

Lab 4

Course goals

- To implement and use data structures and algorithms in application programs: graph algorithms.
 - Shortest path algorithms.
 - Minimum spanning tree algorithms.

Preparation

Please ensure you read the entire lab description thoroughly, before starting any exercises, including the preparation steps given below.

Completing the following tasks before the Lab4 HA session will help you make the most of your lab time.

- Download the [files for this exercise](#) from the course website. Like previous labs, you can then use CMake to create a solution with two projects for this lab, one for the exercise in part A and another for the exercise in part B.
- For each project, it's possible to compile, link, and execute the program. There is a `main.cpp` file with the `main` function, for each of the projects.
- Implement the graph algorithms requested in [Part A](#).
- Read [Part B](#) and understand the code given for this part.

Please note that in the exercises for this lab, there is no vertex labeled zero. Therefore, slot zero in data structures such as the `dist` vector should remain unused.

If you have any specific question about the exercises, then send us an e-mail. Be short and concrete, otherwise you won't get a quick answer. You can write your e-mail in Swedish. Add the course code to the e-mail's subject, i.e. "TND004: ...".

Part A: shortest path algorithms

In this first part, you are requested to implement the UWSSSP and the Dijkstra's algorithm (in the lectures, we also named the Dijkstra's algorithm as PWSSSP). To this end, follow the steps below.

- Review [lecture 12](#) and [lecture 13](#).
- Read sections 9.1 and 9.3.1, 9.3.2, and 9.3.5 of the course book.
- Study carefully the code given for this exercise: `edge.h`, `digraph.h`, and `digraph.cpp`.
- **Implement** functions `Digraph::uwssp`, `Digraph::pwssp`, and `Digraph::printPath`.
 - Use container [std::queue](#) for the implementation of function `Digraph::uwssp`.

An example of the input graph for the exercises in this part is shown below.

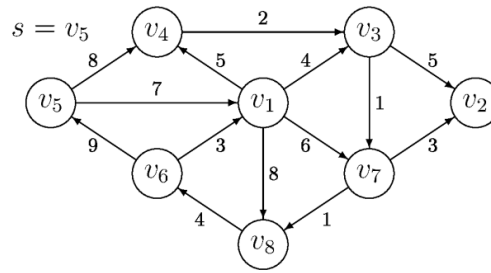


Figure 1: A (weighted) directed graph G and a start vertex s .

The corresponding output, a shortest path-tree, is exemplified in figure 2.

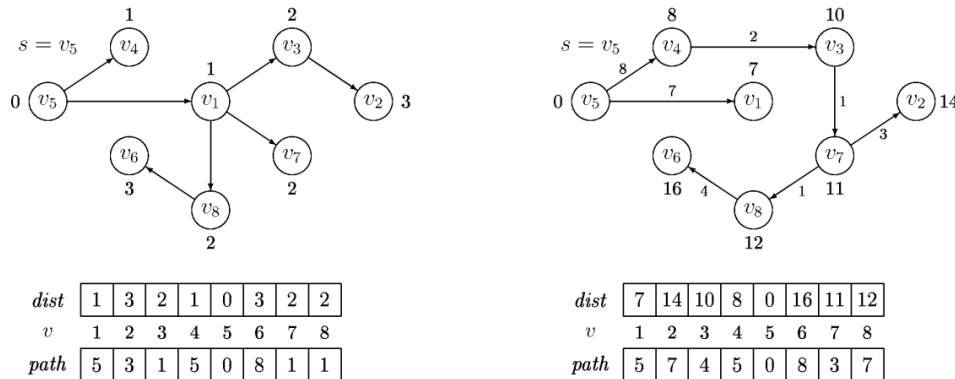


Figure 2: Unweighted (*left*) and weighted (*right*) shortest path-tree for s .

In this lab, graphs are represented with adjacency lists. This graph representation was introduced in [lecture 12](#). Note that the loop below (in pseudo-code) to iterate through the adjacency list of a vertex v (used in several of the graph algorithms you have seen during the lectures of the course)

```
for all  $(v, u) \in E$  do {
    ...;
}
```

can be implemented as (table is a [std::vector](#) with all vertices of the graph and [std::list](#) is used to store the adjacency lists)

```
for (auto& e : table[v]) {
    int u = e.to;    // e = (v, u)
    ...;
}
```

The files `digraph1_test_run.txt` and `digraph2_test_run.txt` contain an example of the program execution, for the graphs provided in the files `digraph1.txt` and `digraph2.txt`, respectively.

Part B: minimum spanning tree algorithms

In this second part, you are requested to implement the Prim's and Kruskal's algorithms. To this end, follow the steps below.

- Review [lecture 14](#).
- Read sections [8.1-8.5] and 9.5 of the course book.

- Study carefully the code given for this exercise: `dsets.h`, `dsets.cpp`, `edge.h`, `graph.h`, and `graph.cpp`.
- Implement functions `Graph::mstPrim` and `Graph::mstKruskal` and test that they work as expected. Both functions build a minimum spanning tree¹ (MST) from a given connected weighted undirected graph $G = (V, E)$. They should display the edges of a minimum spanning tree and the total weight of the MST built. The implementation of `Graph::mstKruskal` requires a min-heap of vertices and [STL functions for heap operations](#) can be used (see also [std::make_heap](#)).
- Re-implement function `Dsets::join` so that **union by size** is performed. Re-implement also function `Dsets::find` so that find with path compression is performed. The code distributed with this lab uses simple union and simple find for these functions.

Like as for [Part A](#), graphs are represented with adjacency lists.

Input and output for the Prim's and Kruskal's algorithms are exemplified in the figure below. In particular, the output (i.e. a MST) is essentially just a list of $|V| - 1$ graph edges.

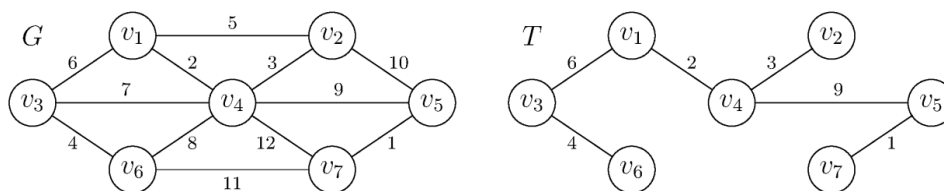


Figure 1: A connected weighted undirected graph G and a minimum spanning tree T for this graph (in this case, the tree is unique).

The files `graph1_test_run.txt` and `graph2_test_run.txt` contain an example of the program execution, for the graphs provided in the files `graph1.txt` and `graph2.txt`, respectively.

Presenting lab and deadlines

The exercises in this lab are compulsory and you should demonstrate your solutions during the lab session *Lab4 RE*. Read the instructions given in the [labs webpage](#) and consult the course schedule.

As usual, we expect that good programming practices are followed. Your code must be readable and well-indented.

If your solution for lab 4 has not been approved in the scheduled lab session *Lab4 RE* then it is considered a late lab. All groups have the possibility to present one late lab on the extra RE lab session scheduled in the end of the course.

Note that **we can only guarantee that each group can present one late lab**. In the extra RE lab session, priority is given to presentation of lab 4, then lab 3, and finally lab 2.

¹ i.e. a spanning tree for graph G of minimum total weight.