Shortest Path Algorithms

Using Wrocław Public Transport Data

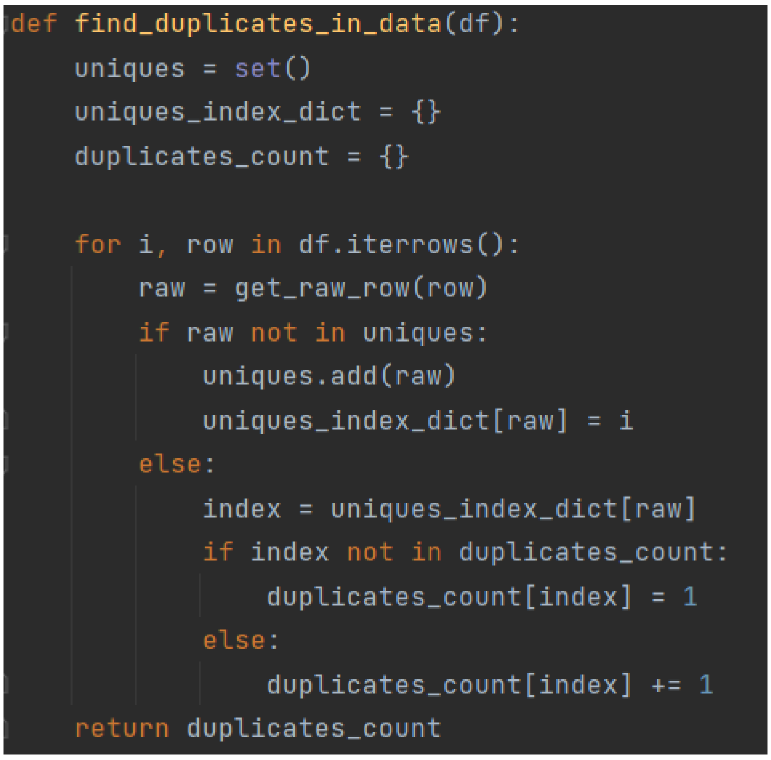
Wiktor Kubera

Contents

1. Data Preparation

1.1. Data Exploration

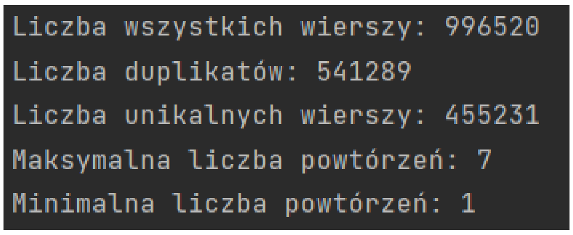
After an initial review of the data contained in the file `connection_graph.csv`, I observed that it pertains to **connections between stops in Wrocław**.
Upon examining the number of records, my first thought was that nearly **a million records related to connections between stops seem excessive for Wrocław**. It is likely that they correspond to **different days (within a week? throughout the year?)**, but this information is not included in the file.
Therefore, I decided to programmatically check whether (and if so, how many) rows have duplicates.
I load the data into the program using the **Pandas library**.
The Pandas library is used for general data analysis, allowing us to load various file formats, which are presented as tables on which we can perform many useful operations. To examine duplicates, I use a script that I wrote myself:



```python
def find_duplicates_in_data(df):
    uniques = set()
    uniques_index_dict = {}
    duplicates_count = {}

    for i, row in df.iterrows():
        raw = get_raw_row(row)
        if raw not in uniques:
            uniques.add(raw)
            uniques_index_dict[raw] = i
        else:
            index = uniques_index_dict[raw]
            if index not in duplicates_count:
                duplicates_count[index] = 1
            else:
                duplicates_count[index] += 1
    return duplicates_count
```

After running the script, I calculated the values of interest and compiled a summary:

