Institute of Theoretical Informatics
Scalable Algorithms Group

Adrian Feilhauer,
Patrick Steil,
Michael Zündorf

Worksheet

Practical Course: Route Planning (WS 26/27)

Launch Wednesday, October 28, 2026
Deadline Tuesday, November 17, 2026, 23:59 (CET/MEZ)

Deadline & Points

We announce all deadlines well in advance. Therefore, we insist that the exercise sheet and other materials are submitted on time. Any submission after the deadline will be considered as not submitted!

Anyone who scores less than 2.5 million points on this exercise sheet is not admitted for the practical course.

General Stuff

With your ITI account, you can log in to the computers in the pool room. Alternatively, you can also access the computers via SSH. To connect to the chair's network from outside, connect to `i11ssh.iti.kit.edu`. However, no computation should be done on this machine! Therefore, connect to one of the pool room computers. The computers have the addresses `i11poolX`, where `X` ranges from 1 to 12. The commands for SSH access are as follows:

```
ssh username@i11ssh.iti.kit.edu
ssh i11poolX
```

If the SSH access doesn't work, it might be because the corresponding computer has been turned off. In that case, try another one. Before performing computations, please check with `htop` that the computer isn't already being used by someone else. If it is, please switch to another computer.

Your ITI account is created with a default password. Before you start, please change the password. To do so, run `passwd` on one of the pool room computers.

To transfer files via SSH, you can use `scp`:

```
scp von_datei_daheim username@i11ssh.iti.kit.edu:nach_datei_im_poolraum_home
scp username@i11ssh.iti.kit.edu:von_datei_im_poolraum_home nach_datei_daheim
```

On `i11ssh` (not on the pool room computers) under `/algoDaten/praktikum/graph` are various input files relevant for the practical course. Please only use these within the scope of the practical course. For code management, we provide a Git repository. The submission of the worksheet is also done through the repository. For each task, you should submit both the source code of your program and the computed solutions.

Implementation

For the implementation of the tasks, you can use either Rust or C++. For both languages, we provide a small base framework that you can use for I/O, etc. (see below). The `README.md` file in the respective repository contains further information on how to use it.

C++

The source code should be compilable by running `./compile.sh` on one of the pool room computers. The pool room computers have GCC 14.2.0 installed, which fully supports C++20. Newer C++ features that are not supported by the compiler are not allowed. Using external libraries is not permitted. The C++ standard library is not considered external.

Rust

The source code should be compilable by running `cargo build --all` on one of the pool room computers. The pool room computers have rustc version 1.75.0 installed.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh # rustup installieren
```

The use of unstable Nightly features is not allowed. Using external crates is not allowed. The Rust standard library is not considered external.

Git

To access your Git repository, run the following command on your local machine:

```
git clone https://username@i11git.iti.kit.edu/git/Praktika/Routenplanung/Teilnehmer/<username>
```

This creates a local directory called `<username>`, which is initially empty. Now, import the C++ or Rust base framework with one of the following commands:

```
git pull https://username@i11git.iti.kit.edu/git/Praktika/Routenplanung/cpp-base master # C++
git pull https://username@i11git.iti.kit.edu/git/Mitarbeiter/Tim-Zeitz/stud-rust-base master # Rust
```

Now you can make any changes you like in your local directory. To synchronize these changes with our server, you need to commit them. First, add the modified or newly created files:

```
git add file1.cpp
git add file1.h
```

Then, bundle the changes into a commit by running:

```
git commit -m "Created file1"
```

This commit initially exists only locally on your machine. To synchronize it with the server, run the following command:

```
git push
```

For more information on Git, you can refer to: `https://rogerdudler.github.io/git-guide/index.html`.

Important: Do not add files larger than 100MB to a Git repository.

Graph Format

Nodes and edges are identified by numerical IDs ranging from 0 to $n-1$ and 0 to $m-1$, where n is the number of nodes and m is the number of directed edges. We store weighted, directed graphs using an adjacency array representation. A weighted directed graph consists of the vectors `first_out`, `head` and `weight`. To iterate over the outgoing edges of a node, you can use the following C++ code (for Rust, example code can be found in the `README`):

