The final assignment for week 7, 8 and 9 of CXX01 is to implement an application using the graph library you produced in week 5 and 6. This assignment should be completed in groups of at most two students.

1 Application

Your group will write an application which uses your own graph library. You may choose your own application (which should be approved of by your teacher). Choosing an original application (different from all other groups in your class) will add 0.5 bonus points to your grade for this assignment. If you don't have inspiration or you do not want the bonus, then you can choose one of the applications described in Chapter 4. Your choice should be approved of by your teacher.

2 Test Driven Approach

You should follow a test driven approach when developing your application. You must use a test framework (e.g. µnit, see: `https://nemequ.github.io/munit/`) to manage your testing.

3 Documentation

Document the application you have chosen and describe the algorithms and data structures you used in your own words. Use proper citations and references when appropriate. Also explain the design decisions you made while implementing the application. Describe your testing approach and testing results in your report. Commit and push your report in pdf format into your repository. If you like you can use Doxygen[1] to generate the documentation for your code.

[1] See: `http://www.doxygen.org/`.

4 Application Ideas

If you are not able to come up with a proper idea of an application of graphs yourself, pick one of the applications described in this chapter. Ask your teacher to confirm your choice before you start. You will have to analyze the application and figure out how to implement the algorithms and data structures yourself.

4.1 Graph Coloring

If you have a undirected graph of, say, 20 vertices. How can each vertex be assigned a color in such a way that no two adjacent vertices have the same color? Part of the problem is the fact that there's a limited amount of color options available (e.g. 3).

In the Figure 1 this problem is represented graphically. Each box correspond to a vertex in the graph. Each border between two different boxes corresponds to an edge in the graph.

Please note that it is possible that multiple solutions exist or no solutions at all. Your algorithm needs to find at least one possible solution if such a solution exists. The test data will consist of solvable and unsolvable sets of data.

The program should have two input parameters: the name of a file which contains the graph in a certain format described below, and the number of available colors. The graph should be read from an input file which complies with the GDF format[2]. The big advantages of using this format is that we can use the Gephi[3] program to visualize and analyze the graph. A simple example of an input file is given in Listing 1. This graph can be visualized in Gephi as shown in Figure 2.

The output of the program should be a message reporting if the presented coloring problem is solvable. If the problem is solvable a file which contains the solution must be produced. This file must be in the GDF format and should contain color

[2] See: `https://gephi.org/users/supported-graph-formats/gdf-format/`.

[3] See `https://gephi.org/`

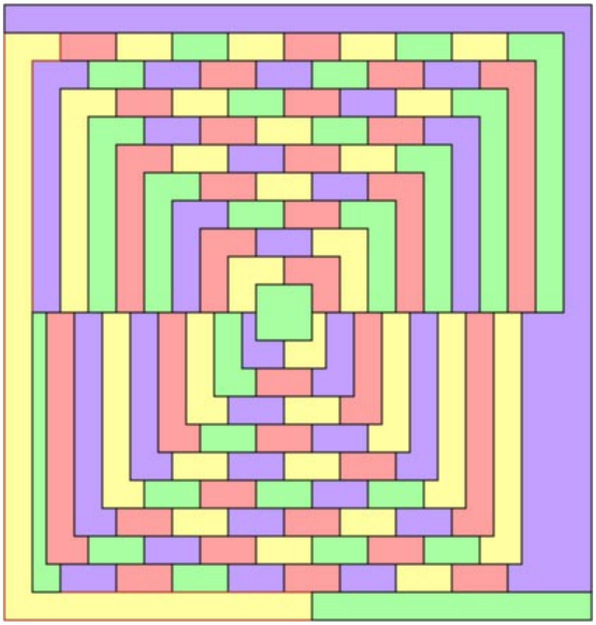

Figure 1: An example of graph coloring.

```
nodedef> name VARCHAR
name1
name2
name3
edgedef> node1,node2
name1,name2
name1,name3
name2,name3
```

Listing 1: Input file `circleOfThree.gdf` in the GDF format which is supported by Gephi.

information. For example the input file `circleOfThree.gdf` can be colored using three colors and a possible solution is presented in Listing 2. This colored graph can be visualized in Gephi as shown in Figure 3.

