

Assignment 7: Pathfinder

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

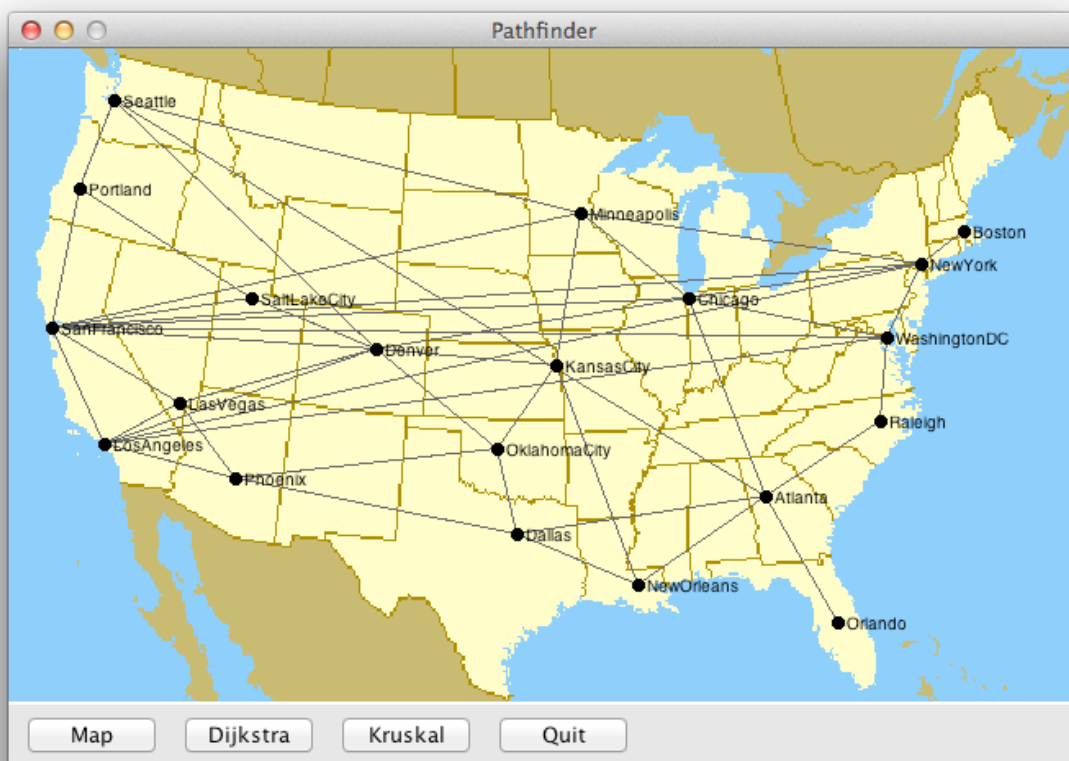
Have you ever wondered how Google, Bing, and Yahoo Maps work their magic?

Well, get ready to find out. For your final challenge, you will create the Pathfinder application, which (in addition to other useful functions) finds and displays the shortest path between any two locations.

Due: December 7th at 3:00 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:



The buttons along the bottom control strip are used as the primary controls used to advance through the application. The **Map** button, for example, allows the user to enter the name of some other map, which it then loads into the window to replace the existing one. The **Dijkstra** button executes Edsger Dijkstra’s algorithm for finding the shortest path between two nodes in the graph, as described in Section 18.6 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 sub-graph connecting all nodes (as illustrated below. Kruskal’s algorithm is described in some 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 several files, if you count both headers and implementations. The good news is that you will need to modify just three of them: **pathfinder.cpp**, **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 their implementations. The CS106B library header files—which I leave in the starter code files even though the starter code doesn’t make direct use of all of them—are documented online at <http://cs106b.stanford.edu>. 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, you'll likely struggle to manage the myriad bugs you're sure to get. 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. Write the code to read map data from a file into its internal representation as a graph. Examine the **graphtypes.h** header file within the project, and read through the **graph.h** documentation on the course web site.
2. Use facilities from **pathfinder-graphics.h** to display the map on the screen.
3. Add buttons to the control strip, making sure that they call the appropriate functions (even if they're initially placeholder functions to be fleshed out in later phases).
4. Re-implement Dijkstra's algorithm—it's presented in the reader—so it fits into the Pathfinder application.
5. Design and implement the **Path** class and integrate it into your Dijkstra code.
6. Implement Kruskal's algorithm for finding minimum spanning trees.

