CISC220 Project Spring 2013

Project Checkpoint 1 due April 18.
Project Checkpoint 2 due May 2.
Final Project due May 12.

You may not help other people debug their programs, except in the very limited way described in the syllabus. You may not use any code written by others, whether they are students or not. The TA and the instructor are available for office hours, and by appointment if you can't make hours because of a class conflict.

Groups

You may work with any person from any lab section in groups of 2 for this project. The professor reserves the right to change the group roster at any time. Permission is required from the professor to work in a group of 3 (this permission will not be granted except in extreme circumstances). You are free to work on your own if you prefer.

Overview

The goal of this project is to help with railway planning for a next generation high speed railway in the US. Several assumptions will be made to simplify the domain so that we can directly apply techniques learned in class:

- All rail links in the network are assumed to be straight lines between two endpoints on a cartesian graph, and capacity is not a concern (it is assumed that adding capacity is a much easier task than laying new track and can be done by simply adding railcars or additional trains to an existing route).
- All links are bi-directional (so the resulting graph is undirected).
- All links will originate and terminate at railway stations.
- Each railway station in the rail network provides an "on-ramp" and "off-ramp" for travelers. In this way we do not care about the interface or implementation of regional train/tram networks. Indeed, a traveler may arrive at the railway station by taxi, ride the train to an intermediate destination, and travel to their final destination by car. We are only concerned with the traveler from the point they enter the rail network to the point they depart the rail network.

Using these simplifications, we create a physical representation of the existing network as such:

- S = set of all rail stations
- L = set of all existing links, where each link has endpoints in S

Part 1 - Implementing a Heap Structure

Consider the min ordered BinaryHeap implementation available on Sakai. This implementation uses a C++ STL Vector (http://www.cplusplus.com/reference/stl/vector/) to store BinaryHeapNode instances. We will need a completed implementation of a min ordered BinaryHeap for our algorithms in Part 2 of the project.

Each BinaryHeapNode has an index (its location in the heap) and a data value, which is a pointer to a Data instance.

Your task is to implement the following functions on a MinBinaryHeap. The headers for the functions are listed in the BinaryHeap.h and BinaryHeap.cpp files. There are various TODO flags placed in the BinaryHeap.cpp where you need to provide implementation:

```
/**
 * Returns the BinaryHeapNode that is the left child of the given node.
 */
BinaryHeapNode * MinBinaryHeap::getLeftChild(BinaryHeapNode * node) const

/**
 * Returns the BinaryHeapNode that is the right child of the given node.
 */
BinaryHeapNode * MinBinaryHeap::getRightChild(BinaryHeapNode * node) const

/**
 * Returns the BinaryHeapNode that is the parent of the given node.
 */
BinaryHeapNode * MinBinaryHeap::getParent(BinaryHeapNode * node) const

/**
 * Updates the position of the given node in the heap.  It is
 * assumed that the position is the same or has increased in heap order.
 *
 * (this is a min heap, so it is a percolate/bubble down operation)
 */
void MinBinaryHeap::increaseKey(Container * container)

/**
 * Updates the position of the given node in the heap.  It is
 * assumed that the position is the same or has decreased in heap order.
 *
 * (this is a min heap, so it is a percolate/bubble up operation)
 */
void MinBinaryHeap::decreaseKey(Container * container)
```

All of the functions listed follow the basic min ordered binary heap rules discussed in class. Keep in mind that you need to maintain correct indices in the underlying vector data structure in addition to the index property of a BinaryHeapNode. Also note that the insert and deleteMin operations of the BinaryHeap are already implemented but depend heavily on your correct completion of the above 5 functions.

For examples please consult the included BinaryHeapTest.cpp. The basic test uses the following min ordered heap:

```
   0
  / \
 1   2
```

The values 0, 1, and 2 are placed at indexes 0, 1, and 2 in the Vector. Value 1 is the left child of value 0 because it is at $index * 2 + 1$ where index is 0 for value 0.

