1 Euler Tours

(a) List ranking can also be used to do list prefix sum. In fact the recursive version shown in class after one level of recursion is already doing a sum.

Associate each edge with the node below it. Each down edge grabs the node value and each up edge uses 0. Do a list prefix sum on this. Subtract the down edge from the up edge. This is the desired value since it accounts for all the values added while traversing the subtree, but excluding the value when entering the subtree. It takes linear work and logarithmic span.

This can also be done directly with list ranking by first list ranking, and then moving each link to position i in an array where i is the rank, doing a plus scan on values as above, and again taking the difference.

(b) First consider generating a pointer from each word i to the word that would start the next line if word i were at the beginning of a line. Note that this pointer structure forms a forest—every word points to at most one other word, and every word can be pointed to by many other words. By adding an artificial word of length L at the end, we can make it a tree. Let's call this the *break tree*. Now we consider three steps: first making the break tree, then converting it into a Euler tour, and finally using the Euler tour to identify the line starts.

(a) To make the tree, let S (for Start) be the result of a plus scan on W. Now create the two arrays $E[i] = S[i] + W[i]$ (for End of word) and $N[i] = S[i] + L$ (for start of Next line). To generate the break tree, for each i we could binary search the value $N[i]$ in the array $E[i]$ to find the parent. However, this would require $O(n \log n)$ work. Instead, merge E and N into M (for Merged). Ties are broken so elements from E go first. Let's call the elements from E and N in M black and white, respectively. Note that for each white element in M its parent in the break tree is the next black element. We have effectively got the same result as the binary searches but in linear work. We can pass the parent index to its children, by creating an array

$$A[i] = \begin{cases} i & \text{black} \\ -\infty & \text{white} \end{cases}$$

and doing a backwards min-scan on it. Each white element will get the index of the next black element in M, i.e. its parent. Note that each black (white) element can easily find its matching white (black) element: when doing the merge, keep the original word location as auxiliary data, and rendezvous at that location.

(b) To convert to an Euler tour, we make each black element in M the down edge for the node below it (the word it corresponds to), and each white element an up edge for the node below it. We need to link these to form the Euler tour. We assume the children of a black element (down edge), i.e., its immediate white predecessors in M, are ordered right-to-left. If a black element (down edge) is a leaf (preceded by another black element) it points to its own white element (its up edge), otherwise it points to the black element corresponding to the previous white element in M (the down edge of its first child). If a white element (up edge) i has a preceding white element j, it

points to j's black element (the down edge of its next sibling), otherwise it is the last child and points to i's parent's white element (the up edge of its parent). The element i knows its parent from the backwards min-scan. The list is formed.

(c) To use the tree to find the breaks, place a 1 at the very first word and 0 elsewhere. Calculate for each node in the tree the sum of its subtree. If your result is 1 you are the first word on a line, and otherwise you are not.

All steps take linear work and logarithmic span.

2 Parallel SSSP

(a) BFS solves it in $O(m)$ work and $O(\mathsf{diam} \log n)$ depth.

(b) Repeated Squaring:

- A^2 can be computed in $O(n^3)$ work and $O(\log n)$ depth. Computing each entry takes $O(n)$ work and $O(\log n)$ work; all entries can be computed in parallel. Summing over the $O(n^2)$ entries gives the bound.
- The proof is by induction. When $k = 1$, the 1-hop distances are given by the weights in $A = A^1$ (note that all $w_{ij} = 1$ by assumption). For the inductive step assume that A^k_{ij} gives the minimum k-hop distance from i to j and is 0 if i and j are not reachable within k hops. Consider what happens to some row i when we multiply A^k by A. The row represents the distances from vertex v_i to all other vertices; if A^k_{ij} is non-zero, then this is the correct k-hop distance by our inductive assumption, otherwise there is no path from v_i to v_j using at most k hops. A^{k+1}_{ij} is computed by multiplying A_{i*} with A_{*j}, which is considering the k-hop shortest path to each neighbor of v_i, and then the weight one edge from this neighbor to j, if it exists, and so $\min_{l=1}^{n} A^k_{il} + A_{lj}$ paths gives the minimum directed $k+1$ hop path to v_j.
- A^n can be computed similar to the exponentiation-by-squaring algorithms for computing x^n in $\lfloor \log n \rfloor$ steps. We first compute A^2, and then square this matrix, repeating until A^n is computed. The same method can be used when n is not a power of two by multiplying the matrices corresponding to the binary representation of n. We need to compute at most $O(\log n)$ multiplications, so the total cost is $O(n^3 \log n)$. Each multiplication costs $O(\log n)$ depth, so the overall depth is $O(\log^2 n)$.