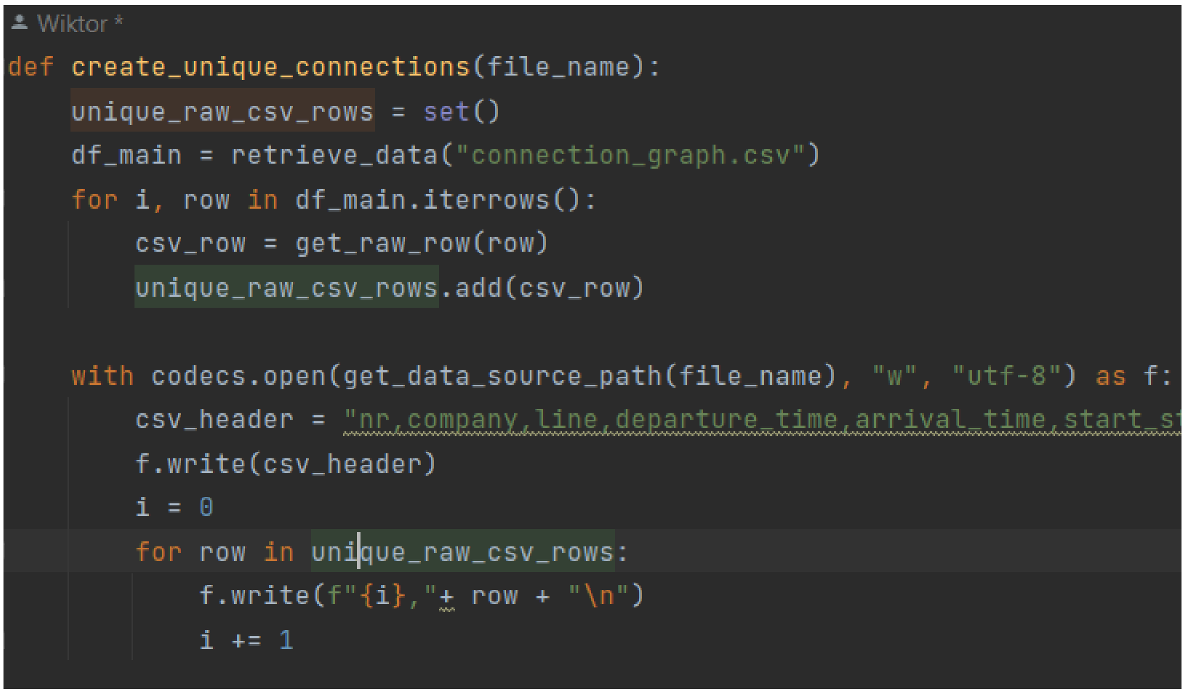


```
Liczba wszystkich wierszy: 996520
Liczba duplikatów: 541289
Liczba unikalnych wierszy: 455231
Maksymalna liczba powtórzeń: 7
Minimalna liczba powtórzeń: 1
```

Above output is in polish, but it says that:

- Number of all rows: 996520
- Number of duplicates: 541289
- Number of unique rows: 455231
- Maximum number of repeats (of a row): 7
- Minimum number of repeats (of a row): 1

Thus, we can eliminate more than half of the file in the context of the issues considered in this dataset.
Using the script I wrote myself, I create a new file that contains no duplicates:



```python
def create_unique_connections(file_name):
    unique_raw_csv_rows = set()
    df_main = retrieve_data("connection_graph.csv")
    for i, row in df_main.iterrows():
        csv_row = get_raw_row(row)
        unique_raw_csv_rows.add(csv_row)

    with codecs.open(get_data_source_path(file_name), "w", "utf-8") as f:
        csv_header = "nr,company,line,departure_time,arrival_time,start_st
        f.write(csv_header)
        i = 0
        for row in unique_raw_csv_rows:
            f.write(f"{i},"+ row + "\n")
            i += 1
```

