Assignment 7: Trailblazer

This assignment has a long history with major contributions by Eric Roberts, Julie Zelenski, Keith Schwarz, Leonid Shamis (UC Davis), Dawson Zhou, Marty Stepp, Chris Piech, and Chris Gregg.
This handout adapted from an one compiled by Chris Piech and Chris Gregg.

For your final assignment of the quarter, you will use your skills to implement three classic graph algorithms – breadth-first search, Dijkstra's algorithm, and A* search – as you essentially build your own version of Google Maps! In the course of doing so, you'll get to see how pathfinding algorithms work in the real world. Thematically, this assignment is all about how to represent abstract concepts like graphs and paths in software and how to use your understanding of those abstractions to drive forward the code. We hope that this serves as a fitting coda to CS106B!

Due Friday, March 17th at the start of class.

No late submissions will be accepted. No late days may be used on this assignment.

You are *encouraged* to work in pairs on this assignment.

The Assignment at a Glance

In this assignment, you will implement the following four functions:

```
Path breadthFirstSearch(const RoadGraph& graph, RoadNode* start, RoadNode* end);
Path dijkstrasAlgorithm(const RoadGraph& graph, RoadNode* start, RoadNode* end);
            Path aStar(const RoadGraph& graph, RoadNode* start, RoadNode* end);
  Path alternativeRoute(const RoadGraph& graph, RoadNode* start, RoadNode* end);
```

Each one of these functions takes as input a road network (described later on) and a start and end node, then uses a pathfinding algorithm to find a good route from the start node to the end node.

We recommend that you complete this assignment on the following timetable, which has you finish everything by Wednesday with comfortable breathing room in case you get stuck at some point:

- Aim to complete the implementations of BFS and Dijkstra's algorithm by ***Monday, March 13th***, getting used to the starter code in the process.
- Aim to complete A* Search by ***Tuesday, March 14th***. This code will be quite similar to what you write for Dijkstra's algorithm, so hopefully this won't be too much of a stretch.
- Aim to complete alternate pathfinding by ***Wednesday, March 15th***. If you structure your approach correctly, this should not require you to write too much code.
- Put in the finishing touches by ***Thursday, March 16th***, and submit everything!

The Starter Program

The starter files contain a nifty little driver program that, when run, pops up a display window containing a map of the United States, as shown here:

In the top-left of the window is a dropdown menu you can use to choose a pathfinding algorithm to run (it defaults to breadth-first search). If you click on a start node and then click a node as a destination, it will run the specified pathfinding algorithm and show you the resulting path in the window. There's an animation delay slider you can use to control the speed of the algorithm – slide to the right to slow it down, slide to the left to speed it up.

At the bottom of the window is a dropdown menu you can use to load in different maps. Just choose the map you'd like and click the Load button. We've provided some sample maps that are simplified versions of real maps (such as the default USA transportation grid or a simplified map of campus), plus some real transportation maps pulled from openstreetmaps.org.

If you actually try running any of these pathfinding algorithms you'll find that nothing much happens, and that's because you haven't implemented any of the functions yet. Once you do get things working, though, we think you'll have a lot of fun watching the algorithms in action.

The Shape of Things to Come

This assignment comes with a good deal of starter code, including a number of custom types for representing road networks. The graph structure you'll be working is represented by the type `RoadGraph`:

```
class RoadGraph {
public:
    /* Returns the set of all nodes adjacent to the given node. */
    Set<RoadNode*> neighborsOf(RoadNode* v) const;

    /* Given a start and end node, returns the edge that links them, or
     * nullptr if there is no such edge.
     */
    RoadEdge* edgeBetween(RoadNode* start, RoadNode* end) const;

    /* Returns the highest speed permitted on any road in the network. */
    double maxRoadSpeed() const;

    /* Returns the "straight-line" distance between the two nodes; that is,
     * the distance between them if you just drew a line connecting them.
     */
    double crowFlyDistanceBetween(RoadNode* start, RoadNode* end) const;
};
```

Each node in the `RoadGraph` type is stored as a pointer to an object of type `RoadNode`. The relevant parts of `RoadNode` are shown here:

```
class RoadNode {
public:
    string nodeName() const; // Name of this node, for debugging
    Set<RoadEdge*> outgoingEdges() const; // All edges leaving this node

    void setColor(Color color); // Color::GRAY, Color::YELLOW, or Color::GREEN.
                                // Note: there is no function to read colors.

    string toString() const;    // For debugging
};
```

The `RoadEdge` type referenced above represents an edge between two nodes:

```
class RoadEdge {
public:
    RoadNode* from() const; // Which node this edge starts from
    RoadNode* to()   const; // Where node this edge ends at
    double    cost() const; // The cost associated with this edge

    string toString() const; // For debugging
};
```

Finally, there's the type `Path`, which is defined as follows:

```
using Path = Vector<RoadNode*>;
```

This uses a C++ feature called ***type aliases***. This line essentially says that any time we write the name `Path`, it's literally as if we'd just written `Vector<RoadNode*>`. We've included this type alias so that you can get some familiarity with the idea and to make clearer what that vector would represent.