Please make sure to consult the BinaryHeapTest.cpp tests if you need clarification for the expected result of each function.

- Download the code from Sakai. Compile/run the BinaryHeapTest.exe to see that the existing implementation does not properly update the nodes to maintain heap ordering.
- Edit BinaryHeap.cpp to implement the above MinBinaryHeap functions. All code changes should be at locations marked with the TODO flags.
- Re-make the test and run it to ensure it is successful.

Part 2 - Implementing Graph Algorithms

The physical representation for the rail network will be read from a file format that looks like this:

```
1 0.0 5.0 First
2 1.0 2.0 Second
3 3.0 4.0 Third
#
1 2 Edge1
2 3 Edge2
```

Each station has a line before the #. The station contains 4 values: numerical identifier for the station, x position, y position, and label for the station. Each link has a line after the #. The link contains 3 values: numerical identifiers for both stations and a label for the link.

Provided Classes

- Network.h + Network.cpp = Provides data structures that represent the following:
  - Network = The graph structure of the rail network. Includes a function that can fully read and write a rail network from/to a file. Uses a vector to store pointers to all vertices in the graph.
  - Station = The vertex structure. Represents different stations in the rail network. Contains a vector of link pointers.
  - Link = The edge structure. Contains a weight property which should be used in Dijkstra's algorithm.
- Dijkstra.h + Dijkstra.cpp = Provides a function for running Dijkstra's algorithm and a class to store results from the algorithm (StationRecord)

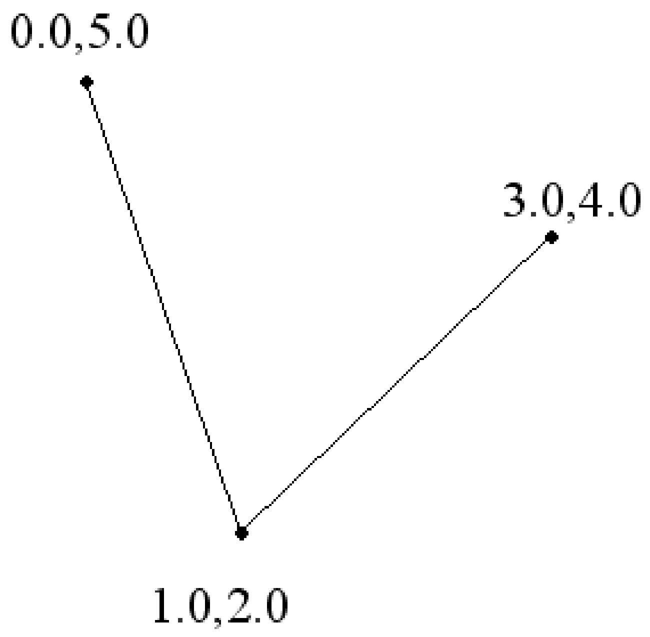

Figure 1: Example of a simple rail Network with 3 Stations and 2 Links

The provided code for the Network class is able to read and write the file format described earlier.

Now that we have the basic BinaryHeap structure you built in Part 1 and the basic graph structure created with the Network/Station/Link classes, we can write some algorithms to process our graph.

(2.A) Shortest Path Calculations

Your task is to implement Dijkstra's algorithm in the dijkstra_getShortestPaths function. Make sure to use the weight property of the Link. Also make sure to use the dist/pred/container properties of the StationRecord (which can be gotten from the Map). The header for this function is in Dijkstra.h and a TODO flag has been placed in the Dijkstra.cpp where you need to provide the implementation.

- Download the code from Sakai and copy over your correct BinaryHeap.cpp implementation.
- Edit Dijkstra.cpp to implement Dijkstra's algorithm for shortest path computation.
- Make the DijkstraTest.exe and run it to ensure it is successful. It will run by default on cities.txt, but you can specify a different source file by giving that file as the first argument:

```
./DijkstraTest.exe cities.txt
```

Your final result should match the output in DijkstraTest_output.txt.

