# Homework 4

15-210: Parallel and Sequential Data Structures and Algorithms (Spring'12)

`tag` — Version 1.0

***due:** tue, feb 21 @ 11:59pm*

## 1 Introduction

In this "mini" assignment, you will generate an interface for finding shortest paths in unweighted graphs. Computing shortest paths in graphs is 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 the shortest synonym path between them in a thesaurus. Some of these paths are quite unexpected.

### 1.1 Submission

This assignment is distributed in a number of files in our `git` repository. Instructions on how to access that repository can be found at `http://www.cs.cmu.edu/~15210/resources/git.pdf`.

This assignment requires that you submit both code and written answers.

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.

For the programming part, the only files you're handing in are

`table_shortest_paths.sml` and `spthesaurus.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 Naming Modules

The questions that follow ask you to organize your solutions in a number of modules. Your modules must be named exactly as stated in the handout. Correct code inside an incorrectly named structure, or a structure that does not ascribe to the specified signatures, *will not receive any credit*.

You may not modify any of the signatures or other library code that we give you. We will test your code against the signatures and libraries that we hand out, so if you modify the signatures your code will not compile and you will not receive credit.

### 1.3 The SML/NJ Build System

This assignment includes a substantial amount of library code spread over several files. The compilation of this is orchestrated by CM through the file `sources.cm`. Instructions on how to use CM can be found at on the website at `http://www.cs.cmu.edu/~15210/resources/cm.pdf`.

### 1.4 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. **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.
2. **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.
3. **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 generate 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 `SHORTEST_PATHS.sig`.

Before you begin, carefully read through the specifications for each function in the `SHORTEST_PATHS` signature. You will need to come up with your own representations for the `'a graph` and `'a spt` types. 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

```
make_graph :  'a edge seq -> 'a 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 unique vertex labels in the edge sequence. For full credit, `make_graph` 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

```
num_edges :  'a graph -> int
num_vertices :  'a 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

```
out_edges :  'a graph -> vertex -> 'a edge seq
```

which returns a sequence $E_{\text{out}}$ containing all directed edges which have the input vertex as their source. If the input vertex is not in the graph, `out_edges` returns an empty sequence. For full credit, `out_edges` must have $O(|E_{\text{out}}| + \log|V|)$ work and $O(\log|V|)$ span, where $V$ is the set of vertices in the graph.

### 2.1.3 Shortest Path Preprocessing

**Task 2.4** (14%). Implement the function

```
make_tree :  'a graph -> vertex -> 'a spt
```

which generates a shortest path tree from the input vertex $v$ to all other reachable vertices. If $v$ is not in the graph, the resulting tree will be empty. For full credit, `make_tree` must have $O(|E|\log|V|)$ work and $O(D(T)\log^2|V|)$ span, where $E$ and $V$ are the sets of edges and vertices in the graph, respectively, and $D(T)$ is the depth of the shortest path tree (i.e., the greatest to any vertex from the source).

You will surely use some form of breadth-first search (BFS) to do this problem. Note, however, that you will have to extend what we covered in class to deal with storing values on the edges (the `'a` type), and to deal with returning a BFS tree so that you can search for paths.

### 2.1.4 Shortest Path Reporting

**Task 2.5** (8%). Implement the function

```
report :  'a spt -> vertex -> 'a edge seq option
```

which, given a shortest path tree from a source vertex $u$, returns the shortest path from $u$ to the input vertex $v$ as a sequence of edges $P = \langle(u, p_1, e_1), (p_1, p_2, e_2), \ldots, (p_{k-1}, v, e_k)\rangle$ where each edge was in the input graph. If no such path exists, `report spt v` evaluates to `NONE`. For full credit, `report` must have $O(|P|\log|V|)$ work and span, where $V$ is the set of vertices in the graph.

**Task 2.6** (0%). For extra credit, improve the time of `report spt v` to $O(|P| + \log|V|)$ work and span. If you choose to complete this task, you only need to hand in this solution. You will also need to indicate in `hw04.pdf` that you did it and explain why it matches these bounds.

## 3 Thesaurus Paths

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

### 3.1 Specification

#### 3.1.1 Thesaurus Construction

**Task 3.1** (6%). 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`.

#### 3.1.2 Thesaurus Lookup

**Task 3.2** (2%). Implement the functions

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

where `num_words` 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 Shortest Paths

**Task 3.3** (6%). Implement the function

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

such that `query T w1 w2` returns the shortest path from `w1` to `w2` as a sequence of strings with `w1` first and `w2` last. If no such path exists, `query` returns `NONE`.

For full credit, your function `query` must be *staged* so that when you apply the second argument (`w1`), it immediately does all the work generating the shortest path tree. Then, when you apply the third argument (`w2`) it only needs to report the path. For example:

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

will generate the tree with cost proportional to `make_tree`, and then

```
val poisonous_path = earthly_connection "POISON"
```

only needs to find the path with cost proportional to `report`.

## 3.2 Testing

We have provided you with `input/thesaurus.txt` to test your implementation of the thesaurus problem. Again, 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 `parse_string` 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`.

A correct implementation should return a sequence of length 10 from "CLEAR" to "VAGUE" as well as from "LOGICAL" to "ILLOGICAL". There's really no good reason to try "EARTHLY" to "POISON".