Assignment #7: Pathfinder

Due: Wed, May 31st in class
Absolutely **no** assignments will be accepted after Friday June 2nd

Have you ever wondered why it's so difficult to be on time for your morning lecture? Maybe you could shave off a few critical seconds by going around the Math corner rather than cutting through the Quad. Or maybe the answer is to give up walking and invest in a pair of rollerblades or a bike. But then again, bikes and blades are expensive and money doesn't grow on trees. And given the usual circus in front of the Clock Tower, you might be taking your life into your own hands. Getting there unharmed is probably worth taking a little longer, and maybe you should just be setting your alarm a bit earlier anyway. But back to the original problem: what if you truly needed to know what was the shortest or quickest route between your dorm and Gates? Or what about the similar task of plotting out the cheapest way to get you back home to Potsdam for the summer?

This task of exploring choices in an attempt to optimize a path is a graph traversal problem of the type discussed in Chapter 16 of the text. With a bag full of handy ADTs, you can make quick work of writing a program to solve all variants of path optimization problems. This is your last CS106 assignment, the last fling, the last hurrah. It is a chance to pull together your now-superior C skills, design a complicated data structure, create and use several ADTs, and implement one of the classic algorithms of computer science. Good luck!

The Pathfinder application

Your task in this assignment is to implement a program that finds optimal routes between nodes in a graph using Dijkstra's algorithm. Since a version of the code for the algorithm is given on page 725 of the text, the real work required for this assignment consists of filling in the details and embedding the algorithm into an interactive application called Pathfinder. Your implementation of Pathfinder must perform the following steps:

1. Ask the user for the name of a data file containing the information about the graph.
2. Read in the data from the map file and display a diagram showing the nodes in the graph and the interconnecting arcs. As an example, Figure 1 shows the graph generated by the `Stanford` data file supplied with the assignment. For each pair of connected nodes, there are usually several arcs corresponding to different modes of travel. For example, on the Stanford map, there are separate arcs for walking, biking, roller-blading, and taking the Marguerite bus.
3. Allow the user to select a source and a destination node by clicking on the map with the mouse.
4. Compute and display the optimum path from the source and the destination under four different metrics: distance, time, cost, and hop count. The individual segments of the path are recorded in the console window and the actual route is highlighted in red on the display.

As an example, suppose that you wanted to get from Gates to Tresidder. If you needed to get there as quickly as possible, you could bicycle straight across the campus. On the other hand, if you were being lazy and wanted to expend as little energy as possible, you could take the Marguerite bus down Serra Street and then all the way around Campus Drive until it stops at Tresidder.

Figure 2 shows the output of the Pathfinder application for the trip from Gates to Tresidder. The one taking the least time, the one that "costs" the least, presumably measured in terms of the energy required, the one covers the least distance and the path with the smallest number of hops. In addition, to the text output, the optimal path will be highlighted on the graphical display.

