CSE548, AMS542: Analysis of Algorithms, Fall 2023 **Date:** Nov 28

Homework #4

(Due: Dec 11)

Task 1. [100 Points] Randomized Min-Cuts: Weighted and Faster Unweighted

Consider the randomized min-cut algorithm we saw in the class that returns an *unweighted min-cut* with probability $\geq 1 - \frac{1}{e}$. Let us call the algorithm Rand-Unweighted-Mincut, and it works as follows. Given a connected undirected unweighted multigraph G with n vertices and m edges as input, it runs the following subroutine $\frac{n^2}{2}$ times and returns the smallest cut (smallest in terms of the number of edges) identified by those runs. Each run of the subroutine that starts with the original n-vertex graph and performs a sequence of $n - 2$ edge contractions. Each contraction is performed on an edge chosen uniformly at random from the current set of edges. A contraction step contracts the two endpoints of the given edge into a single vertex and removes all edges between them, but retains all other edges (and thus leading to a multigraph). After $n - 2$ contraction steps only 2 vertices remain, and all edges between those two vertices are returned as a potential min-cut.

(*a*) [**25 Points**] Argue that after an $\mathcal{O}(m + n)$ time preprocessing step at the beginning, each contraction step of the subroutine can be implemented to run in $\mathcal{O}(n)$ time, and thus Rand-Unweighted-Mincut takes $\mathcal{O}\left(m + n^4\right)$ time to return a min-cut with probability $\geq 1 - \frac{1}{e}$. Explain how in every contraction step you would choose an edge uniformly at random from the set of existing edges in $\mathcal{O}(n)$ time.

(*b*) [**30 Points**] Suppose that you are given a weighted undirected graph G as input, where the edge weights are integers chosen from $[1, W]$ for some $W \geq 1$. Let n be the number of vertices and m be the number of edges in G. Now you want to find a *weighted min-cut* of G, i.e., a cut with the smallest total edge weights (which may have more edges than in an unweighted min-cut of G). Argue that this problem can be reduced to an instance of finding an *unweighted min-cut* of a multigraph which can be solved using Rand-Unweighted-Mincut. Let us call this new algorithm Rand-Int-weighted-Mincut. Show that the running time of Rand-Int-weighted-Mincut matches the time Rand-Unweighted-Mincut takes for an n-node m-edge multigraph. Do you still choose contraction edges uniformly at random? If not, at what probability do you choose any given edge? Make sure that in every contraction step you can choose an edge as above in $\mathcal{O}(n)$ time.

(*c*) [**10 Points**] Based on your results from part (*b*) argue that even if the edge weights are arbitrary real numbers in $[1, W]$ a weighted min-cut can still be found with probability at least $1 - \frac{1}{e}$ without asymptotically exceeding the running time of Rand-Int-weighted-Mincut on an n-node m-edge graph with integer edge weights.

There is a deterministic weighted min-cut algorithm that can return a weighted min-cut (with certainty) in $\mathcal{O}\left(n^{\frac{8}{3}}\right)$ worst-case time. So, the randomized algorithms described above run much slower than the deterministic algorithm and also do not always produce a correct solution!

In order to speed up our randomized algorithms, we can use the following hybrid approach. Starting with the n-vertex graph we keep performing random edge contractions until we are able to reduce the number of vertices in the graph to r for some predetermined $r < n$. We then apply the deterministic algorithm on that r-vertex graph to find a min-cut.

For simplicity, the rest of this task will be about RAND-UNWEIGHTED-MINCUT only.

- (d) [**15 Points**] Show that a single run of the hybrid version of RAND-UNWEIGHTED-MINCUT produces a unweighted min-cut with probability at least $\left(\frac{r-1}{n}\right)^2$ in $\mathcal{O}\left(n^2 + r^{\frac{8}{3}}\right)$ time.
- (e) [**10 Points**] Show how the hybrid algorithm from part (d) can be used to produce a unweighted min-cut in $\mathcal{O}\left(\frac{n^4}{r^2} + n^2 r^{\frac{2}{3}}\right)$ time with probability at least $1 - \frac{1}{e}$.
- (f) [**10 Points**] Based on the result from part (e) design a Monte-Carlo algorithm that runs in $o\left(n^{\frac{8}{3}}\right)$ time and can produce a unweighted min-cut w.h.p. in n.

