Assignment 7: Trailblazer

Updated by Anton Apostolatos, Chris Gregg, Chris Piech and Keith Schwarz. Thanks to Julie Zelenski, Nick Parlante, Jerry Cain, Eric Roberts, Dawson Zhou, Leonid Shamis (UC Davis), Marty Stepp, and Marissa Gemma

Due: **Thursday, August 17 at 12:00 PM (noon).** Note that you **cannot use late days** on this assignment.

This is an individual assignment. Write your own solutions and **do NOT work in pairs.**

Assignment Overview and Starter Files

For your final assignment of the quarter, you will use the skills you've learned in this course 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. This assignment gives you lots of practice in thinking about how to represent abstract concepts like graphs and paths in software, and how to use your understanding of those abstractions to develop the code. We hope you'll have fun and see this as a capstone to your CS106B experience!

In `trailblazer.cpp`, you will implement the following four functions:

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

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

Starter Code **Expected Output**

We provide you with several support files, but you should not modify them. Turn in the following files:

1. `trailblazer.cpp`, code to perform graph path searches
2. `map-custom.txt`, a world map of your own creation
3. `map-custom.jpg`, a world map of your own creation

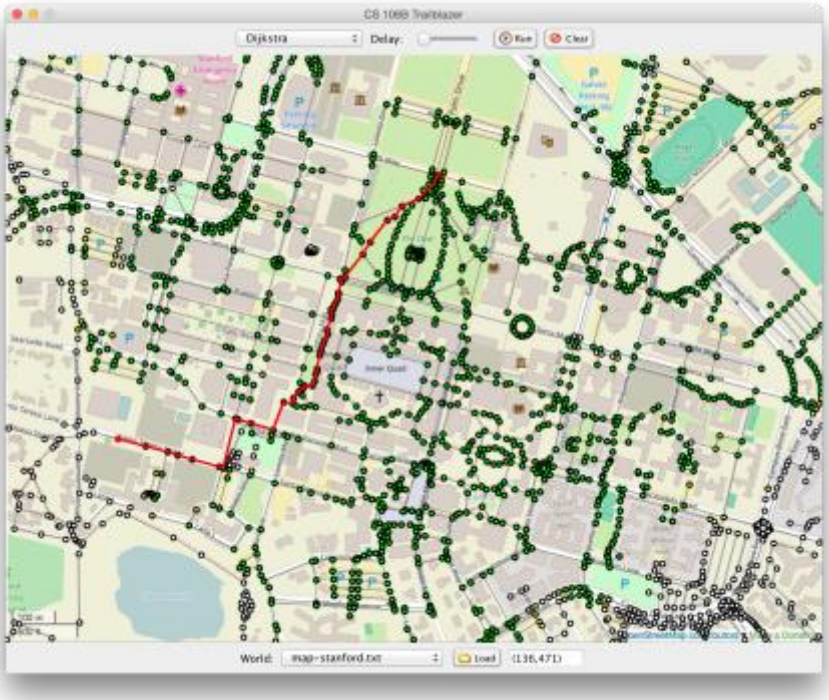
Map Display

We've provided you with a program to display various road maps and allow you to see the shortest path between any two nodes that you select. In the window that pops up when you run the program, you can select one of four path-searching algorithms from the drop-down menu at the top:

1. Breadth-first search (BFS)
2. Dijkstra's algorithm
3. A* search
4. Alternative Route

You can load other maps by selecting them from the bottom drop-down menu and clicking "Load." The maps for this assignment use real road map data; the Stanford, San Francisco, and Istanbul maps are open source, provided by www.openstreetmaps.org.

As an example of how the interactive display works, if you loaded the Stanford map and selected one node at the top of the Oval and another at FroSoCo, your program should display the best route:



If you click on any two nodes in the world (once you've implemented at least one of the algorithms successfully, that is), the program will find a path from the starting position to the ending position. As it does so, it will color the vertexes green and yellow based on the colors the algorithm assigns to them (more on these colors below). Once the path is found, the program will highlight it and display information about the path cost in the console.

