CS106B Handout 30
Summer 2011 August 28th, 2010

Assignment 5: Pathfinder

This assignment was developed and written by Julie Zelenski and Eric Roberts.

Have you ever wondered Google Maps or those tiny little GPS units work their magic? Get ready to find out. In this final CS106B challenge, you will create the Pathfinder application, which (in addition to other useful functions) finds and displays the shortest path between two locations on a map.

The Pathfinder assignment is designed to accomplish the following objectives:

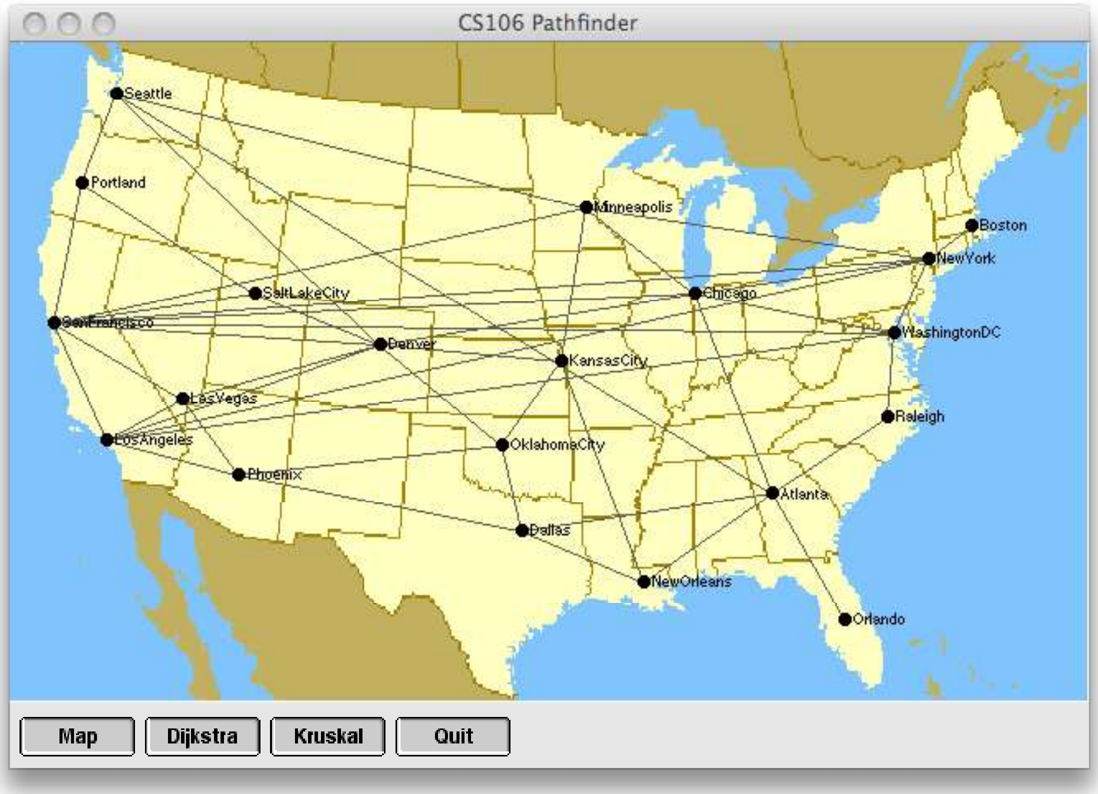
- To give you practice working with graphs and graph algorithms.
- To offer you the chance to design and write a class of your own.
- To continue making use of the handy CS106 class libraries.
- To let you practice using callback functions in an application.

Due: August 9th at 4:00 p.m.

