This problem set has a total of nine problems, and was last updated on Thursday August 27. Solutions for these will be accepted until Thursday September 24. Problems marked with (Imp.) are implementational, and should be sent to the instructor via email. For the rest, solutions can be either handed in or presented to the instructor during office hours.

1 (Imp.) Performance of Linked List Traversal

The goal of this exercise is to make clear the limitations of the analyses of algorithms done in this class.

An implementation of the $O(n)$ time prefix sum algorithm discussed at the start of Lecture 1 is given at http://www.cc.gatech.edu/~rpeng/8803SA/code/pointerChase.cc.

It performs a traversal on a linked list of $N = 10^7$ nodes. By modifying the initialization of pointers in the lines between the comments, exhibit two inputs where the running time of the traversal differs by a factor of more than 5 when compiled using $-O3$. The validity of the linked list is checked using the assert statements at the end of the file.

NOTE: These results are very much machine dependent. On my laptop I was able to get a $15 \times$ performance difference. If differences are not evident with $10^7$, you can set $N$ to $10^8$ instead. If that still only gives a mild difference, let me know.

2 List ranking in $O(\log n)$ depth

Consider a random event that succeeds with some constant probability $c$. Show that for any constants $c_1$ and $d$, there exists a constant $c_2$ such that after $c_2 \log n$ such independent events, the probability of having $c_1 \log n$ or more successes is at least $1 - n^{-d}$.

Use this to show that the randomized list ranking algorithm presented in Lecture 1 terminates in $O(\log n)$ rounds with high probability. Recall that each round decreases the size of the list by a factor of $9/10$ with probability at least $1/6$.

3 (Imp.) Tuning Sampling Based List Ranking

Recall the randomized list ranking algorithm from Lecture 1: nodes declare themselves odd or even at random; then even ones with odd predecessors are shrunk into them. Implement a sequential version of this algorithm (you may ignore the summing steps), and run it with $10^7$ initial vertices.
• What is the (average) total work / number of rounds? How does this compare against the deterministic halving algorithm?

• Does changing probabilities of odd/even give better performances?

4 Alternate Tree Decomposition Schemes

Recall low-depth decompositions of trees from lecture 2: they are $O(\log n)$-depth hierarchical decompositions of trees into subtrees.

Exhibit trees where the following algorithms may generate decompositions with depth at least $n^{0.1}$. Your examples should work for infinitely many values of $n$.

1. Pick a random vertex $r$, treat it as the root of the decomposition. Recurse on the subtrees formed by removing this vertex, and combine the decompositions using $r$.

2. Pick the longest path (diameter), decompose it into a balanced binary tree. Repeat on the subtrees formed by removing this path, and attach their decompositions below this binary tree.

5 More Graph Decompositions

Consider the following graph decomposition scheme:

• fix a distance $r$,

• pick a random permutation of the vertices, $\pi(u)$.

• for each vertex $u$, assign it to the vertex $v$ within distance $r$ that’s earliest in the permutation. Aka, assign $u$ to

$$\arg \min_v \{\pi(v) | d(u,v) \leq r \}.$$ 

Give an algorithm for finding such a decomposition on an unweighted undirected graph that runs in $O(m \log^3 n)$ time or faster.

NOTE: this decomposition scheme is used in the optimal $O(\log n)$-stretch tree metric approximation: http://research.microsoft.com/pubs/74347/06-journal.pdf.

6 Dijkstra’s Algorithm and Breadth-First Search

Breadth first search can be viewed as an $O(m)$ time routine for computing single shortest path in a graph where all edges have the same positive weight.
Give an $O(m)$ time algorithm for computing shortest paths in a directed graph where all edges have lengths either $w_1$ or $w_2$ ($w_1, w_2 > 0$).

NOTE: This is a quite general phenomenon: an $O(m \log k)$ time algorithm is possible if there are only $k$ different positive edge weights. Combining this with bucketing also gives that an $O(1)$-approximate shortest path can be computed in $O(m \log \log n)$ time.

7 Heuristic for Low Stretch Spanning Trees

Consider the following heuristic for low stretch spanning trees: pick a random vertex with probability proportional to degree, run shortest path from it with randomized tie breaking, and take the shortest path tree as a candidate tree.

This heuristic does quite well on most real graphs with relatively small edge weight ratios, but we will show that it’s problematic in the worst case. Construct an unweighted graph with $n$ vertices and $m$ edges where the total stretch of any shortest path tree is at least $mn^{0.2}$.

8 (Imp.) Randomized Minimum $s-t$ Cut

Consider adapting the randomized minimum cut algorithm to finding $s-t$ cuts in undirected graphs: at each step, pick a random edge (w.p. proportional to weight) that’s not between $s$ and $t$, contract it, and update edge weights.

Implement this algorithm, and exhibit a graph with between 20 and 50 vertices and integer capacities between 1 and 100 on which this algorithm doesn’t find the minimum cut after $10^5$ iterations. You should output your graph in the form of:

- Line 1: $n$, the number of vertices.
- Line 2 - $n$: line $i + 1$ contains $n - i$ integers. The $j^{th}$ number on line $i + 1$ gives the capacity of the edge between vertices $i$ and $j$.

The vertices are 1-indexed, with $s = 1$ and $t = 2$.

NOTE: I was able to produce an instance with 30 vertices on which the best result produced in $10^7$ iterations is $OPT + 3$. I would be very interested in seeing cases where the output produced is consistently more than $2OPT + 10$.

A code that computes the minimum 0-1 cut as well as the global minimum cut is at http://www.cc.gatech.edu/~rpeng/8803SA/code/mincut.cc. It repeatedly reads graphs from the standard input in the format specified. A file with 100 such graphs is given in http://www.cc.gatech.edu/~rpeng/8803SA/code/cut.in. After compiling mincut.cc, you may run it on this file using the command ./mincut<cut.in.
9 Structure of Cuts

Let $c(u, v)$ denote the size of the minimum cut separating $u$ and $v$ in an undirected, unweighted graph $G$, aka:

$$c(u, v) = \min_{S \subset V, u \in S, v \notin S} |\partial(S)|,$$

where $\partial(S)$ denotes the set of edges leaving $S$. Prove that for any triples of vertices $u$, $v$, and $w$ we have:

$$c(u, v) \geq \min\{c(u, w), c(v, w)\}.$$

Building upon this fact, show that given any undirected graph, we can find a weighted tree (with perhaps added vertices), $T$, so that for any $u$ and $v$, $c(u, v)$ equals to the minimum weight along the path between $u$ and $v$ in $T$. 