Step One: Implement Breadth-First Search

Your first task in this assignment is to implement the function

```
Path breadthFirstSearch(const RoadGraph& graph, RoadNode* start, RoadNode* end);
```

This function takes in a road network, a start location, and an end location, then uses BFS to find a path between the start and end nodes. It should then return that path, or an empty path if no such path exists. As an edge case, if the start and end nodes are the same, it should return a path of just the start node.

In lecture, we presented you this version of breadth-first search:

```
breadth-first-search() {
  make a queue of nodes.
  enqueue the start node.
  color the start node yellow.

  while (the queue is not empty) {
    dequeue a node from the queue.
    color that node green.

    for (each neighboring node) {
      if (that node is gray) {
        color the node yellow.
        enqueue it.
      }
    }
  }
}
```

Your task is to adapt this version of BFS in three ways:

1. Our `RoadNode` type has a function (`setColor`) you can use to set the color of a node, which will then update the display to make it easier to see what your implementation is doing. However, the `RoadNode` type does *not* have a function that lets you *read* the color of a node. This is intentional. You will need to use some sort of auxiliary data structure to store color information.
2. The version of BFS given in the pseudocode above runs until it has reached all the nodes in the graph. The version you'll want to implement in the assignment should stop as soon as it's discovered a path from the start node to the end node – any searching beyond that point is unnecessary. Think about where you should check for when you have the best path to the given node. Also, think about what happens if you don't find any paths at all, which is entirely possible in some of the graphs we've provided.
3. The version of BFS given in the pseudocode above works on *nodes* rather than *paths*. You will need your solution to ultimately return a path from the start node to the end node. Think about how you might modify this code to do this. One option would be to have the queue work with paths rather than nodes. Another option, which is a trickier but faster, is to use parent pointers.

To help you test your solution, we've provided some images of sample runs of the program. Visit

https://cs106b.stanford.edu/materials/assignment7/

to see those screenshots. Compare your outputs against our own to see whether you've got a match. If you search for a path between point *A* and point *B*, you'll visit a different set of nodes than if you find a path between point *B* and point *A* (do you see why?), so if your output doesn't match, try reversing which node is the start node and which one is the end node.

We've included three "huge" maps as part of the starter files (Stanford, Istanbul, and San Francisco). Your code might take a while to find paths in those maps, simply because the cost of animating the colors can be high. Your code will run much faster on those maps if you set the speed slider all the way to the left.

Step Two: Implement Dijkstra's Algorithm

Your next task is to implement the function

```
Path dijkstrasAlgorithm(const RoadGraph& graph, RoadNode* start, RoadNode* end);
```

This function takes in a road network, a start location, and an end location, then uses Dijkstra's algorithm to find a path between the start and end nodes. As with breadth-first search, the function should then return the path that it finds, or an empty path if no such path exists. As an edge case, if the start and end nodes are the same, you should return a path consisting of just the start node.

As a reminder, here's the pseudocode for Dijkstra's algorithm that we came up with in lecture:

```
dijkstra's-algorithm() {
  make a priority queue of nodes.
  enqueue the start node at distance 0.
  color the start node yellow.

  while (the queue is not empty) {
    dequeue a node from the queue.
    if (that node isn't green) {
      color that node green.

      for (each neighboring node) {
        if (that node is not green) {
          color the node yellow.
          enqueue it at the new distance.
        }
      }
    }
  }
}
```

Before you sit down to code this up, make sure that you understand the differences between Dijkstra's algorithm and breadth-first search. The changes are subtle but critically important. By far the most common bugs we see in the LaIR stem from simply not translating this pseudocode correctly, often as a result of copying and pasting logic from BFS without fully understanding the difference between the two. Those bugs can be frustrating to track down, especially if you're shaky on how the algorithm works.

As with breadth-first search, you'll need to make three modifications to this pseudocode:

1. You'll need to track node colors on your own.
2. You will need to stop your search as soon as you find the shortest path to the destination node, rather than just letting the algorithm run and find shortest paths to each node in the graph. Where can you safely stop your search?
3. You will need to return a path to the node that you found and must handle the case where no path exists. You can either use a priority queue of paths or use parent pointers. It's slightly harder to use parent pointers here, but it's also a lot more efficient, so we'll consider it an extension if you choose to implement the algorithm this way.