Figure 1. Stanford campus map and route graph

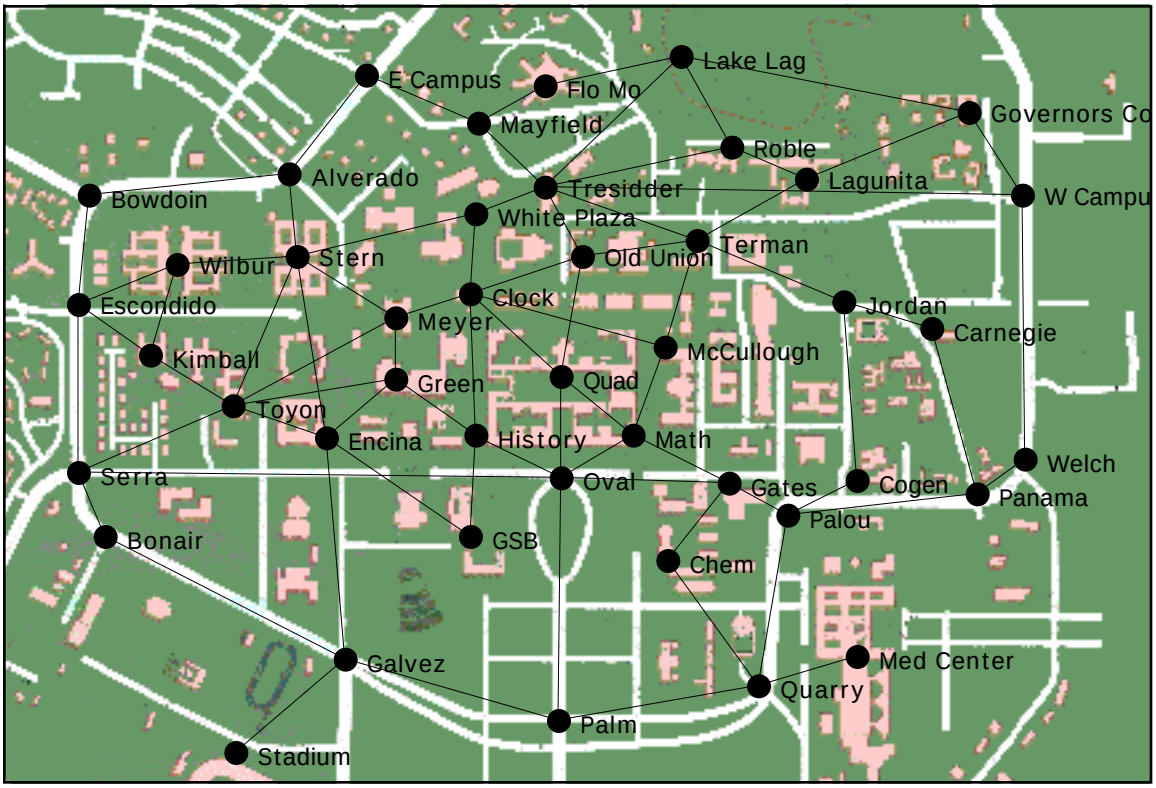

Figure 2. Sample run of Pathfinder showing minimum paths from Gates to Tresidder

```
Enter the name of data file: Stanford ↲

Click a starting node... Gates chosen.
Click on your destination node... Tresidder chosen.

The quickest path takes 3.2 minutes.
     bike from Gates to Math
     bike from Math to Quad
     bike from Quad to Old Union
     bike from Old Union to Tresidder
Hit return to continue... ↲

The cheapest path will set you back $0.00.
     marguerite from Gates to Oval
     marguerite from Oval to Serra
     marguerite from Serra to Escondido
     marguerite from Escondido to Bowdoin
     marguerite from Bowdoin to Alvarado
     marguerite from Alvarado to E Campus
     marguerite from E Campus to Mayfield
     marguerite from Mayfield to Tresidder
Hit return to continue... ↲

The path with the minimum distance is 0.43 miles
     walk from Gates to Math
     walk from Math to Quad
     walk from Quad to Old Union
```

```
    walk from Old Union to Tresidder
Hit return to continue... ↵

The path with the fewest hops has 4 segments.
    walk from Gates to Math
    walk from Math to McCullough
    walk from McCullough to Terman
    walk from Terman to Tresidder
```

Optimal path search

One strategy you could employ to find the optimal path is to exhaustively construct all possible paths between the two nodes and then compare them all to find the best one. However, given that there are a lot of possible paths between any two sites, a completely exhaustive search would be time consuming; in fact, the running time would be exponential, and therefore, intractable for a graph of any significant size. Instead, you're going to make use of a classic computer science algorithm to quickly and efficiently find the best path.

The basic idea is to keep a queue of all paths constructed so far, arranged such that the best one rises to the top. At each stage, you extract the best path so far from the queue and construct new paths by appending the edges leading away from the last node in that path. You can then consider each of the new paths you have created. You discard the path if you already know a better way to get there (i.e. you have already found a more optimal path to that node). After updating your queue with the new paths, you again choose whichever path has risen to the top of the queue and explore from there. At each stage, you always choose to go outward from the best path so far until you finally hit a path that leads to the destination node.