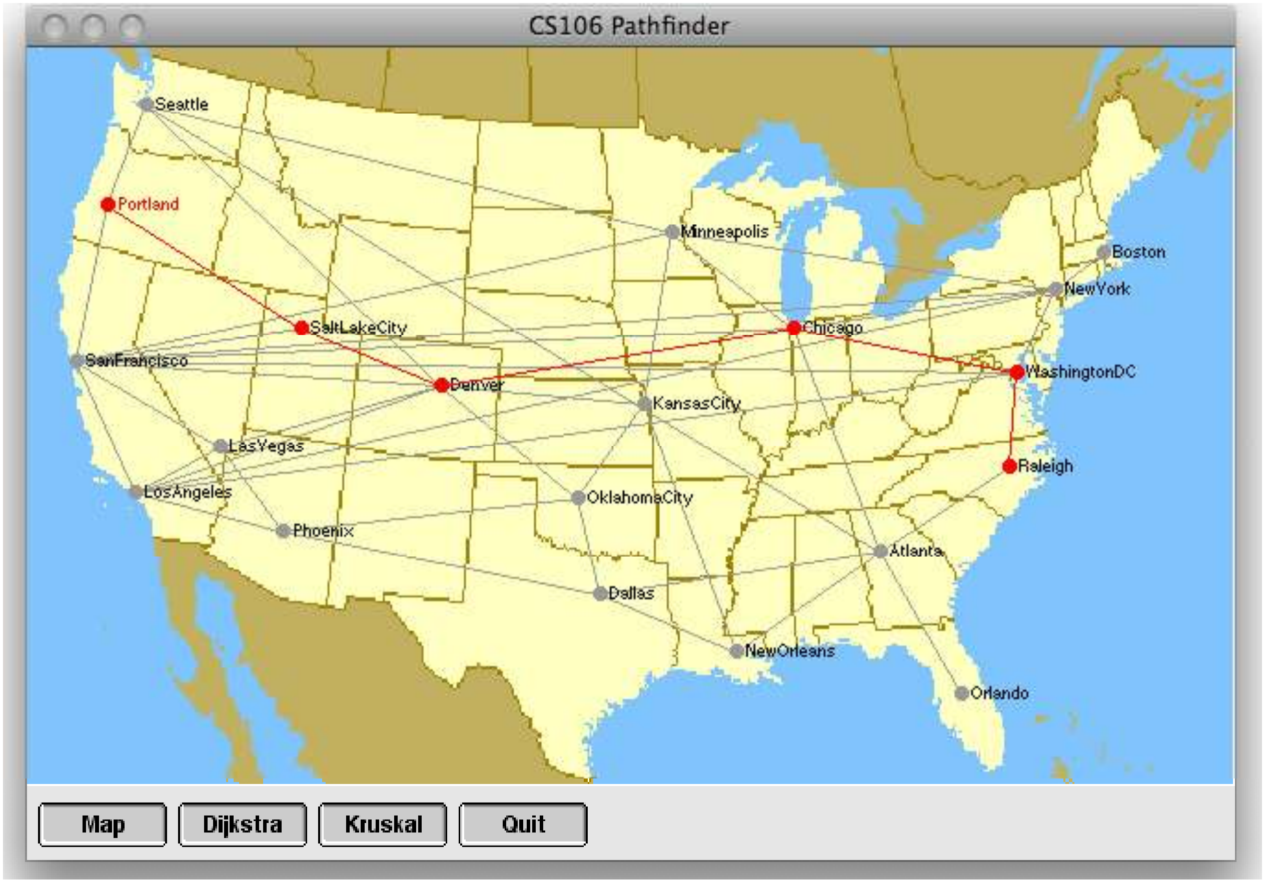
Hard Deadline: August 11th at 5:00 p.m.

The Pathfinder application from the user's perspective

The Pathfinder application begins by asking the user to specify a filename containing data for a map consisting of a background image and a graph linking locations in the map. The application then displays the graph in the graphics window. For example, if the user asks for the map **`USA.txt`**, Pathfinder will create the display shown below:



User interaction in the Pathfinder program is controlled by the buttons in the control strip at the bottom of the window. The **Map** button, for example, allows the user to enter the name of some other map, which it then loads into the window, replacing the existing one. The **Dijkstra** button executes Edsgar Dijkstra's algorithm for finding the shortest path between two nodes in the graph, as described in section 16.6 of the reader. For example, if the user selects **Dijkstra** and then clicks on the city circles for Portland and Raleigh, the Pathfinder application will highlight the shortest path between those cities, as shown below. The **Kruskal** button executes an algorithm by Joseph Kruskal that finds the lowest-cost cycle-free subgraph that connects all the nodes. Kruskal's algorithm is described in more detail later in this handout. Clicking the **Quit** button exits from the program.



