Homework 6

15-210: Parallel and Sequential Data Structures and Algorithms (Spring'13)

Version 1.0 `tag — 1390a4f129` ***due:** Wed, Mar 20 @ 11:59pm*

1 Introduction

You have recently been hired as a TA for the course Parallel Cats and Data Structures at the prestigious Carnegie Meow-llon University. There is a lot of work that you need to do in order to keep this course afloat. Obviously, most of this work will come in the form of writing and reasoning with graph algorithms. Fortunately, your time in 15-210 has prepared you well for this.

Over the course of this "abridged" lab, you will solve **two** coding problems, A* Search and Bridges, instead of the usual one. Start early to avoid post-spring break blues.

2 Files

After downloading the assignment tarball from Autolab, extract the files from it by running `tar -xvf abridgedlab-handout.tgz` from a terminal window. You should see the following files:

1. `lib/`
2. `Makefile`
3. `support.cm`
4. `sources.cm`
5. `abridgedlab.pdf`
6. `BRIDGES.sig`
7. `ASTAR.sig`
8. `MkTableDijkstra.sml`
9. `abridged.sml`
10. * `MkBridges.sml`
11. * `MkAStarCore.sml`

You should only modify the last 2 files, denoted by *. Additionally, you should create a file called `abridgedlab-written.pdf` which contains the answers to the written part of the assignment.

3 Submission

To submit your assignment: open a terminal, `cd` to the `abridgedlab-handout` folder, and run `make`. This should produce a file called `handin.tgz` which contains the following files: `MkBridges.sml`, `MkAStarCore.sml`, and `abridgedlab-written.pdf`. Open the Autolab webpage and submit this file via the "Handin your work" link.

4 Bridges

Since you are a TA for Parallel Cats and Data Structures, Meow Zoidberg of Facelook Corporation decides to interview you himself for a summer internship. At the end of the interview, you ask him if you can play around with the in-house version of Graph Search for a while. Even though it's a blatant violation of user privacy rights, Zoidberg sportingly agrees.

After spending some time using Graph Search, you exclaim – "Look, the social graph of all users from Carnegie Meow-llon is connected!", and after a moment's pause, "I wonder if removing a single edge could disconnect this graph." Zoidberg replies – "I'm afraid Graph Search doesn't have that feature currently. If you can solve this problem, the job is yours." *Do you have what it takes?*

4.1 Specification

Let $G = (V, E)$ be some **unweighted, undirected**, simple[1] graph. G is not necessarily connected. An edge $\{u, v\} \in E$ is a *bridge* if it is not contained in any cycles (equivalently, if a bridge is removed there will no longer be a path which connects its endpoints). Your task is to find all bridges in G.

Task 4.1 (5%). Define the type `ugraph` representing an undirected graph and write the function

```
val makeGraph :  edge seq -> ugraph
```

in `MkBridges.sml` which takes in a sequence S representing the edges of a graph G as described above and returns that same graph under your `ugraph` representation. For full credit, `makeGraph` must have have $O(|E| \log |V|)$ work and $O(\log^2 |V|)$ span.

S contains no duplicate elements — that is to say, for any two vertices u, v, at most one of $\{(u, v), (v, u)\}$ is in the sequence S. Vertices are labeled from 0 to $|V| - 1$, where $|V|$ is the maximum vertex label in the edge sequence, plus one. You may assume that all vertices in the graph have at least one neighbor.

Task 4.2 (30%). Implement the function

```
val findBridges :  ugraph -> edge seq
```

in `MkBridges.sml` which takes an undirected graph and returns a sequence containing exactly the edges which are bridges of G. For full credit, `findBridges` must have $O(|V| + |E|)$ work and span. The edges need not be ordered in any way, but for any edge $\{u, v\}$, at most one of $\{(u, v), (v, u)\}$ should appear in the output sequence. Our solution is around 40 lines with comments.

You should make use of the argument structure ascribing to `ST_SEQUENCE`, assuming the costs for `MkSTSequence` and `ArraySequence` in the library documentation.

4.2 Testing

For this lab, you will *not* need to submit test cases. You should, however, test your code locally before submitting to Autolab. In `abridged.sml`, we have provided some structure definitions for your testing convenience. In particular, you should make use of `Bridges.makeGraph` and `Bridges.findBridges`.

Finally, you should submit your code to Autolab to run it on our test cases. Note that passing the Autolab testing suite makes up a large portion of your grade for this assignment.

