CS106B
J Zelenski

Handout #34
Mar 5, 2007

Assignment #7: Pathfinder

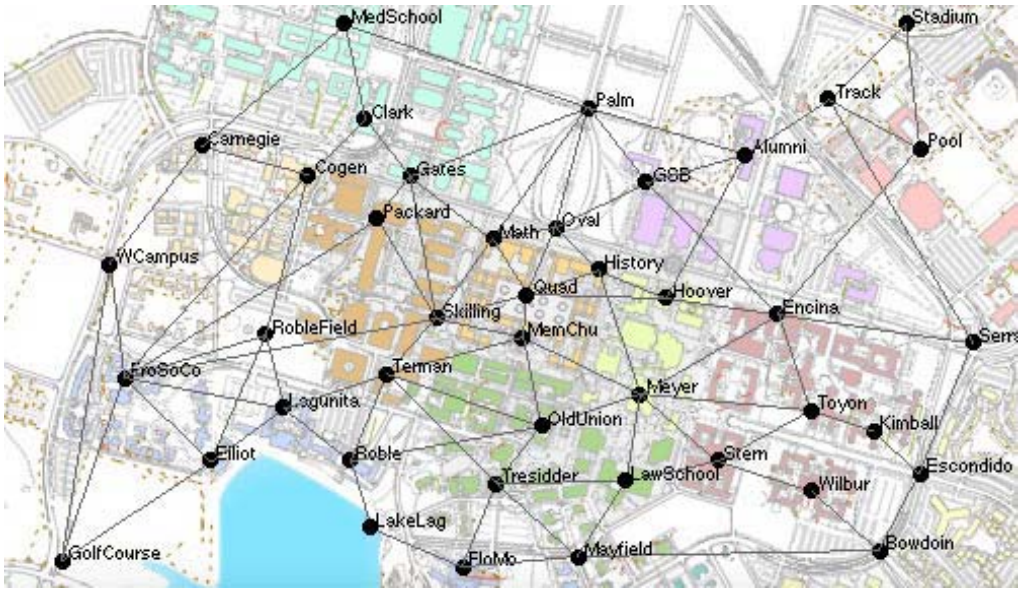
Due: Fri Mar 14 2:15pm

Absolutely **no** late assignments accepted after Mon Mar 17

Have you ever wondered how MapQuest and Google Maps or those tiny little GPS units work their magic? Get ready to find out! In this final CS106B challenge, you will take your now-superior C++ skills, employ a variety of classes (including writing a class template of your own) and implement two classic and elegant graph algorithms. I can't think of a better way to cap off our intense journey. When you look back at where you were at the end of CS106A, did you imagine you'd be ready for something this fancy just 10 weeks later? Wow!

The program from a user's perspective

From the user's point of view, the program draws a graph and provides two operations: finding the shortest path between two locations and constructing the minimal cost network. The user chooses the graph data file for the program to read and display. Below, the **`Stanford`** data file is displayed:



The program offers the user a menu of choices, as shown below:

```
Your options are:
    (1) Choose a new graph data file
    (2) Find shortest path using Dijkstra's algorithm
    (3) Compute the minimal spanning tree using Kruskal's algorithm
    (4) Quit
Enter choice:
```

If the user chooses the shortest path search, the program prompts the user to click on two locations. The program finds the shortest path connected the two and displays it. If 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. The user can do additional searches and/or change data files. The program exits when the user chooses to quit.

The program from an implementer's perspective

Behind the scenes, this assignment is designed to accomplish the following objectives:

- To give you practice working with graphs and graph algorithms.
- To give you more mileage with C++ pointers and dynamic memory. Many of you commented on the mid-quarter evals that you felt a bit of mystery still surrounds these topics. I'm hoping that conquering the chunklist improved your confidence and this assignment should further solidify your understanding.
- To learn how to adapt existing code (in this case, the PQueue from the previous assignment) for use in a new context. The majority of programming that people do in the industry consists of modifying existing systems rather than creating them from scratch.
- To continue making good use of our handy class library. Several of the classes have a role to play and your spiffy PQueue is critical for the two priority-driven algorithms.

This assignment covers material from the entire quarter and is an opportunity to show your mastery over the complete CS106B repertoire. Set your personal goal to make your final project truly shine!

The graph data structure

The data structure being modeled is a *graph*. A graph is a recursive data structure consisting of a collection of nodes, each of which can have any number of links (called *arcs*) to other nodes. Unlike a tree, a graph doesn't have a root node— all nodes are peers. Within a graph, there may be more than one distinct path connecting two nodes and a graph may contain cycles—that is, tracing a path away from a node may at some point return back to that same node.

