Europe by Rail

Prerequisites, Goals, and Outcomes

Prerequisites: Students should have mastered the following prerequisite skills.

- *Graphs* - Knowledge of graph representation, particularly adjacency lists
- *Graph Algorithms* - Knowledge of Dijkstra's shortest path algorithm

Goals: This assignment is designed to reinforce the student's understanding of the implementation of a fundamental graph algorithm
Outcomes: Students successfully completing this assignment would master the following outcomes.

- Understand graph representation
- Understand how to implement Dijkstra's shortest path algorithm

Background

Traveling through Europe by rail is a cheap and effective way to experience the sights, sounds, and culture of a wide array of countries and cities. Generally, travelers purchase rail passes that allow unlimited travel on the rail system. Individual tickets, however, can be purchased.

Description

The program for this assessment calculates the cheapest route between two cities in a mock European rail system. The graph that represents the rail system is pictured below.

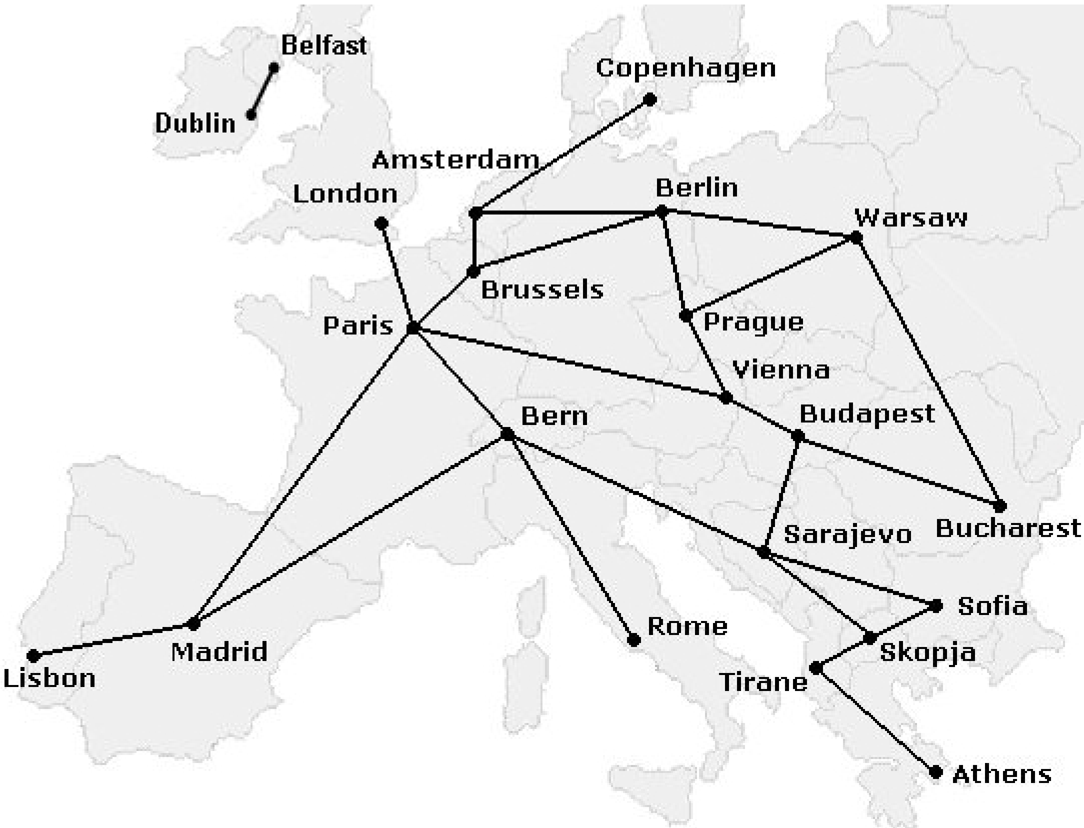

Figure 1 A mock European rail system

In this program, a weighted graph is constructed to represent rail services to and from European cities. Each service from a city has an associated destination city, a fee (in *Euros*), and a distance (in kilometers). The program processes user input of a source city and a destination city. The program then displays the cheapest route from the source city to the destination city. In addition, for each route, the total cost and total distance are displayed. The file `services.txt` contains the data for the available services.

