Sample Solutions to Problem set 3

1. (6 points) Determining the end of an array

You are given a long array A, in which the first n indices contain arbitrary integers, and the remaining entries (indices n and higher) are all ∞. Give an algorithm that determines n, while accessing at most $O(\log n)$ of the entries of A.

Answer: This can be solved by searching through the array in geometrically increasing indices and then binary search.

1. Set $i = 1$. Double i while $A[i] < \infty$. If $i = 1$, then return $n = 0$.
2. Let $\ell = \lfloor i/2 \rfloor$ and $r = i$. We now conduct a search in $A[\ell, r-1]$.
   - While $r-1-\ell+1 > 1$: if $A[\lfloor (r+\ell)/2 \rfloor] < \infty$ then $\ell = \lfloor (r+\ell)/2 \rfloor + 1$, else $r = \lfloor (r+\ell)/2 \rfloor$.
   - Return $n = r - 1$.

The $O(\log n)$ time follows from the fact that i doubles until it reaches at most $2n$ in the first step, and the second step is essentially binary search.

2. (4 + 4 = 8 points) Graph coloring

This exercise is primarily to help you reason about graphs.

Call an undirected graph *k-colorable* if one can assign each vertex a color drawn from $\{1, 2, \ldots, k\}$ such that no two adjacent vertices have the same color. We know that bipartite graphs are 2-colorable.

(a) Prove that if the degree of every vertex of G is at most Δ, then the graph is $(\Delta+1)$-colorable. Show how to contruct, for every integer $\Delta > 0$, a graph with maximum degree Δ that requires $\Delta + 1$ colors.

Answer: Here is an algorithm for constructing a coloring for G using at most $(\Delta+1)$ colors. Repeat the following step until all vertices are colored.

- Pick an uncolored vertex v. Assign it a color from $\{1, 2, \ldots, \Delta+1\}$ that has not been assigned to any of its neighbors. Since the degree of v is at most Δ, such a color always exists.

Clearly, the algorithm terminates in n iteration. And since the number of colors used is at most $\Delta + 1$, we have proved the desired statement.

Here is a family of graphs of degree Δ that requires $\Delta + 1$ colors: the complete graph over $\Delta + 1$ vertices (i.e., every pair of vertices has an edge). Every vertex has to have a distinct color since it has an edge to every other vertex. So the number of colors needed is $\Delta + 1$.

(b) Prove that if G has n vertices and $O(n)$ edges, then it can be colored with $O(\sqrt{n})$ colors.

Answer: We will follow the steps in the Piazza post. (There are other ways to approach this problem.)

Let the number of edges in G be m. We know that $m \leq cn$ for some constant $c > 0$. We first obtain a bound on the number k of vertices with degree at least $\sqrt{n}$. Since $k\sqrt{n} \leq 2m \leq 2cn$, we obtain that $k \leq 2c\sqrt{n}$. We assign each of these k vertices distinct colors from $\{1, \ldots, k\}$.

This leaves vertices with degree less than $\sqrt{n}$. We use the algorithm of part (a) to assign them colors from a new pallete of $\lfloor\sqrt{n}\rfloor + 1$ colors $\{k+1, k+2, \ldots, k+\lfloor\sqrt{n}\rfloor+1\}$. The total number of colors used is at most $k + \lfloor\sqrt{n}\rfloor + 1$, which is at most $(2c+1)\sqrt{n} + 1 = O(\sqrt{n})$, thus completing the proof.

3. (8 points) Number of shortest paths in social networks

Chapter 3, Exercise 10, page 110. (*Hint:* Use breadth-first search.)

Answer: We will solve the more general problem of finding the number of shortest paths from v to every other node.

We perform a BFS from v, obtaining a set of layers $L_0, L_1, \ldots$, where $L_0 = \{v\}$ and L_i is the set of nodes i hops away from v. For a node x, say in L_i, every path from v to x that goes through a node in L_1, then a node in L_2,..., then a node in L_{i-1}, and then to x is a shortest path. (This follows from the definition of BFS.)

We use the BFS routine to compute the number of shortest paths for each node x. Let $S(x)$ denote this number for a node x. For each node x in L_1, $S(x)$ is 1 since the only shortest-path consists of the single edge from v to x. Now consider a node y in layer L_j, for $j > 1$. The shortest paths from v to y all have the following form: they are a shortest path to some node x in L_{j-1}, and then they take one more step to get to y. Thus, $S(y)$ is the sum of $S(x)$ over all nodes x in layer L_{j-1} with an edge to y.

After performing BFS, we can thus compute all these values in order of the layers; the time spent in computing $S(y)$ is at most the degree of y. Since the sum of the degrees is twice the number of edges, the total running time is $O(m)$ plus the running time of BFS, which is $O(n+m)$. So the total running time is $O(n+m)$.

4. (8 points) A walk through the entire graph

Give an algorithm that takes as input an undirected graph $G = (V, E)$ and returns a path that traverses every edges of G exactly once in each direction. Your algorithm should run in $\Theta(n+m)$ time in the worst-case, where m is the number of edges and n is the number of vertices.

Answer: We can solve the problem using either BFS or DFS.

Here is the solution using DFS. We consider any edge (u, v) as two directed edges (arcs) $\langle u, v\rangle$ and $\langle v, u\rangle$. We first obtain a depth-first search of the graph, starting from an arbitrary vertex s. Since the graph is connected, this traversal visits every vertex and edge in the graph. Furthermore, the depth-first search also classifies the edges as tree and back. (Note that since the graph is undirected, there are no forward or cross edges.)

We need to find a path that goes through every edge in each direction exactly once. We first note

that using the depth-first search tree, we can easily find a path starting from the root s that goes through every *tree edge* exactly once in each direction. In order to add the back edges, we do the following: if (u, v) is a back edge from descendant u to ancestor v, after the completion of DFS(u) and before returning to DFS(u), we go through the edge in both directions.

We now describe the algorithm in more detail. We maintain a path P during the depth-first search algorithm, which will finally consist of all edges in both directions exactly once. We initially set P to nil. A call to DFS(u) goes through every edge (u, v) adjacent to u. If v is newly discovered, then the depth-first search sets the parent of v to u and calls DFS(v). In the modified algorithm, we add arc $\langle u, v\rangle$ to P just before calling DFS(v). If v is discovered but not yet finished, then there are two cases: (i) if v is the parent of u, then we do nothing; (ii) if v is not the parent of u (i.e., (u, v) is a back edge), we add the arcs $\langle u, v\rangle$ and $\langle v, u\rangle$ in this order to P. If v is already finished, this implies that (v, u) is a back edge that we have already explored in each direction when we were performing DFS(v). So we do nothing. When DFS(u) completes, we add $\langle u, \text{parent}[u]\rangle$ to P.

If m is the number of edges, then the above algorithm takes $O(m)$ time, since each edge is explored exactly twice.

Here is the solution using BFS. We first compute the breadth first traversal and mark each edge as tree edge or cross edge. We first note that just like above, using the breadth-first search tree, we can easily find a path starting from the root s that goes through every *tree edge* exactly once in each direction. (Visit all children before returning to parent.) In order to add the cross edges, we do the following: if (u, v) is a cross edge between u and v, we visit this edge in each direction from whichever of u or v that we reach first.

If m is the number of edges, then the above algorithm takes $O(m)$ time, since BFS takes time $O(m)$ and in the construction of the path, each edge is explored exactly twice.