`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 your 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.

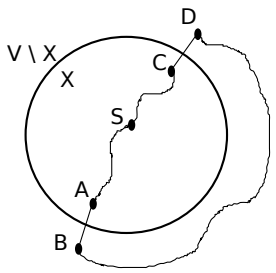
Solution 2.3 Let the notation $|u - v|$ denote the euclidean distance between any vertices u and v. Given any directed edge (u,v), let t be the target with minimum euclidean distance to v. So, $h(v) = |v - t|$ and $h(u) \leq |u - t|$ (it may be that u has a closer euclidean distance to some target that isn't t). Note that $w(u,v) = |u - v|$. From the triangle inequality, $|u - t| - |v - t| \leq |u - v|$, which by our equalities and inequality above implies that $h(u) - h(v) \leq w(u,v)$.

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

Solution 2.4 The heuristic $h(v) : V \to \mathbb{R}_+ \cup \{0\} = x \mapsto c$ for any constant c causes A* to perform exactly as Dijkstra's algorithm would. This is because Dijkstra uses "$\delta(s,v)+w(v,w)$" as it's priority, A* uses "$\delta(s,v) + w(v,w) + h(w)$", and the ordering of priorities shifted by a constant amount is isomorphic to the ordering of unshifted priorities.

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.

Solution 2.5 Solution 1:



We are going to prove by induction that when A* visits a vertex v outside of X it has found the shortest path to v.

The base case is when $X = \emptyset$. Whichever vertex is chosen, the distance to itself is 0, so this trivially is the shortest path to that vertex (we are working with graphs where all edges are positive).

The step case is when X is not empty and A* has to visit a vertex in $V \setminus X$. Let's say that vertex is D as in the figure above. The vertex right before D on the path from S is C and C is in the set of visited vertices X. We know by induction that the path from S to C is the shortest such path $\delta(S,C)$. Now we want to show that the path from S to D is the shortest such path.

Since A* chose to visit D, we know that for all other vertices $A \in X$ and $B \in (V \setminus X)$, the inequality below holds. By induction we also know that the path from S to A is the shortest such path $\delta(S,A)$.

$h(B) + w(A,B) + \delta(S,A) > h(D) + w(C,D) + \delta(S,C)$

This is equivalent to $h(B) - h(D) + w(A,B) + \delta(S,A) > w(C,D) + \delta(S,C)$ (1)

From the consistency property we know that for every directed edge $(v,w) \in E$, $h(v) - h(w) \leq w(v,w)$. If we have a path made of the vertices $s_1, s_2, s_3, s_4, ..., s_n$ that is the shortest path from s_1 to s_n, we get that

$h(s_1) - h(s_2) + h(s_2) - h(s_3) + h(s_3) - h(s_4) + ... + h(s_{n-1}) - h(s_n) \leq w(s_1,s_2) + w(s_1,s_2) + ... + w(s_{n-1},s_n)$.

This means that $h(s_1) - h(s_n) \leq \delta(s_1,s_n)$, or the difference between h applied to the first and last vertices of a path is $\leq$ than the weight of that path. We use this insight for (1) and we get:

$\delta(B,D) + w(A,B) + \delta(S,A) > w(C,D) + \delta(S,C)$

So the path to D that A* chose is indeed shorter than all other possible paths to D. This is what we wanted to prove in the step case of induction.

By the induction principle, when A* visits a vertex in $t \in T$ it has found the shortest path to t.

Solution 2.5 Solution 2:

If there is no path from the any source vertex s to any target t, then it easily follows that A* will return the correct result. For the rest of the proof, we assume that there is some path from a source vertex to a target vertex.

We use $\delta(v)$ to mean the shortest path distance from any source vertex to vertex v.

Lemma A: For any edge (u,v) on the shortest path, $\delta(u) + w(u,v) + h(v) \leq \delta(t) + h(t)$, where t is any target vertex of a shortest path.
Proof: Consider the last edge of a shortest path, (v,t). By definition, $\delta(v) + w(v,t) = \delta(t)$, and because h is a consistent heuristic, $h(v) - h(t) \leq w(v,t)$. The lemma follows from these equations.

