Lukas Ruppert
Raphael Braun
Timo Sachsenberg
Wolfgang Fuhl
Dimitrios Koutsogiannis
Frieder Wallner
Martin Röhm

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Programming in C++

Sheet 11

Submission date: 13.09.2023 16:00

11.1 Shortest Paths (80 Points + 20 Points + 10 Bonus Points)

C++

As the final task, we tackle the problem of computing shortest paths within graphs.

Here, the graph is given as a full adjacency matrix. Every node is a `class Location` with a name and a position (for the A* heuristic). It also has a `std::vector<std::optional<float>> distances`, representing the distances to each other node. You can access edges in the adjacency matrix using e.g. `adjacency_matrix[1][2]`. Here, `1` and `2` are the source and target node ids. All potential connections between all nodes are represented explicitly. For nodes which are not connected, the `std::optional` has no value. For the length of the shortest path, only the values in the `distances` vector matter, not their positions. While we could have again derived from `std::vector` to remove some boilerplate code, here, we kept things a bit simpler and self-contained.

a) You probably all already know about Dijkstra's algorithm, but as a quick refresher, you need to perform the following steps: (In the following, list usually means `std::vector`, never `std::list`.)

Setup a list of points to explore (the "queue") and initialize it with the starting point.

While there still are points in the queue, fetch the point with the shortest distance from the start point. Remove that point from the list and remember that this point has been processed. Do not process points twice.

Add all neighbors of that point to the queue. Check if the distance to that neighbor via the current point is shorter than their current shortest path. If yes, set the current point as that neighbor's predecessor and update its distance.

You can terminate the search once you visit the goal point. All shorter paths that may lead to the goal have already been explored at that point, as we do not allow negative distances.

After completing the search, construct a list of points explored along the way by tracking back through the predecessors of each point. Return that list ordered from start to finish, including start and finish. (You may have to `std::reverse` the list.)

If the goal cannot be reached, return an empty list instead.

Implement all this in `shortest_paths.cpp` in

```
std::vector<size_t> compute_shortest_path(uint32_t from, uint32_t to) const;
```

You can test your code in the main function and, of course, on InfoMark.

Please note: The location names are just there for having a nice, understandable output for a concrete application. In your algorithm, operate *only* on ids. Not all city names are unique and you may run into an infinite loop if you use them. Also, the distances between two nodes given by the edges are not necessarily the same as the euclidean distance between the two locations. Generally, the distance values should be larger or equal to the euclidean distances.

b) As an extension, the A* algorithm adds a heuristic to each point's distance that represents the expected distance towards the goal.

Add the distance heuristic and compare for yourself how many steps are needed to search for shortest paths between cities compared to the basic Dijkstra algorithm. Your submission will be graded based on the number of steps taken (number of nodes explored) by your implementation. (Finding the shortest path from Berlin to Munich should take 323 steps with Dijkstra's algorithm, but only 41 steps with A*. Both should return the same path, once found.)

Do not submit your A* implementation separately. Just extend your implementation of Dijkstra. (Maybe make a copy to fall back to, in case things go wrong.)

c) C++ is all about being efficient. Don't waste time or space and earn 10 bonus points for implementing the queue as a (max-)heap.