```
vector<unsigned> first_out = load_vector<unsigned>("first_out_file_name");
vector<unsigned> head = load_vector<unsigned>("head_file_name");
vector<unsigned> weight = load_vector<unsigned>("weight_file_name");

unsigned my_node = 42;
for(unsigned out_arc = first_out[my_node]; out_arc < first_out[my_node+1]; ++out_arc){
    cout<< "There is an arc from " << my_node
        << " to " << head[out_arc]
        << " with weight " << weight[out_arc]
        << endl;
}
```

Note: `head` and `weight` have m elements. `first_out` has $n + 1$ elements. The first element of `first_out` is always 0, and the last one is always m. For all graphs, there are two different edge weights: travel time and distance. Furthermore, some graphs also include the geographical position for each node. This is stored as two `float` vectors, which represent the latitude and longitude for each node.

In the directory `/algoDaten/praktikum/graph`, you will find several graphs in this format. Some are for testing purposes, while others are used for task evaluation. The test graphs roughly correspond to Stupferich, Karlsruhe & surroundings, Germany & surroundings, and (Western) Europe. The task graphs are of the size of the Germany graph.

Warning: The Europe graph may be too large for the main memory of some pool room computers.

Task 1: Dijkstra's Algorithm (2 000 Points)

Implement Dijkstra's Algorithm to compute the shortest distance between two nodes in a graph.

To test the correctness of your implementation, test data can be found in the `test` subdirectories of the 4 test graphs. These consist of 4 vectors: `source`, `target`, `travel_time_length`, and `geo_distance_length`. They can be loaded using `load_vector<unsigned>`. The data is structured as follows: At position `travel_time_length[i]` and `geo_distance_length[i]` the travel time and distance respectively from node `source[i]` to `target[i]`. If your implementation computes the same value as the provided reference solutions for each test query and graph, your program is likely correct.

For Task 1, there is a graph in `/algoDaten/praktikum/graph/aufgabe1` whose `test` subdirectory does not contain `travel_time_length` nor `geo_distance_length` files. It is your task to create these two files. Save the files you compute in your Git repository under `/abgabe/aufgabe1/travel_time_length` and `/abgabe/aufgabe1/geo_distance_length`.

We will evaluate the submitted files `/abgabe/aufgabe1/travel_time_length` and `/abgabe/aufgabe1/geo_distance_length` by comparing them with our reference solution. For Task 1, you can therefore achieve up to 2 000 points, as there are 1 000 queries and two edge weights. If the answer for any query differs even slightly from the correct value, you will not receive

any points for that query.

Important: Generate the files using the `save_vector` function (C++) or `Vec::write_to()` (Rust). There will be no points awarded for custom file formats.

Task 2–4: Contraction Hierarchies (2 000 000 Points each)

These tasks follow the same structure as the first one: Compute the `travel_time_length` and `geo_distance_length` vectors for the remaining graphs. However, there are 1 000 000 queries for each of these graphs, making Dijkstra's Algorithm too slow for the computation. Therefore, use Contraction Hierarchies (CH) to accelerate the computations. The three tasks differ in terms of which parts of the CH you need to implement.

- For Task 2, the node order and the graph augmented with shortcuts are given. The data can be found in the `travel_time_ch` and `geo_distance_ch` subdirectories. In this task, you only need to implement the query algorithm. The order is given as a list of node IDs in the order they were contracted (and not as the rank of each node). To get the ranks, you need to invert this permutation.
- For Task 3, the augmented graph is no longer provided, only the node order. So, you need to compute the augmented graph yourself. You can reuse the implementation of the query algorithm from Task 2.
- For Task 4, only the input graph is provided. So, you need to compute both the augmented graph and the node order.

Details about the CH algorithm can be found in the introduction slides.

Submission Checklist

Before submitting your answers, please ensure that…

- …the submission deadline has not passed. Late submissions will not be graded.
- …the submitted files are 4 000 or 4 000 000 bytes long and conform to the required format. No points will be awarded if we cannot determine which answer corresponds to which query.
- …the submission files are located in your Git repository at the following path:

  ```
  /abgabe/aufgabe{1,2,3,4}/{travel_time,geo_distance}_length
  ```

- …your Git repository contains the source code you wrote.
- …running `./compile.sh` or `cargo build --all` compiles all programs.
- …your code can be compiled on a pool room computer.

Then, send an e-mail to `michael.zuendorf@kit.edu` with a message stating what you have submitted, when, and where.