Task 2. [70 Points] Counting Prime Factors in Parallel

Figure 1 shows a not-so-efficient algorithm for counting the number of prime factors for all positive integers up to some given bound $n > 1$.

- (a) [**10 Points**] Give a tight asymptotic bound on the worst-case serial running time of SIEVE-NAÏVE for any given n.
- (b) [**15 Points**] Give a tight asymptotic bound on the worst-case serial running time of COUNT-PRIME-FACTORS for any given n.
- (c) [**20 Points**] Parallelize SIEVE-NAÏVE and analyze its work, span, and parallelism.
- (d) [**25 Points**] Parallelize COUNT-PRIME-FACTORS and analyze its work, span, and parallelism.

```
SIEVE-NAÏVE( $P$, $n$ )
Input: $P$ is a pointer to an array of length $n$ indexed from 1 to $n$.
Output: For each $k \in [1, n]$, $P[k]$ is set to TRUE provided $k$ is prime, otherwise it is set to FALSE.
1. $P[1] \leftarrow$ FALSE
2. for $i \leftarrow 2$ to $n$ do
3.     $P[i] \leftarrow$ TRUE
4. for $i \leftarrow 2$ to $n$ do
5.     $j \leftarrow 2 \times i$
6.     while $j \leq n$ do
7.         $P[j] \leftarrow$ FALSE
8.         $j \leftarrow j + i$
9. return
```

```
COUNT-PRIME-FACTORS( $P$, $Q$, $n$ )
Input: $P$ is a pointer to a pre-initialized array of length $n$, where for each $k \in [1, n]$, $P[k] =$ TRUE provided $k$ is prime, otherwise $P[k] =$ FALSE. Also, $Q$ is a pointer to another array of length $n$, indexed from 1 to $n$.
Output: For each $k \in [1, n]$, $Q[k]$ is set to the number of unique prime factors of $k$.
1. for $i \leftarrow 1$ to $n$ do
2.     $Q[i] \leftarrow 0$
3. for $i \leftarrow 1$ to $n$ do
4.     if $P[i] =$ TRUE then
5.         $j \leftarrow i$
6.         while $j \leq n$ do
7.             $Q[j] \leftarrow Q[j] + 1$
8.             $j \leftarrow j + i$
9. return
```

```
PRIME-FACTORS( $Q$, $n$ )
Input: $Q$ is a pointer to an array of length $n$, indexed from 1 to $n$.
Output: For each $k \in [1, n]$, $Q[k]$ is set to the number of unique prime factors of $k$.
1. allocate a Boolean array $P$ of length $n$
2. SIEVE-NAÏVE( $P$, $n$ )
3. COUNT-PRIME-FACTORS( $P$, $Q$, $n$ )
4. free $P$
5. return
```

Figure 1: [Tasks 2/3] When called with parameters $n \geq 1$ and $Q[1 : n]$, PRIME-FACTORS sets $Q[k]$ to the number of unique prime factors of k for each $k \in [1, n]$.

Task 3. [30 Points] Counting Prime Factors I/O-efficiently

Consider again the not-so-efficient algorithm for counting prime factors given in Figure 1. We will not consider their parallel versions in this task.

(*a*) [**5 Points**] Analyze the I/O complexity of SIEVE-NAÏVE.

(*b*) [**5 Points**] Analyze the I/O complexity of COUNT-PRIME-FACTORS.

Now suppose that I give you an I/O-efficient priority queue that supports INSERT and EXTRACT-MIN operations in $\mathcal{O}\left(\frac{1}{B}\log_M n\right)$ amortized I/Os each, where n is the maximum number of items the priority queue contains during its lifetime, M is the size of the external memory (or cache), and B is the block transfer size (or cache line size).

(*c*) [**10 Points**] Modify SIEVE-NAÏVE to improve its I/O complexity using the given priority queue. What I/O-complexity do you get?

(*d*) [**10 Points**] Repeat part (*c*) for COUNT-PRIME-FACTORS.