`tag` — Version 1.0 ***due:*** *Mon, Oct 1st @ 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 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.

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>/assn4/`

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

`/afs/andrew.cmu.edu/course/15/210/bin/check/04/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 `hw04.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

```
AllShortestPaths.sml
AllShortestPathsTest.sml
ThesaurusASP.sml
ThesaurusASPTest.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 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 in the `ALL_SHORTEST_PATHS` signature. You will need to come up with your own representations for the `graph` and `asp` types. **You must comment your code and explain why you chose the representations that you did.** Reading the cost specifications beforehand should help you make an informed decision. You may assume that you will be working with simple graphs (i.e., there will be no self-loops and no more than one directed edge in each direction between any two vertices).

2.1 Specification

2.1.1 Graph Construction

Task 2.1 (6%). 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.

2.1.2 Graph Analysis

Task 2.2 (4%). 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 2.3 (4%). 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.

2.1.3 All Shortest Paths Preprocessing

Task 2.4 (14%). 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. You must define the type `asp` yourself. 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).

2.1.4 All Shortest Paths Reporting

Task 2.5 (8%). 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 longest shortest path from u to v.

Task 2.6 (5%). Briefly give a reasonable worst case bound (in Big-O) for $|P|$, that is, the number of shortest paths in the previous section. Give an example graph where this worst case scenario occurs.

Task 2.7 (5%). Test your `ALL_SHORTEST_PATHS` implementation in the file `AllShortestPathsTest.sml`.

3 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`.

3.1 Specification

3.1.1 Thesaurus Construction

Task 3.1 (4%). 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.

3.1.2 Thesaurus Lookup

Task 3.2 (2%). 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.

3.1.3 Thesaurus All Shortest Paths

Task 3.3 (8%). 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.

3.2 Testing

Task 3.4 (5%). Test your implementation of `THESAURUS` in the file `ThesaurusASPTest.sml`. We have provided you with `input/thesaurus.txt` to test your implementation of the thesaurus problem. You should use the helper functions in `ThesaurusUtils` to parse input files.

The file is formatted such that each line is a word followed by its synonyms, separated by spaces. You should use the `parseString` helper in the functor `ThesaurusUtils`, which converts thesaurus strings into a sequence of pairs `(w,S)` such that each word `w` is paired with its sequence of synonyms `S`.

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".