[1] no self-loops; at most one edge between any two vertices.

5 Paths "R" Us

You've decided that Facelook Corporation isn't your ideal company. Instead, this summmer you'll be joining a YC (YowCat) funded startup located in Silicat Valley named Paths "R" Us. Your intern project is to implement the swiss army knife of graph searches, having many features that standard shortest-path algorithms don't have. In particular, it will support multiple sources and destinations, as well as the widely-used A^* heuristic. Using this tool, Paths "R" Us plans to initiate a hostile takeover of Facelook and eventually rule the world.

5.1 Using Dijkstra's

Your intuition tells you that you should modify Dijkstra's algorithm, as covered in lecture, to solve this problem. Recall that Dijkstra's algorithm solves the *single source* shortest paths problem (SSSP) in a **weighted, directed** graph $G = (V, E, w)$, where $w : E \to \mathbb{R}$ is the weight function. However, it only works on graphs with no negative edge weights — that is, when $w : E \to \mathbb{R}^+$.

Task 5.1 (5%). Give an example of a graph on ≤ 4 vertices with negative edge weights where Dijkstra's algorithm fails to find the shortest paths. List the priority queue operations up to the point of failure.

Solution 5.1 Google it.

Task 5.2 (5%). Assuming there are no negative-weight cycles in G, how would you modify Dijkstra's to accomodate negative edge weights and return the correct solution?

Solution 5.2 Don't stop early. Check visitied vertices to see whether a shorter distance has been found.

5.2 The A^* Heuristic

Dijkstra's Algorithm searches evenly in all directions from a source vertex — you cleverly observe that this can be optimized when you have more information about a certain graph. For example, suppose that you are driving from Pittsburgh to New York. Using Dijkstra's to find the shortest route, you would end up exploring cities such as Cleveland, Columbus, Washington D.C., and Philadelphia, before finally arriving at New York. Some of these cities are't even in the right direction!

This intuition motivates the A^* algorithm, which makes use of a *heuristic* function $h : V \to \mathbb{R}^+$ to estimate the distance from a vertex to some destination. Suppose that $d : V \to \mathbb{R}^+$ gives the shortest path distance to a vertex from some source. Like Dijkstra's, A^* maintains a priority queue of vertices to search, but ordered by minimizing $d(v) + h(v)$ instead of just $d(v)$.

To find the optimal distance from a source to a destination using A^*, the heuristic must be both **admissible**[2] and **consistent**. An *admissible* heuristic is one that is guaranteed to return a smaller-or-equal distance than the actual shortest path distance from a vertex to a destination vertex. Furthermore, a heuristic h is *consistent* if for every edge $e = (u, v) \in E$ (a directed edge from u to v), $h(u) \leq h(v) + w(e)$.

[2] There may be times when you choose to use an inadmissible heuristic to gain a faster search, if not an optimal path (eg. in a game where speed is preferable to a best path). In addition, it is possible to change the heuristic dynamically to alter the choice between speed and accuracy, depending on the situation.

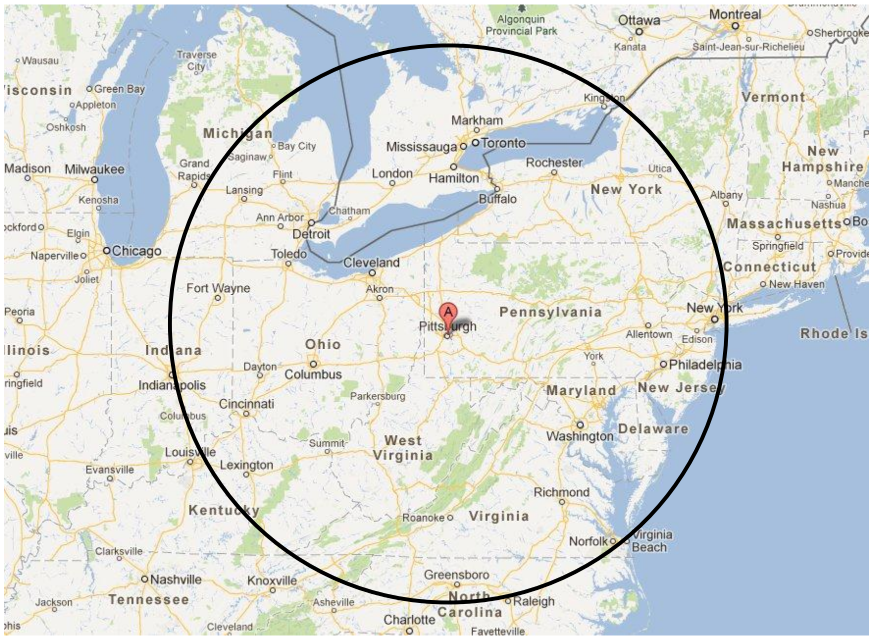

Figure 1: Area explored using Dijkstra.