1.2. Data Modeling

1.2.1. Vertex - Stop

A vertex in the graph, representing a stop, is modeled as a triplet $(name, latitude, longitude)$.
Initially, I wanted the vertex to be unique based on these three values. However, after further consideration and testing, I realized that such an approach would make testing significantly more difficult. Instead of referring to a stop by its name alone, one would have to provide precise geographic coordinates. Moreover, the algorithms I implemented only consider movement using transit lines, meaning that walking between stops is not taken into account.
Therefore, I decided to generalize the stops as follows:

- I collected all stops based on their names and grouped them accordingly.
- I calculated the arithmetic mean of the geographic coordinates for each stop.
- I assigned all connections to each stop based on its name.

Through this generalization, the number of unique stops was reduced from 2284 to 939.
Due to the necessity of maintaining information about vertex adjacency, in the final model, a single vertex in the graph is represented as a mapping from a vertex to a list of neighboring vertices.

1.2.2. Edge - Connection to a List of Implementations

A single edge in the graph connects two stops in a directed manner.
This introduces a challenge—after all, a trip from Plac Grunwaldzki to Galeria Dominikańska can occur at different times!
Thus, my edge model represents a connection as a mapping from a given route to a list of possible implementations.
An implementation is modeled as a quadruple $(company, line, departure\_time, arrival\_time)$.
Meanwhile, a connection is represented as a pair $(start, stop)$.

2. Finding the Shortest Connections

2.1. Dijkstra's Algorithm

The key components used in the implementation of the algorithm are:

- Priority queue (minheap)
- Cost map
- Predecessor map
- Algorithm for selecting the best connection execution
- Algorithm for calculating the cost
- Algorithm for reconstructing the path

Each component will be discussed in the following sections.

2.1.1. Priority Queue

This is a crucial element that allows the algorithm to easily select the node to visit next. This data structure is essential for finding the shortest path (with the lowest cost).
I use a built-in priority queue of type minheap, which is **provided by the heapq library**.
A minheap is a heap that contains comparable elements and always keeps the smallest element at the top.
Extracting the smallest element is computationally inexpensive, although it requires rearranging the heap elements afterward.
Thus, in this algorithm, the heap maintains the stop with the lowest cost as its smallest element.

2.1.2. Cost Map

I use a dictionary to store the cost associated with each vertex, allowing easy tracking and updating of the current path cost. In other words, for each vertex, the cost of reaching it from the predecessor is stored.

2.1.3. Predecessor Map

For each visited vertex (except the first one), I store its predecessor along with the execution that allowed reaching it.
This is necessary for reconstructing the path.

2.1.4. Algorithm for Selecting the Best Connection Execution

In Dijkstra's algorithm, when selecting a connection execution between two stops, the choice is made according to the following criteria:

- **The execution must take place in the future**
- **The execution with the shortest travel time plus waiting time is selected**

```
def find_best_future_execution_time(current_time, executions):
    lowest_cost = infinity
    best = None
    for execution in executions:
        if execution.departure_time >= current_time:
            cost = execution.departure_time - current_time +
            (execution.arrival_time - execution.departure_time)
            cost_int = int(cost.seconds)
            if cost_int < lowest_cost:
                lowest_cost = cost_int
                best = execution
    return best
```

2.1.5. Algorithm for Calculating the Cost

The cost is essentially the execution time plus the waiting time. This cost, along with the corresponding vertex, is added to the priority queue.

2.1.6. Algorithm for Path Reconstruction

This involves traversing the predecessor dictionary until reaching the source node:

```
current = destination
path = [previous[current]]

while True:
    current = previous[current][0]
    if current == source:
        break
    path.append(previous[current])

path.reverse()
return path
```

At the end, the path is reversed, as we need it from the starting point to the destination rather than the other way around.

