

# PPA-F25 hw6

## Logistics

This homework consists of 100 points total, with points for individual tasks as indicated below. There are 2 coding tasks.

Please submit your work on Gradescope.

For the coding tasks, you will edit and submit `IntersectionCount.sml` and `TriangleCount.sml`.

**Due Date:** This submission is due at **5:00pm EDT on Monday, Oct 20**. Course policy on late submissions is available on the course website. (<https://cs.nyu.edu/~shw8119/courses/f25/3033-121-ppa/>)

## Setup

- Follow the instructions on the course website to access one of the **crunchy** compute servers.
- Install MaPLe. For the **crunchy** servers, you can:
  - Download <https://cs.nyu.edu/~shw8119/courses/f25/3033-121-ppa/resources/mpl-845f76f.tgz> and copy it to the server
  - Unpack by running `tar xzf mpl-845f76f.tgz`. This will create a directory `mpl-845f76f/`
  - The compiler, `mpl`, is located at `mpl-845f76f/bin/mpl`
  - Add `mpl-845f76f/bin` to your `PATH` so that you can access it easily: `export PATH="$(pwd -P)/mpl-845f76f/bin:$PATH"`. We recommend updating your `~/.bashrc` file or other configuration file as appropriate.

## Intersection Reminds Me of Merging

This homework assignment is primarily about graphs, but before we get there, we need to take a little detour. In this part of the homework, you will develop an important subroutine which will be useful in the next part.

We previously saw an efficient parallel algorithm for merging two sorted sequences. In this task, you will implement a slight variation on that algorithm to instead *count the number of elements that are the same* across two sorted sequences, i.e., to count the size of their intersection.

**Task 1 (40 points):** In `IntersectionCount.sml`, implement the function `intersection_count cmp (s, t)` which counts the number of elements that appear in both `s` and `t`. You may assume that the inputs are sorted. You may also assume that, within each input, there are no duplicates.

For example, on inputs  $s = \langle 1, 2, 4, 5, 42 \rangle$  and  $t = \langle 2, 3, 4, 5, 10 \rangle$ , `intersection_count Int.compare (s, t)` should return 3, because there are 3 elements in common (2, 4, 5).

Your implementation should have polylogarithmic span, and should meet the following work recurrence for inputs of size  $\ell$  and  $r$ :

$$W(\ell, r) \leq \begin{cases} O(1), & \text{if either } \ell = 0 \text{ or } r = 0 \\ W(\ell/2, k) + W(\ell/2, r - k) + O(\log r), & \text{otherwise, } \forall k : 0 \leq k \leq r \end{cases}$$

This recurrence should remind you of the merging algorithm. Your implementation will probably be pretty similar.

## Understanding the Cost of Intersection Counting

Note that the above work recurrence can be bounded by  $W(\ell, r) \leq O(m \log(1 + n/m))$ , where  $m = \min(\ell, r)$  and  $n = \max(\ell, r)$ . (You don't need to prove this. The proof is intricate.)

This bound,  $O(m \log(1 + n/m))$ , is quite interesting. For some intuition, we recommend thinking about two extreme cases:

- Suppose one input is significantly smaller than the other, i.e.,  $m \ll n$ . As  $m$  gets smaller,  $O(m \log(1 + n/m))$  approaches  $O(\log n)$ . In other words, intersection counting is more efficient when one of the inputs is much smaller than the other.
- Suppose the two inputs are similar sizes, i.e.,  $m \approx n$ . Then  $O(m \log(1 + n/m))$  simplifies to approximately  $O(m) = O(n)$ . In other words, intersection counting is never more expensive than linear.

More generally,  $O(m \log(1 + n/m))$  sits somewhere between  $O(\log n)$  and  $O(n)$ , and the intersection algorithm seamlessly transitions between these depending on the input sizes.

## Triangle Counting

In a directed graph, there are two possible kinds of “triangles”. We’ll call them **pointed** and **cyclic**.

