1 Introduction

In this assignment, you will implement solutions to both the *unweighted* and *weighted* variety of the *all single-source shortest paths* problem. Given a graph and a source, you will need to compute every shortest path to every other vertex in the graph. These solutions can then be used to find the shortest path between two words using their synonyms.

There is no written portion to this lab. Have fun coding!

2 Files

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

```
tar -xvf shortlab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. The files denoted by * will be submitted for grading.

1. * `MkUnweightedASP.sml`
2. * `MkWeightedASP.sml`
3. `Sandbox.sml`
4. `Tests.sml`

3 Submission

To submit your assignment to Autolab, generate a handin archive `handin.tgz` with the command

```
make package
```

then submit your `handin.tgz` file on Diderot.

Note that your code may not be graded until after the due date. It is your responsibility to test your code thoroughly before submitting.

Before submission, always, *always* verify that the files you will submit are indeed the versions that you intended to submit. Be very careful to NOT submit blank files, or old versions of the files. **We will not entertain "*Oh no, I submitted the wrong file!*" emails or posts on Diderot after the due date**. We will adopt a *zero-tolerance* approach to this, and you will receive the grade of your most recent submission (which may be subject to late submission penalties!) *You have been warned*.

The questions below ask you to organize your solutions in a number of modules written almost from scratch. 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.

This assignment requires the use of library code spread over several files. The compilation of this is orchestrated by the compilation manager through several `.cm` files. Instructions on how to use CM can be found in the file `cm.pdf` under Resources on Diderot.

Style grading for this assignment will be evaluated on a 5 point scale: -2 through +2. Style points will be added to your final grade from the rest of your homework. **You should review the full style guide available on the Diderot under Resources.**

You should submit your solutions on or before the deadline on the top of the first page. Failure to do so will cause substantial deductions of your grade. Late submissions of up to a day will scale your homework grade by 700% (so if you were to get 100% as your grade, you will now get 70%). Late submissions of up to 2 days will scale your homework grade by 50%. Beyond 2 days, you will receive 0 as your homework grade. This scaling will be handled outside Autolab.

4 Unweighted Shortest Paths

Suppose you are interested in finding how closely related two words are. In order to find the similarity between two words, you decide to find the shortest path of synonyms relating the first word to the second word. In particular, you are only interested in finding the relationship between your favorite word and all other words so you preprocess all possible shortest paths from you favorite word to the remaining words you are interested in.

4.1 Implementation

For cost bounds, assume

- the `ArraySequence` implementation of `SEQUENCE`, and
- the `MkTreapTable` implementation of `TABLE`.

The graphs in this section are *directed*, *simple*, and *unweighted*.

Task 4.1 (75 pts). In `MkUnweightedASP.sml`, implement the following functions. Note that you will first need to decide on a representation for the types `graph` and `asp`. Choose these carefully so that you can meet the given cost bounds.

```
val makeGraph : edge Seq.t → graph
```

Given a sequence of directed edges E, (`makeGraph` E) should produce the graph induced by E. Your implementation should require $O(|E| \log |E|)$ work and $O(\log^2 |E|)$ span.

```
val makeASP : graph → vertex → asp
```

Given a graph G, (`makeASP` G s) should return an `asp` containing information about all shortest paths in G beginning at the source s. Assuming G represents the graph (V, E) and that all vertices are reachable from the source, your implementation should require $O(|E| \log |V|)$ work and $O(D \log^2 |V|)$ span, where D is the diameter of the graph.

```
val report : asp → vertex → vertex Seq.t Seq.t
```

Given an `asp` A which was constructed for the graph (V, E) with source s, (`report` A t) should return all shortest paths from s to t. Each path is a sequence of vertices beginning at s and ending at t. Your output doesn't have to be ordered in any particular way. Your implementation should require $O(k\ell \log |V|)$ work and span where

- k is the number of shortest paths between s and t, and
- ℓ is the length of the shortest path between s and t.

5 Weighted Shortest Paths

When testing your system, you realize that some of the paths generated contain obscure synonyms. To make your system more robust, you decide that you want to prioritize the use of more relevant synonyms. Luckily, your dictionary already keeps track of the proximity between two synonyms, so you use this additional information to generate the most likely list of synonyms connecting your two chosen words.

5.1 Implementation

This section is identical to the previous except that the input graph is now *weighted*, and the desired cost bounds of `makeASP` are now sequential. Note that if you use a similar `asp` type as in the unweighted section, your `report` function will be similar too. Feel free to copy-paste, but make sure you do not have any correctness or bounds mistakes! (Otherwise, you'll be penalized in both sections for the same error...)

For cost bounds, assume

- the `ArraySequence` implementation of `SEQUENCE`,
- the `MkTreapTable` implementation of `TABLE`, and
- the `MkLeftistHeapPQ` implementation of `PQ`.

The graphs in this section are *directed*, *simple*, and *weighted*. You may assume the weights are strictly positive (no edge will have an edge weight of 0).

Task 5.1 (50 pts). In `MkWeightedASP.sml`, decide on a representation for the types `graph` and `asp`, then implement the following functions.

```
val makeGraph : edge Seq.t → graph
```

Given a sequence of weighted directed edges E, (`makeGraph` E) should produce the graph induced by E. Your implementation should require $O(|E| \log |E|)$ work and $O(\log^2 |E|)$ span.

```
val makeASP : graph → vertex → asp
```

Given a graph G, (`makeASP` G s) should return an `asp` containing information about all shortest paths in G beginning at the source s. Assuming G represents the graph (V, E) and that all vertices are reachable from the source, your implementation should require $O(|E| \log |V|)$ work and span.

```
val report : asp → vertex → vertex Seq.t Seq.t
```

Given an `asp` A which was constructed for the graph (V, E) with source s, (`report` A t) should return all shortest paths from s to t. Each path is a sequence of vertices beginning at s and ending at t. Your output doesn't have to be ordered in any particular way. Your implementation should require $O(k\ell \log |V|)$ work and span where

- k is the number of shortest paths between s and t, and
- ℓ is the maximum *number of edges* among any of the shortest paths. Specifically, if $P =$ (`report` A t), then

$$\ell = \max_{p \in P} |p|$$

6 Testing

There are three ways to test your code.

1. In `Sandbox.sml`, write whatever testing code you'd like. You can then access the sandbox at the REPL:

```
- CM.make "sandbox.cm";
...
- open Sandbox;
```

2. You can load `thesaurus.cm` and print the shortest paths from one word to another, as follows:

```
- CM.make "thesaurus.cm"; open Thesaurus;
...
- val goodPaths = unweightedQuery "good";
val goodPaths = fn : string -> unit
- goodPaths "evil";
UNWEIGHTED PATHS FROM good TO evil:
good, value, appreciate, prize, prey, spoil, damage, harm, evil
good, value, rate, tempo, beat, throb, ache, hurt, evil
good, value, appreciate, prize, prey, spoil, damage, hurt, evil
good, right, accurate, exact, extort, wring, pain, hurt, evil
good, value, treasure, wealth, fortune, chance, accident, mishap, evil
good, worth, merit, reward, tip, upset, bother, nuisance, evil
val it = () : unit
```

3. In `Tests.sml`, add test cases according to the instructions given. We have provided some basic test cases that you may use as examples. Then run the autograder:

```
- CM.make "autograder.cm";
...
- Autograder.run ();
```