Brian Ozawa Burns, 4637799
Discussed with: Rithwick Adicherla

1. **a) Give an algorithm that takes as input an undirected graph G = (V, E) as an adjacency list, and an edge uv ∈ E, and computes a shortest cycle (i.e. minimum length cycle) that uses the edge uv (or determines that no such cycle exists).**

   The algorithm starts by constructing a new graph/adjacency list, call it G' that takes the same set of vertices V and a new set of edges E' such that E' is E with uv and vu removed. Now vertices u and v are no longer neighbors. Then, we run a breadth first search on u, storing the distances each vertex is from u in an array, call it D (we can even terminate the breadth first search early if the vertex v is found before BFS is done). So, all vertices of degree 1 (1 edge from u) store 1 in D, all vertices of degree 2 (2 edges from u) store 2 in D, and so on. Now, check if v is in D. If v is not in D (or D[v] = +inf), then there does not exist any cycle from u to v that uses edge uv, and the algorithm will terminate. If v is in D, then the algorithm finds a neighbor, n, of v in D such that D[n] = D[v] - 1 (ie n is 1 edge closer to u from v). This process repeats, traversing backward until u is reached (D[u] = 0), storing the path as it returns to u. The last step is to append the edge uv to the path being drawn. At this point, the algorithm finishes and returns the path constructed.

   b) Briefly argue correctness of your algorithm from 1a.

   A cycle is defined as a sequence of vertices such that the first and last vertices are the same. Thus, a cycle from u to v that includes the edge uv, must be able to reach u from v without the uv edge.

   Taking the BFS algorithm to be true, we know that we can utilize it to construct the array D of all vertex distances from u. In addition, we know that the BFS algorithm will terminate once it has visited every vertex, or once it has reached v if we add that case to the algorithm. In addition, the backward traversal portion of our algorithm will always terminate. It either never starts, in the case that D[v] does not exist (or D[v] = +inf), or it terminates once it has backtracked all the way to 0 at D[u], the starting vertex.

   And finally, the algorithm indeed returns the shortest cycle from u to v by definition of breadth first search algorithm which takes the shortest path to reachable unvisited vertices in the graph.

   c) Briefly analyze the running time of your algorithm from 1a.

   The algorithm's running time is dependent on BFS, array insertion and lookup, and backward traversal of the array. We know that BFS is always O(n + m) with n and m

representing the number of vertices and number of edges, respectively. The array insertion and lookup operations will be constant time, and the backward traversal of the array will be O(n) time worst case. Thus, total running time is O(n + m) + O(1) + O(n) = **O(n + m).**

2. **a) Give an algorithm that takes as input a connected undirected graph G and vertex s and outputs a set $E' \subseteq E$ such that (i) For every vertex $v \in V$, the shortest path distance from v to s in $G' = (V, E \setminus E')$ is equal to the shortest path distance from v to s, and (ii) subject to (i), $|E'|$ is maximized.**

   The algorithm starts by doing a breadth first search on s. As the BFS traverses through the vertices, we will store the distances each vertex is from s in an array, call it D. Take E'' = E\E' that was given in the problem statement. E' will start as an empty array and we will store all unnecessary edges from G in it. The algorithm will iterate backward through D. For each vertex, v, find the first neighbor n such that D[n] = D[v] - 1. For any other neighbor of v, n', that satisfies D[n'] = D[v] - 1, store vn' in E'. In addition, any edges vk such that D[k] = D[v] should be stored in E'. After finishing this traversal of D, the algorithm returns E', and E'' can be constructed by removing all edges e from E such that $e \in E'$ and $e \in E$.

   b) Briefly argue correctness of your algorithm from 2a.

   Since the BFS algorithm is correct, we can use it to construct the array D of all vertex distances from u and know that it will terminate once it has visited every vertex. In addition, the backward iteration portion of our algorithm will always terminate when it has backtracked all the way through the array.

   The algorithm is correct because it returns only the set of edges E' that represent the edges to remove from the edge set E to get the shortest path distances from any vertex v to the input vertex s. Firstly, because BFS is used, only the shortest distances from s to any vertex are inserted into the array D. Also, since the algorithm only inserts one edge from each vertex to a vertex closer to s in E'', all unnecessary edges are essentially removed. And, since E'' contains edges that trace every single vertex, except for the starting vertex s, back one degree of separation (ie one degree closer to s), every vertex thus has a single optimized path to s. Also, as stated in the problem, $E' \subseteq E$ because E' contains all or less of the edges in E.

   c) Briefly analyze the running time of your algorithm from 2a.