As an example of a heuristic that is both admissible and consistent, 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 both properties is the Euclidean distance from each vertex (i.e. straight line distance from Pittsburgh to New York) to a single target. Multiple targets can be handled by simply taking the minimum Euclidean distance to any target.

Figure 1 shows the area (circled) of the map explored using Dijkstra's algorithm when computing the shortest path from Pittsburgh to New York, and Figure 2 shows the area of the map potentially explored using the A^* algorithm with a good heuristic (e.g. the Euclidean distance to New York). As you can see, in this case Dijkstra's algorithm ends up exploring all locations within d of Pittsburgh where d is the distance of the shortest path from Pittsburgh to New York. By narrowing the search towards the locations closer to New York, the A^* algorithm explores fewer locations before finding the shortest path.

Task 5.3 (5%). Briefly argue why the Euclidean distance heuristic is both admissible and consistent for edge weights that represent distances between vertices in Euclidean space.

Solution 5.3 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 5.4 (5%). Give a heuristic that causes A^* to perform exactly as Dijkstra's algorithm would.

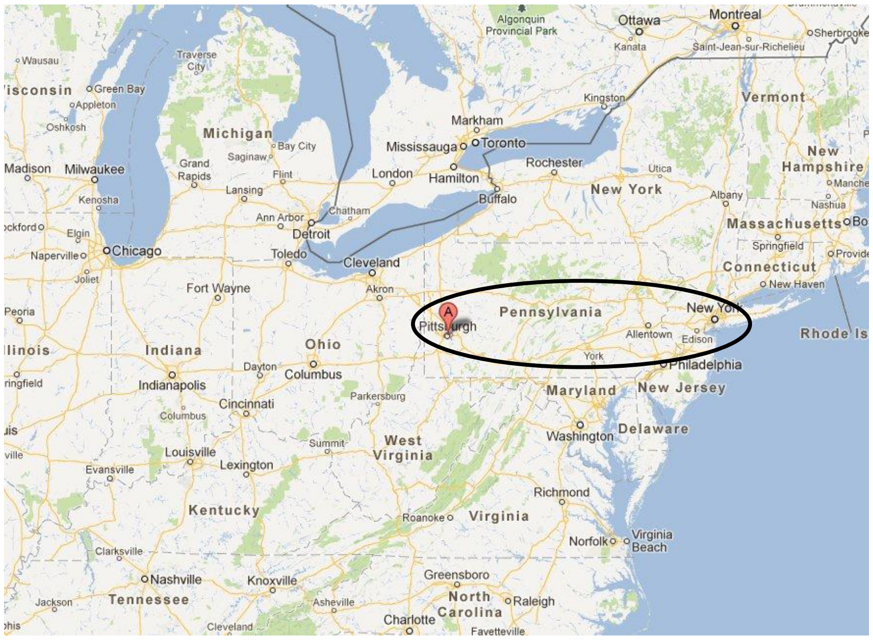

Figure 2: Area explored using A^*.

Solution 5.4 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 5.5 (5%). Give an example of a weighted graph on ≤ 4 vertices with a heuristic that is *admissible but inconsistent*, where the Dijkstra-based A^* algorithm fails to find the shortest path from a single source s to a single target t. Label each vertex with its heuristic value, and clearly mark the vertices s and t.

Solution 5.5 Consider finding the shortest path from s to g in the following graph.

```
        b
   1  /   \ 2
     /     \
    s ----- c ------- g
        4         4
```

Let $\delta(v,g)$ be the actual shortest path from v to g and $h(v)$ the heuristic shortest path cost from v to g.

$$\delta(s,g)=7 \qquad h(s)=6$$
$$\delta(b,g)=6 \qquad h(b)=5$$
$$\delta(c,g)=4 \qquad h(c)=1$$
$$h(g)=0$$

Note that the heuristic defined is admissible because $\delta(v,g) > h(v), \forall v \in V$. But it is not consistent because $h(b)=5 > w(b,c)+h(c)=3$.

