CS202 – Data Structures

Dr. Ihsan Ayyub Qazi

Assignment 5

Graphs

(Due: 11pm, Friday May 5th, 2017)

In this assignment, you will be implementing different graphs algorithms.

The course policy about plagiarism is as follows:

1. Students must not share actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students must indicate any assistance they received.
4. Students cannot copy code from the Internet
5. All submissions are subject to automated plagiarism detection.

Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee

Overview

From the user's point of view, the program loads a graph from the description in a text file and builds a graph data structure in memory. The program offers the user a menu of choices, as shown below:

(1) Choose a new graph data file
(2) Compute the minimum spanning tree using Kruskal's algorithm
(3) Compute the minimum spanning tree using Prim's algorithm (Bonus)
(4) Run Dijsktra's algorithm
(5) Quit
Enter choice:

The user chooses the graph data file for the program to read and display. When the user chooses the minimal spanning tree option, the program determines the set of links that connect all locations with the minimal overall cost and displays them; it also displays the sum of the costs of these links. The user can do additional searches and/or change data files. The program exits when the user chooses to quit.

The data structure being modeled is a graph. A graph consists of a collection of nodes, each of which can have any number of links (called arcs/edges) to other nodes. Unlike a rooted tree, a graph doesn't have a root node. In a graph, there can be more than one distinct path connecting two nodes and a graph may contain cycles.

Ideally the node and arc data structures allow easy and direct access to each other. For example, an arc could track two pointers to its start and end nodes, and node could maintain a collection of pointers to its outgoing arcs.

The graph is populated with information read from a graph data file. The line "NODES" indicates the beginning of the node entries. The nodes are listed one per line, each with a name and x-y coordinates. The line "ARCS" indicates the end of the node entries and beginning of the arc entries. Each arc identifies the two endpoints by name and gives the distance between the two. The arc is a bi-directional connection between the two endpoints.

```
NODES  // NODES marks beginning of list of nodes
Denver 2.54 3.25 // one-word name (no spaces) and x-y coordinates
SanJose 1.05 2.66
...
ARCS  // ARCS marks beginning of list of arcs
Denver SanJose 1780 // two nodes connected by this arc and distance between
LA Denver 1890 // note each arc is a bi-directional connection
```

The format is designed to be easily read using the stream extraction >> operation. As a reminder, extraction skips over whitespace by default (exactly as desired in this case). You can assume all input files are properly formatted.

Tasks

- Task 1: Design your data structures
- Task 2: Read graph from file and store in your data structure
- Task 3: Implement Kruskal's algorithm to compute the minimal spanning tree
- Task 4: Implement Dijsktra's algorithm to find the optimal path
- Task 5: Implement Prim's algorithm to compute the minimal spanning tree (*Bonus*)

In the start, you are **required** to make your heap class generic so that it can handle any data type. This will make it easier to use later in your other tasks. Use the file '**pqueue.h**'.

Task 1: Design your data structures

The next thing to do is to determine which data structure you will be using. The graph data structure you design will need to store raw connectivity plus associated information such as the distance for an arc or the name of a node. A node will store its outgoing arcs. The arcs could be stored in a raw array or a linked list, but the convenience and safety of a Vector or Set will make your job easier. Similarly figure out what information an arc will store. Learning how to leverage and reuse components is an essential goal of the assignment; don't feel you need to reinvent the wheel!

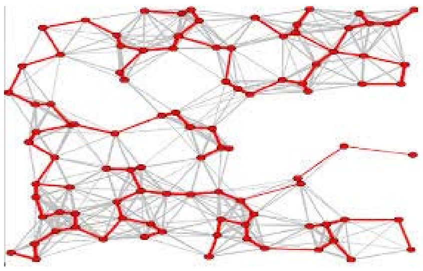
Task 2: Read the data file, store into your structure

Loading the information into the graph is your next job. In the data file, each arc identifies its start and end nodes by name. Use a Map to store nodes keyed by name to make it simple to find the node for a name when reading the data file. Once you have done that lookup, each arc can permanently store pointers to the start and end nodes for direct access later.

Before moving on, confirm you have correctly built the graph. There's no point in writing more code if you aren't sure you are operating on the correct graph to begin with. Add debugging code to print the nodes and connections and verify all is as expected.

