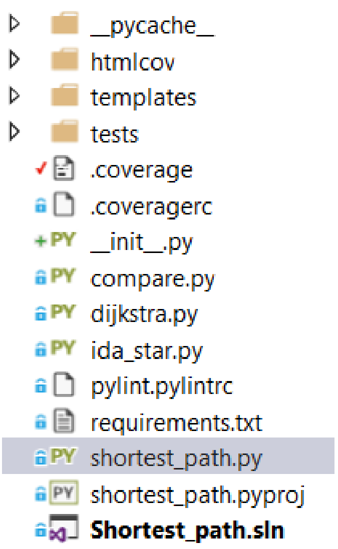
Toteutusdokumentti _ Implementation document

Data Structures Lab Course (start of summer 2023)

Structure of the program:

The current files are structured as follows in Visual Studio.



```
__pycache__
htmlcov
templates
tests
.coverage
.coveragerc
__init__.py
compare.py
dijkstra.py
ida_star.py
pylint.pylintrc
requirements.txt
shortest_path.py
shortest_path.pyproj
Shortest_path.sln
```

- **dijkstra.py:** Dijkstra algorithm implementation, priority queue implemented on heap.
- **ida_star.py**: the file of implementing IDA* (Iterative Deepening A*) algorithm.
- **shortest_path.py**: the application file which contains all the codes for the application, incl. user interface, various functions and route. It can work with either IDA_star.py or Dijkstra.py, depending which algorithm you prefer; choose from below lines:

```
# Use Algorithm:
if start_coordinates not in graph.nodes or end_coordinates not in graph.nodes:
    # Handle the case when the coordinates are not present in the graph
    flash('Invalid start or end coordinates. Please provide valid coordinates.', 'error')
    return redirect(url_for('home'))
else:
    # if use ida_star function (IDA* Algorithm) to find the shortest path, run below line:
    shortest_path = ida_star(graph, start_coordinates, end_coordinates)
    # if use Dijkstra’s Algorithm to find the shortest path, run below line:
    # shortest_path = dijkstra(graph, 'start', 'end')
```

- **compare.py**: not a part of the program. check the execution time of each algorithm, compare the execution time.

Processes to implement the program:

- Import the necessary libraries: Flask, folium, polyline, geopy.geocoders.Nominatim etc.
- Create a Flask application instance and html file as the user interface.

- Define the geocoding function (geocode) that takes an address as input, and returns corresponding coordinates using the Nominatim geocoding service.
- Define the route handling for the home page ("/") that handles both GET and POST requests. In the GET request, render the initial index.html template. In the POST request, retrieve the start and end addresses from the user interface’s form, geocode the addresses, and proceed with finding the shortest path.
- Implement the find_shortest_path function with IDA* (Iterative Deepening A*) algorithm or Dijkstra algorithm, which is the core and target for this course. This function sends a request to the external API to calculate the driving route between the start and end coordinates. It retrieves the encoded polyline from the response and decodes it using polyline.decode.
- Implement the plot_shortest_path function, which creates a Folium map, adds markers for the start and end coordinates, and plots the polyline representing the shortest path. The map is saved as an HTML file to templates folder, which will be rendered in browser.
- Run this Flask application using app.run(). Users can test the program by entering start and end addresses, submitting the form, and verifying that the map with the shortest path is displayed.

Time and space complexities:

Because this project uses geocoding and HTTP requests, complexities can be impacted by these both actions. It requires space to store the addresses and coordinates. The biggest impact would be the path finding algorithm. The time complexity for geocoding each address would be in the range of O(1) to O(log n), where n is the size of the geocoding database; space complexity for each address is O(1).

The previous codes implemented with networkx library (built-in Dijkstra's algorithm) bring the time complexity of O((V + E) log V), V is the number of nodes, and E is the number of edges. The space complexity is O(V + E). In the last step of plotting, the amount of the polyline and nodes will impact on the time complexity, and space complexity is decided by the amount of the coordinates and the map file.

Below table lists the main differences between the Iterative Deepening A* (IDA*) algorithm and Dijkstra's algorithm:

	Dijkstra's algorithm	**IDA* algorithm**
Searching	explores nodes in increasing order of the distance from the start node. It considers all unvisited nodes and selects the one with the shortest distance as the next node to explore. breadth-first manner.	Explores in a depth-first manner, guided by a heuristic function that estimates the remaining cost to the goal. It gradually increases the search depth (threshold) to find the shortest path.
Heuristic	not require heuristic function, using a binary heap. An element	requires a heuristic function that provides an estimate of the remaining cost from each node to the goal.

	with high priority is dequeued before an element with low priority.	
Memory/ Space Complexity	maintains a priority queue to keep track of the nodes to be explored, based on the distances from the start node. O(V + E)	requires less memory compared to Dijkstra's algorithm, as only keeps track of the current path being explored. In this program, dominated by the number of nodes in the graph, resulting in O(V + E)
Time Complexity	time complexity O((V + E) log V), where V is the number of nodes (vertices) and E is the number of edges.	In the worst case, IDA* could have an exponential time complexity if heuristic function does not provide a good estimate. Sorting the neighbors based on the combined cost of the path from the start node and the heuristic estimate takes O(d log d) time, d is the number of neighbors. Reconstructing the shortest path from the visited nodes takes O(V) time, V is the number of nodes.

Plotting shortest path has a linear time complexity and space complexity of O(n).

Overall, the time complexity and space complexity for this program depends on the geocoding, the chosen shortest path algorithm (IDA* or Dijkstra's), and the number of coordinates in the path.

Performance:

Based on above analysis, this program's performance depends on the geocoding, the chosen shortest path algorithm (IDA* or Dijkstra's), and the number of coordinates in the path.

If the internet connection slows down or the map API works inefficiently, it will impact on how quick geocoding and html file can be executed.

Possible flaws and improvements:

In order to display the updated html file correctly, codes were modified to allow generate new html file to local folder for new address's submission, which is not an ideal solution if considering the huge memory demand for saving those html files. Thus, this current solution is not for a big amount of users. From another perspective, the advantage is service administrator is able to record and retrieve all the path finding records. As an alternative, in real life the service administrator can remove or delete those html files with a certain frequency if those are not needed for user management.

This program is capable to display the main turning points, but not each individual node. But it gives users an overall direction. If the distance between the start and end points is very long, it can be challenging to display all the nodes along the path due to visualization limitations.

Sources:

Flask: Quickstart — Flask Documentation (2.3.x) (palletsprojects.com)

Flask Tutorial in Visual Studio Code: https://code.visualstudio.com/docs/python/tutorial-flask