Let $(x,(v,d))$ be the entries on the Priority Queue, where d is the current best estimate for the distance from vertex s to vertex v and x is $d+h(v)$. The table below shows the priority queue PQ and the visited set D at the end of each iteration of the A*.

deleteMin	**PQ**	**visited**	**comment**
	(6, (s, 0))		
(6, (s, 0))	(5, (c, 4)), (6, (b, 1))	{s}	add nghrs of s
(5, (c, 4))	(6, (b, 1)), (8, (g, 8))	{s,c}	add ngbr of c
(6, (b, 1))	(4, (c, 3)), (8, (g, 8))	{s,b,c}	update dist to c
(4, (c, 3))	(8, (g, 8))	{a,b,c}	don't revisit c
(8, (g, 8))		{a,b,c}	dequeued goal and return 8

In the this example there are two paths spreading from a single vertex s and joining again at a single vertex c. The path a,b,c is the shortest path to c. But because the heuristic at b is inconsistent, the estimated distance to the goal along the longer path is under estimated relatively more than the shorter path estimate, enough so that their positions in the priority queue are switched. Since both of these paths go through c, when the shorter path to c is dequeued it is discarded because c has already been visited by the longer path.

Task 5.6 (5%). Give an example of a weighted graph on ≤ 4 vertices with heuristic values that are *inadmissible*, where the A^* algorithm fails to find the shortest path from a single source to a single target. Again, clearly label your vertices with heuristic values.

$\boxed{\textbf{Solution 5.6}}$ If the heuristic is inadmissible, the above change may fail to find the shortest path distance to the goal. This failure occurs if the heuristic value of a vertex on the shortest path is a so large that its estimated distance is larger than any estimate distances on longer paths. For example, it will not find the shortest distance from s to g for the following graph.

```
        b
   1  /   \ 2
     /     \
    s ----- g
       4
```

If $h(b)=5$ then the improved algorithm will put $(4,(g,4))$ and $(5,(b,4))$ on the priority queue and will dequeue $(4,(g,4))$ because the inadmissible heuristic on b is too large. Since g is the goal the algorithm terminates, returning the distance 4 instead of 3.

5.3 Specification

Task 5.7 (5%). Define the type `graph` representing a directed graph and write the function

```
val makeGraph :  edge seq -> graph
```

which takes a sequence of edges representing a graph and returns the same graph under your `graph` type. Each edge is represented as a triple (u, v, w) representing a directed edge from u to v with weight w. You may assume that all weights are non-negative. For full credit, `makeGraph` should have $O(|E| \log |V|)$ work and $O(\log^2 |V|)$ span.

Task 5.8 (30%). You will implement the function

```
val findPath : heuristic -> graph -> (set * set)
                -> ((vertex * real) option * int)
```

which augments Dijkstra's Algorithm to accept the following arguments:

1. An A^* heuristic, h, assuming that h is both admissible and consistent.
2. Multiple source vertices, $S \subseteq V$.
3. Multiple target vertices, $T \subseteq V$. If multiple sources and destinations are given, your algorithm should return the shortest path distance between any $s \in S$ and any $t \in T$ (a shortest $S - T$ path).

Specifically, `findPath` $h\ G\ (S, T)$ should evaluate to a pair of values consisting of:

1. `SOME (v,d)` if the shortest $S - T$ path in G ends at vertex $v \in T$ with distance d, or `NONE` if no such path exists. If there are multiple shortest paths, you may return any one.
2. A count of the number of vertices visited, to give you a sense of the efficiency of your search given a certain heuristic. Your count should include both source and destination vertices (given that a path was found), but count no vertex more than once.

You may use the the code for Dijkstra's Algorithm in `MkTableDijkstra.sml` as a starting reference. Our solution is for `findPath` is around 25 lines long.

5.4 Testing

As for Bridges, you will not need to submit test cases for A^*, but you should test your code locally. In `abridged.sml`, we have provided some convenience structure and functor definitions:

```
functor MkAStar(structure Vtx : HASHKEY) : ASTAR
structure IntAStar : ASTAR
structure StringAStar : ASTAR
```

For example, you should use `IntAStar.makeGraph` and `IntAStar.findPath` to define graphs on integer-labeled vertices and perform searches in them.

Again, passing the Autolab testing suite is a large part of your grade for this assignment, so you should submit early to ensure that you pass all of our test cases.