Homework 5

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

Version 1.0 ***due:*** *Mon, Oct 7th @ 11:59pm*

1 Introduction

In this assignment, you will implement an interface for finding shortest paths in unweighted graphs. Shortest paths are used all over the place, from finding the best route to the nearest Starbucks to numerous less-obvious applications. You will then apply your solution to the thesaurus problem, which is: given any two words, find all of the shortest synonym paths between them in a given thesaurus. Some of these paths are quite unexpected! :-)

2 Files

After downloading the assignment tarball from Autolab, extract the files from it by running `tar -xvf thesauruslab-handout.tgz` from a terminal window. In addition to your standard `Makefile`, `bobbfile.py`, `config.py`, and `lib/`, you should see the following files:

1. `thesauruslab.pdf`
2. `input/`
3. `support/`
4. * `MkAllShortestPaths.sml`
5. * `MkThesaurusASP.sml`
6. * `Tests.sml`

You should only modify the last 3 files, denoted by *. Notice that there is no written part to this lab. :-)

3 Submission

To submit your assignment: open a terminal, `cd` to the `thesauruslab-handout` folder, and run `make`. Then visit Autolab to check that you got the autograde results that you expected.

If you prefer to submit manually, run `make package` and upload the generated `handin.tgz` file.

4 Unweighted Shortest Paths

The first part of this assignment is to implement a general-purpose interface for finding shortest paths. This will be applied to the thesaurus path problem in the next section, but could also be applied to other problems. Your interface should work on directed graphs; to represent an undirected graph you can just include an edge in each direction. Your job is to implement the interface given in `ALL_SHORTEST_PATHS.sig`. Before you begin, carefully read through the specifications for each function there.

4.1 Specification

4.1.1 Graph Construction

For these tasks, you may assume that you will be working with *directed*, *simple*, *connected* graphs (*directed, simple*: no self-loops and no more than one directed edge in each direction between any two vertices; *connected*: each graph is comprised of a single component). You may also assume that the graph has at least 1 node.

Task 4.1 (2%). In `MkAllShortestPaths.sml`, define the type `graph` that would allow you to implement the following functions within the required cost bounds. **Leave a brief comment explaining why you chose the representation that you did.**

Task 4.2 (8%). Implement the function

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

which generates a graph based on an input sequence E of directed edges. The number of vertices in the the resulting graph is equal to the number of vertex labels in the edge sequence. For full credit, `makeGraph` must have $O(|E|\log|E|)$ work and $O(\log^2|E|)$ span.

4.1.2 Graph Analysis

Task 4.3 (6%). Implement the functions

```
numEdges :  graph -> int
numVertices :  graph -> int
```

which return the number of directed edges and the number of unique vertices in the graph, respectively.

Task 4.4 (6%). Implement the function

```
outNeighbors :  graph -> vertex -> vertex seq
```

which returns a sequence V_{out} containing all out neighbors of the input vertex. In other words, given a graph $G = (V, E)$, `outNeighbors` $G\ v$ contains all w s.t. $(v, w) \in E$. If the input vertex is not in the graph, `outNeighbors` returns an empty sequence. For full credit, `outNeighbors` must have $O(|V_{\text{out}}| + \log|V|)$ work and $O(\log|V|)$ span, where V is the set of vertices in the graph.

4.1.3 All Shortest Paths Preprocessing

Task 4.5 (2%). In `MkAllShortestPaths.sml`, define the type `asp` that would allow you to implement the following functions within the required cost bounds. **Leave a brief comment explaining why you chose the representation that you did.**

Task 4.6 (23%). Implement the function

```
makeASP : graph -> vertex -> asp
```

to generate an `asp` which contains information about all of the shortest paths from the input vertex v to all other reachable vertices. If v is not in the graph, the resulting `asp` will be empty. Given a graph $G = (V, E)$, `makeASP` G v must have $O(|E| \log |V|)$ work and $O(D \log^2 |V|)$ span, where D is the longest shortest path (i.e., the shortest distance to the vertex that is the farthest from v).