The program from an implementer's perspective

The Pathfinder application is a substantial piece of code comprising 14 files, if you count both headers and implementations. The good news is that you will need to change only four of those files: **pathfinder.cpp**, **graphtypes.h**, **path.h**, and **path.cpp**. You will, however, need to use the rest of the files and must therefore understand their public interfaces, even if you can ignore the implementations for the most part. The complete list of files appears at the end of this handout and offers an overview of how the individual pieces of the project fit together. With Pathfinder—as with any large project—the most important advice we can offer is to subdivide the project into phases so that you can implement and test each phase independently. In general, if you write the code for Pathfinder all at once and then try to get it working, the complexity of the project will almost certainly make your life difficult as you debug your code. On the other hand, if you take the time to test each new piece of your code as you write it, the assignment will go much more smoothly.

We recommend that you implement Pathfinder in the following phases:
1. Determine what extensions you need to make in the graph data structures.
2. Write the code to read map data from a file into its internal representation as a graph.
3. Use the facilities from **`gpathfinder.h`** to display the map on the screen.
4. Add buttons to the control strip, making sure that they call the appropriate functions.
5. Re-implement Dijkstra's algorithm so it fits into the Pathfinder application.
6. Design and implement the **`Path`** class and integrate it into the Dijkstra code.
7. Implement Kruskal's algorithm for finding minimum spanning trees.

The sections that follow describe each of these phases in turn.

Phase 1: Determine what extensions you need to make in the graph data structures

The files in the starter folder include two interfaces that, taken together, define a graph abstraction that is both flexible and easy to use. The **`graphtypes.h`** interface introduces two structure types—**`nodeT`** and **`arcT`**—that represent the nodes and arcs of the graph. The **`graph.h`** interface defines a **`Graph`** class that takes the node and arc types as template parameters.

In the version of **`graphtypes.h`** supplied in the starter folder, the **`nodeT`** and **`arcT`** structures contain only those fields that are required to represent any graph and contain no data specific to Pathfinder. Depending on the design of your code—and also on what extensions you choose to implement—these types will need to include additional information. If nothing else, you won't be able to display the nodes on the graphics window unless you know the coordinates at which that node appears, which means that this information needs to be part of the data structure. You therefore need to change the **`graphtypes.h`** interface so that the structure types contain whatever information your application needs.

Beyond the additional data required for nodes and arcs, it is also likely that you will want to associate additional information with the graph as a whole. To do so, your initial instinct would probably be to change the files that define the **`Graph`** abstraction, adding new private instance variables to store the necessary data along with new methods for gaining access to that information. If, however, you followed this instinct and opened the source files, you would encounter the following notice:

```
//////////////////////////////////////////////////////////////////////
//                         IMPORTANT NOTE                          //
//                                                                 //
// Next year, the Graph class defined by this interface will be    //
// part of the CS106 class library, which means that clients will  //
// have to use this code "as is" and will not be able to change    //
// it to suit the requirements of a particular application.  In    //
// anticipation of this change, you are not allowed to change the  //
// contents of this file as you implement Pathfinder.              //
//////////////////////////////////////////////////////////////////////
```

So much for that idea! Fortunately, the fact that **Graph** is a class means that you can still extend its behavior without changing its definition. All you need to do is define a subclass of **Graph** that extends the base class by adding whatever additional data you need. Defining a **Graph** subclass has the further advantage of allowing you to incorporate the template parameters as part of the base type. If you don't define a new class, everywhere you need to use a graph, you have to include the type parameters, as in **Graph<nodeT,arcT>**. However, you can equate a Pathfinder-specific class with a particular instantiation of the **Graph** template using this line:

```
typedef Graph<nodeT, arcT> PathfinderGraph;
```

By doing so, you no longer need to specify the template parameters when you are working with the **PathfinderGraph** class.

Your mission in Phase 1 is therefore to read through the assignment, figure out what additional information you need to maintain at each level of the graph structure, and then make the necessary extensions to **graphtypes.h** or your new **PathfinderGraph** subclass to incorporate that information into the data structure.