Provided data types

As we noted above, in your `trailblazer.cpp` file, you must write the following 4 functions for finding paths in a graph:

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

Each of the first three implements a path-searching algorithm taught in class; the fourth will be explained in greater detail below. For all of them, you will need to make use of a few special data types that we provide for you: `Path`, `RoadGraph`, `RoadNode`, and `RoadEdge`.

The first is a `Path`, which is an alias for a `Vector<RoadNode*>`. In other words, `Path` is a typedef, which is essentially a shorthand for another type; we use typedefs for readability— it's much easier to manage the expression `Path` than `Vector<RoadNode*>` in your code.

The road network world, in turn, is represented by a `RoadGraph`, which is a thin wrapper around the `BasicGraph` we saw in class. Each vertex or `RoadNode` represents a specific location in the world. An edge (`RoadEdge`) between two vertices means that there is a direct road between the two. The cost of the path is the time it takes to traverse that road.

The vertexes and edges of a given map world are contained inside a `RoadGraph` object passed to each of your algorithm functions. Here are the member functions contained in RoadGraph, which you must use to implement your algorithms:

RoadGraph Methods	**Description**
`Set<RoadNode*> neighborsOf(RoadNode* v)`	get all RoadNodes that can be reached by a single edge from v
`RoadEdge* edgeBetween(RoadNode* start, RoadNode* end)`	return the edge that connects start to end
`double maxRoadSpeed()`	the fastest speed of any edge in the graph
`double crowFlyDistanceBetween(RoadNode* start, RoadNode* end)`	the distance between two nodes as the crow flies.

Each vertex in the graph is represented by an instance of the `RoadNode` structure, which has the following members for you to use:

RoadNode Member	Description
`string nodeName()`	vertex's name, such as "r34c25" or "vertex17"
`Point location()`	returns the position of the node on the map
`void setColor(Color c)`	sets the color of the node in the display
`string toString()`	returns a printable string representation of the vertex for debugging

When you call `setColor` on a `RoadNode`, the GUI is automatically updated, the program's internal count of the number of verticies that you touch is increased, and a delay is added (so you can see your algorithm animate). All vertex colors are reset before each of your path-searching algorithms is called.

Finally, each edge in the graph is represented by an instance of the `RoadEdge` structure, which has the following members:

RoadEdge Member	Description
`RoadNode* from()`	the starting vertex of this edge
`RoadNode* to()`	the ending vertex of this edge
`double cost()`	cost to traverse this edge
`string toString()`	returns a printable string representation of the edge for debugging