Classes

This program utilizes three main classes, class `City`, class `Service`, and class `RailSystem`. The implementations for class `City` and class `Service` are given. Class `City` maintains information about a city. Class `Service` models a rail service from the rail system. This class contains public data members for a destination city, a fee, and a distance. Both of these classes are used by class `RailSystem`.

Class `RailSystem` models the rail system using an adjacency list representation. These adjacency lists are represented using the STL class `map`. Specifically, a `map` of type `string` to type `list<Service*>` represents the rail system. The following image represents how a portion of the rail system is represented. Note that the ticket fees and distances are left out to simplify the picture.

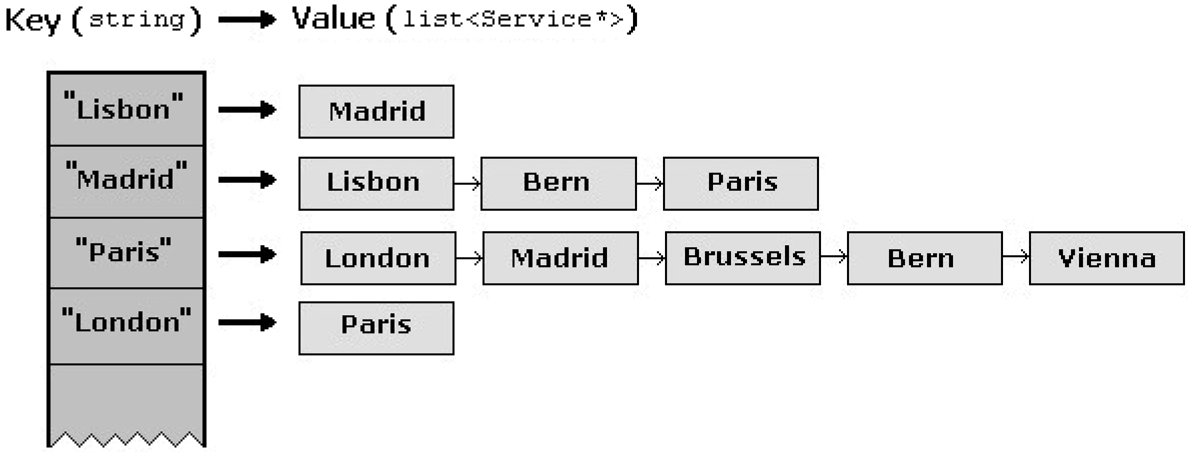

Figure 2 Adjacency list representation of the rail system

Essentially, in the above representation, a linked list of outgoing services can be indexed using a string containing the source city name. For example, notice in the above picture, Madrid has three outgoing services: one to Lisbon, one to Bern, and one to Paris. This matches the picture of the rail system at the top of this handout. In addition to this type of map object, another type of map is used to map city names to pointers to their respective `City` objects. This map object is used in the search algorithm.

Sample output from this program is shown in the figure below.

