4 and [2,3,0,7,6,5,1,4,8] for August 2nd for the cost of $1.556,01.

IV. Results

As shown in part III. F., the shortest path to traverse most mentioned locations in *The Adventures of Sherlock Holmes* calculated using the proposed method with Brute Force Algorithm approach is:

1. Sacramento -> Savannah -> Tallahassee -> Streatham -> London -> Paris -> Prague -> New Delhi -> Beijing at July 2nd for the cost of $1.611,4 or Rp 27.267.810,57 with the exchange rate of Rp 16.412,55 per USD.
2. Tallahassee -> Savannah -> Sacramento -> Streatham -> London -> Paris -> Prague -> New Delhi -> Beijing at August 2nd for the cost of $1.556,01 or Rp 25.538.091,93 with the exchange rate of Rp 16.412,55 per USD.

As stated before, this result and calculation ignores where the traveller starts among with many other assumptions, hence this might not be the real, exact cheapest route.

V. Conclusion

This paper presents how the problem of traversing most mentioned real locations in *The Adventures of Sherlock Holmes* can be solved using Brute Force Algorithm. However, there have been few assumptions during this study, such as the plan ticket fees are fixed to only a certain date to traverse all locations, without considering the travel, departure, and arrival times, resulting in the cheapest route not being optimal. The methods chosen are using NER to identify the locations, data cleansing to ensure accurate data, data transformation to increase accuracy, data visualization for better understanding of the problem, graph making to prepare for the shortest path algorithm and lastly, shortest path algorithm to solve the problem at once. The result shows that it is possible to travel at the cost of $1.556,01 or Rp 25.538.091,93 with the exchange rate of Rp 16.412,55 per USD on August 2nd with route: Tallahassee -> Savannah -> Sacramento -> Streatham -> London -> Paris -> Prague -> New Delhi -> Beijing. This is the slightly cheaper route than the route on July 2nd with slight

change of route, where Tallahassee is visitted third rather than first while Sacramento is visitted first rather than third.

ACKNOWLEDGMENT

The author is extremely grateful to God for all the guidance during the process of learning leading to the competion of this paper. The author would as well thank his lecturer of Discrete Mathematics IF1220, Dr. Rinaldi Munir and Dr. Arrival Dwi Sentosa, for sharing their knowledge and for their patience throughout the whole lecture. Furthermore, the author treasures the constant support from his loved ones, family and friends, which undoubtedly complements his efforts.

REFERENCES

[1] Abdishakur, "How to Extract Locations from Text with Natural Language Processing", [Online]. Available: https://medium.com/spatial-data-science/how-to-extract-locations-from-text-with-natural-language-processing-9b77035b3ea4 [Accessed 18-Jun-2025]

[2] Doyle, Arthur Conan, The Adventures of Sherlock Holmes, Project Gutenberg, 1999. [Downloaded from https://www.gutenberg.org/cache/epub/1661/pg1661.txt and accessed 18-Jun-2025]

[3] spaCy, "Linguistic Features", [Online], Available: https://spacy.io/usage/linguistic-features#named-entities [Accessed 18-Jun-2025]

[4] The Editors of Encyclopædia Britannica, "Bohemia", [Online], Available: https://www.britannica.com/place/Bohemia [Accessed 18-Jun-2025]

[5] GeeksforGeeks, "How to get Geolocation in Python?", [Online]. Available: https://www.geeksforgeeks.org/python/how-to-get-geolocation-in-python/ [Accessed 18-Jun-2025]

[6] https://google.com/travel/flights

[7] https://www.rome2rio.com/

[8] Yse, Diego Lopez, "Graphs | Solving the Travelling Salesperson Problem (TSP) in Python", [Online]. Available: https://lopezyse.medium.com/graphs-solving-the-travelling-salesperson-problem-tsp-in-python-54ec2b315977 [Accessed 19-Jun-2025]

[9] GeeksforGeeks, "Named Entity Recognition", [Online]. Available: https://www.geeksforgeeks.org/named-entity-recognition/ [Accessed 18-Jun-2025]

[10] Wilson, Robin J., Introduction to Graph Theory, 4th ed., Harlow, 1996, pp. 1-3

[11] https://github.com/paulbrodersen/netgraph/archive/dev.zip.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Juni 2025

Ray Owen Martin - 13524033