(2.B) Computing Efficiency of a Rail Network

We will now use an additional file containing travel statistics as input for our Scenario. The provided RailSchedule class will read this file into a weighted edge list of RouteSchedules showing the travel patterns of passengers. The physical representation for the travel statistics will be read from a file format that looks like this:

```
1 2 400
2 1 300
2 3 100
```

This example shows that on average, 400 people travel from Station 1 to Station 2 each day, 300 people from Station 2 to Station 1 each day, and 100 people from Station 2 to Station 3 each day.

- Edit Project.cpp to implement the **totalDistanceTraveled** function that calculates the total distance traveled by passengers in the Network. For each RouteSchedule you must compute the shortest path distance from the source to target Station, multiply this distance by the number of passengers, and sum these distances. Make sure to not run Dijkstra's algorithm except when necessary – the RouteSchedules appear in order in the file by source Station (so you only need to run Dijkstra's algorithm once per source Station).
- Make the ProjectTest.cpp and run it to ensure it is successful. It will run by default on cities.txt and routes.txt, but you can specify different source files by giving those files as the first two arguments:

  ```
  ./ProjectTest.exe cities.txt routes.txt
  ```

  Your output should be: `Total distance is 958192`

Part 3 - Optimizing a solution to a rail Network

Devise a cost-effective strategy for adding new rail links to an existing rail Network. You will now run a Scenario that includes another file representing the yearly government budget for growth of the national rail Network. The funding file will look like:

```
500
200
100
```

This file indicates that during the first year you may purchase and build a total of 500 distance units of new Links between Stations. The following year you may purchase and build 200 more distance units, and finally 100 distance units of Links.

Each event provides for X distance worth of new rail Links which can be split however you deem appropriate. Unused funding units will be carried over to future years, but your end result will include total distance traveled by passengers for each year in the simulation.

Your goal is to decrease the total distance traveled by passengers in the Network. For each funding year, the Scenario will call your **addLinks** function, output the Network to a file, and display the **totalDistanceTraveled**.

- Download the final project code from Sakai. Copy your BinaryHeap.cpp, Dijkstra.cpp, and Project.cpp files from the second part into your final project folder.
- Edit Project.cpp to add the **addLinks** function. If you leave the body of the function empty, you should now be able to compile and run the basic Scenario:

  ```
    ./Scenario.exe cities.txt routes.txt shortterm_funding.txt
  ```

  Your output should be:

  ```
  Year 0 Total Distance Traveled=958192 (958192)
  Year 1 Total Distance Traveled=958192 (1.91638e+06)
  Year 2 Total Distance Traveled=958192 (2.87458e+06)
  Funding left over=850
  Sum of Total Distance Traveled=2.87458e+06
  ```

  This shows that you have not used any of the funding to improve the rail network.
- Optimize your solution by coming up with a strategy for your implementation of **addLinks**. You are able to do anything you want that would help compute which links to add, but you cannot remove existing links and you cannot add more links than your budget allows (the Scenario will exit with an error message if you try). **When you know you want to add a link, call the addLink method that is in the Project (see Project.h).**
- You do not have to use all of the funding every year, the remaining funding units will carry over to the next year of the simulation.
- Upload your code to Sakai! I will setup a periodic run of the whole class's solutions to see how well your solution performs against the class. Re-submit a new version to Sakai and it will be included in the next run.
- A Scenario can be run with different cities, routes, and funding. You will be graded on how your solution performs (Sum of Total Distance Traveled) for various scenarios. You should test to make sure your code works for:

  ```
    ./Scenario.exe cities.txt routes.txt shortterm_funding.txt
    ./Scenario.exe cities.txt routes.txt longterm_funding.txt
    ./Scenario.exe cities.txt full_routes.txt shortterm_funding.txt
    ./Scenario.exe cities.txt full_routes.txt longterm_funding.txt
  ```

  In addition, there are a few that I will run that you do not have, so your code should not be hard-coded for a specific scenario.

Submission of the final Project is due May 12th at midnight.