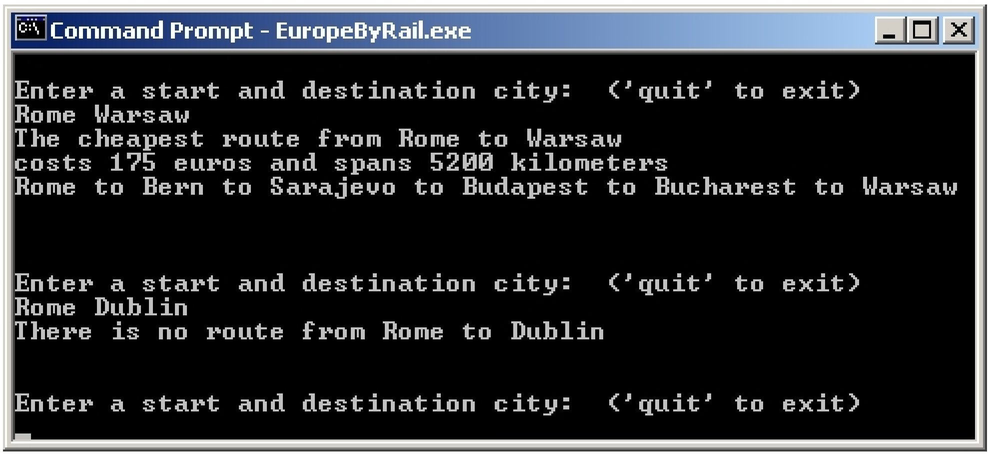

Figure 3 Output from a sample solution

The algorithm to be used is a positive-weighted, shortest path algorithm. Each `City` is a node in a graph, and each `Service` is an edge. Edges have an associated cost: the fee for the ticket. The goal is to find the cheapest route from the source city to the destination city.

To perform a new search, a pointer to a `City` object representing the start city should be added to an initially empty candidates queue. The algorithm continues by exploring the services of each candidate city, possibly adding new candidates to the list. The algorithm labels each `City` object with a number representing the cost of the cheapest route to it from the origin found so far. This is stored in member `total_fee` of class `City`. This value may decrease as new routes that are cheaper are found, but the value never increases.

To recover the actual path found by the search, every `City` object contains a string `from_city`. This string will be updated by the search algorithm to contain the name of the city by which it was reached through the cheapest path. In other words, if examining *Brussels* reveals a cheaper path to *Paris*, update variable *total_fee* of the `City` object that corresponds to *Paris*, and set its `from_city` equal to "Brussels". Once the search is complete, the cheapest route to the destination has been found, and the `from_city` strings can be traversed from the destination to the origin to reconstruct the path.

Files

Following is a list of files needed to complete this assessment.

- *`me7.arj`* contains all of the following necessary files:
  - *`main.cpp`* - This file contains the main routine.
  - *`City.h`* - This defines class `City`.
  - *`Service.h`* - This defines class `Service`.
  - *`RailSystem.h`* - This declares a class to represent the rail system.
  - *`RailSystem.cpp`* - This is a partial implementation of class `RailSystem`.
  - *`services.txt`* - This file contains data that defines the rail system services.

Tasks

To complete this assessment, you need to complete the implementation of class `RailSystem`.

To begin, verify the files needed for this assessment.

1. **Extract** the archive to retrieve the files needed to complete this assessment.

Following is an ordered list of steps that serves as a guide to completing this assessment. Work and test incrementally. Save often.

1. **Begin** by examining the implementation of class `City` and `Service`.
2. **Next**, complete function `load_services` of class `RailSystem`. The data file of services is *`services.txt`*. Function `load_services()` must correctly build populate the `cities` and `outgoing_services` maps.
3. **Then**, finish the implementation of function `RailSystem::reset`. This function should iterate through the `cities` map and reset the data members of the `City` objects.
4. **Next**, complete the `RailSystem` destructor implementation. The destructor should ensure that all `City` and `Service` objects allocated using `new` are deallocated using `delete`.
5. **Then**, complete the definition of function `calc_route()`. This function accepts the names of start and destination cities, uses a positive-weighted shortest path algorithm, and returns a `pair<int, int>` object with the total fee and total distance from the start city to the destination city. If there is not a path from the start city to the destination city, the function returns the pair`(INT_MAX, INT_MAX)`.
6. **Finally**, implement function `recover_route()`. Function `recover_route()` accepts the name of a destination city as an argument, and returns a string which contains, in order, the city names from the source city from which the destination city was reached through the destination city. The path can be recovered through the `from_city` strings, which should have been set by function `calc_route()`. The city names must be separated by the four-character string `" to "`. See the screen shot of sample output above for samples of strings created by function `recover_route()`.

Submission

Submit **only** the following.

1. *`RailSystem.cpp`* - finished implementation of class `RailSystem`