   The algorithm's running time depends on BFS, array insertion and lookup, and backward traversal of the array. We know that BFS is always O(n + m) with n and m representing

the number of vertices and number of edges, respectively. The array insertion and lookup operations will be constant time, and the backward traversal of the array will be O(m). Thus, total running time is O(n + m) + O(1) + O(m) = **O(n + m).**

3. **a) Give an algorithm that takes as input a connected undirected graph G = (V, E) as an adjacency list and computes a shortest cycle (i.e. minimum length cycle) in G, or determines that no such cycle exists.**

   The algorithm is essentially the same algorithm from problem 1a), call it alg1, except it is used on every edge returned in E' from the algorithm from problem 2a), call it alg2.

   First, the algorithm retrieves the edges E' from G by running it through alg2 because these are all edges that form cycles in G. Then, for each edge in E', the algorithm runs the edge through alg1 with the original graph G and stores the cycle based on its size, which is calculated by iterating through the cycle's edges. The algorithm then selects the cycle of smallest length and returns it.

   Furthermore, the algorithm can be slightly optimized by combining the BFS used in both alg1 and alg2, but we will omit this in this explanation for simplicity.

   b) Briefly argue correctness of your algorithm from 3a.

   The algorithm is correct because it answers the question of and returns the correct shortest cycle of a connected undirected graph G.

   Assuming the BFS is correct, and that alg1 and alg2 both terminate properly, then this algorithm terminates immediately after returning the shortest cycle generated by alg1.

   In addition, taking alg1 to be correct, we get the shortest cycle containing any edge that we send into it as input. Therefore, running alg1 with the cycle edges returned by alg2 will give us all shortest cycles in the entire graph G. Thus, all that is left to do is identify the shortest path and return it.

   c) Briefly analyze the running time of your algorithm from 3a.

   The algorithm's running time is dependent on alg1, alg2, calculating the size of each cycle, and selecting the shortest cycle. Alg1 and alg2 each, independently, require O(n + m) with n and m representing the number of vertices and number of edges, respectively. For each cycle generated by alg2, calculating its size requires iterating through its n elements. Finally, selecting the cycle of shortest size will be constant time. Therefore, O(n + m) + O(n(n + m)) + O(1) = O((n + 1)(n + m) + 1).

This, however, can be simplified to **O(n(n + m)).**

4. **a) Design a data structure, which takes as input a tree T = (V, E) (in adjacency list format) together with a root vertex r. The data structure may then spend some time to pre-process the input.**

   The new data structure will consist of **two arrays, one for pre-order traversal and the other for post-order traversal** of the tree taken as input.

   Pre-processing step:
   Pre-order traversal is done recursively. It starts at the root r, storing 1 in the pre-order array at the index of the root r. It then makes a recursive call to the left, storing 2 in the pre-order array at the index of this left vertex. Then, it makes recursive calls to the right. In other words, this pre-order array stores values 1 through |V|, with the vertices hashed to these values in the order in which they were visited (ex. If v is visited first, then pre-order_array[v] = 1; if v was visited fourth, then pre-order_array[v] = 4; if v was visited last, then pre-order[v] = |V|). Each parent vertex will be visited before its children and thus will have a smaller value in the pre-order array.

   Post-order traversal is also done recursively. It is similar to pre-order except that this function calls itself on its left and right children before adding itself to the post-order array. This function, thus, starts at the leftmost child, stores 1 at the index of this vertex in the post-order array. It then moves to its parent's right child, then to the parent. This process continues until every vertex in the graph is reached. Just as for the pre-order array, this post-order array stores values 1 through |V|, with the vertices acting as the hash keys in the order in which they were visited (ex. If v is visited first, then pre-order_array[v] = 1; if v was visited fourth, then pre-order_array[v] = 4; if v was visited last, then pre-order[v] = |V|). Each parent will be visited after its children and thus will have a larger value in the pre-order array.

   Now that we have the two arrays, one storing the pre-order and the other storing the post-order, we can determine whether a vertex u is a descendant of vertex v. In pre-order_array, if u is a descendant of v, then u will store a value that is greater than v. In post-order_arry, if u is a descendant of v, then u will store a value that is less than v. In other words, check if pre-order_array[v] < pre-order_array[u] and if post-order_array[v] > post-order_array[u]. If both of these conditions are satisfied, then u is indeed a descendant of v.

   b) Briefly analyze correctness of your data structure from 4a

This data structure is correct because it answers the question of and returns the correct result to the question of whether a given vertex is a descendent of another.