Task 3: Implement Kruskal's algorithm

Once the graph is built and displayed, you would require Kruskal's algorithm for constructing a minimal spanning tree. In many situations, a minimum-cost path between two specific nodes is not as important as minimizing the overall cost of a network. As an example, suppose your company is building a new cable system that connects 10 large cities in Philadelphia. Your preliminary research has provided you with cost estimates for laying new cable lines along a variety of possible routes. Your job is to find the cheapest way to lay new cables so that all the cities are connected through some path.



To minimize the cost, one of the things you need to avoid is laying a cable that forms a cycle. Such a cable would be unnecessary, because some other path already links those cities, and thus you might as well leave such arcs out. The remaining graph, given that it has no cycles, forms a tree. A tree that links all the nodes of a graph is called a spanning tree. The spanning tree in which the total cost associated with the arcs is as small as possible is called a minimum spanning tree. The cable network problem is therefore equivalent to finding the minimum spanning tree of the graph.

There are many algorithms in the literature for finding a minimum spanning tree. Of these, Joseph Kruskal devised one of the simplest in 1956. In Kruskal's algorithm, you consider the arcs in the graph in order of increasing cost. If the nodes at the endpoints of the arc are unconnected, then you include this arc as part of the spanning tree. If, however, some path already connects the nodes, you can discard this arc.

Kruskal's is an example of a greedy algorithm. Since the goal is to minimize the overall total distance, it makes sense to consider shorter arcs before the longer ones. To process the arcs in order of increasing distance, the heap will come in handy again.

The tricky part of this algorithm is determining whether a given arc should be included. The strategy you will use is based on tracking connected sets.

- Initially, each node is in its own set all by itself. Which means there will be "n" sets comprising of individual nodes. Each set contains the nodes, which are connected by some edge.

- The algorithm then considers each edge, e, in turn, ordered by increasing weight. (Use getMin() of the priority queue for this)

- If an edge e connects two different sets, then e is added to the set of edges of the minimum spanning tree, and the two sets of nodes that are now connected by e are merged into a single set.

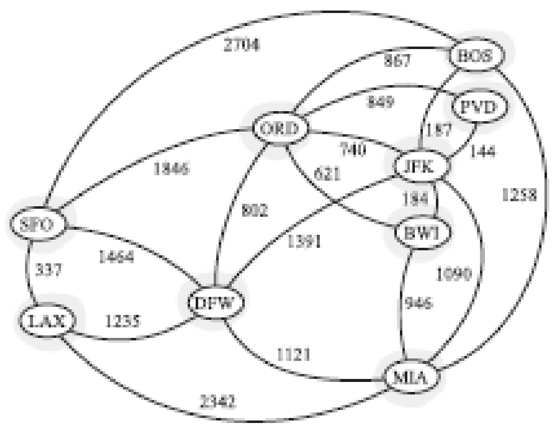
- If, on the other hand, e connects two vertices/nodes that are already a path between them using some earlier edge, then e is discarded. (Check to see if the two end points of the edge e belong in the same set)

- Once the algorithm has added enough edges to form a spanning tree (meaning that only one set of nodes remain and that set contains all the nodes), it terminates and outputs this tree as the minimum spanning tree.

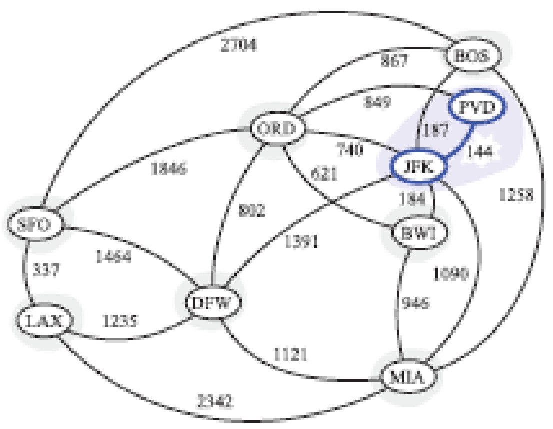
A good data structure for tracking the connected sets is the Set class. This algorithm requires a heap data structure that you have worked with previously to set the ‘priorities’ or ordering of elements.

As you add each new arc, print it to the console. When you have finished, the entire minimal spanning tree will be displayed. You also must display the total cost of all these links combined.