At its most basic, a graph is simply a collection of nodes and arcs. 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.

Graph data file format

The graph is populated with information read from a graph data file. The first line is the filename of the background image. The next 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.

```
USA.bmp                     name of image file to display background picture
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.

Drawing the graph

The simple drawing and event-handling needed for this program is supported by the CS106 graphics library. The function **`DrawNamedPicture`** is used to display the background image onto which you will draw circles and lines for the nodes and arcs. The starter code includes some utility functions to get you started on theses tasks.

Optimal path search

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, visually highlighting 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. 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 any 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.

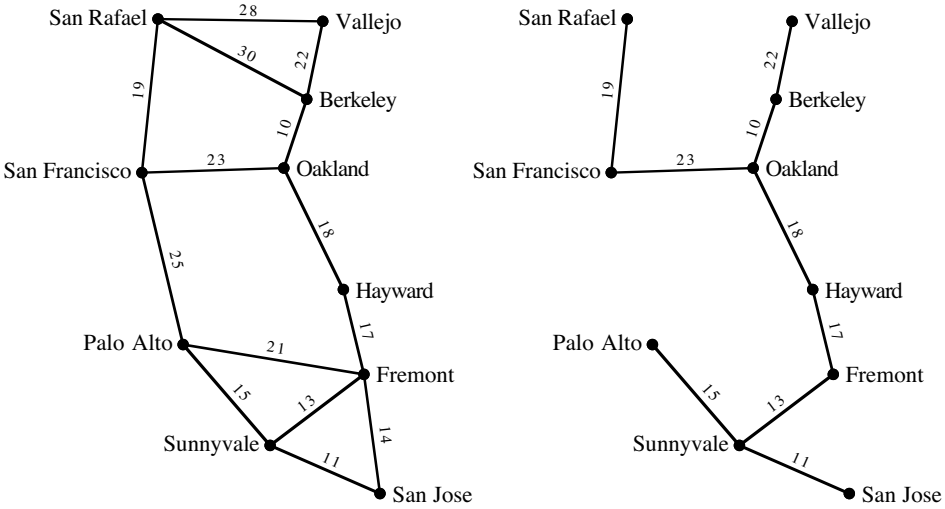
The idea is to keep a queue of all paths constructed so far, prioritized in terms of distance. (A perfect use for the priority queue!) At each stage, you dequeue the shortest path; if it leads to the destination, your work is done. Otherwise, you use that path as the basis for constructing new extended 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 our 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 that the algorithm considers paths guarantees you will find the shortest such path first.

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.

Dijkstra's algorithm should feel familiar, because it is a minor variation on the breadth-first algorithm you wrote to solve a maze in Assignment 2. 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 substitute a priority queue 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.

Minimal spanning tree

Your second algorithm is another graph classic: 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 cost of a network as a whole. As an example, suppose your company is building a new cable system that connects 10 large cities in the San Francisco Bay Area. Your preliminary research has provided you with cost estimates for laying new cable lines along a variety of possible routes. Those routes and their associated costs are shown in the graph on the left below. 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, which is shown in the right side of the figure above.

There are many algorithms in the literature for finding a minimum spanning tree. Of these, one of the simplest was devised by Joseph Kruskal 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, the nodes are already connected by some path, you can discard this arc. The steps in the construction of the minimum spanning tree for the graph above are shown below.