This is a "greedy" strategy since we always focus on our attention on what currently appears to be the best thing going so far. It is guaranteed to return the optimal path from the starting node to the destination and it does not require an exhaustive search to do so. This graph search technique is known as *Dijkstra's algorithm*, after its inventor, Edsgar Dijkstra. Dijkstra's Algorithm is discussed in more detail in Chapter 16 of the text, Before you start implementing this assignment, we recommend reading section 16.5 carefully.

Task breakdown

Straight from the CS106 Task Breakdown Task Force in Geneva, Switzerland, here's our suggested order of attack. This may look rather long and daunting, but that’s mostly because we're trying to provide you with a lot of guidance on how to approach this program. We highly recommend following our plan to keep from getting lost in the middle of all the various modules.

Task 1	Familiarize yourself with the provided ADTs, design your data structures
Task 2	Read graph from file and store in your data structure
Task 3	Draw the graph nodes and arcs in graphics window
Task 4	Allow user to choose nodes by clicking
Task 5	Implement and test the priority queue module
Task 6	Implement and test the path module
Task 7	Implement Dijkstra’s algorithm to find the optimal path
Task 8	Print and display results

Task 1—Design your data structures

Before you undertake any of the coding, the first thing you need to do is determine what data structures you need to represent the data in the graph. Much of this is made easier by use of pre-written ADTs, such as the symbol table, graph, set, and iterator. The graph as a whole is clearly a **`graphADT`**, but this is not the entire story. At each level—the graph itself, the individual nodes,

and the arcs connecting the nodes—you will need to store additional data beyond the connectivity information maintained by the graph abstraction. This additional data will consist of pointers to application-defined records associated with each level of the graph through the use of the functions **`SetGraphData`**, **`SetNodeData`**, and **`SetArcData`**. (see p. 706 of text for background information).

Think carefully about the data that you need to store in conjunction with each level of the structure. For the arcs, you clearly need to keep track of the mode of travel, travel time, distance, and cost. For the nodes, you need to maintain information about the name and location of the node on the display. For the graph as a whole, it is extremely convenient to keep a symbol table that allows you to easily translate the names of nodes into the corresponding **`nodeADT`** value.

Figuring out how to set all this up is an important first step for the assignment.

Task 2—Reading the data file

The point of this phase is to make sure that you can read in the data for the graph from the data files. To do so, you will have to open the file, read in the lines corresponding to the nodes and the arcs, and add the relevant structures to the graph

The first line of the file is a string that corresponds to the background image displayed in the graphics window behind the graph representing the map. The next line must consist of the string **`"NODES"`**, which marks the beginning of the node data. Each line of node data includes the name in quotes and the x-y coordinates of this node for the display. The word **`"ARCS"`** indicates the end of the node entries and beginning of the arc entries. Each arc connects two nodes via a means of transportation with an associated distance, time, and cost.

```
USA picture                 <-- name of map picture
NODES                       <-- word "NODES" marks begin of node listing
"Denver" (2.54 3.25)        <-- name of node and x-y coordinate in parens
"San Francisco" (1.05 2.66)
ARCS                        <-- word "ARCS" marks begin of arc listing
 "fly" from "Denver" to "San Francisco" distance: 1780  time: 1 cost: 15.00
 "drive" from "LA" to "Denver" distance: 1890 time:  3 cost: .29
```

Although it is possible to parse the file using the token scanner, it is more convenient in this case to use the **`scanf`** functions from **`stdio.h`** to do the job, primarily because the lines are specified in a rigidly controlled format. The **`scanf`** functions are described beginning on page 146 of the text. The starter code provides a **`scanf`** template to use in reading the file format to save you the trouble of messing with formatting details. Be sure to use it and spare yourself grief.

In order to create an arc, you will need to be able to access its start and end nodes by name. Rather than repeatedly iterating through all nodes in the graph to find the one with the correct name, it is more convenient if you have created a symbol table to store nodes under their name as the key to make lookup quick and easy.