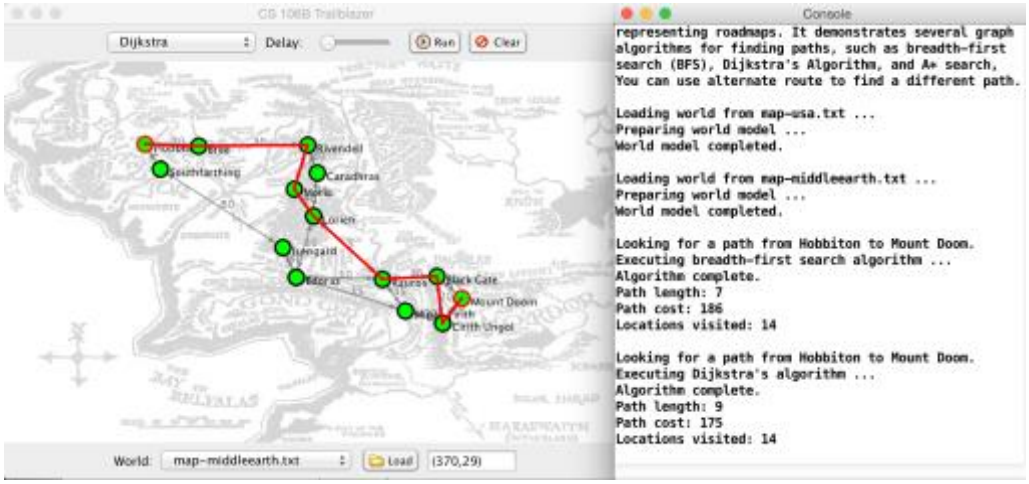
Graph algorithm details

For each of your path-finding algorithms, you should search the given graph for a path from the given start vertex to the given end vertex. If you find such a path, you should return a list of all vertices along that path, with the starting vertex first (index 0 of the vector) and the ending vertex last.

If no path is found, return an empty path. If the start and end vertices are the same, return a one-element vector containing only that vertex. Though some graphs may be undirected (i.e., all edges go both ways), your code should not assume this. You may, however, assume that the graph passed in is not corrupt.

Our provided main client program will allow you to test each algorithm one at a time before moving on to the next. You can (and should) add more functions as helpers, particularly to remove redundancy between some algorithms containing similar code.

Your algorithm should work on any `RoadGraph` instance, such as this map of Middle Earth that can help Frodo get from the Shire to Mount Doom:



Coloring: In addition to searching for a path in each algorithm, we also want you to add some code to give colors to various vertices at various times. This coloring information is used by the GUI to show the progress of your algorithm and to provide the appearance of animation. To give a color to a vertex, call the color member function on that vertex's `RoadNode` object, passing it a global color constant such as `Color::GRAY`, `Color::YELLOW`, or `Color::GREEN`. For example:

```
// set the start vertex's color to green
start->setColor(Color::GREEN);
```

Here is a list of which colors you should use and when:

- **enqueued = yellow**: Whenever you enqueue a node to be visited for the first time, such as in BFS and Dijkstra's algorithm when you add a node to a data structure for later processing, color it yellow (`Color::YELLOW`).
- **visited = green**: Whenever your algorithm directly visits and examines a particular vertex, such as when it is dequeued from the processing queue in BFS, color it green (`Color::GREEN`).

The provided GUI has an animation slider that you can drag to set a delay between coloring calls. As long as the slider is not set all the way to the left, each call to `setColor` on a vertex will pause the GUI briefly, causing the appearance of animation, so that you can watch your algorithms run.

Path Finding

See the expected output files for reference on what your program should do. For Dijkstra's and A* you should get the same path costs as shown in the expected outputs. But you do not need to exactly match our path itself, nor its "locations visited", so long as your path is a correct one.

Check out the search handout for an example of the search algorithms. You can choose to rely on the pseudocode we go over in lecture, or try to come up with your own pseudocode of the algorithms on your own.

Breadth-first search implementation notes: Implement the version of the algorithm pseudo-code from lecture. Make sure to keep track of nodes that you have already visited.

Dijkstra's algorithm implementation notes: The version of Dijkstra that we talk about in class is slightly different than the version in the book. Your implementation of Dijkstra's algorithm can follow any version. The one we talked about in lecture is probably the easiest. The priority queue should store paths, and once you find a path that ends in the destination, you should reconstruct the shortest path back. See the lecture slides for more details.

A* implementation notes: As discussed in class, the A* search algorithm is essentially a variation of Dijkstra's algorithm that uses heuristics to fine-tune the order of elements in its priority queue to explore more likely desirable elements first. So when you are implementing A*, you need a heuristic function to incorporate into the algorithm.

One handy feature of the display is that it lets you easily compare the behavior of Dijkstra's algorithm and A* (or any pair of algorithms). Simply try performing a search between two points using Dijkstra's algorithm, then select A* and press the "Run" button at the top of the GUI window. This will repeat the same search using the currently selected algorithm, so you can see how much more efficient A* is.

A* Heuristic

In order for A* to work, you need to define a *heuristic* function which under-estimates the cost of travelling between two vertices.