(c) Ullman-Yannakakis:

- It turns out we don't need a Chernoff bound for this problem. Consider a path of length $k = c\sqrt{n} \log n$. Recall that the probability some node is selected as a hop node is $1/\sqrt{n}$. The probability that all k nodes are not selected as a hop node is therefore $(1 - 1/\sqrt{n})^k$. Expanding k we have:

$$(1 - 1/\sqrt{n})^{\sqrt{n}^{c \log n}} \leq (1/e)^{c \log n} = 1/n^c$$

The probability that at least one of the nodes is a hop node is therefore

$$1 - 1/n^c$$

which proves the high probability bound.

(d) Given s and t we can compute the shortest path distance between them using H as follows. First, we run a k-limited BFS from both s and t (run the search for t using in-edges). If we discover t during s's search then just return the distance found by the search. Otherwise, the distance from s to t is $> k$, and so we route the path through H. Let the hop nodes discovered by s be $H(s) = \{h_{s1}, \ldots\}$ and similarly for $H(t)$. The distance between s and t is given by computing

$$\min_{h_s \in H(s), h_t \in H(t)} d(s, h_s) + d_H(h_s, h_t) + d(h_t, t) \tag{1}$$

A simple argument shows that this computes the correct shortest path distance between s and t. If $d_G(s,t) \leq k$, then we report the correct distance immediately. Otherwise, consider some shortest path between s and t. Consider decomposing this path P into $|P|/2k$ pieces. Each piece will have a hop node w.h.p. Each hop node in a piece of the path will find its adjacent hop nodes when conducting its k-limited search. Therefore, the path between the hop node in s's piece and the hop node in t's piece can compute the correct shortest path distance using the all-pairs shortest path distance graph on the hop nodes, H. This quantity will be one of the hop-pairs computed in Equation (d) and so Equation will report the correct st distance.

Finally, we have to compute how many 'bad events' we're union-bounding over and make sure that we still have that all paths are preserved w.h.p. Note that trying to ensure that *every* path of length k has a hop node on it is hopeless; on a complete graph there are $O(n^k)$ paths of length k, and given our hop nodes a significant fracion of these will be bereft of hop nodes. We can be smarter about this, and instead ensure that for each pair of vertices that are distance $k/2$ away from each other in G, there is a hop node on some shortest path between them. There are at most $O(n^2)$ such pairs so we can just union bound over all such pairs as long as $c > 2$.

(e) We select $O(\sqrt{n})$ hop nodes in expectation. Each does a k-limited search which takes $O(m)$ work and $O(k \log n)$ depth in the worst case. Building H takes $O(n^{1.5})$ work in expectation $O(\log^2 n)$ depth. Therefore the total expected work is $O(\sqrt{n}m)$ and the depth is $O(k \log n)$, assuming that $m = O(n)$.

For a single shortest path query we perform 2 k-limited searches which costs $O(m)$ work and $O(k \log n)$ depth. For the hop nodes found, we intersect their distances using Equation (d), which gives us $O(n)$ work in expectation.

The work and depth for computing all (s,t) shortest paths is $O(nm + n^2)$ and $O(k \log n)$ depth by a similar argument.

(f) k should be set to $n/\rho \cdot \log n$.

In terms of ρ, the preprocessing requires $O(m\rho)$ work and $O(n/\rho \log^2 n)$ depth. A query requires $O(m + \rho^2)$ work and $O(n/\rho \log^2 n)$ depth.