Phase 2: Read map data from a file into its internal representation

The next step is to write the code necessary to read the information defining a Pathfinder graph from a data file. The starter folder includes four data files: **USA.txt**, **Small.txt**, **Stanford.txt**, and **MiddleEarth.txt**. The figure below, for example, shows the contents of the **Small.txt** data file, along with a few explanatory notes. With only four nodes and six arcs, this file is ideal for testing.

```
USA.jpg                             Name of image file to display background picture
NODES                               Marks the beginning of list of nodes
WashingtonDC 536 176                Each city is a one-word name with x-y coordinates
Minneapolis 349 100
SanFrancisco 26 170
Dallas 310 296
ARCS                                Marks beginning of list of arcs
Minneapolis SanFrancisco 1777       Each arc specifies two nodes and a distance
Minneapolis Dallas 935              Note that each arc is a bidirectional connection
Minneapolis WashingtonDC 1600
SanFrancisco WashingtonDC 2200
Dallas SanFrancisco 1540
Dallas WashingtonDC 1319
```

Each of the data files shares a common data format. The first line is the name of a file containing the background image. The next line consists of the word **NODES**, which indicates the beginning of the node entries. The nodes are listed one per line, each with a name and the *x* and *y* coordinates of the node, with each of the fields separated by spaces. 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 end of the **ARCS** section is simply the end of the file.

Reading the Pathfinder data files is similar to reading the various other data files you've used this quarter. In general, the simplest approach is to read each line from the file using **getline** and then use a scanner to interpret the information on each line. As you go through the file, you also have to make the appropriate calls to **`addNode`** and **`addArc`** methods in the **`Graph`** class to create the appropriate data structure.

As you create the graph structure, one point to keep in mind is that the arcs in the data file are *bidirectional,* in the sense that each arc indicates a connection both from the first node to the second and from the second node back to the first. Because the **`Graph`** class itself assumes that arcs run in one direction, you will need to add two arcs for each line in the data file: one from the first node to the second and one in the opposite direction.

Given that Phase 2 comes before you have any means of displaying the graph on the screen, you will need to write additional code to test whether you have read the graph data correctly. That code is not required in the final assignment, but it is good practice to leave this kind of code in place so you can use it again if, for example, you need to change Pathfinder to work with a different file format.

Phase 3: Display the map on the screen

In Phase 3, your mission is to write whatever code is necessary to take a graph of the sort you've created in Phase 2 and display it on the screen. For example, assuming that you have read the graph data from the file **`USA.txt`**, there needs to be some function you can call that will produce the window show on the first page of this handout. Such a function, however, is not sufficient in and of itself. At some point, you will need to be able to highlight individual nodes and arcs by drawing them in a different color, as shown in the graphic back on page 2. If you think about what drawing capabilities you are going to need and design this part of the assignment so that those capabilities are easy to achieve, you will have an easier time implementing the algorithms in the later phases of this assignment.

You are not, however, entirely on your own for Phase 3. We haven't made that much use of the **`graphics.h`** and **`extgraph.h`** interfaces in CS 106B, and it wouldn't be fair to force you to figure out all the graphical facilities on your own. To make this assignment manageable, we've given you a **`gpathfinder.h`** interface that includes quite a few useful methods for implementing the graphical parts of this assignment. At least for the parts of this assignment before you start adding extensions, everything you need is available in **`gpathfinder.h`**, and you won't need to work with the **`graphics.h`** and **`extgraph.h`** interfaces at all.

In contrast to earlier assignments, however, we are not going to describe in detail what functions are available in **`gpathfinder.h`**. You need to look at the interface for that. In today's software development environment, programmers are *always* building on top of existing facilities whose structure they have to figure out on their own. To help you with that process, **`gpathfinder.h`** includes extensive comments, but you need to learn how to use it by looking at those comments and prototypes.

Phase 4: Add buttons to the control strip

For the last couple of decades, the Pathfinder application has used a text-based menu to determine what operation it should execute. That style of user interface dates from the 1970s and seems entirely out of place nearly forty years later. As you can see from the first two screenshots, the new Pathfinder (courtesy of Eric Roberts' work last quarter) has buttons, which at least drags it some distance into the modern world.

Although producing an application that seems a little less dated is useful in itself, the decision to incorporate buttons into this quarter's Pathfinder assignment has an additional advantage. In modern user interfaces, mouse clicks, button activation, and other similar user-generated activities represent actions that can occur at times not of the programmer's choosing. Such actions are collectively called **events**. Programs respond to those events by designating some function that should be called whenever that event occurs. The program as a whole typically does nothing except wait for events and then call the appropriate function.

Event-driven programs of this sort depend on the idea of function pointers (sometimes, as in Java, embedded in classes as dynamically bound methods but nonetheless implemented using function pointers) to trigger the action. The programmer begins by providing the event manager with a pointer to a function that must be called whenever a particular event occurs. At some later time, when the user performs an action that triggers the event, the event manager then uses that function pointer to trigger the appropriate action.

