1 Applying BFS 1

For later usage, we provide an algorithm which computes a shortest path between two given vertices $s, t \in V$. To do this, we use a modified version of BFS which keeps track of how the vertices are discovered. In particular, each vertex keeps track of which of its neighbors was first to "discover" it during BFS. After running BFS on G with t as the starting vertex, if s has not been discovered, then we return an empty/null path. If s has been discovered, then we construct a path from s to t in G by backtracking from s through each vertex's "discoverer" until we reach t.

Algorithm 1. Input is a graph $G = (V, E)$ and vertices $s, t \in V$. Output is a path from s to t in G (when it exists) in the form of a list of vertices, such that each pair of consecutive vertex is adjacent in G. If no such path exists (i.e., if s and t are in different connected components), the output is an empty list.

```
1. Let $F$ be a queue of vertices, initialized with $t$.
2. Let $S$ be an array of vertices indexed by $V$, with values initialized to $\varnothing$ (null).
3. Set $S[t] = t$.
4. While $F$ is not empty:
5.     Let $u$ be the vertex dequeued from $F$.
6.     For each $v \in V$ adjacent to $u$ with $S[v] = \varnothing$:
7.         Enqueue $v$ onto $F$.
8.         Set $S[v] = u$.
9. If $S[s] = \varnothing$, return an empty list.
10. Let $P$ be an empty list of vertices.
11. Set $P[0] = s$.
12. Let $i = 0$ be an index.
13. While $P[i] \neq t$:
14.     Set $P[i+1] = S[P[i]]$.
15. Return $P$.
```

(a) Give an algorithm that takes as input an undirected graph $G = (V, E)$ as an adjacency list, and an edge $uv \in E$, and computes a shortest cycle (i.e. minimum length cycle) that uses the edge uv (or determines that no such cycle exists). Full points awarded for algorithms with running time $O(n + m)$.

We first use Algorithm 1 to find a path from v to u in the graph obtained by removing the edge uv from G. We can then append v to this path to form a cycle in G, based at v.

Algorithm 2. Input is a graph $G = (V, E)$ and edge $uv \in E$. Output is a cycle in G based at v which uses uv (when it exists) in the form of a list of vertices, such that each pair of consecutive vertices are adjacent in G. If no such cycle exists, the output is an empty list.

```
1. Let $G' = (V, E \setminus \{uv\})$.
2. Let $P$ be the output of Algorithm 1 applied to $G'$ and $v, u$.
3. Append $v$ to $P$.
4. Return $P$.
```

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

Given a cycle in G which uses a given edge, removing the edge from the cycle gives a path between its endpoints which do not use the edge. Similarly, given a path between the endpoints of an edge, adding the edge to the end of the path gives a cycle in G which uses the edge. In other words, the cycles in G which use uv are in correspondence with the paths in G between u and v which do not use uv.

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

In Algorithm 1, lines 1-8 are comparable to the usual BFS algorithm, whose running time is $O(n+m)$. The remainder of the algorithm constructs the path from s to t by backtracking through S. The running time of this section increases linearly with the number of vertices in the path, which is n is the worst case. Hence, the running time of Algorithm 1 is $O(n+m)$.

Arguably, G' is not necessary for Algorithm 2, but allows Algorithm 1 to be more general. If we are clever, we could construct G' without copying G. However, even if we do copy G, this is $O(n+m)$ The call to Algorithm 1 is also $O(n+m)$. The remainder is constant time, so Algorithm 2 is $O(n+m)$.

2 Applying BFS 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.

We use the first part of Algorithm 1 to compute S, starting at s. Then E' is taken to be all the edges not encoded by S.

Algorithm 3. Input is a connected graph $G = (V, E)$ and a vertex $s \in V$. Output is a subset E' of E satisfying conditions (i) and (ii).

```
1. Let $S$ be computed as in Algorithm 1 lines 1-8, with $t = s$.
2. Let $E'$ be a copy of $E$.
3. For each $v \in V$:
4.         Remove from $E'$ the edge between $v$ and $S[v]$.
5. Return $E'$.
```

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

Note that the BFS in Algorithm 1 explores an entire connected component of the graph; since G is connected, this is all of G. This means that we can use the data of S to compute a shortest path in G from any vertex to s. Since E' is constructed to be the edges in E which are not used by any path derived from S, we know that $E \setminus E'$ is precisely the set of edges which are used by paths derived from S. In particular, S encodes shortest paths in G from every vertex to s, so all such paths will be present in G', i.e., (i) is satisfied.