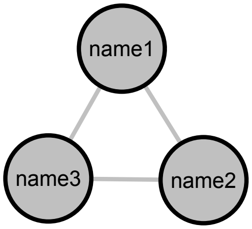

Figure 2: The visualization of the file `circleOfThree.gdf`.

```
nodedef> name VARCHAR,color VARCHAR
name1,'255,0,0'
name2,'0,255,0'
name3,'0,0,255'
edgedef> node1,node2
name1,name2
name1,name3
name2,name3
```

Listing 2: Possible output file `circleOfThreeSolved.gdf` which colors the graph shown in Figure 2 with three colors.

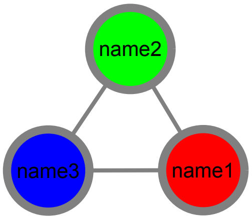

Figure 3: The visualization of the file `circleOfThreeSolved.gdf`.

Figure 4 shows the provinces of the Netherlands. If we assume that Flevoland is only connected to Overijssel and Friesland, then this map can be colored with three colors. The input file in GDF format is given in `provincesNetherlands.gdf`. The graph is visualized in Gephi as shown in Figure 5.

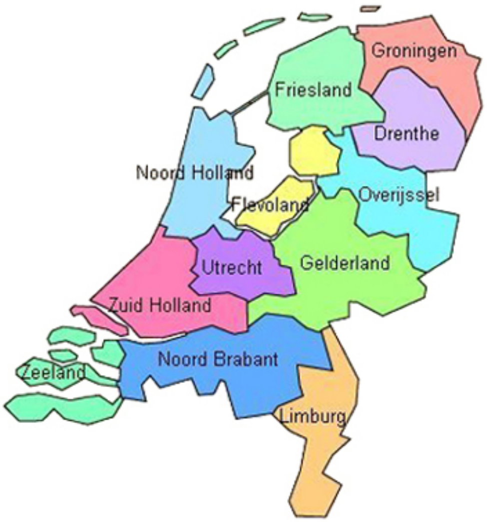

Figure 4: The provinces of the Netherlands.

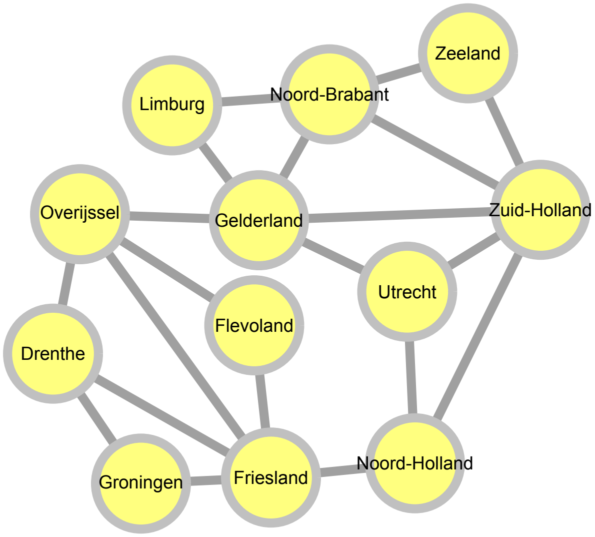

Figure 5: The visualization of the file `provincesNetherlands.gdf`.

4.2 Word Ladders

There is a certain children's game that works with words. The goal is to get from a source word (e.g. ape) to a target word (e.g. man) by hopping between intermediate words that only differ by one letter. Such a chain of words is called a word ladder. Figure 6 shows a word ladder from ape to man: ape, apt, opt, oat, mat, man.

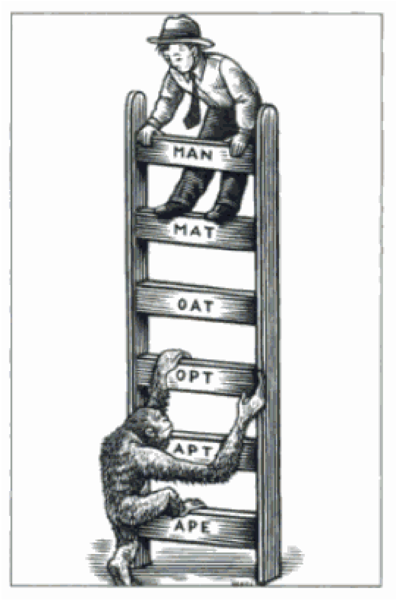

Figure 6: A word ladder from men to ape.