A simple form of this mechanism is illustrated by the contents of the **`pathfinder.cpp`** file in the starter folder. The main program in the starter file

```
int main() {
   InitPathfinderGraphics();
   AddButton("Quit", QuitAction);
   PathfinderEventLoop();
   return 0;
}
```

where **`QuitAction`** is defined as

```
void QuitAction() {
   exit(0);
}
```

In this code, the call

```
AddButton("Quit", QuitAction);
```

creates a new button in the control strip and gives it the label **`Quit`**. More importantly, the call also registers the function **`QuitAction`** as the function to be invoked whenever the user activates the **`Quit`** button. The **`PathfinderEventLoop`** function simply watches the mouse and waits for the user to click on one of the buttons. When the user selects a button, **`PathfinderEventLoop`** has to invoke the associated function. In the case of the **`Quit`** button, that function is **`QuitAction`**, which invokes the standard **`exit`** function to terminate the application.

The **`AddButton`** and **`PathfinderEventLoop`** functions are exported by **`gpathfinder.h`** and have no knowledge of functions like **`QuitAction`** that live in the main program. The implementations of **`AddButton`** and **`QuitAction`** lie on opposite sides of the abstraction barrier that the **`gpathfinder.h`** interface is designed to create. The function is provided by the main program and passed in pointer form across the abstraction barrier as a parameter to the **`AddButton`** function. At some later time, the implementation of **`PathfinderEventLoop`** has to retrieve the function pointer and make a call back across the abstraction barrier to the function defined in the main program. Functions transmitted across an abstraction barrier and then invoked across that same barrier in the opposite direction are called **callback functions**. You're already familiar with the notion of a callback function, because we've discussed them as the very mechanism used by the **`Set`** template to order its elements when those elements can't be compared using **<** and **`==`**.

This simple model, however, is not sufficient to implement the other buttons you need for the Pathfinder application. The **`QuitAction`** function, after all, is extremely simple and requires no information from the application. The functions that implement the other buttons, however, must have access to the graph representing the current map. The crux of the problem is therefore how to share information between the main program and the callback function triggered by an event.

One possible strategy—and indeed one that ends up being used in far too many user- interface packages—is to share information in global variables. This strategy, however, is a poor choice, because global variables can be manipulated in any part of the program. The sharing is therefore far too general. What we want instead is a mechanism that shares a data structure only between the cooperating parties: the main program that defines the button and the callback function invoked by **`PathfinderEventLoop`**. To accomplish this objective, the usual approach is to give the callback function an argument and pass this argument in both directions as a reference parameter.

The definition of **`AddButton`** in **`gpathfinder.h`** implements this strategy by allowing an optional third argument that represents the data structure you want to share. The callback function must also then take an argument of that same type. When the user clicks a button, the registered callback function receives this information and can therefore communicate with the **`main`** program without using global variables.

To illustrate this idea, suppose that your graph data structure is stored in a variable named **`g`** whose type is the **`PathfinderGraph`** class described in the description of Phase 1. You could then create a **`Clear`** button (not required for this assignment) by adding the line

```
    AddButton("Clear", ClearAction, g);
```

to the main program. You could then define **`ClearAction`** like this:

```
void ClearAction(PathfinderGraph & g) {
   g.clear();
   DrawPathfinderMap("");
   UpdatePathfinderDisplay();
}
```

Note that the parameter **`g`** is passed by reference, so the call to **`clear`** actually clears the graph defined in the main program, and not some temporary copy of it. The creation of the buttons on the screen requires just a few lines of code. The harder part is writing the callback functions. At this point, you don't yet have enough of the assignment in place to implement the **`Dijkstra`** and **`Kruskal`** buttons, but you can put them on the screen and assign them to functions whose bodies you will flesh out later. As part of Phase 4, however, you can (and indeed should) get the **`Map`** button working by having its callback function invoke the code you wrote in Phases 2 and 3.