Before you move on to the next phase of the assignment, it is important to make sure that you are able to read in the data correctly. How would you know if your code works without writing the rest of the Pathfinder application? The easiest way to test your code is to write debugging functions that allow you to display the data associated with a particular node or arc. For example, you could write a simple main program that reads in the graph and then calls a function to print the structure out so you can verify you have correctly read and constructed the graph.

Task 3—Draw the graph

As soon as you are confident that you can read the data for a map, you can turn to the problem of drawing the actual graph. Recall that the first line of each graph data file is the name of the picture to use as the background. The background picture for the map is stored on the Macintosh as a resource in the **`pathfinder.rsrc`** file and on the PC as a separate image file. In either case, you can draw the background by using the **`DrawNamedPicture`** function from the extended graphics library. Since the picture covers the entire graphics window and has its lower-left corner at the origin, you can draw the background by making the following calls:

```
MovePen(0, 0);
DrawNamedPicture(name);
```

where *name* is the name of the picture given on the first line of the data file.

Once you have drawn the picture, the next step in the process is to display the individual nodes and arcs in the graph. Since you don't have direct access to the nodes and arcs (they are behind the wall of abstraction), you will use iterators over the arc and node sets to gain access to them. For each node, you will need to draw a small filled circle centered at the coordinates specified in the data file and then add a label off to the side using the **`DrawTextString`** function. For each arc, all you have to do is draw a line connecting the coordinates of each endpoint.

Task 4—Allow user to choose nodes by clicking

You will ask the user to choose the starting site and desired destination by clicking on the displayed graph. To wait for a mouse click, you need to use the following code, which depends on functions from the extended graphics library:

```
double x, y;

WaitForMouseDown();
WaitForMouseUp();
x = GetMouseX();
y = GetMouseY();
```

When the mouse is pressed and released, the program will read its coordinates into the variables **`x`** and **`y`**. Given this location, you need to find if there is a node there, and if so, which one. Again, the appropriate tool for cycling through the nodes is an iterator. For test purposes, you might try printing out the name of the node that the user clicks on.

Task 5—Implement the priority queue ADT

Before you can turn your attention to Dijkstra's algorithm itself, you need to complete the implementation of some necessary ADTs. The priority queue is a variation on the standard FIFO queue described in Chapter 10 of the text. In some cases, a FIFO strategy may be too simple for the activity being modeled. A hospital emergency room, for example, needs to schedule patients according to priority. A patient who arrives with a more serious problem will pre-empt others even if they have been waiting longer. This is a *priority queue*, where elements are added to the queue by priority and when asked to dequeue one, it is the highest priority element in the queue that is removed. Such an ADT would be useful in a variety of situations; in fact, you can even sort data by inserting all the values into a priority queue and then dequeueing them one by one, which will pull them out in order.

A priority queue will be used in the search for the optimal path. At each stage you dequeue the best path so far and enqueue all paths leading away from its end for later processing. Note that the "best" path is the shortest, so a smaller distance path will have higher priority in our scheme.

Here are the functions that make up the interface of the priority queue, the formal specification for each function is detailed in the **`pqueue.h`** interface file given in the starting project folder:

```
pqueueADT NewPQueue(cmpFn cmp);
void FreePQueue(pqueueADT queue);
bool PQueueIsEmpty(pqueueADT queue);
bool PQueueIsFull(pqueueADT queue);
void Enqueue(pqueueADT queue, void *newElem);
void *DequeueMax(pqueueADT queue);
```

Note the priority queue is polymorphic (i.e. it stores elements of **`void*`** type) which means the implementation assumes nothing more about the elements other than that they are some kind of pointer. But since the implementation needs to know how to order queue elements relative to each other, the client will have to supply a compare-by-priority function that can be applied to the queue elements to determine their ordering. Since that function is chosen once and must remain consistent throughout the lifetime of the ADT, it is appropriate that it be passed just once when a new priority queue is created.