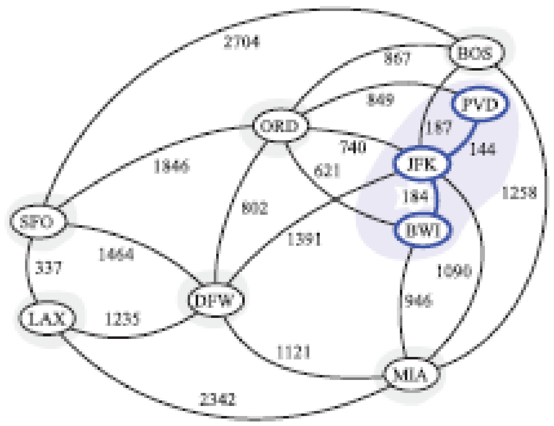
Here is a sample execution:



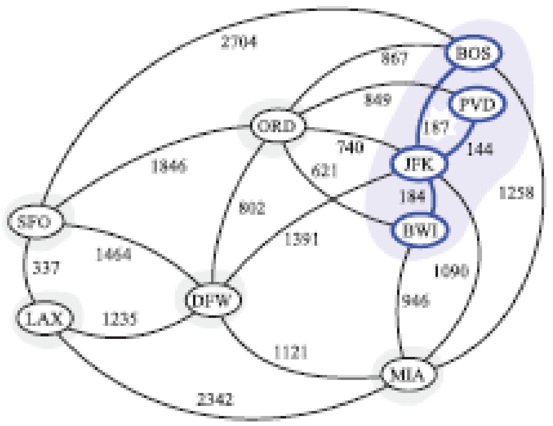
(a)



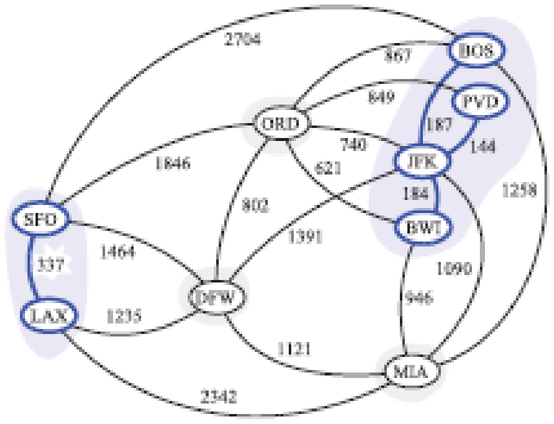
(b)



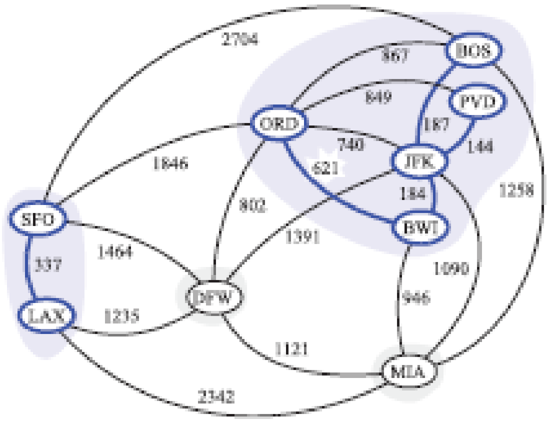
(c)



(d)



(e)



(f)

Now, take a great idea from Kruskal, mix in a template heap, and one minimal spanning tree is coming right up! The basic strategy is to store all arcs into a heap, and then pull each out in order and decide whether to include it as part of the minimal spanning tree, using the merge-tree strategy described earlier to determine whether an arc is redundant. A shorter arc is higher priority, so be sure that the callback function correctly reports the ordering.

Task 4: Implement Dijsktra's algorithm to find the optimal path

Once the graph is built and displayed, the Dijkstra option prompts the user to select two nodes and computes the shortest path between the two, displaying the optimal route. One strategy to find the best path would be to construct every possible path and compare them all to determine the shortest path. However, there are far too many paths for this to be efficient. Such an exhaustive search requires exponential time, which is intractable for a graph of a reasonable size. Instead, a better approach is to organize the search to pursue the promising short paths first and only explore the longer routes if nothing better is found earlier.