Phase 5: Reimplement Dijkstra's algorithm so it fits into the Pathfinder application

Figure 16-8 on page 591 of the reader has a complete implementation of Dijkstra's algorithm for finding the shortest path between two nodes. Your task in Phase 5 is simply to adapt that implementation so that it fits your data structure. As you do so, you'll need to make one change from the code in the text. The notes on page 590 ask you to assume that the **`Queue`** class has been redesigned so that it contains an overloaded version of **`enqueue`** that takes a priority. Although we are still considering a possible redesign in this direction, the Pathfinder assignment uses our implementation of the previous assignment's **`PQueue`**. You will therefore need to use the supplied **`PQueue`** class in place of the standard **`Queue`**.

Copying the code for **`FindShortestPath`**, even with that slight modification, is not your primary task in Phase 5. The more challenging part is integrating the code into the application. When you click the **`Dijkstra`** button in the user interface, the program should ask the user to click on two nodes on the graph. These nodes then serve as the **`start`** and **`finish`** arguments to **`FindShortestPath`**. More challenging still is the problem of highlighting the path after **`FindShortestPath`** returns.

Phase 6: Design and implement the `Path` class and integrate it into the Dijkstra code

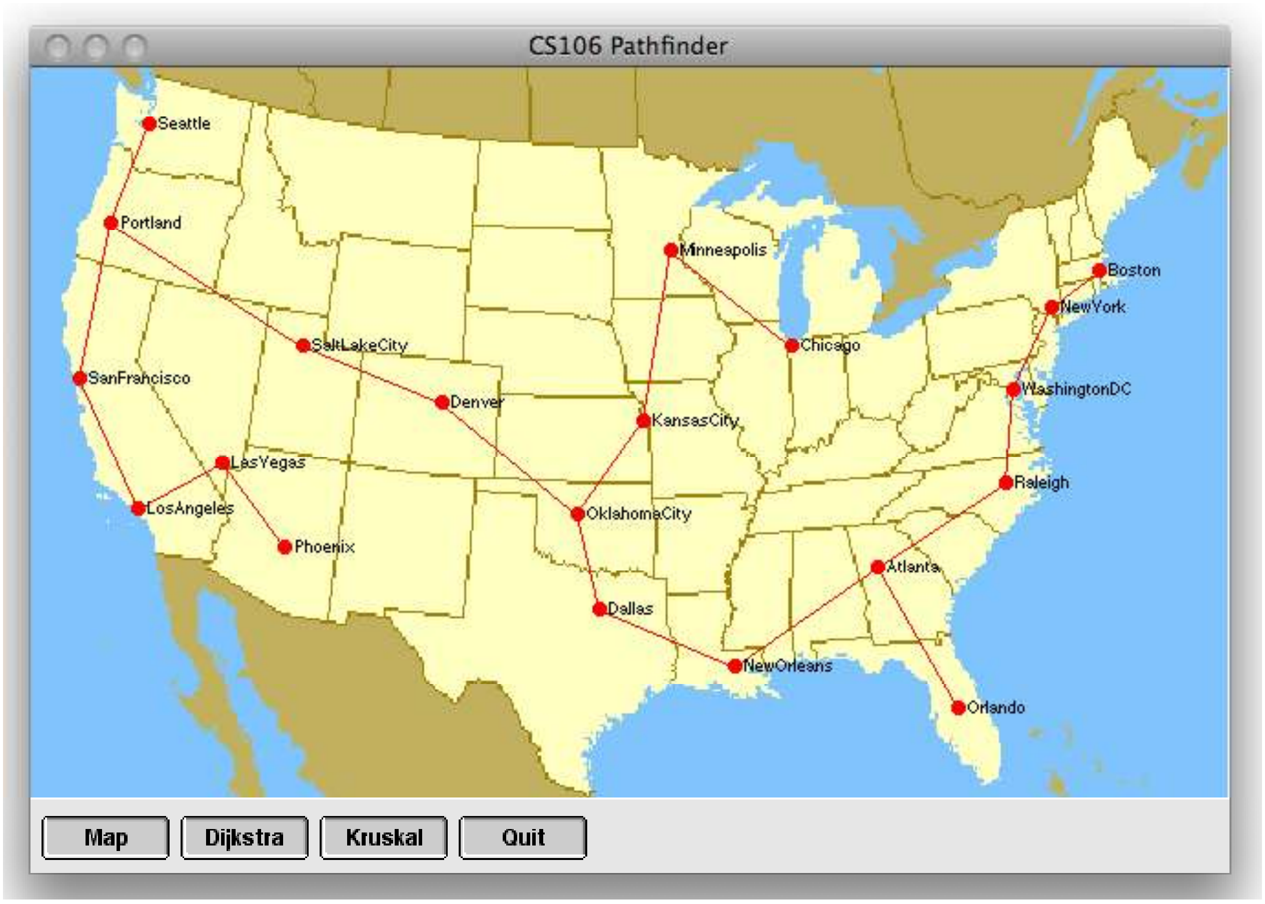
The code for Dijkstra's algorithm embodied in the **`FindShortestPath`** function has some notable inefficiencies. One of these is that calculating the total distance along a path requires adding up the individual distances (or costs, in the language of the graph abstraction) of each of the arcs along the way. Because this operation must be performed several times for each path as it evolves, looping over the arcs in a path generates redundant computation that could easily be eliminated. Unfortunately, as long as the path is represented as a **`Vector`** instead of as a separate class of its own, there is no obvious place to store the total distance alongside the list of arcs.