There are many data structures you could choose to represent and manipulate a priority queue, with various tradeoffs between amount of memory required, speed of the Enqueue and Dequeue operations, complexity of code to get the job done, etc. You have total freedom to choose the implementation strategy with the tradeoffs that you prefer. We aren't requiring anything sophisticated, the only mandate is that it correctly meet the functional specification given in the interface file. It is perfectly fine if your queue has *O(N)* performance on Enqueue or Dequeue in order to support the priority behavior. In the starter folder, we have provided both the linked-list and ring-buffer implementations of an ordinary FIFO queue as given in Chapter 10 of the text. You may want to adapt code from one of those implementations. You will likely only need to modify the Enqueue and/or Dequeue operations. You could also build something of your own design, such as a binary tree or the nifty heap discussed in Chapter 16 p 728-732.

It makes good sense to write test code 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 other modules.

Task 6— Implement the path module

The second ADT you need is the pathADT module, a useful abstraction described in the text but never implemented. Here are the functions that make up the interface of the path, the formal specification for each function is detailed in the **`path.h`** interface file given in the starting project folder:

```
pathADT NewPath(distanceFn fn);
pathADT CopyPath(pathADT path);
void FreePath(pathADT path);
nodeADT StartOfPath(pathADT path);
nodeADT EndOfPath(pathADT path);
void AppendToPath(pathADT path, arcADT arc);
pathADT NewExtendedPath(pathADT path, arcADT arc);
double TotalPathDistance(pathADT path);
void MapPath(pathADT path, arcMappingT fn, void *clientData);
```

This version of the **`pathADT`** is somewhat more general than the one described in the text. For example, it includes a mapping function that allows you to apply a function to each edge in the path—this will be useful when you are trying to print or draw a path. Another important change is that the **`NewPath`** function takes a client-supplied function that computes the distance traversed by each arc in the path. In this context, the word *distance* has a generic meaning. In some cases, the distance function may represent geographic distance; in others, it may represent travel time, cost, or

any similar metric that can be applied to arcs in a path. I retained the word *distance* in this interface to minimize the changes between the assignment and the description in the text. By supplying different distance functions, you can use the **`pathADT`** to sum over different criteria.

The internal structure of a **`pathADT`** is probably most easily represented using a simple singly-linked list, so that it what we recommend for your implementation. Try to organize your functions to make maximal re-use of each other where possible, rather than repeating or re-implementing functionality. For example, what other functions might be useful in making quick work of implementing the **`NewExtendedPath`** function?

Task 7 — Finding the optimal path

With all your infrastructure in place, you're finally ready to work on the heart of the program—the priority-driven traversal to find the optimal path via Dijkstra's nifty algorithm. In quick summary, here is how the algorithm works. We create a priority queue of paths, prioritized in terms of total distance (or time, etc.). When we have determined the shortest distance from the starting point to some other node (not necessarily the desired destination), we will say we have "fixed" the distance to that node. Using this terminology, our goal is to fix the distance to the desired destination.

Suppose the starting node is called A. We begin by fixing the distance to A as 0. Then we place in the queue the paths from A to all nodes directly connected to it. Now we choose the shortest of these paths--suppose it is to node D. Guess what— we have just fixed to length of the path to node D! There can't possibly be a shorter way to D, because all the other paths leading out of A are at least as long this one. We now mark D as fixed at that distance. Now we add to the queue the paths that go from A to D to each of D's direct neighbors.

Now we just repeat this process over and over. We dequeue a path, and since this is a priority queue, we will get the shortest path we have found so far. This allows us to fix the distance to the node at the end of the path, since no other path betters this one. Now we extend that path, placing the resulting paths (one for each neighbor) in the priority queue unless they end at a node whose distance we have already fixed, in which case we discard the extended path. Then we do it all over again with the next shortest path. When we finally dequeue a path ending at the desired destination, we have found our optimal solution. The graphs in the data files are *fully connected*, meaning every pair of nodes can be connected by some path (i.e. there is no discontinuity where "you can't get there from here"), and thus you are guaranteed to find some path from the start to the destination.