```
Process edges in order of cost:
10: Berkeley -> Oakland
11: San Jose -> Sunnyvale
13: Fremont -> Sunnyvale
14: Fremont -> San Jose (not needed)
15: Palo Alto -> Sunnyvale
17: Fremont -> Hayward
18: Hayward -> Oakland
19: San Francisco -> San Rafael
21: Fremont -> Palo Alto (not needed)
22: Berkeley -> Vallejo
23: Oakland -> San Francisco
25: Palo Alto -> San Francisco (not needed)
28: San Rafael -> Vallejo (not needed)
30: Berkeley -> San Rafael (not needed)
```

Kruskal's is another 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 priority queue 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. For each node, maintain the set of the nodes that

are connected to it. At the start, each node is connected only to itself. When a new arc is added, you merge the sets of the two endpoints into one larger combined set that both nodes are now connected to. When considering an arc, if its two endpoints already belong to the same connected set, there is no point in adding that arc and thus you skip it. You continue considering arcs and merging connected sets until all nodes are joined into one set. The perfect data structure for tracking the connected sets is the Set class, since it has the handy high-level operations (such as **`unionWith`**) that are exactly what you need here.

As you add each new arc, draw it on the graphical display. When you have finished, the entire minimal spanning tree will be highlighted.

A polymorphic priority queue

Both algorithms need a priority queue and the priority queue you worked on for the last assignment is almost the perfect tool, save the limitation that it was written to store only integer elements. You must first upgrade the priority queue to be polymorphic (i.e. a class template) and capable of storing any type of element. You've been the client of class templates all quarter, and how here is your chance to implement your own template.

Note that since the priority queue needs to know the relative ordering elements, the client will have to supply a compare-by-priority comparison function that can be applied to any two elements to determine their ordering. Since that function is chosen once and must remain consistent throughout the lifetime of the object, it is appropriate that it be passed to the constructor when a new priority queue is created and stored internally to the implementation, just as the **`Set`** class template does. The default argument for the comparison function should be the **`OperatorCmp`** comparison function from **`cmpfn.h`** that applies the standard relational operators. The interface and behavior of the PQueue should not change when converting to template form. It still enqueues and dequeues in order of *maximum* priority. If a client desires to dequeue in order of minimum priority, they can easily affect this change by providing the desired ordering in their comparison callback function.

Remember that class defined as a template follows different compilation rules: the template **`pqueue.h`** will #include the template **`pqueue.cpp`** file and the **`pqueue.cpp`** file is no longer added to the list of files compiled for the project.

Task breakdown

Not sure where to begin? Take a look at our suggested order of attack:

Task 1	Templatize your pqueue class
Task 2	Design your data structures
Task 3	Read graph from file and store in your data structure
Task 4	Display the graph, allow user to choose nodes
Task 5	Implement Dijkstra's algorithm to find the optimal path
Task 6	Implement Kruskal's algorithm to compute a minimal spanning tree

Task 1—Make PQueue generic

This task is an extension of your last assignment: take your PQueue class and convert it into a generic form that can store elements of any type. When finished, it should be a class template that uses a comparison callback to compare elements. You can adapt whichever of the four implementations you'd like (one of our original ones or either of the two you wrote). Mostly this task just involves minor syntactic changes, but nothing is simple when C++ templates are involved, so do take care here. It makes good sense to exercise the priority queue in isolation before you integrate it into the larger program. This will allow you to find and fix its bugs without having to wade through the camouflage of the rest of the code. The pqueue client test code we gave you with the previous assignment could be adapted and used for this purpose.

Task 2—Design data structures
The next thing to do is determine what data structures 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. Be sure to take advantage of all the CS106 classes that can make your job easier. For example, 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. Learning how to leverage and reuse components is an essential goal of the assignment; don't feel you need to reinvent the wheel!

Task 3—Reading 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 4—Draw the graph, allow user to click nodes
The background picture is drawn first and the nodes and arcs are overlaid on top. The user will choose locations by clicking. To determine which node was clicked, examine each graph node and test if the click location was close enough to the node's coordinate. There are a few helper function pre-written in the starter code that help with the mundane drawing and event-handling tasks.

Task 5—Finding the optimal path
With your infrastructure in place, you're ready for Dijkstra's nifty algorithm. The user will choose start and end nodes by clicking, and you will find the shortest path between the two and highlight it. This will be your chance to leverage your newly templatized PQueue. 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.

Task 6—Kruskal's algorithm
And lastly, take a great idea from Kruskal, mix in a template priority queue, and one minimal spanning tree is coming right up! The basic strategy is to enqueue all arcs into a priority queue, 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 N+1—Pat yourself on the back!
Voila! Think back to where you were at the beginning of CS106B, when pointers were mysterious and you knew little of recursion or the power and beauty of ADTs. Today you completed a program that shows your mettle with over a half-dozen classes, spicy pointers, recursive data structures, elegant algorithms, and impressive skills for designing, implementing, and debugging a sophisticated project. Congratulations! I hope you are as proud as I am of what you have accomplished!