In Phase 6 of the assignment, your goal is to replace the **`Vector<arcT *>`** used to store the path with a new **`Path`** class that you define in **`path.h`** and then implement in **`path.cpp`**. The versions of these files in the starter project are essentially empty. We don't tell you what methods the **`Path`** class must export or what data you need to maintain in the private section of the class. We only require that your class meet the following conditions:

1. The class must not export any public instance variables. All data required to store the path must be private to the class.
2. The method that returns the total cost of the path must run in constant time.
3. The class must export a `toString` method that returns a string composed of the nodes on the path separated by arrows formed by the two-character sequence `->`.
4. Any heap storage allocated by this class must be freed when the object is deleted.

Beyond those restrictions, you are free to design and implement this class as you see fit. There is, however, one more requirement. Whenever you design a class, there are many different choices you could make in terms of what methods to export, what types to use, how flexible you want the interface to be, and so forth. Typically, each choice has advantages and disadvantages, so that the interface designer must evaluate the tradeoffs among the different possibilities. All too often, those design decisions are not reflected in the code, which means that future developers have to rediscover the underlying logic all over again. Thus, as the comments in the starter files make clear, one of the requirements of this assignment is that you include comments in the `path.cpp` file describing why you made the choices you did in your design of the `Path` class. This comment will be treated as an essay and will count for 25% of the style grade on the Pathfinder assignment.

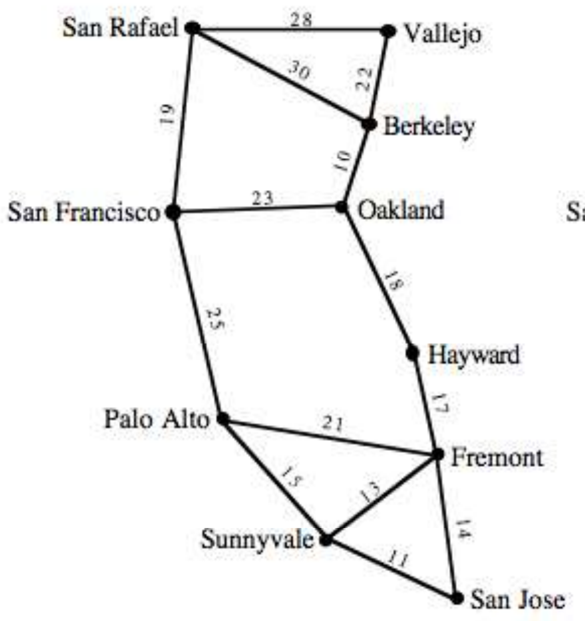
Phase 7: Implement Kruskal's algorithm for finding minimum spanning trees