2.1.7. Implementation Challenges

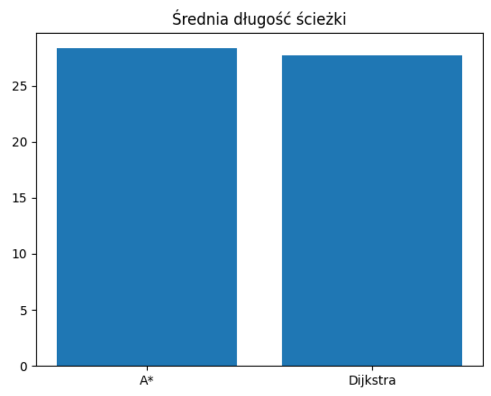
Initially, I represented vertex adjacency using a set.
This caused an issue where, for the same start node A, destination B, and time t, the algorithm sometimes returned different paths. This is because there may be multiple shortest paths in the graph, and the algorithm terminates as soon as it reaches the destination. Consequently, the order of vertices in the adjacency structure mattered. However, since **a set does not guarantee any order**, I decided to use a list instead to ensure the algorithm remains deterministic.

2.2. A* Algorithm

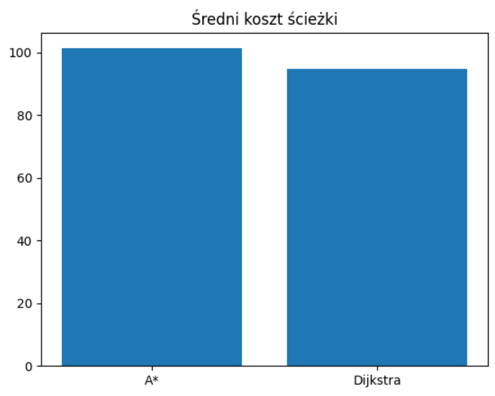
The A* algorithm is essentially Dijkstra's algorithm with an additional component, **a heuristic**.
This is a function that estimates the cost of the path from the next vertex to the destination.
Thus, the total cost is the local criterion plus the heuristic value.
In my case, I use the **Euclidean distance heuristic** since our space allows for infinitely many directions (geographical coordinates are real numbers).

2.2.1. A* with Time Criterion

This algorithm functions almost identically to Dijkstra's algorithm (with the time criterion), with the difference that the heuristic (in this case, the Euclidean distance between the next and destination nodes) is added to the cost.
The selection of the connection execution in A* with the time criterion is the same as in Dijkstra's algorithm.
One might think that since it differs so little from Dijkstra's algorithm, it should yield similar results—let's find out.
As before, **I ran Dijkstra and A* with the time criterion for 750 test cases and recorded data such as path length, path cost, and computation time**. Let's analyze the results using the graphs below. The above graphs present average values for 750



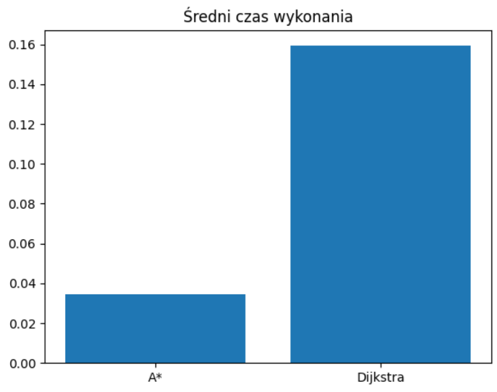
Rysunek 1: Average path length



Rysunek 2: Average path cost

executions.
As observed, **A* exhibits slightly worse quality metrics regarding the path itself**, which is expected.

However, the comparison of computation time remains. The above graph compares



Rysunek 3: Average computation time

the average computation time for the paths. As we can see, **the A* algorithm is significantly faster than Dijkstra's, with only small deviations from optimal quality metrics**!