The idea is to keep a queue of all paths constructed so far, prioritized in terms of distance (A perfect use for the priority queue ☺). At each stage, you dequeue the shortest path; if it leads to the destination, your work is done. Otherwise, you use that path as the basis for constructing new extended paths that add an arc leading away from the node at the end of the path. You discard the extended path if you already know a better way to get there (i.e. you have already found a shorter path to that node), otherwise you enqueue the extended path to be considered with the others. After updating the queue, you dequeue the next shortest path and explore from there. At each step, the search expands outward until you find a path that ends at the destination. The graphs in the data files are fully connected, meaning, every pair of nodes is connected by some path, and thus you will eventually find a path. The order of the paths considered by the algorithm guarantees that you will find the shortest such path first.

Remember that the shorter of two paths is considered higher priority (i.e. the shortest path is the first one dequeued), so be sure the return value for your comparator appropriately reflects that.

This is a greedy strategy since it chooses the shortest known path, the locally best option, in hopes of it leading to the shortest overall path, the globally best option. This priority-driven approach will efficiently find the shortest path without an exhaustive search. This

algorithm is called Dijkstra's algorithm, after its inventor, Edsgar Dijkstra. Much of the structure is the same — a Stack can be used to represent a path, the queue-driven search stops at the first path that reaches destination, and so on.

There are a few differences to note. Instead of an ordinary queue, you can use a heap as you used earlier so that paths are prioritized in terms of total distance, not just number of hops, as in maze. Instead of using the four neighbors to extend the current path, you use the graph connectivity. Whereas a perfect maze has only a single path between two nodes and never has a loop, a graph can have multiple paths and cycles, so you will need to take care to avoid wastefully re-exploring nodes that have already been visited earlier in the search. The ever-handy Set might be useful for that purpose.

Task 5 (BONUS): Implement Prim's algorithm

Implement Prim's algorithm for finding the Minimum Spanning Tree of a given graph. Prim's is a greedy algorithm which works by selecting the cheapest adjacent vertex while looking at a single vertex. You are encouraged to use binary heap for selecting the cheapest adjacent vertex. Here, the word "cheapest" means the one with the smallest edge cost. Pick out a starting point, and keep on adding vertices until the heap is empty. You will need to initialize keys of all vertices except the starting point to -INF, and update the key of a given vertex when it is discovered by its neighboring vertex. Take a look at https://en.wikipedia.org/wiki/Prim%27s_algorithm#Description for more examples and description of the algorithm.

General hints and suggestions

- The general expectation is that you will provide a main menu to offer the user the choice between algorithms, gracefully handle invalid user input, and allow the user to switch data files, and so on.
- Take care with your data structure. Plan what data you need, where to store it, and how to access it. Think through the work to come and make sure your data structure adequately supports all your needs. Be sure you thoroughly understand the classes that you are using. Ask questions if anything is unclear.
- Forward references handle circularities. It is likely that a node will store pointers to arcs and an arc will store pointers to its start and end nodes, creating a mutually recursive set of structures. This situation results in a circular reference. One has to be defined first, but how you can you define a type that depends on something that hasn't yet been seen? In C++, the mechanism for dealing with this is the forward reference. When declaring the ***nodeT struct,*** you can precede it with a forward reference to arcT using this bit of syntax:
- ***struct arcT***; This forward reference informs the compiler that there will be a struct named arcT. This allows the nodeT to declare a field of type arcT* since the compiler has been assured such a struct will exist and will be seen later.
- Equality comparisons. Make sure you override the == operator in your classes so that they compare by value. This will make a difference because the Set and Heap classes use the == method to compare for equality, which, if you're using pointers, would compare memory addresses by default and not values.
- Careful planning aids re-use. This program has a lot of opportunity for unification and code reuse, but it requires some careful up-front planning. You'll find it much easier to do it right the first time than to go back and try to unify it after the fact. Sketch out your basic attack before writing any code and look for potential code-reuse opportunities in advance so you can design your functions to be all-purpose from the beginning.
- Bring your best pointer game. You are likely to have quite a few pointers within your graph data structures, which brings opportunity for errors (forgetting to allocate/deallocate are probably the most common). Proceed with caution and develop incrementally so you can ferret out difficult bugs earlier rather than later. Your program is expected to deallocate heap-allocated memory. We recommend you do this last. Only when you have your program running properly should you go back and add delete statements one at a time, testing your program at each step.
- Templates:
    - Don't add .cpp template files to be compiled into the project.
- Test on smaller data first. There is a "Small" data file with just four nodes that is helpful for early testing. The larger USA.txt and Stanford.txt data files are good for stress-testing once you have the basics in place.

Deliverables:

The submission should include all your code files.

GOOD LUCK!