Dawson Zhou
CS106B Notes

Pathfinder Assignment Overview Notes

The Pathfinder assignment has a few major steps. I recommend approaching them in the following order:

1. Read in the graph files.
2. Set up the window with graphics and buttons.
3. Implement Dijkstra's algorithm.
4. Implement the `Path` class and incorporate it into Dijkstra's algorithm.
5. Implement Kruskal's algorithm.

Getting started

The project has a lot of different files. Even though a lot of them are already implemented for you, you'll want to skim through them to gain an understanding of what's available. `pathfinder-graphics.h` contains a lot of graphics and window routines you'll need to invoke in order for your application to respond to user input. `pathfinder-graph.h` describes the `PathfinderGraph` class, which you will use to store the data for your graph (prefer this to `SimpleGraph`; it implements a lot of useful functionality for you). `graph-types.h` describes other data types that you'll need to use, just as the `Node` and `Arc` `structs`.

When exploring the `PathfinderGraph` class, you'll notice that most of the information is actually described by the templatized class, `Graph`. The interface is described in full in the CS106B library documentation, located at http://www.stanford.edu/class/cs106b/materials/cppdoc.

Read in the graph files

The format of each graph file is described in detail in the assignment handout. You have full access to the `TokenScanner` class, which will make your job a lot easier. You can construct a `TokenScanner` with a `string` as input as follows:

```
string line;
// initialize line...
TokenScanner scanner(line);
```

To pull out "tokens" from the scanner (space separated portions of the string), simply invoke `scanner.nextToken()`, which returns a string. However, spaces will be treated as tokens of their own. In order to skip spaces, you should call `scanner.ignoreWhitespace()` after you create the scanner.

Scanners will treat text and numbers differently. Since some of the input files contain non-integer values, you'll want to invoke `scanner.scanNumbers()` before calling `scanner.nextToken()` when the next token is a number. It will still return a `string`, but it processes decimal points differently. You can pass the result to `stringToReal` to convert it to a `double`.

As you read in the data from the graph files, you'll be able to construct `Node` and `Arc` objects to populate your `PathfinderGraph` object.

Set up the window with graphics and buttons

The graphics routines are relatively straightforward; buttons can be a little tricky. The starter code demonstrates how the quit button might be implemented. You can add a button with a particular label using:

```
addButton("Quit", quitAction);
```

"`Quit`" describes the string that labels the button. `quitAction` is the name of the zero-argument function that should be called whenever the quit button is clicked. This is an example of a function being passed around as data; `addButton` will remember the name `quitAction` so that when the button is clicked, it can call the appropriate procedure. We write `quitAction` ourselves to make the appropriate response.

When adding buttons for "Map", "Dijkstra", and "Kruskal", you'll likely need to pass additional information into the callback functions. For example, "Map" needs to populate the `PathfinderGraph` with new nodes and arcs from the specified file, which means it needs to be able to access the `PathfinderGraph` in the first place. To accommodate this design, we provide a different version of `addButton`:

```
template <typename T>
void addButton(const string& name, void (*fn)(T& data), T& data);
```

In this version, the first argument is still the label for the button. The second argument is still a function, but instead of a zero-argument function, it should be a function that accepts a single parameter by reference. The third argument to `addButton` is the auxiliary data that will eventually be forwarded on to the callback function we also provide; those data types must match.

This means that if we want to allow the callback function for the map button to manipulate a `PathfinderGraph`, then we need to pass that `PathfinderGraph` to `addButton` when we initially create the button. That same object will be passed on to the callback function when it's invoked every time we click the map button.

Implement Dijkstra's algorithm

Dijkstra's algorithm is provided in the course reader. You can pretty much it exactly as provided, except adapting it to the Pathfinder application.

Implement the `Path` class and incorporate it into Dijkstra's algorithm

Instead of using raw `Vector<Arc*>`s to represent paths, we want to encapsulate that information in the `Path` class. `Path` can itself store an internal `Vector`, as long as it's kept private. Then, you'll just need to augment the `Vector` with information that can give you constant time total-cost lookup.

It's best to avoid storing paths in other ways; if you end up needing to dynamically allocate memory (e.g. if you store the path using a linked list or a raw array), then you'll run into a world of problems when it comes time to copy paths. `Vector` objects know how to deep copy themselves properly; linked lists and raw arrays don't.

Implement Kruskal's algorithm

The algorithm is described in the handout. It should strike you as being familiar to the MazeGenerator project from earlier in the quarter. The algorithm is actually the same, but it's being used in a different context.

Here's an overview of how you might approach Kruskal's algorithm:

1. Maintain a separate set for each node, representing connected nodes. At the beginning, nothing is connected to anything but itself.
2. Order the arcs by increasing cost.
3. For each arc:
    a. Extract the two endpoints. It doesn't matter which is the start and which is the finish.
    b. Determine if their connected sets are the same.
        i. If they are, that means they're connected, and this arc is not needed.
        ii. If they aren't, that means we should use this arc. Include it, and merge the two nodes' connected sets together.

By the time all arcs are processed, only the ones that are absolutely necessary are included. Since we processed them in order of increasing cost, the total cost will be minimized.

Good luck!