As before, check your answers against the reference images on our website and make sure you've got a match before moving on to the next step in this assignment.

Some notes on this part of the assignment:

- The lengths of the edges in the graphs are `double`s, not `int`s. Make sure you don't use `int`s to store intermediate distances; that's a really easy way to get inexplicably wrong answers.
- If you run your code on the larger maps, you'll likely find that it takes a while to complete unless you crank the speed slider all the way to the left. That's okay – it's just a consequence of the code to recolor the nodes being a bit slow.

Step Three: Implement A* Search

You now have a working implementation of Dijkstra's algorithm. Awesome! As your next task, you'll implement A* search by coding up the following function:

```
Path aStar(const RoadGraph& graph, RoadNode* start, RoadNode* end);
```

The inputs to and outputs from this function should be the same as for Dijkstra's algorithm, since, after all, A* search is just an improved version of Dijkstra's algorithm.

In order to implement A* search, you will need some sort of heuristic function you can use to estimate the distance from a given node to the destination node. That heuristic must never overestimate the distance to the target, which can be tough given that you're pathfinding on a road network.

One particularly useful heuristic for pathfinding in road networks is the following. Imagine that you're at a node and that, magically, a new road mysteriously appears that lets you drive directly to the destination node. And, oh happy day, that road's speed limit happens to be the maximum speed limit of any road in the entire network! This would be the absolute best possible way to get from the current node to the destination, since anything else you did either (1) wouldn't be as direct or (2) wouldn't be as fast. (Do you see why?) That means that *the travel time down this Magically Perfect Road will never overestimate the remaining travel time*, so the hypothetical time required to drive down a road like this one works fantastically as a heuristic for A*.

So what exactly *is* that travel time? Well, it would be the length of a road from your current location directly to the end node (whose length would be the "as the crow flies" distance from the current node to the end) divided by the maximum speed allowed on any road in the network. Mathematically:

$$h(\text{node}, \text{end}) = \frac{\text{crow's-flight distance from node to end}}{\text{maximum speed allowed on any road}}$$

And hey – don't you have some functions lying around to make that happen? (Hint: yes you do. Go back to Page 3 for more details. ☺)

Some notes on this part of the assignment:

- Before you sit down to code things up, make sure you have a good understanding of how A* search works. How is it similar to Dijkstra's algorithm? Where does it differ?
- As before, make sure not to use `int`s to hold distances or heuristic estimates – those should be stored in `double`s, since they're not necessarily integer-valued. That's particularly important for when you're computing the above quotient, since if you divide and round down you'll end up with weird rounding errors.
- The priority you use when enqueuing a node or path into the priority queue should be the estimated distance to that node (just like Dijkstra's algorithm) plus the heuristic distance to the end. If you try to reuse the priority of one path as a baseline for the priority of the next path, remember that the priority of the preceding path is itself a sum of a true distance and heuristic distance. If you forget this, you can easily end up with incorrect path costs, usually by adding in the heuristic distance multiple times.

Step Four: Find an Alternate Path

If you've ever used Google Maps to find directions, you've probably seen that it offers alternate directions from your start point to the endpoint. Those alternate directions are usually a little bit slower than the main directions it gives (after all, it always tries to find you the best possible route), but typically use a very different set of roads to accomplish the goal. For example, if you wanted to drive from Stanford to San Francisco, you could choose to take US-101, which is more direct but not the most scenic drive in the world (unless a bunch of tech company billboards is your idea of "scenic,"), or you could choose to take I-280 (absolutely gorgeous and usually less-trafficked, but also less direct). Similarly, if you wanted to drive from Stanford to Sacramento, you could take I-80 (direct, but a lot of traffic) or CA-82 to CA-237 to I-880 to I-680 to I-80 (less direct, much more pleasant). In short, a good alternate path should be

- sufficiently different from the fastest possible path (so that it represents a "fundamentally" different route), and
- as efficient as possible, subject to the previous constraint.

For the purposes of this assignment, we'd like you to use the following approach. Start off by finding the best possible path you can from the start node to the end node. This will be the "main" path. Then, do the following. For each edge in the main path, find the best possible route you can from the start node to the end node *that does not use that edge*. Each path you get back will be slightly different from the main path, and (depending on which edge you removed) potentially *significantly* different from the main path. At the same time, each of those paths will still be very efficient.