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

Phase 1: Read the map data from a user-specified file

The first 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	<i>Name of image file to display background picture</i>
NODES	<i>Marks the beginning of list of nodes</i>
WashingtonDC 536 176	<i>Each city is a one-word name with x-y coordinates</i>
Minneapolis 349 100	
SanFrancisco 26 170	
Dallas 310 296	
ARCS	<i>Marks beginning of list of arcs</i>
Minneapolis SanFrancisco 1777	<i>Each arc specifies two nodes and a distance</i>
Minneapolis Dallas 935	<i>Note that each arc is a bidirectional connection</i>
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 x and y coordinates, each of the fields separated by spaces. The line **ARCS** indicates the end of the node entries and the beginning of the arc entries. Each arc

identifies the two endpoints by name and gives the distance between the two (expressed as a real number with a fractional part—see **Stanford.txt**'s **ARCS** section to confirm). The end of the **ARCS** section is 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 to then use a **TokenScanner** (as you did for the **json-lite** problem in section) to subdivide each line into meaningful subparts. As you go through the file, you also have to make the appropriate calls to **addNode** and **addArc** methods in the **PathfinderGraph** class (which extends 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 to the second and from the second back to the first. Because the **PathfinderGraph** class 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 1 comes before you have any means of displaying the graph on the screen, you should write additional code to test whether you have read the graph data correctly. That code is not required of your final submission, 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 2: Display the map on the screen

In Phase 2, your mission is to write the code necessary to take a graph of the sort you've created in Phase 1 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 shown 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. 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 2. To make this assignment manageable, we've given you a **pathfinder-graphics.h** interface that includes a good number of useful functions for implementing the graphical elements of this assignment. Everything you need is available in **pathfinder-graphics.h**, and you won't need to work with the **gwindow.h** interfaces much at all.

In contrast to earlier assignments, however, we are not going to describe in detail what functions are available in **pathfinder-graphics.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, **pathfinder-graphics.h** includes extensive comments, but you need to learn how to use it by looking at those comments and the function prototypes.

Phase 3: Add buttons to the control strip

For most of the last 15 years, the Pathfinder assignment 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 over forty years later. As you can see from the first two screenshots, the new Pathfinder (courtesy of Eric Roberts' work this past summer and this past week) has buttons, which at least drags it some distance into the modern age.

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 to be invoked whenever an event occurs. The program as a whole typically does nothing other than wait for events and call the appropriate functions.

Event-driven programs of this sort depend on the idea of 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() {
    GWindow gw(kWindowWidth, kWindowHeight + kControlStripHeight);
    initPathfinderGraphics(&gw);
    addButton("Quit", quitAction);
    pathfinderEventLoop();
    return 0;
}
```

where **quitAction** is defined as

```
static void quitAction() {
    exitGraphics(); // ends the program
}
```

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 (e.g. clicks on) 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 clicks a button, **pathfinderEventLoop** knows to invoke the registered function. In the case of the **Quit** button, that function is **quitAction**, which invokes our custom **exitGraphics** function to terminate the application.

The **addButton** and **pathfinderEventLoop** functions are exported by **pathfinder-graphics.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 **pathfinder-graphics.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**.

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 minimal information. The functions that implement the other buttons, however, must have access to the relevant graph. 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 very poor one, because global variables can be manipulated by any part of the program. That global-variable approach is therefore too general and bad software engineering. 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 **pathfinder-graphics.h** implements this strategy by allowing an optional third argument that represents the data structure being shared. 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 **PathfinderGraph**. 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:

