Lab 6 - *AbridgedLab*

Due: *Fri, Mar 21st, 2014 @ 11:59 pm*

Parallel and Sequential Data Structures and Algorithms 15-210 (Spring '14)

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 lab, you will solve **two** coding problems, A* Search and Bridges, instead of the usual one. Note that you have a little longer than usual. Take advantage of that and start early to avoid *misery*.

2 Files

After downloading the assignment tarball from Autolab, extract the files by running:

```
tar -xvf abridgedlab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones handed in by the submission script.

1. `Makefile`
2. `support/MkTableDijkstra.sml`
3. * `MkBridges.sml`
4. * `MkAStarCore.sml`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment.

3 Submission

To submit your assignment to Autolab, open a terminal, `cd` to the `abridgedlab` folder, and run:

```
make
```

Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the "*Handin your work*" link.

4 Bridges

Since you are a TA for Parallel Cats and Data Structures, Meow Zoidberg of Facelook, Inc. decides to interview you for a summer internship, where you get the chance to play around with the in-house version of Graph Search. You exclaim: "Look, the social graph of all users from Carnegie Meow-llon is connected! 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 Implementation

Let $G = (V, E)$ be some **unweighted, undirected**, simple (no self-loops; at most one edge between any two vertices) 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%). In `MkBridges.sml`, define the type `ugraph` representing an undirected graph and implement the function

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

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 and 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 follow this convention, and that they all also have at least one neighbor.

Task 4.2 (30%). In `MkBridges.sml`, implement the function

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

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.

Hint: you will want to make use of DFS numberings to solve this problem. Think about how we can use vertex numbers to determine if an edge is a bridge.

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 `Tester.sml`, we have provided some structure definitions for your testing

convenience. In particular, you should make use of the `Tester.Bridges` structure, which contains the functions you have written. Note that this testing structure is different from previous labs, as we do not provide functions which explicitly test your code against a reference solution. Here's a good time to put those *well-trained testing chops* to work!

5 Paths "R" Us

You've decided that Facelook, Inc. isn't your ideal company. Instead, this summer you'll be joining a YC (Y Concatenator) 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, this method 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 (i.e. insertions and updates of shortest path lengths) up to the point of failure. Explain why the algorithm has failed and what it should have done instead.

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

5.2 The A* Heuristic

Dijkstra's Algorithm searches evenly in all directions from a source vertex, but 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**[1] 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. A heuristic h is *consistent* if for every edge $e = (u, v) \in E$ (a directed edge from u to v), $h(u) \leq h(v) + w(e)$.

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

[1] There may be times when you choose to use an inadmissible heuristic to gain a faster search, if not an optimal path (e.g. 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.

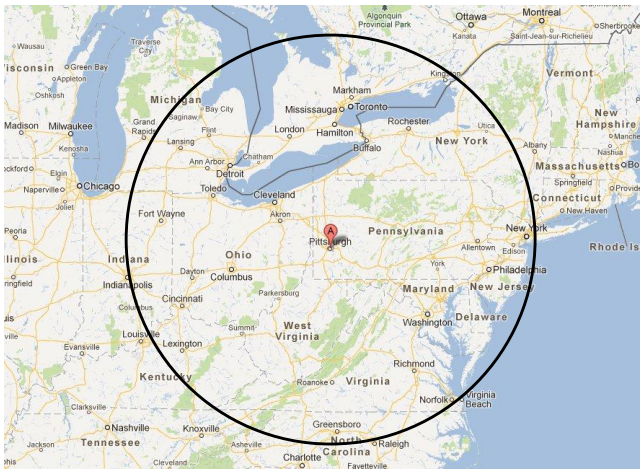

Figure 1: Area explored using Dijkstra.

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.

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

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. In 2-3 clear sentences, explain why the shortest path is not found (e.g. when does the algorithm fail, and what exactly it does wrong).

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 and explain why the shortest path is not found.

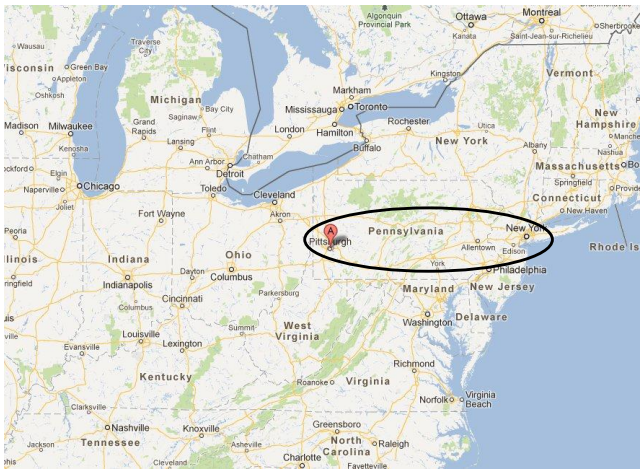

Figure 2: Area explored using A*.

5.3 Implementation

Task 5.7 (5%). In `MkAStarCore.sml`, implement the function

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

which takes a sequence of edges representing a graph and returns the same graph conforming to the provided `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%). In `MkAStarCore.sml`, implement the function

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

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

1. An admissible and consistent A* heuristic h
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 $t \in T$ (i.e. a shortest $S - T$ path).

Specifically, `findPath h G (S, T)` evaluates to `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. The asymptotic complexity of your algorithm should not exceed that of Dijkstra's algorithm as discussed in lecture.

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

For extra fun (not required for this lab, but a very instructive exercise), try proving that the A* algorithm is correct. In other words, prove that, given an admissible and consistent heuristic function, when the algorithm visits a vertex t in the target set, it has indeed found the shortest path to t.

5.4 Testing

As for Bridges, you will not need to submit test cases for A*, but you should test your code locally. In `Tester.sml`, we have provided some convenience structures:

```
structure IntAStar : ASTAR
structure StringAStar : ASTAR
```

For example, you should use `Tester.IntAStar.makeGraph` and `Tester.IntAStar.findPath` to define graphs on integer-labeled vertices and perform searches in them. Again, feel free to further modify `Tester.sml` to make it easier to test your code.

Good luck!