The condition that $|E'|$ is maximized is equivalent to $|E \setminus E'|$ being minimized. For each $v \in V$, S uniquely encodes a shortest path in G from v to s. By construction, the only path in G' from v to s is the one encoded by S. In other words, each vertex of G' has a unique path in G' to the basepoint s, i.e., G' is a tree. More specifically, G' is a spanning tree, which implies that $|E \setminus E'| = n - 1$ is minimized subject to G' being connected (which is a weaker condition than (i)).

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

Computing S using Algorithm 1 once again is $O(n + m)$. Copying E is $O(m)$. Looping over V is $O(n)$. Thus, Algorithm 3 is $O(n + m)$.

3 Applying BFS 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. Full points awarded for algorithms with running time $O(n(n+m))$. *Hint: use the algorithm from Task 2a.*

We apply Algorithm 3 to construct spanning trees G' based at each vertex of G. We look at each edge of G which is not in G' and compute the length of the path in G' between its endpoints and passing through the basepoint. If this length is the smallest seen so far, we record the length and the edge in question. Once we have checked all of these cases, we use Algorithm 2 to compute the shortest cycle in G using the most recently recorded edge.

Algorithm 4. Input is a connected graph $G = (V, E)$. Output is a shortest cycle in G in the form of a list of vertices such that each pair of consecutive vertices is adjacent in G, or an empty list if no such cycle exists.

```
1. Let $\ell = +\infty$.
2. Let $e = \varnothing$.
3. For each $s \in V$:
4.     Let $E'$ be the output of Algorithm 3 applied to $G$ and $s$.
5.     Let $D$ be computed as in the given BFS algorithm applied to $G'$ and $s$.
6.     For each $\overline{uv} \in E'$:
7.         If $D[u] + D[v] < \ell$:
8.             Set $\ell = D[u] + D[v]$.
9.             Set $e = \overline{uv}$.
10. If $e = \varnothing$, return an empty list.
11. Let $P$ be the result of Algorithm 2 applied to $G$ and $e$.
12. Return $P$.
```

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

Given a shortest cycle P in G, let s be a vertex in P. Let E' be the output of Algorithm 3 applied to G and s, and let $G' = (V, E \setminus E')$. Since G' is a tree, there must be some edge $\overline{uv}$ of P which is in E'. Then P consists of the edge $\overline{uv}$ and two shortest paths: one from s to u and another from s to v. These path may not be in G', but G' does contain some shortest paths, say A from s to u and B from s to v.

Note that A and B must be disjoint except for s. If this this not the case, then a cycle shorter than P may be found starting at u and following A towards s until we hit a vertex in B, at which point we head towards v. If the first vertex in B we encounter is not s, then we have skipped some of both A and B, giving a path from u to v which does not use $\overline{uv}$ and is shorter than that contained in P.

This means that the length of P is the sum of the lengths of A and B, plus 1 for the edge $\overline{uv}$. In other words, with D as in line 5, the length of P is precisely $D[u] + D[v] + 1$. In which case, if such a short cycle has yet to be found, we set ℓ to $D[u] + D[v]$ and e to $\overline{uv}$.

This means that $\overline{uv}$ is present in a shortest cycle, so when we use Algorithm 2 to compute a shortest cycle containing $\overline{uv}$, we in fact get a shortest cycle with respect to all of G.

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

The calls to Algorithm and BFS are both $O(n+m)$. Iterating over E' is at most $O(m)$, and the interior is constant time. Iterating over V is $O(n)$ and its interior is $O(n+m)$, hence the whole loop is $O(n(n+m))$.

The call to Algorithm 2 is $O(n+m)$.

Thus, Algorithm 4 is $O(n(n+m))$.

4 Who's your Great Great Great Grandfather?

Let T be a tree and r be a vertex of T, called the root. We will say that a vertex u is a descendant of a vertex v if the path from u to r in T passes through v.