Write a program that asks the user for a source and a target word. Next the program will list all possible word ladders from the source to the destination word (if any).

The game is the most fun when the source and target words have some relation e.g. try to find the word ladder to get from “target” to “source” or from “graph” to “laugh”.

You program will need a list of words to work with. A good word list to use is SOWPODS[4]. This is the word list used in English-language tournament Scrabble in most countries.

You can check your solution by using this on-line word ladder solver: `http://ceptimus.co.uk/wordladder.php`.

4.3 Dijkstra's Algorithm

A very common problem within graph theory is to find the shortest path between two vertices in a graph.

Figure 7 shows a graph in which the vertices are European cities. The connections between the cities which you are allowed to follow with their associated costs (distances) are shown as edges.

For example, the shortest way to go from Amsterdam to Florance in Figure 7 is via the path: Amsterdam, Brussels, Paris, Geneva, Milan, Florence at costs 1705.

The program should have three input parameters: the name of a file which contains the graph in a certain format described below, the name of the starting vertex and the name of the destination vertex. The graph should be read from an input file which complies with the GDF format[5]. The big advantages of using this format is that we can use the Gephi[6] program to visualize and analyze the graph. A simple example of an input file is given in Listing 3. This graph can be visualized as shown in Figure 8.

The output of the program should be the shortest path from the starting to the destination vertex. The program should use Dijkstra's algorithm, see `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`.

[4] See: `https://en.wikipedia.org/wiki/SOWPODS`.

[5] See: `https://gephi.org/users/supported-graph-formats/gdf-format/`.

[6] See `https://gephi.org/`

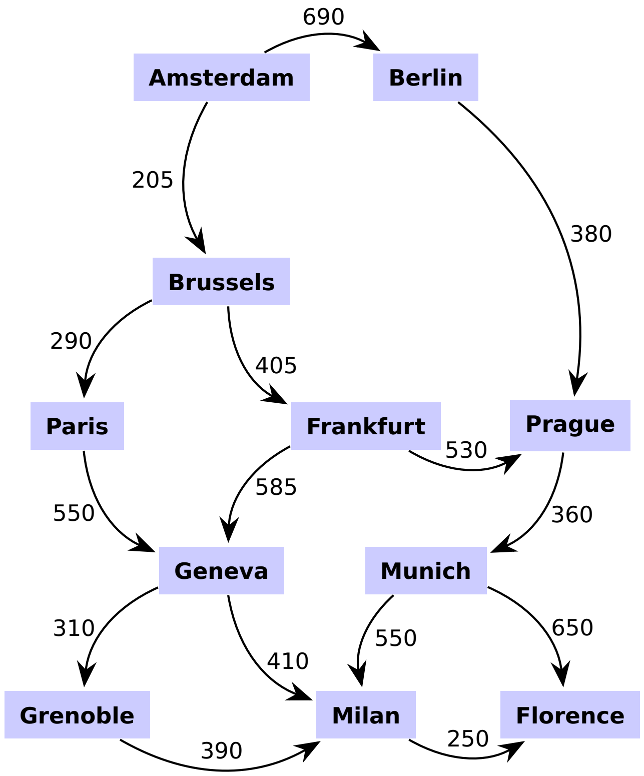

Figure 7: Can you find the shortest way from Amsterdam to Florence in this graph?

```
nodedef> name VARCHAR,label VARCHAR
1,Amsterdam
2,Brussels
3,Paris
edgedef> node1,node2,weight INTEGER,directed BOOLEAN
1,2,1,true
2,3,2,true
1,3,4,true
```

Listing 3: Input file `weightedDiGraph.gdf` in the GDF format which is supported by Gephi.

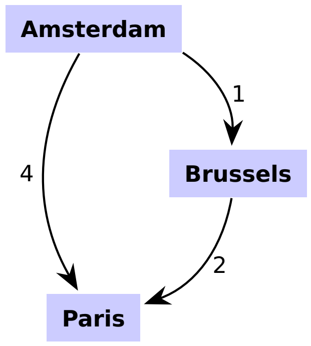

Figure 8: A visualization of the graph which is defined in Listing 3.

The graph shown in Figure 7 is described in the file `cities.gdf`. The shortest path is colored red in the file `citiesShortestPath.gdf`. Figure 9 shows the contents of this file visualized with GUESS[7].

