`tag` — Version 1.0 ***due:*** *Wed, Oct 17th @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

In this assignment, you will implement an interface for finding shortest paths in *weighted* graphs. Your algorithm will be the Swiss army knife of searches, having many features that standard shortest path algorithms don't have. In particular it will support multiple sources and destinations (finding the shortest path from any of the sources to any of the destinations), it will support the widely used A* heuristic and it will support negative weight edges. It will do this all in one concise piece of code.

1.1 Submission

Submit your solutions by placing you solution files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn5/`

Name the files exactly as specified below. You can run the check script located at

`/afs/andrew.cmu.edu/course/15/210/bin/check/05/check.pl`

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw05.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

For the programming part, the only files you're handing in are

```
distance.sml
distanceTest.sml
```

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.2 Style

As always, you will be expected to write readable and concise code. If in doubt, you should consult the style guide at `http://www.cs.cmu.edu/~15210/resources/style.pdf` or clarify with course staff. In particular:

1. **Code is Art.** Code *can* and *should* be beautiful. Clean and concise code is self-documenting and demonstrates a clear understanding of its purpose.
2. **If the purpose or correctness of a piece of code is not obvious, document it.** Ideally, your comments should convince a reader that your code is correct.
3. **You will be required to write tests for any code that you write.** In general, you should be in the habit of thoroughly testing your code for correctness, even if we do not explicitly tell you to do so.
4. **Use the check script and verify that your submission compiles.** We are not responsible for fixing your code for errors. If your submission fails to compile, you will lose significant credit.

2 Paths"R"Us®

In class we covered Dijkstra's algorithm for finding single source shortest paths in a weighted graph $G = (V, E, W)$. As discussed, it only works with graphs with no negative edge weights. Your goal is to extend this algorithm in various ways as described below. Each of these extensions are worth points. We will run tests on each separately, although the interface is fixed (see the next section). We also have various written questions included below. These must be put in your written solution file.

Multiple Sources. The first extension is to allow multiple source vertices $S \subseteq V$. The goal is to find the shortest shortest path length from any of these sources.

Task 2.1 (10%). Implement multiple sources in your code.

Multiple Targets. Your function will take a set of targets $T \subseteq V$ (think of them as destinations) and return the length of the shortest of the shortest paths to these targets. If multiple sources and destinations are given your algorithm should return the shortest shortest path length between any $s \in S$ and any $t \in T$. The purpose of supplying target vertices is to reduce the part of the graph that needs to be searched, especially if a target is close to the source. Your code therefore needs to take advantage of this.

Task 2.2 (10%). Implement multiple targets in your code.

The A* Heuristic. The A* technique is a very effective heuristic to further reduce the portion of a graph that needs to be searched to find a shortest path to a particular target (or targets). The idea is to narrow the search toward the target vertices. The heuristic is widely used in practice.

The A* technique is a variant of Dijkstra's algorithm. Recall that Dijkstra's idea is to maintain a set of visited vertices $X \subseteq V$ and on each step to add a new vertex $w \in V \setminus X$ to X for which

$$\min_{v \in X, (v,w) \in E} (\delta(s, v) + w(v, w))$$

is minimum. He showed that for such a vertex the given path length is minimum.

In A* we are additionally given a heuristic $h(v) : V \to \mathbb{R}_+ \cup \{0\}$ over the vertices. The A* variant then changes each Dijkstra step so it adds a new vertex $w \in V \setminus X$ for which

$$\min_{v \in X, (v,w) \in E} (\delta(s, v) + w(v, w) + h(w))$$

is minimum. Note that the only difference is the added $h(w)$. For this variant to produce correct path lengths the function $h(w)$ needs to have the following property:

consistency property: for every directed edge $(v, w) \in E$, $h(v) - h(w) \leq w(v, w)$.

If the heuristic satisfies the consistency property, then when the A* approach adds any target $t \in T$ to the visited set, it has found the shortest path length to t.

As an example of heuristic that satisfies the consistency property consider edge weights that represent distances between vertices in euclidean space (e.g. the length of road segments). In this case a heuristic that satisfies the consistency property is the euclidean distance from each vertex (i.e. as the bird flies) to a single target. Multiple targets can be handled by simply taking the minimum euclidean distance to any target.

Task 2.3 (5%). Briefly argue why this heuristic satisfies the consistency property.

Task 2.4 (5%). Give a heuristic that causes A* to perform exactly as Dijkstra's algorithm would.

Task 2.5 (10%). Prove in a paragraph or less that the A* heuristic works: i.e. if the consistency property is true, then when A* visits a vertex in $t \in T$ it has found the shortest path to t.

Task 2.6 (20%). Implement the A* heuristic in your code by accepting a user supplied heuristic $h(v)$. You need not verify that the function satisfies the consistency property.

Negative Edge Weights. In class we will go or have gone over the Bellman Ford algorithm for graphs that have negative edge weights. As it turns out, however, Dijkstra's algorithm can be modified so that it handles negative edge weights. But, this requires allowing a vertex to be visited multiple times and in the worst case ends up requiring the same work as the Bellman Ford algorithm. The modified variant of Dijkstra's has an important advantage—if most edges are non-negative it can do significantly less work than Bellman Ford. Your job is to implement this variant. The idea is to select the next vertex to visit based on the exact same approach as Dijkstra (or A*) but now due to the negative edge weights we are no longer guaranteed the distance is indeed the best path. We therefore allow the vertex to be visited again if the path length decreases.

Task 2.7 (5%). Can you still stop early (i.e. as soon as you visit a target) when using negative edge weights? If not, what is the right stopping condition?

Task 2.8 (5%). If there are no negative edge weights, how many vertices will your modified algorithm visit?

Task 2.9 (20%). Implement this variant that allows negative edge weights in your code. Make sure it terminates even for negative weight cycles.

2.1 Specification

You need to implement an algorithm that supports the single source shortest path problem augmented in the ways described. You will get points for correctness of each of the parts implemented correctly. The interface you need to implement is defined in `DISTANCE.sig`. The main function has the following interface:

```
findPath (c : vertex -> real) (G : graph) (S : Set.set) (T : Set.set)
         : ((vertex * real) option * int)
```

The function returns a pair of values consisting of:

1. The target vertex that is found and the distance to that target. If S or T are empty, if no vertex in T is reachable from S, or if there is a negative weight cycle this value needs to be `NONE`.
2. A count of how many times a vertex is "visited". We say a vertex is visited the first time it is added to the visited set X or if it has already been visited and its distance is decreased (can only happen with negative edges weights).

The reason for counting the number of vertices visited is that the count gives you a sense of the efficiency of your algorithm. After all the goal of specifying target vertices and using the A* heuristic is simply to reduce the number of vertices visited. We will reduce points for algorithms that visit too many vertices or take more than $O(\log n)$ work for each edge they "relax".

2.2 Testing

Task 2.10 (10%). We have given you a file with some tests. You need to augment these.