Once you've done this, you'll end up with a collection of alternate paths that you can choose from. So which one should you pick? Well, we'd ultimately like to pick one that's "sufficiently different" from the original path. One way to do that is to assign a ***uniqueness score*** to each of those alternate paths, where the uniqueness score of that path is defined as the fraction of nodes on that path that aren't on the main path. Mathematically:

$$\textit{uniqueness}(\text{alternate path}) = \frac{\text{number of nodes in the alternate path that don't appear in the main path}}{\text{number of total nodes in the alternate path}}$$

The higher a path's uniqueness score, the more dissimilar it is from the original path. Consequently, we'd like you to choose as your alternate path ***the shortest path of the ones you found earlier whose uniqueness score is at least 20%***. To summarize:

1. Begin by finding the optimal path from the start node to the end node. This is the main path.
2. For each edge in the main path, find the shortest path from the start node to the end node that does not use that edge.
3. Return the best path you found in step (2) whose uniqueness score is at least 20%.

It's entirely possible that you won't find a path in Step (2) whose uniqueness score is at least 20% (or, in extreme cases, that you'll even find any alternate paths at all!) If that happens, follow the convention from the rest of the assignment and just return an empty path to indicate that you couldn't find a good alternate.

Some thoughts on this part of the assignment:

- The key here is ***code reuse***. You already have working pathfinding algorithms from parts one, two, and three. Can you find a way to reuse that code, or at least, find some way to factor out the common code so that you're not rewriting everything from scratch?
- When computing the uniqueness score, make sure that you don't succumb to the woes of integer division and accidentally have the quotient divide and round down. Cast the numerator or the denominator to a `double` before doing the division.
- This algorithm is slow on the Stanford, Istanbul, and San Francisco maps. Don't worry if it takes a while to complete there. Do worry if it takes a long time to complete on the other maps, though.

Step Five: Build Your Own Map!

You now have a nifty program that's essentially your own version of Google Maps! To cap off your CS106B experience for the quarter, we'd like you to create your own custom map. The good news is that our provided starter code has a map file reader built into it, so all you'll need to do is create some custom data files and bundle a custom image along with it.

Update the `map-custom.txt` and `map-custom.jpg` representing a map graph of your own. Put the files into the `res/` folder of your project. The text file contains information about the graph's nodes (vertices) and edges. The graph can be whatever you want, so long as it is not essentially the same as any of the provided graphs. Your image can be any (non-offensive) JPEG image you like; we encourage you to search around and find a good one. (Google Image Search is your friend here!)

For full credit, your file should load successfully into the program without causing an error and be searchable by the user. The text file containing the graph structure should match the expected file format exactly – unfortunately, as with many programs that read custom data files, even minor typos can cause the whole file to be rejected. Here's a sample data file (`map-small.txt`), annotated (in ***blue italics***) with the expected format:

```
IMAGE
map-usa.jpg          Name of the image file
654                  Width of the image, in pixels
399                  Height of the image, in pixels

VERTICES
Washington, D.C.;536;176  CityName;x;y
Minneapolis;349;100       CityName;x;y
San Francisco;26;170      CityName;x;y
Dallas;310;296            CityName;x;y

EDGES
Minneapolis;San Francisco;1777          StartCity;EndCity;Weight
Minneapolis;Dallas;935                    (or, for directed edges:
Minneapolis;Washington, D.C.;1600               StartCity;EndCity;Weight;true)
San Francisco;Washington, D.C.;2200
Dallas;San Francisco;1540
Dallas;Washington, D.C.;1319
```

(Optional) Step Six: Extensions!

There are all sorts of ways you can improve upon the basic assignment, and we'd encourage you to try some out if you have some free time. Here are some suggestions to help you get started:

- ***Use parent pointers***. As mentioned earlier in the handout, using parent pointers to reconstruct the path you find to destination node is much faster than enqueuing paths into a queue or priority queue. Update your solution to use parent pointers instead of paths.
- ***Find a better heuristic for A* search***. The heuristic we recommended in the section on A* is one possible heuristic, but by no means the only possible heuristic. See if you can find a better heuristic to use on these graphs. Just make sure that your heuristic is admissible (that is, it never overestimates the distance to the target!)
- ***Implement bidirectional search***. A common alternative to A* search is *bidirectional Dijkstra's algorithm*, where you search outward both from the start node to the end node and from the end node to the start node. As soon as the two searches compute shortest paths to some common node, you can join those two paths together to form the overall shortest path. Implement this algorithm and compare it against A* search. How does it do?
- ***Find better alternate paths***. The algorithm presented in this handout is one that, in the words of its creator, is a bit "janky." Can you come up with a better algorithm?

Submission Instructions

To submit this assignment, submit your `Trailblazer.cpp` source file, along with your `map-custom.txt` and `map-custom.jpg` files, on Paperless. And that's it! You're done with the very last assignment of the quarter! Congratulations!

Good luck, and have fun!