Note: Dijkstra's algorithm can only be used when there are no edges with a negative weight (this can occur when you can earn something by traveling an edge). So your program should check if all weights are positive. It is no problem when there are cycles in the graph.

4.4 Topological Search

A very common problem within graph theory is to find the shortest path between two vertices in a graph.

Figure 7 shows a graph in which the vertices are European cities. The connections between the cities which you are allowed to follow with their associated costs (distances) are shown as edges.

For example, the shortest way to go from Amsterdam to Florance in Figure 7 is via the path: Amsterdam, Brussels, Paris, Geneva, Milan, Florence at costs 1705.

[7] GUESS is part of the Network Workbench, see `http://nwb.cns.iu.edu/`.

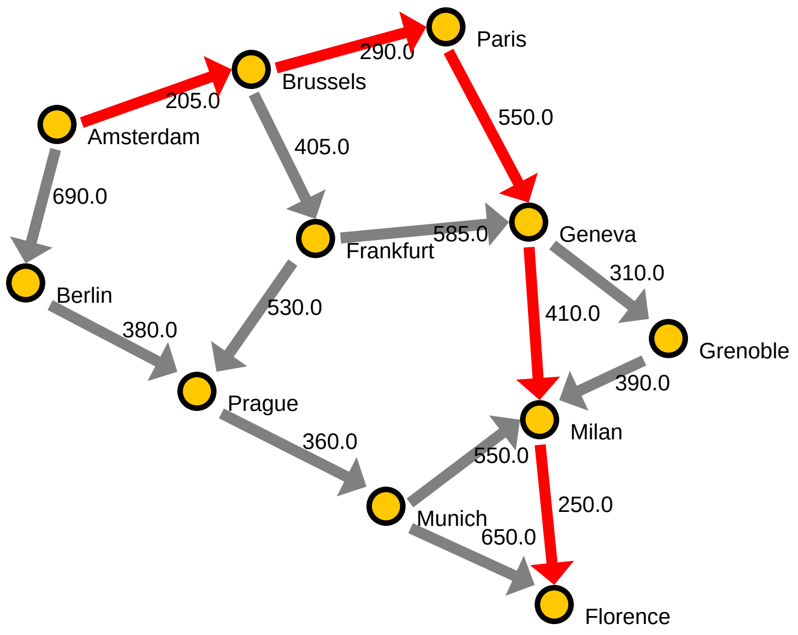

Figure 9: The visualization of the file `citiesShortestPath.gdf`.

The program should have three input parameters: the name of a file which contains the graph in a certain format described in Section 4.3, the name of the starting vertex and the name of the destination vertex.

The output of the program should be the shortest path from the starting to the destination vertex. The program should use the topological sort algorithm, see https://en.wikipedia.org/wiki/Topological_sorting.

Note: A shortest path algorithm based on a topological sort can only be used when there are no cycles in the directed graph. So your program should check if there are no cycles in the graph. A directed graph without cycles is called a DAG (Directed Acyclic Graph). As you can see for yourself, the graph in Figure 7 is a DAG. It is no problem when edges have a negative weight (this can occur when you can earn something by traveling an edge).

4.5 Bellman–Ford Algorithm

A very common problem within graph theory is to find the shortest path between two vertices in a graph.

Figure 7 shows a graph in which the vertices are European cities. The connections between the cities which you are allowed to follow with their associated costs (distances) are shown as edges.

For example, the shortest way to go from Amsterdam to Florance in Figure 7 is via the path: Amsterdam, Brussels, Paris, Geneva, Milan, Florence at costs 1705.

The program should have three input parameters: the name of a file which contains the graph in a certain format described in Section 4.3, the name of the starting vertex and the name of the destination vertex.

The output of the program should be the shortest path from the starting to the destination vertex. The program should use the Bellman–Ford algorithm, see `https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm`.

Note: The Bellman-Ford algorithm can only be used when there are no negative cycles in the directed graph. A negative circle is a circular path in which the sum of all weights is negative. There is no shortest path in this case. You can cycle the negative cycle eternally and the path will get shorter and shorter. Therefore, your program should detect and report a negative circle.

It is no problem when edges have a negative weight (this can occur when you can earn something by traveling an edge). It is also no problem when there are cycles in the graph, as long as every cycle has a positive weight.