The pre-processing step is correct because the graph that is inputted, as an adjacency list, must be finite. Thus, the pre-order and post-order traversals will finish (ie terminate) after iterating through every vertex. In the pre-order traversal, the parent is visited before its children, and thus any vertex u that is a descendant of v will have a value greater than that of v. However, in pre-order traversal, the left children are visited before the right children; however, this does not make the right children descendants of the left children. This is solved with the post-order traversal, in which the children are visited before the parents. In post-order, any vertex u that is a descendant of v will have a value less than that of v. Therefore, this post-order array corrects the problem where a vertex has a lower value in pre-order but is not actually a descendant.

c) Briefly analyze the running time of your preprocessing procedure from 4a.

The pre-order traversal visits each node once and finishes once every node has been visited making it O(n). The post-order traversal visits each node once and finishes once every node has been visited making it O(n). Thus, the running time of the preprocessing procedure is O(2n) and can be simplified to **O(n)**.

d) Briefly analyze the running time of your algorithm for the descendant query from 4a.

To check if u is a descendant of v: we must hash with key u for pre-order, v for pre-order, then compare them, then we must hash u for post-order, v for post-order, then compare them. This is O(6). Thus, the running time for the descendant query is simply **O(1)**.

5. **a) For every fixed value v ∈ V . What is the expected number of keys hashed to v? In other words, what is E[|{k ∈ K : f(k) = v}|]?**

We have that the set of keys, |K| = 10, and that the hash value set, |V| = 20. We define |{k ∈ K : f(k) = v}| to be a random variable X_k. Since the probability of a key being hashed to any particular value, v ∈ V, is $\frac{1}{|V|} = \frac{1}{20}$. We have that

$$E[X_k] = \sum_{i=1}^{|K|} \frac{1}{20} = \frac{|K|}{20} = \frac{10}{20} = \mathbf{0.5}$$

b) What is the probability that there are no collisions when we sample a random function f : K → V from a key set K of size 10 to a hash value set V of size 20?

We have that the set of keys, |K| = 10, and that the hash value set, |V| = 20.

$$Probability(\#collisions = 0) = \frac{|V|}{|V|} \cdot \frac{|V|-1}{|V|} \cdot \frac{|V|-2}{|V|} \ldots \frac{|V|-|K|+1}{|V|} = \frac{|V|!}{|V|^{|K|}(|V|-|K|)!}$$

$$\frac{20!}{20^{10}(20-10)!} = \mathbf{0.06547}$$

c) What is the expected number of collisions when we sample a random function f : K → V from a key set K of size 10 to a hash value set V of size 20?

$$E[\#collisions] = 0 \cdot P(K_0\ collision) + 1 \cdot P(K_1\ collision) + 2 \cdot P(K_2\ collision) + \ldots + (|K| - 1) \cdot P(K_{|K|-1} collision) = 0 + \frac{1}{|V|} + \frac{2}{|V|} + \ldots + \frac{|K|-1}{|V|}$$

$$= \sum_{i=0}^{|K|-1} \frac{i}{|V|} = \frac{|K|(|K|-1)}{2|V|} = \frac{10(9)}{2(20)} = 9/4 = 2.25\ collisions$$

However, based on our definition of a collision (ex. (1,3) represents 2 collisions), we actually expect to have $2.25 \cdot 2 =$ **4.5 collisions**

d) What is the smallest possible value of m so that the expected number of collisions is less than or equal to 1?

$E[\#collisions] \leq 1$, let m = |V|

$$E[\#collisions] = \sum_{i=0}^{|K|-1} \frac{i}{m} = \sum_{i=1}^{|K|-1} = \frac{|K|(|K|-1)}{2m} \leq 1$$

$$\frac{|K|(|K|-1)}{2} = \frac{10(9)}{2} = m = 45$$

However, based on our definition of a collision (ex. (1,3) represents 2 collisions), we actually need to have $m = 45 \cdot 2 =$ **90** to have collisions less than or equal to 1.

e) What is the smallest possible value of m so that the probability of no collisions is at least 1/2?

$$\frac{1}{2} \leq \frac{|V|}{|V|} \cdot \frac{|V|-1}{|V|} \cdot \frac{|V|-2}{|V|} \ldots \frac{|V|-|K|+1}{|V|} = \frac{|V|!}{|V|^{|K|}(|V|-|K|)!} = \frac{m!}{m^{|K|}(m-|K|)!}$$

$$\frac{25!}{15! \times 25^{10}} \quad 0.12437869075365888$$

$$\frac{50!}{40! \times 50^{10}} \quad 0.38170668055855104$$

$$\frac{75!}{65! \times 75^{10}}$$ 0.5341568918686291015

$$\frac{65!}{55! \times 65^{10}}$$ 0.48252073026454359

$$\frac{70!}{60! \times 70^{10}}$$ 0.509623898933184337

$$\frac{69!}{59! \times 69^{10}}$$ 0.5044202224050745

m = 69