(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. After the pre-processing the data structure should be able to efficiently answer any number of descendant queries. Each descendant query consists of two vertices u and v, and the data structure should report whether u is a descendant of v or not. Full score given for data structures that spend $O(n)$ time for preprocessing and are able to answer each descendant query in time $O(1)$. *Hint: compute the pre-order and the post-order DFS-traversals of T from r, see "Tree Traversal" on Wikipedia*

The data structure, denoted R, will store boolean values and be indexed by ordered pairs of vertices, i.e., elements of the cartesian product $V \times V$. In other words, R will be a 2-dimensional array: for each $u \in V$, $R[u]$ is an array of booleans indexed by V. We will process the data such that $R[u, v] := R[u][v]$ is true if and only if u is a descendent of v.

We perform a modified version of DFS, where we store the fact that each vertex is a descendent of each vertex on the path from r. The paths starting at r are iterated over in the form of the DFS stack.

```
 1. Let $K$ be a stack of vertices, initialized with $r$.
 2. Let $R$ be an array of booleans indexed by $V \times V$.
 3. While $K$ is not empty:
 4.     Let $v$ be the top of $K$.
 5.     For $u \in K$:
 6.         Set $R[u, v] =$ true.
 7.     If $v$ has a neighbor $u \in V$ with $R[u, u] =$ false:
 8.         Push $u$ onto $K$.
 9.     Else:
10.         Pop $v$ off of $K$.
11. Return $R$.
```

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

At any given time, the stack K represents a path from the root r at the bottom of the stack to the vertex at the top of the stack. This is because we only ever push a new vertex onto the stack when it is adjacent to the previous top vertex. Since T is a tree, each pair of vertices has a unique path between them. In particular, u is a descendant of v if and only if the unique path in T from r to u passes through v. Correspondingly, we set $R[u, v]$ to true if and only if v lies on the path between r and u.

Additionally, we are using the value of $R[u, u]$ to represent whether or not a vertex has been visited. This ensures that the path represented by the stack is always moving away from r.

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

Note that a tree on n vertices has $n-1$ edges, so DFS on a tree is $O(n+n-1)=O(n)$.

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

The query of R is the same as accessing two arrays, each of which is $O(1)$, so the running time of the query is $O(1)$.

5 Hash Calculations

(a)

With 10 keys across 20 values, we expect each value to be hashed to by $10/20 = 0.5$ keys.

(b)

There are 20^{10} possible functions $K \to V$; we count the number of function with no collisions. There are 20 possible values for $f(1) \in V$. To avoid collisions, we must have $f(2) \neq f(1)$, so there are 19 possible values for $f(2)$, etc. There are $20 \cdot 19 \cdots 11 = 20!/10!$ possible functions with no collisions. Thus, the probability that there are no collisions is

$$\frac{20!/10!}{20^{10}} = \frac{19!}{10!20^9}.$$

(c)

The expected number of collisions can be broken down into the sum of the expected number of collisions of the form (k, k') as k ranges over K and $k \neq k'$, i.e.,

$$E[\#\text{collisions}] = \sum_{k \in K} E[\#\text{collisions } (k, -)] = \sum_{k \in K} \sum_{k' \neq k} E[\#\text{collisions } (k, k')].$$

Let $\Omega = V^K$ denote the set of functions $f : K \to V$.

$$E[\#\text{collisions } (k, k')] = \sum_{f \in \Omega} \frac{\delta_{f(k), f(k')}}{|\Omega|} = \frac{1}{|\Omega|} |\{f \in \Omega : f(k) = f(k')\}|,$$

where $\delta_{i,j}$ is the Kronecker delta function, i.e., $\delta_{i,i} = 1$ and $\delta_{i,j} = 0$ for $i \neq j$. There are as many functions $f : K \to V$ with $f(k) = f(k')$ as there are functions $K \setminus \{k'\} \to V$, since the value of $f(k')$ is determined by the value of $f(k)$. Hence,

$$|\{f \in \Omega : f(k) = f(k')\}| = \left| V^{K \setminus \{k'\}} \right| = |V|^{|K \setminus \{k'\}|} = |V|^{|K|-1},$$

so

$$E[\#\text{collisions } (k, k')] = \frac{|V|^{|K|-1}}{|\Omega|} = \frac{|V|^{|K|-1}}{|V|^{|K|}} = \frac{1}{|V|}.$$

Finally,

$$E[\#\text{collisions}] = \frac{1}{|V|} \sum_{k \in K} \sum_{k' \neq k} 1 = \frac{1}{|V|} \sum_{k \in K} (|K| - 1) = \frac{|K|(|K| - 1)}{|V|}.$$

(d)

(e)