2.2.2. A* with Transfer Criterion

This version of the algorithm aims to minimize the number of transfers during the journey.
Compared to A* with the time criterion, the following modifications are necessary:

- Modify the algorithm for selecting the best connection execution
- Modify the function for calculating the connection cost based on transfers

Thus, compared to the previous version of the algorithm, the connection execution selection algorithm first searches for the best time-based executions that do not require a line change. However, if none exist, it delegates the selection to the previous function version, meaning that if no execution allows continuing on the same line, it simply chooses the one that is best in terms of time.
Regarding the modification of the cost calculation function, it operates similarly to the previous one by calculating the cost based on time. However, **if a transfer is required, an additional cost is added**.
This additional cost partially overrides the heuristic. When the cost is low, the probability of choosing a transfer increases; when it is high, the probability decreases.

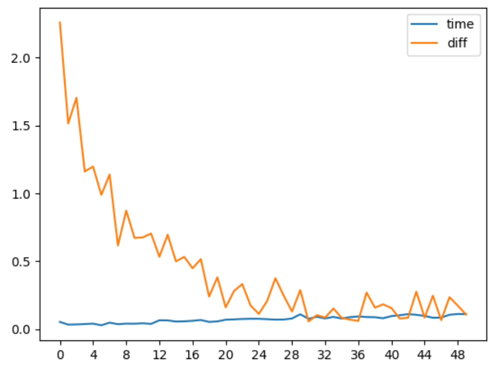
However, as the cost increases, the number of visited vertices also rises, since the heuristic has less influence.

2.2.3. Modification of the A* Algorithm to Reduce the Cost Function Value or Computation Time

The simplest modification here is to demonstrate that if the transfer cost is reduced, computation time decreases.
However, such an answer does not satisfy me, and I would like to conduct statistical studies to show the relationship between the transfer cost value and quality metrics. Thus, as part of my research, I will conduct two tests:

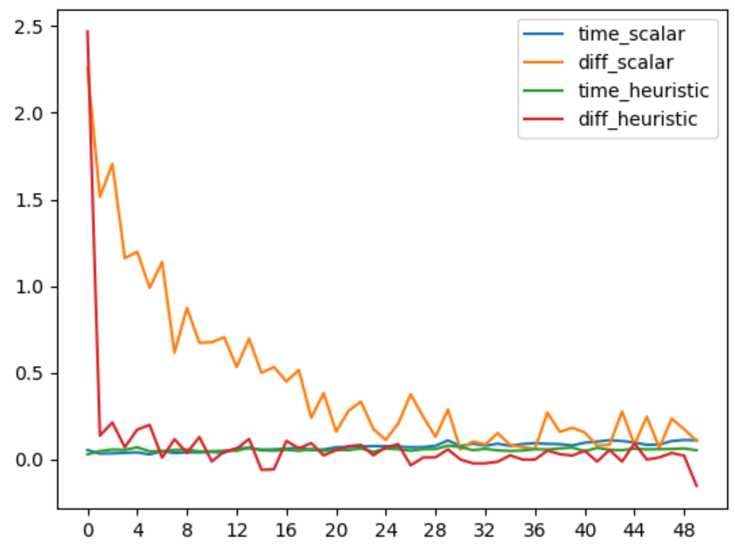
- Quality analysis for cost in the range $< 0; 49 >$
- Quality analysis for cost as a function of the heuristic – the cost value will be multiplied by the current heuristic value $c = n * h$

For each case, I will calculate the number of transfers for the cost $c = 500$, since for such a high value, avoiding transfers will have a very high priority. I will compare this number with the current algorithm result—the smaller the difference, the better. Another value I will compute is the algorithm's execution time. For each test cost value, I will analyze over 90 random cases to obtain an average.