The code given in Figure 16-9 of the text presents a simple implementation of Dijkstra's algorithm. This code isn't exactly what you need, but it is a pretty good start. A few adjustments you will need to make:

1. The function should incorporate a changeable distance function, which allows the algorithm to use a client-supplied metric for the cost of a particular path.
2. The interface for our version of the priority queue is slightly different than that of the text, so you will need to make adjustments to fit ours.
3. You will need to add calls to **`FreePath`** and **`FreeQueue`** to reclaim memory when it is no longer needed.

To optimize for other criteria than just shortest travel time is actually just a small step: by adding a function parameter which specifies how to weight the arc to the search, you can control how a path computes its sum, and thus how paths measure up relative to one another. Optimizing for a different criterion simply means writing a new arc weighting function and then making another call to your optimize function. You will optimize for four different criteria: shortest distance, minimum time, minimum cost, and minimum number of hops.

Task 8—Reporting the optimal path

The Pathfinder application displays the optimal path in two different ways. First, it writes out a description of the path to the console window in a format that looks like this:

```
bike from Gates to Math
bike from Math to Quad
bike from Quad to Old Union
bike from Old Union to Tresidder
```

Second, the program needs to highlight the path in the graphics window. Both of these operations require you to make use of the **`MapPath`** facility, which applies a function to each arc along a path.

General hints and suggestions

- *Take care with your data structure*. It is worthwhile to take time in the first task and carefully plan what data you will need, where it will be stored and how it will be accessed. Think through the work to come and make sure your data structure adequately supports all your needs. Be sure you thoroughly understand the ADTs that you will be using as well.
- *Careful planning aids re-use*. This program has a lot of opportunity for code-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.
- *On small functions*. When making use of the different mapping and weighting needs, you will find you need to create several small functions that are used for mapping or evaluating paths. Although some of these functions may be as small as one line, this is to be expected and nothing to be concerned over. As mentioned above, try to construct these functions for optimal re-use where possible.
- *You are responsible for freeing memory*. Both your path and pqueue ADTs should have properly implemented free functions. As you search for paths, you are required to clean up after yourself and free any heap-allocated memory when you are done with it. The best way to go about doing this is to write your program first without actually freeing any memory, but putting comment placeholders where you think memory should be freed (e.g., "/* need to free this set */"). Only when you have your program running properly should you go back and add memory-freeing lines of code one place at a time, testing your program at each step. You do not have to free the graph and associated structures at the end.
- *Test on smaller data first*. There is a “Small Data” file with just 4 nodes that is especially helpful for testing in the early stages. The “US Cities” and “Stanford” data files are both quite a bit larger and good for stress-testing once you have the basics in place.

Accessing Files

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

pqueue.h/.c	Interface file and empty implementation for pqueue ADT.
path.h	Interface file and empty implementation for path ADT.
pathfinder.c	Starter file for main module.
pathfinder.rsrc	Resource file containing background pictures (Mac). This is replaced by individual picture files on the PC.
graph.h/.c	Interface and implementation for graphs as given in Chapter 16.

set.h/.c	Interface and implementation for sets as given in Chapter 15 (used by graph)
bsd.h/.c	Interface and implementation for binary search trees as given in Chapter 13 (used by set)
symtab.h/.c	Interface and implementation for symbol table as given in Chapter 11.
cmpfn.h/.c	Comparison functions for basic types (used by sets and graphs)
iterator.h/.c	Interface and implementation for iterators as given in Chapter 11
foreach.h/.c	Implementation for "foreach" macro (optional, but can make iterating a little cleaner)
Standard queues	Folder with source for usual FIFO queue as array and linked list.
Pathfinder Demo	Compiled version of a working program.

To get started, create your own starter project and add all the .c files. On the Mac, you will also need to add the image resource file to your project.

Deliverables

At the beginning of Wednesday's lecture (or at the absolute latest to Julie's office by Friday at 11am), turn a manila envelope containing a printout of your code and a floppy disk containing your entire project. Everything should be clearly marked with your name, CS106X and your section leader's name!

Due to time constraints, this last program will not be interactively graded.

A student asks: "Why does Hawaii have Interstate highways?" and "Are a cat's feet just THAT much heavier than the rest of its body?" These questions and more to appear on the final...