For a road map (where cost is travel time), a simple heuristic is to calculate the time it would take to get between the nodes assuming that there was a direct super highway between the two nodes. If the speed of a super highway is as fast or faster than the speed on any road, then the algorithm will underestimate the true graph distance (which makes it an admissable heuristic). To calculate how long it would take to travel along a super highway between two nodes, use the two `RoadGraph` functions `crowFlyDistanceBetween` and `maxRoadSpeed`.

Recall that **travelTime = distance / speed** .

Your A* search algorithm should always return a path with the same length and cost as the path found by Dijkstra's algorithm. If you find that the algorithms give paths of different costs, it probably indicates a bug in your solution.

Alternative Route

When travelling between two points on a road map, you may want to take the fastest route, but if there is a reasonable alternative you might prefer that instead. For example, though highway 101 is often the slightly faster way to get to San Francisco from the peninsula, highway 280 is more beautiful. (Chris Gregg thinks that you should also consider going all the way to highway 1.) Your final task is to implement the `alternativeRoute` function.

In this example the shortest path between the oval and the back of MemChu is shown in red, and a next best alternative is shown in blue:

We already know how to find a shortest path from a start to a goal. How do we find an alternate route?

To find an alternate path from a start node to an end node, first we are going to calculate the best (i.e., shortest) path between start and end. Then, for each edge in the best path we are going to

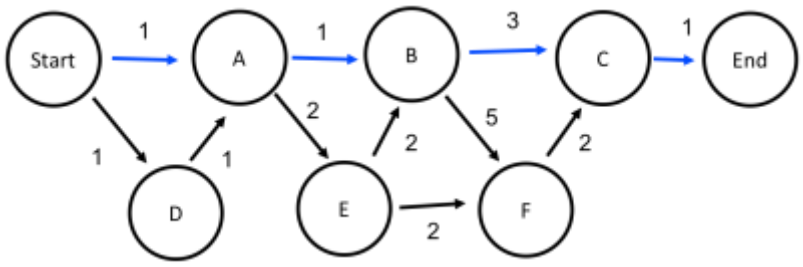
calculate the shortest path from start to end that *ignores* that edge. Each search will thus return a candidate "alternate route". Your function should return the lowest cost alternate route that is sufficiently different from the original best path. (When we say "sufficiently different," we mean according to a threshold that we define for you as a constant, `SUFFICIENT_DIFFERENCE`, with a value of 0.2.)

To calculate the difference between two paths: we define the difference of an alternate path P from the best path B to be the number of nodes that are in P but are not in B, divided by the number of nodes in B:

$$\text{difference}(P,B) = \frac{\#\text{nodes in P that are not in B}}{\#\text{ nodes in B}}$$

For each edge in the best path you will produce one candidate alternate route. Your function should choose, from those candidates, the shortest path that has a difference score greater than 0.2 when compared to the original best path (i.e. greater than the defined constant `SUFFICIENT_DIFFERENCE`).

Example of alternative route calculation: In the graph below, the shortest path from Start to End has four edges. For each edge we compute a candidate alternate route:



Excluded Edge	Alternate Path	Alternate Cost	Difference Score
Start -> A	Start -> D -> A -> B -> C -> End	7	1/5
A -> B	Start -> A -> E -> F -> C -> End	8	2/5
B -> C	Start -> A -> B -> F -> C -> End	10	1/5
C -> End	*No path*	∞	0

The shortest alternate route is the path `Start -> D -> A -> B -> C -> End`. However it has a difference score of 0.2 (recall that we are looking for a path with difference score *greater than* 0.2). The shortest alternative that *also* has a difference greater that 0.2 is:

```
Start -> A -> E -> F -> C -> End
```

So this is the path that your alternative path algorithm should return.