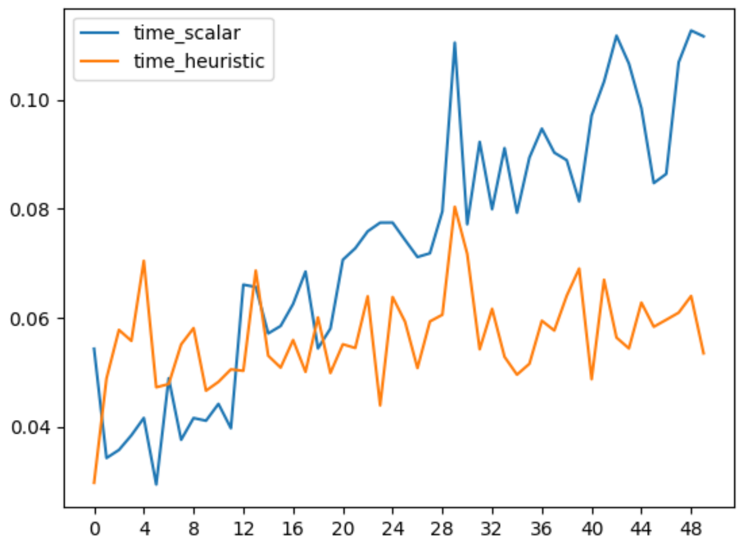
Rysunek 4: Average deviation from the 'optimal' number of transfers and computation time depending on the constant cost value

As seen from the previous graph, as this constant increases, computation time increases, and average deviation decreases.
The point at which deviation stops decreasing significantly is around $c = 20$.
It makes no sense to choose a higher value, as the minor improvement in deviation quality comes at the cost of increased computation time.
Now, let's compare how this behaves when the cost is a function of the heuristic in the form $c = n * h$.



Rysunek 5: Comparison of graphs

As observed, when the cost is a function of the heuristic, it achieves much better results, **even reaching negative values!** – this means it performs better than the constant $c = 500$! (for which, by the way, computation time is similar to Dijkstra's!)
Therefore, it is undoubtedly better to compute the cost as a function of the heuristic, but what value of c should be chosen to sufficiently improve result quality while keeping computation time low?
From the graph, potential critical points are $c = 15$, $c = 26$, and $c = 49$.
It is worth examining a graph scaled to computation time before making a decision.



Rysunek 6: Comparison of computation time graphs

Among the previously mentioned characteristic points, the best in terms of computation time is $c = 26$. Since A* is intended to be fast, I choose this point. Notably, the deviation for this point is negative!
Thus, to optimize the results of the A* algorithm with the transfer criterion, **the transfer cost should be computed as a function of the heuristic**.
Although time constraints allowed me to analyze only the cost function as a linear function of the heuristic, one could explore other function classes, such as logarithmic, and increase the precision of the exhaustive search. However, this is a topic for scientific research that could take a significant amount of time!

3. Summary

The task required proper modeling of the graph to leverage the abstraction of a general graph for this specific problem.
In my opinion, an important aspect of this task was also data exploration. I managed to reduce the file size by over 50% without losing any data!
This significantly sped up the process of loading data into the graph.
Regarding the implementation of the algorithms, the most challenging part for me was debugging. In such a large graph, it was difficult to keep track of what happens where.

3.1. Comments on the Algorithms

Both algorithms—Dijkstra and A*—have their advantages and disadvantages, and the choice of algorithm depends heavily on the circumstances.
If the graph is not very complex or we need optimal results (in terms of path cost), we should use Dijkstra's Algorithm.
However, if we have a large graph and can tolerate some acceptable quality deviations, we should use the A* Algorithm.
It is worth noting that A* is highly dependent on the heuristic. Depending on the specific problem, different heuristics may be used—there is no truly universal one.
Therefore, various studies should be conducted on a given heuristic, as a poorly chosen heuristic can cause A* to perform worse both in terms of time and quality than Dijkstra!
In the case of A*, based on my observations, it is worth considering a hybrid approach—evaluating whether the local criterion can be combined with the heuristic in some way to optimize quality metrics.
It is crucial to make skillful use of data, compile meaningful information, conduct experiments in relevant spaces, and interpret the results accordingly.