```
static void clearAction(PathfinderGraph& g) {
    g.clear();
    drawPathfinderMap("");
}
```

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 transient 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 implementations will be fleshed out in later phases. As part of Phase 3, however, you can (and indeed should) get the **Map** button working by having its callback function execute code you wrote in Phases 1 and 2.

Phase 4: Re-implement Dijkstra's algorithm so it fits into the Pathfinder application

Figure 18-11 on page 805 of the reader has a complete implementation of Dijkstra's algorithm for finding the shortest path between two nodes. Your task for Phase 4 is to adapt that implementation so that it fits into your Pathfinder application architecture. As you cannibalize the supplied implementation, be aware that the code references a **PriorityQueue** class different from those you implemented for Assignment 4 and used for Assignment 5. The **PriorityQueue** used there is actually the one we expect you to use, as it's part of the CS106B library set that includes the now-familiar **Vector**, **Stack**, **Queue**, and so forth.

Copying the code for **findShortestPath**, even with that slight modification, is not your primary task in Phase 4. 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 5: 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 inefficiency. 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**, there is no obvious place to store the total distance alongside the list of arcs.

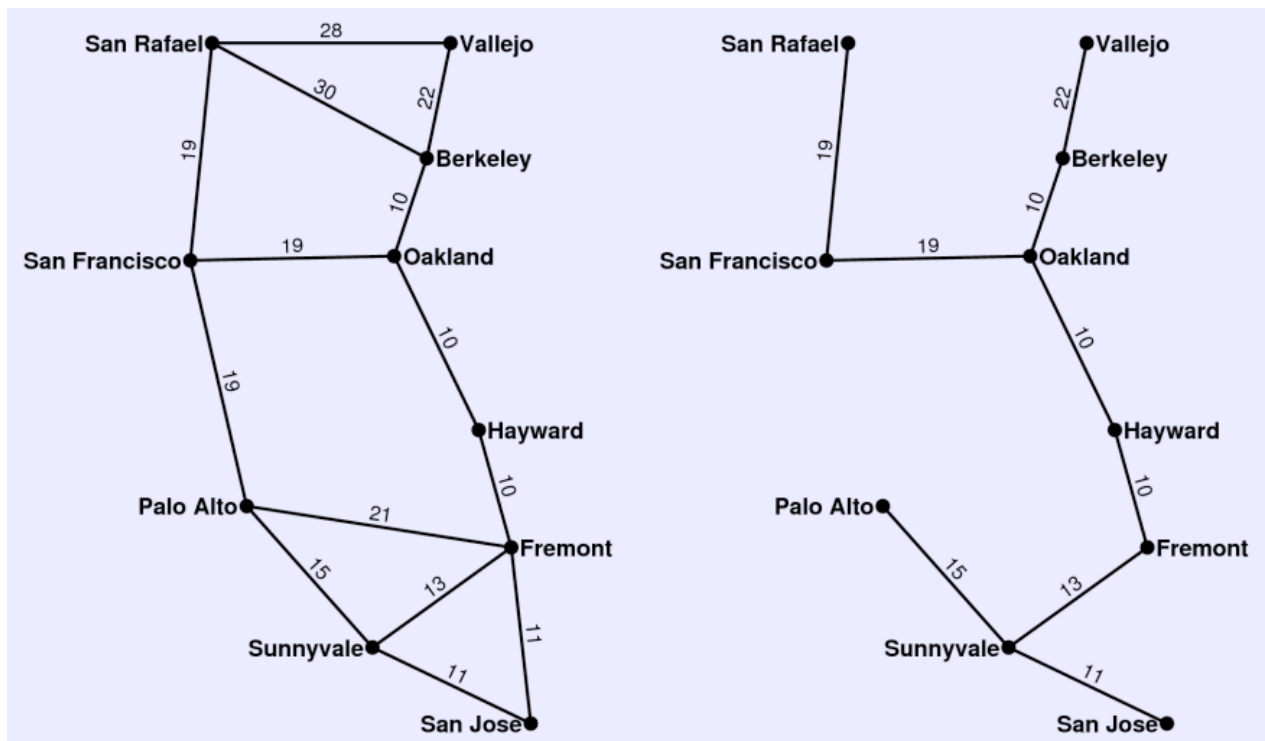
In Phase 5 of the assignment, your goal is to replace the **Vector<Arc *>** used to store the path with a new **Path** class you're to define in **path.h** and implement in **path.cpp**. The versions of these files in the starter project are essentially empty. We don't tell you what methods the **Path** 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.

Phase 6: 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 sub-tree 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 in the figure on page 2. Although a description of Kruskal's algorithm appears in the reader as exercise 18-13, 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 left graph of the two graphs on the next page. 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 is never needed, because some other path already links those cities, so 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, the particular sub-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 for finding a minimum spanning tree. Of these, one of the simplest was devised in 1956 by Joseph Kruskal. 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 discard the arc instead.

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. 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 or excluded. The strategy you'll 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, merge the sets of the two endpoints into one combined set. When considering an arc, if its two endpoints already belong to the same connected set, just skip it. Repeatedly consider arcs and merge connected sets until all nodes are merged into one large set. The perfect data structure for tracking the connected sets is our **Set** class, since it has the handy high-level operations (e.g. **operator+=**) 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. In general, you should emulate the sample application as closely as possible.
- *Careful planning aids reuse.* This program has a lot of opportunity for unification and code reuse, but it requires some careful planning up front. You'll find it much easier to do it right the first time than to go back and try to unify common code patches in the 11th hour. Sketch out your basic attack before writing anything and look for potential 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.