General hints and suggestions

- *Check out the demo*. Run our provided demo to learn how the program should operate. The general expectation is that you will provide a main menu to offer the user the choice between algorithms, gracefully handle invalid user input, allow the user to switch data files, and so on, just as the demo does.
- *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.

- *Client callbacks*. Both the template PQueue and Set expect a client comparator function pointer when constructing the object. If you omit this argument, the default **`OperatorCmp`** is used, which attempts to compare the two items using the built-in relational operators. For storing certain item types (int, string), this comparison may be appropriate. For other types (such as structs), the default is inappropriate and will fail to compile. If storing pointers, use of the default comparator will compile and means the memory address will be compared for equality/ordering. In some situations, this is okay, but if this doesn't fit your needs, remember you can supply a callback function to compare the pointers according to your desired ordering.

- *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 everywhere!* This is your first opportunity to write a template class and you will be a client of many templates. Don't let the syntax bog you down! A few tips:
  - — **Don't add .cpp template files to be compiled into the project**. It is likely that the only source file compiled in your project will be the pathfinder.cpp file.
  - — The reader shows the implementation of several class templates if you need a reference of the correct syntax (Stack and Queue in Ch.10, Map in 11, BST in 13, Set in 15).
  - — Our friendly LaIR staff are great help on template compile errors, so come on by!

- *Test on smaller data first*. There is a "Small" data file with just four nodes that is helpful for early testing. The larger USA and Stanford data files are good for stress-testing once you have the basics in place.

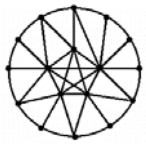
A little extra challenge: extension ideas

If you've completed a beautiful, elegant solution with time and energy to spare, there are many other graph explorations you could dive into, such as:

- *Employ A**. Dijkstra's algorithm is guaranteed to find the shortest path, but it can do unnecessary searching in what will prove to be the wrong direction if there are many short paths that lead away from the goal. One way to avoid this is to guide the search in the right direction by adding in a heuristic. The algorithmic fix is amazingly simple—just change how priority is calculated. Instead of considering only the length of the path, add in an estimate of how close that path gets

you to the goal. One good estimate is the straight-line distance from the end of the path to the goal. A heuristic is a "rule of thumb" that aids in the solution of a problem by employing domain-specific knowledge. In this case, our understanding of the physical world tells us that Cartesian distance between two points as a good estimate of proximity. The search algorithm using this heuristic is an example of a group of algorithms called *A** in the artificial intelligence world. One of the inventors of A* was Nils Nilsson, an Emeritus Professor of our very own Stanford CS department. Add in an option to employ this heuristic and report on the difference in the numbers of paths considered between this and standard Dijkstra's search.

- *Finding the graph center*. The city works department is thinking about adding a new fire station and need your help to find the appropriate central place. The goal is a position so that the shortest paths from the fire station to other locations are relatively small. The *eccentricity* of a node N is the maximum distance any other node is from N (i.e. consider all of the shortest paths between N and each of the other nodes, the longest of those is N's eccentricity). A node is at the *center* of the graph if its eccentricity is the smallest of all nodes (all nodes that tie for smallest are jointly considered the center). Add a feature to your program to find the graph center, which is the set of all nodes tied for the minimum eccentricity.
- *Max clique*. A graph *clique* is a subset of the graph nodes where every pair of nodes in the subset is directly connected. As in high school, it is a tight-knit cluster of friends that all know each other and ignore those outside their circle. Given the graph on the left below, the diagram on the right shows the maximal clique (the largest subset of the graph nodes that form a clique).




Identifying clusters of related objects often reduces to finding large cliques in graphs. For example, a program recently developed by the IRS to detect organized tax fraud looks for groups of phony tax returns submitted in the hopes of getting undeserved refunds. The IRS constructs a graph where each tax form is a node and connects arcs between any two forms that appear suspiciously similar. A large clique in this graph points to fraud. Finding the truly maximal clique is known to require an exhaustive and time-consuming search, although there are good approximation algorithms that can find large cliques fairly efficiently. Add a feature to your program to find the maximal (or large) clique within the graph.

Accessing Files http://see.stanford.edu/see/materials/icspacs106b/assignments.aspx

On the class web site, there are two folders of starter files: one for Mac and one for PC. Each folder contains these files:

pathfinder.cpp	Starter file for main module
Small.txt, USA.txt, Stanford.txt	Graph data files
Pictures	Folder containing background pictures
Pathfinder Demo	Compiled version of a working program

To get started, create your own starter project and copy the pqueue files from your previous assignment into your project folder.

Deliverables

For this final assignment, you need only submit electronically (no paper copies). The submission should include your **pathfinder.cpp file and template pqueue class files**. If using a late day, your program is due by 2:15 pm on Monday March 17th. Absolutely no assignments will be accepted later. Due to end-quarter time constraints, this last program will not be interactively graded.