Lemma B: After every round of relaxation, either the algorithm terminates, or there is always some u in the visited set and some v on the frontier such that (u,v) is an edge in a shortest

path.
Proof: The lemma holds after the first round of relaxation, because the first edge of all shortest paths start at one of the source vertices s and goes to a neighbor of s. On each subsequent round of relaxation, at least one of the following cases about any edge (u, v) that the lemma discusses will apply:

1. If v is not removed from the frontier during the relaxation, (u, v) continues to cause the lemma to hold.
2. If v is removed from the frontier, but has any neighbor w not in the visited set such that (v, w) is an edge in the shortest path, (v, w) is now an edge that causes the lemma to hold (although (u, v) has ceased to be).
3. Otherwise, if v is removed from the frontier but there is no "unexplored" edge (v, w) in a shortest path, then v is a target vertex and the algorithm terminates.

For A* to return an incorrect result, the "wrong path" needs to come out of the priority queue; that is, some element with a priority strictly greater than $\delta(t) + h(t)$ is the lowest element of the priority queue at the end of a round, where t is any target vertex for which that quantity is minimized. This is impossible, because from lemmas A and B, we see that there is always an element in the priority queue with priority $\delta(u) + w(u, v) + h(v) \leq \delta(t) + h(t)$ up until the algorithm terminates. Therefore, A* cannot terminate with an incorrect path, and because we assumed that there is some path from a source vertex to a target vertex, A* must terminate.

Solution 2.5 Solution 3:

Yet another valid solution (although this description is less than formal) is to construct a graph as follows:

1. The weight of each edge (u, v) is replaced with the weight $w(u, v) - h(u) + h(v)$
2. The graph has a new vertex s that has an edge to each source s_i of weight $h(s_i)$.
3. The graph has a new vertex t that has an edge from each target t_i of weight $\text{MH} - h(t_i)$, where MH is the maximum of the heuristic values of the target vertices.

It is guaranteed that this graph contains no negative edge weights (for 1. by the consistency property, and trivially for 2. and 3.).

Performing Dijkstra's algorithm on this graph from s to t will result in a path $\langle s, s_x, a, b, \ldots, z, t_y, t\rangle$ for which $h(s_x) + (w(s_x, a) - h(s_x) + h(a)) + (w(a, b) - h(a) + h(b)) + \cdots + (w(z, t_y) - h(z) + h(t_y)) + \text{MH} - h(t_y) = w(s_x, a) + w(a, b) + \cdots + w(z, t_y) + \text{MH}$ is minimized. This is the shortest path from any s_x to any t_y in the unmodified graph. Lastly, this "Dijkstra on a modified graph" is exactly equivalent to A*: the values entering the priority queue are exactly the same, and the rest of the algorithms besides the queue-entry were identical anyways. Thus, the correctness of A* with a consistent heuristic can be proven from the correctness of Dijkstra'a algorithm.

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?

Solution 2.7 You cannot stop as early; for example, consider the graph $V = \{s, t, v\}$, $E = \{(s,t),(s,v),(v,t)\}$, where $w(s,t) = 0$, $w(s,v) = 1$, $w(v,t) = -9$. Dijkstra's algorithm would find a 0 cost path, but the shortest path has cost -8. A correct stoping condition is to stop when either the priority queue is empty, or the highest priority element in the queue involves a distance $\delta(s,v)$ who's corresponding path involves more than $|V|$ edges. The latter case will occur if and only if a reachable negative cycle exists: if the latter case occurs, then the minimum cost path to some $v \in V$ involves some $u \in V$ twice. We know that the subpath from u to u is necessarily a negative cycle because it's clearly not a positive cost cycle, and it's not zero cost because Dijkstra's algorithm does not re-relax to vertices unless the cost is in fact lower than when the vertex was initially visited. Furthermore, if a reachable negative cycle exists, the algorithm will certainly end up in the aforementioned latter case; new paths will keep being added to the priority queue without it going empty, and there are only a finite amount of paths that use less than $|V|$ edges.

Note that the worst-case complexity of Dijkstra's algorithm with these stopping conditions is significantly higher than the worst-case complexity of the Bellman Ford algorithm—it's exponential for some graphs. It turns out that the "stop after $|V| * |E|$" vertices have been explored is not a valid stopping condition, and may occasionally report the existence of negative cycles when there are none.

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

Solution 2.8 On graphs without negative edge weights, these stopping conditions lead to every vertex being visited once. However, it's within cost bounds to check to see if there is any edge with a negative weight (merely iterating through the edges achieves this), so a modified algorithm could use the normal Dijkstra stopping conditions in the specific case that a graph has no negative edge weights—and thus visit no more vertices than the unmodified Dijkstra algorithm.

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.