- In a **pointed** triangle, there are two edges pointing at the same vertex. Specifically, a pointed triangle is a group of three vertices  $u, v, w$  with three edges:  $u \rightarrow v$ ,  $v \rightarrow w$ , and  $u \rightarrow w$ .
- In a **cyclic** triangle, every edge points at a different vertex. Specifically, a cyclic triangle is a group of three vertices  $u, v, w$  with three edges:  $u \rightarrow v$ ,  $v \rightarrow w$ , and  $w \rightarrow u$ .

By counting these triangles (for example, in a social network graph), we can get a sense for how “tightly knit” a graph is.

**Task 2 (60 points):** In `TriangleCount.sml`, implement the function `triangle_count(g)` which takes a graph `g` and returns a tuple  $(t_p, t_c)$ , where  $t_p$  is the number of pointed triangles, and  $t_c$  is the number of cyclic triangles. Your solution should have  $O(N + M\delta)$  work and  $O(\text{polylog}(N))$  span, where  $N$  is the number of vertices,  $M$  is the number of edges, and  $\delta$  is the maximum degree (in-degree or out-degree) of any vertex.

See the file `lib-local/Graph.sml` for the graph interface. **You may assume that the graph has been preprocessed to store neighbors in sorted order.** That is, `Graph.out_neighbors(g, v)` always returns a sorted sequence, in  $O(1)$  work and span. (Similarly for `in_neighbors(...)`.) You may also assume the neighbor sequences are free of duplicates, and that there are no self-loops (i.e., no edges where a vertex points at itself).

We recommend using the `intersection_count` function. Note that Task 2 will be graded independently of Task 1.

Hint: be careful with symmetries! How many times do you count each triangle?

## Unit Testing

You can test your implementations as follows. Feel free to add more tests to the top of `test.sml`.

```
$ make test
$ ./test
```

There are test graphs inside of `test-graph/...`. Each file is a list of directed edges in human readable format. This is the same format that is commonly used in the [SNAP dataset](#).

## Download and play with some real graphs!

The [SNAP dataset](#) has a number of freely available graphs. Here are a few interesting ones in particular:

- <https://snap.stanford.edu/data/cit-Patents.html> A patent citation network, where each directed edge  $p_1 \rightarrow p_2$  indicates that a patent  $p_1$  cited some other patent  $p_2$ .

- <https://snap.stanford.edu/data/soc-LiveJournal1.html> A social network graph, taken from LiveJournal.
- <https://snap.stanford.edu/data/com-Orkut.html> A social network graph, taken from the short-lived Orkut social network.

You can run these through your implementation as follows:

```
# download the datasets
$ wget https://snap.stanford.edu/data/cit-Patents.txt.gz
$ wget https://snap.stanford.edu/data/bigdata/communities/com-orkut.ungraph.txt.gz
$ wget https://snap.stanford.edu/data/soc-LiveJournal1.txt.gz

# unpack
$ gunzip cit-Patents.txt.gz
$ gunzip com-orkut.ungraph.txt.gz
$ gunzip soc-LiveJournal1.txt.gz

# compile
$ make main

# run
$ ./main @mpl procs 32 -- cit-Patents.txt
Loading graph (if large, this might take a while...)
Warning: graph contains 1 self-loops
Loaded graph in 4.3893s
num vertices 3774768
num edges 16518948
-----
warmup 0.0000
repeat 1
time 1.4619s

average 1.4619s
minimum 1.4619s
maximum 1.4619s
std dev 0.0000s
-----
num pointed triangles 7515027
num cyclic triangles 0
```

For reference, using 32 processors on `crunchy1`, Sam's code takes:

- Approximately ~1.5s on the patent citation network, to count 7515027 pointed triangles (no cyclic triangles).
- Approximately ~90s on the LiveJournal social network graph, to count 924966203 pointed triangles and 238218098 cyclic triangles. (Note that this graph has a significant number of self-loops. Depending on the specifics of how your implementation handles self-loops, you might see different counts.)
- Approximately ~120s on the Orkut social network graph, to count 627584181 pointed triangles (no cyclic triangles).