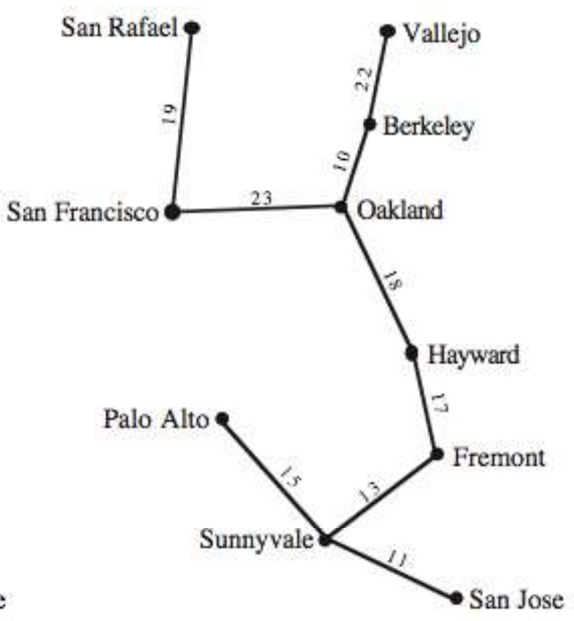


Your final task in the Pathfinder assignment is to implement another graph classic: Kruskal's algorithm for constructing a **minimal spanning tree**, which is a subtree that connects all the nodes in a graph at minimal cost. For example, if you invoke Kruskal's algorithm on the graph from the **`USA.txt`** file, you get the result shown on at the bottom of the previous page. Although a description of Kruskal's algorithm appears in the reader as exercise 16-10, it seems worth repeating that discussion here so that you have all the information in one place.

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.

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 forms a structure that is in some ways like a tree, even though it lacks a root node. Most importantly, the graph you're left with after eliminating the redundant arcs is tree-like in that it has no cycles. For historical reasons, this reduced graph that links all the nodes from the original one 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 are shown in the indented text below, which was generated by a program that traces the operation of the algorithm. 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.

```
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)
```

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.

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 structures*. 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.

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

Deliverables

Because there is no interactive grading for this final assignment, you need only submit electronically (no paper copies). The submission should include the four source files you've written or modified: **`pathfinder.cpp`**, **`graphtypes.h`**, **`path.h`**, and **`path.cpp`**.
To save space on the submission area, you should not include the images we supplied.

Contrary to rumor, you may use late days on the final assignment. Your section leaders aren't likely to grade the final assignments until after finals are over, so we're happy to allow late day use if it simplifies your dead week planning.

Files in the Pathfinder application

`pathfinder.cpp`	This file is the main program for the Pathfinder application. Most of your work in this assignment comes from adding code to this file; the version of **`pathfinder.cpp`** in the starter folder simply initializes the graphics library and puts the **`Quit`** button on the screen so that you have one example of how to use buttons. You have to write everything else. Your implementation of Dijkstra's and Kruskal's algorithms will appear here, along with the code for reading the graph data from the file and for responding to the other buttons.
`path.h` **`path.cpp`**	In the finished Pathfinder project, these files will define the interface and implementation of a **`Path`** class that represents a multistep path in a graph. As they appear in the starter project, however, these files are almost entirely empty, which means that you are responsible for both the design and the implementation of the class. This part of the assignment is described in the discussion of Phase 6 (which covers designing the **`Path`** class) on page 11.
`graphtypes.h`	This file is a simplified version of the structure-based graph interface that appears in Figure 16-3 on page 572 of the reader. The only change is that the **`graphT`** type has been removed, mostly because the Pathfinder assignment requires you to use the object-based **`Graph`** class instead. You will need to add fields to at least one of the structure types (**`nodeT`** and **`arcT`**) to keep track of the extra data the Pathfinder application needs.
`graph.h` **`graphpriv.h`** **`graphimpl.cpp`**	These files define and implement the template version of the **`Graph`** class defined in Figure 16-5 in the reader, with the exception of the low-level types provided in **`graphtypes.h`**. As the comments make clear, you should not make changes to these files, even if you are extending the assignment.
`gpathfinder.h` **`gpathfinder.cpp`** **`gpathfinderimpl.cpp`**	These files provide a library of graphical functions for the Pathfinder assignment. This interface includes all the calls that you need to make in implementing Pathfinder, so there is no reason—unless, of course, you are extending the assignment to add new features—that you would need to use the **`graphics.h`** or **`extgraph.h`** interfaces. As is typical in commercial projects, however, we have provided very little documentation of the Pathfinder graphics package in this handout. To find out what's available, you need to read the **`gpathfinder.h`** interface and use the comments there to figure out what you need to know.
`pqueue.h` **`pqueuepriv.h`** **`pqueueimpl.cpp`**	These files comprise a template version of the priority queue class you dealt with in Assignment 6. You can use the **`PQueue`** as is, although you'll need to adapt to its generalized interface.
`point.h`	This file defines a **`pointT`** structure with **`x`** and **`y`** fields.