4.1.4 All Shortest Paths Reporting

Task 4.7 (15%). Implement the function

```
report :  asp -> vertex -> vertex seq seq
```

which, given an `asp` for a source vertex u, returns all shortest paths from u to the input vertex v as a sequence of paths (each path is a sequence of vertices). If no such path exists, `report asp` v evaluates to the empty sequence. For full credit, `report` must have $O(|P||L| \log |V|)$ work and span, where V is the set of vertices in the graph, P is the number of shortest paths from u to v, and L is the length of the shortest path from u to v.

Task 4.8 (6%). Test your `ALL_SHORTEST_PATHS` implementation in the file `Tests.sml` by adding test cases to the appropriate lists (see the file for reference – there are existing test cases to guide you).

The following functions are defined to run your implementation of functions of `ALL_SHORTEST_PATHS` against your test cases in `Tests.sml`.

```
testNumEdges ()
testNumVertices ()
testOutNeighbors ()
testReport ()
```

5 Thesaurus Paths

Now that you have a working implementation for finding all shortest paths from a vertex in an unweighted graph, you will use it to solve the Thesaurus problem. You will implement `THESAURUS` in the functor `ThesaurusASP` in `ThesaurusASP.sml`. We have provided you with some utility functions to read and parse from input thesaurus files in `ThesaurusUtils.sml`.

5.1 Specification

5.1.1 Thesaurus Construction

Task 5.1 (10%). Implement the function

```
make :  (string * string seq) seq -> thesaurus
```

which generates a thesaurus given an input sequence of pairs `(w,S)` such that each word `w` is paired with its sequence of synonyms `S`. You must define the type `thesaurus` yourself.

5.1.2 Thesaurus Lookup

Task 5.2 (6%). Implement the functions

```
numWords :  thesaurus -> int
synonyms :  thesaurus -> string -> string seq
```

where `numWords` counts the number of distinct words in the thesaurus while `synonyms` returns a sequence containing the synonyms of the input word in the thesaurus. `synonyms` returns an empty sequence if the input word is not in the thesaurus.

5.1.3 Thesaurus All Shortest Paths

Task 5.3 (10%). Implement the function

```
query :  thesaurus -> string -> string -> string seq seq
```

such that `query th w1 w2` returns all shortest path from `w1` to `w2` as a sequence of strings with `w1` first and `w2` last. If no such path exists, `query` returns the empty sequence. **For full credit, your function `query` must be *staged*.** For example:

```
val earthlyConnection = MyThesaurus.query thesaurus "EARTHLY"
```

should generate the `thesaurus` value with cost proportional to `makeASP`, and then

```
val poisonousPaths = earthlyConnection "POISON"
```

should find the paths with cost proportional to `report`.

Invariably, a good number of students each semester fail to stage `query` appropriately. Don't let this happen to you! Staging is not automatic in SML; ask your TA if you are unsure about SML evaluation semantics or how a properly staged function is implemented.

5.2 Testing

Task 5.4 (6%). Test your `THESAURUS` implementation in the file `Tests.sml` as before.

The following functions are defined to run your implementation of functions of `ALL_SHORTEST_PATHS` against your test cases in `Tests.sml`.

```
testNumWords ()
testSynonyms ()
testQuery ()
```

Note that `testQuery` merely prints out the shortest paths your code produces for the given test cases. We do not test against a reference solution, as our non-costbound-meeting solution is too slow. Ask on Piazza to see if you are producing correct results.

The thesaurus used is defined in `input/thesaurus.txt` where each line is an entry associating the first word in the line with the rest of the words in the line.

As reference, our implementation returned only find one path of length 10 from “CLEAR” to “VAGUE” as well as from “LOGICAL” to “ILLOGICAL”. However, there are two length 8 paths from “GOOD” to “BAD”

Don’t try “EARTHLY” to “POISON”.