There are different ways to accomplish this modification to your path algorithm. One option is to simply write a separate and modified Dijkstra or A* function can ignore or exclude an edge. While this option is allowed for your project, it does duplicate a lot of code. A better option would be to *refactor* your code and write a modified function that performs Dijkstra or A*, and can either perform the standard algorithm *or* can ignore a given edge. Then, modify your original Dijkstra or A* function to simply call your new function.

Creative input file

Along with your implementation of the four path-finding functions, you must also turn in the files `map-custom.txt` and `map-custom.jpg`, which represent a map graph of your own choosing. Put the files into the `res/` folder of your project. The text file should contain information about the graph's vertexes and edges, in **precisely** the format given in this example (which comes from `map-small.txt`):

```
IMAGE
map-usa.jpg                          ← image file name
654                                  ← image width, in pixels
399                                  ← image height, in pixels

VERTEXES
Washington, D.C.;536;176             ← vertex format is: name;x;y
Minneapolis;349;100
San Francisco;26;170

EDGES
Minneapolis;San Francisco;1777       ← edge format is: vertex1;vertex2;weight
Minneapolis;Washington, D.C.;1600        (or, for a directed one-way edge:
San Francisco;Washington, D.C.;2200       vertex1;vertex2;weight;true)
```

The map itself can be whatever you want, so long as it is not essentially the same as any of the provided graphs. Choose any (non-offensive) JPEG image you like; we encourage you to use a search engine like Google Image Search to find an interesting landscape. For full credit, your file should load successfully into the program without causing an error and be searchable by the user.

Development Strategy

- Trace through the algorithms by hand on small sample graphs before coding them.
- Work step-by-step. **Complete each algorithm before starting the next one**. (We put this in bold because it's important!) You can test each individually even if others are incomplete. We suggest doing BFS, then Dijkstra's, then A*, and finally Alternative Route.
- Start out with tiny worlds first. It is much easier to trace your algorithm and/or print every step of its execution if the world is small. Try small or middle earth before trying one of the large graphs (especially Stanford, Istanbul or San Francisco).
- Remember that edge costs are `doubles`, not `int`s.
- In both A* and Dijkstra you will need to calculate the cost of a path. You can (but don't have to) memoize the cost calculation.

- In Dijkstra's algorithm, don't stop your algorithm early when you enqueue the ending vertex; stop it when you dequeue the ending vertex (that is, when you color the vertex green).
- The alternate path algorithm that we ask you to implement is slow. Test it on small maps. On the Stanford map, it's normal for it to take a few seconds, and on Istanbul and San Francisco it can take several minutes.

Possible Extra Features

Though your solution to this assignment must match all of the specifications mentioned previously, you are allowed and encouraged to add extra features to your program if you'd like. Here are some ideas for extra features that you could add.

1. **Implement bidirectional search:** A common alternative to using A* search is to use a bidirectional search algorithm, in which you search outward from both the start and end vertices simultaneously. As soon as the two searches find a vertex in common, you can construct a path from the start vertex to the end vertex by joining the two paths to that vertex together. Try coding this algorithm up as a fifth algorithm choice.
2. **Better alternate:** The alternate route algorithm is slow and not guaranteed to find the best alternate. Can you come up with a different algorithm that is faster or that uses a more realistic measure of path distance?
3. **Add more data from open maps:** We generated the maps from openstreetmaps.org. In the graphs you see, we added some nodes and edges, but left out a lot of interesting information (like street names and building). Here is a link to a folder with all of the raw openstreetmap data. Do anything you like with it. Can you print out directions? We haven't tried yet but it seems interesting!
4. **Other:** If you have your own creative idea for an extra feature, go for it!

Indicating that you have done extra features:
If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can easily identify their code).

Submitting a program with extra features:
Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one named `trailblazer.cpp` without any extra features added (or with all necessary features disabled or commented out), and a second one named `trailblazer-extra.cpp` with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Our submission system saves every submission you make, so if you make more than one we will be able to view all of them; your previously submitted files will not be lost or overwritten.