Parallel and Concurrent Algorithms (15-852), Spring 2024
Assignment #2 Due: Thursday, February 29

---

Do any two of the first three problems, and then do the fourth problem.

**Problem 1: Parenthesis Matching**

Assume you are given a sequence of different types of left and right parentheses, such as "{}" "[]" "()". A string of these are *matched* if they are properly nested with matching types of parentheses on the left and right. For example "{()[]{[]([])}}" is matched, but "{[)({}]}" is not. If you have used LaTeX, you understand the importance of such proper nesting.

Describe an linear work algorithm for identifying if a string is matched. For a string of length $n$, It can take $n^\alpha$ span for $\alpha > 0$ (e.g. $\alpha = 1/2$). Hint: consider whether sorting can help.

**Problem 2: Low Diameter Decomposition**

As described in class a $(\beta, d)$-decompositions partitions an unweighted, undirected graph into clusters of diameter at most $d$ and with the probability of an edge being between clusters being at most $\beta$. We then covered an algorithm for generating $(\beta, O(\log n/\beta))$-decompositions.

The idea was for every vertex $v$ to pick $\delta_v$ from an exponential distribution with parameter $\beta$. Every vertex would then start a search at time $\delta_{max} - \delta_v$ and "arrive" at vertex $u$ at time $\delta_{max} - \delta_v + d(v, u)$. Here $\delta_{max}$ is $\max_{v \in V} \delta_v$ and $d(u, v)$ is the shortest unweighted path between $u$ and $v$.

We argued that due to the memoryless property of the exponential distribution, given that first vertex arrives at a vertex $u$ at time $t$, that the probability that the next vertex arrives by time $t + a$ was at most the cumulative distribution function for the exponential distribution up to $a$. The cumulative distribution is $1 - e^{\beta a}$, so for $a = 1$ this is $1 - e^{\beta} \leq \beta$.

It is actually possible to show that after the first vertex arrives, the probability that $k$ more arrive within one unit of time is at most $\beta^k$ (see the course notes on Parallel Algorithms).

Use this fact to argue that if we use a $(1/4, O(\log n))$-decomposition, then contract each cluster into a single vertex and remove redundant edges, only $O(n)$ edges will remain in the contracted graph.

## Problem 3: Maximal Matching

In class we described Luby's algorithm for maximal independent sets (MIS). Here we want to generalize it to maximal matching. A matching in a graph $G = (V, E)$ is a subset of the edges $E$ such that no two share an endpoint. A maximal matching is a matching such that no more edges can be added while still being a matching.

Based on the ideas of Luby's MIS algorithm, and the analysis we did in class (also in the course notes), describe an algorithm for Maximal Matching. It must run in $O((m+n)\log n)$ work and polylogarithmic span. Justify these cost bounds.

The algorithm should be simple (hardly any more complicated than Luby's algorithm).

## Problem 4: Nearest hot-spot

This is a programming problem to be implemented using parlaylib. The hot spot problem on an unweighted undirected graph is the following.

You are given an unweighted undirected graph $G = (V, E)$ and a set $U \subset V$ of "marked" vertices. The problem is to return for every vertex the label of the closest marked vertex. If there are multiple with equal distance, you can return any among the ties. If there is no reachable vertex in $U$ you should return $-1$.

For the programming assignment you can assume the vertices are labeled with ids from 0 to $|V| - 1$, and the graph is represented as a sequence of sequences. The $i$-th element of the outer sequence contains a sequence of the neighbor ids for vertex $i$. The set $U$ is simply a sequence of vertex ids for the set.

Your solution must work in $O(|E|+|V|)$ work and $O(D \log |V|)$ span, where $D$ is the diameter of the graph.

I will post a `hotspot.cpp` file which you can use as a driver and to test your code. You need to implement a submit `hotspot.h` file that is compatible with `hotspot.cpp`. In particular it implements a function with the interface:

```
using vertex = int;

parlay::sequence<vertex>
hotspot(const parlay::sequence<parlay::sequence<vertex>>& G,
        const parlay::sequence<vertex>& U);
```

and returns a sequence of length $|V| =$ `G.size()`, where the $i$-th element is the index of vertex $i$ which is closest.

The student with the fastest solution on our 72 core machine (`aware.aladdin.cs.cmu.edu`) when run on the graph `com_orkut_sym.adj` will get extra points.

More info on the setup will be sent via email. You can start with any code you find in the `parlaylib/examples` directory and you can use `parlaylib/examples/helper/ligra_light.h` if you want (yes it could be helpful, especially if you want a fast implementation).