CSE421: Design and Analysis of Algorithms

Homework 3

Anup Rao Due:

Each problem is worth 10 points:

1. Give a polynomial time algorithm that takes an undirected graph with m edges as input and outputs a coloring of the vertices with 3 colors, so that at least $2m/3$ of the edges are properly colored. An edge is properly colored if its vertices get distinct colors. HINT: Give a greedy algorithm that colors each vertex one by one.

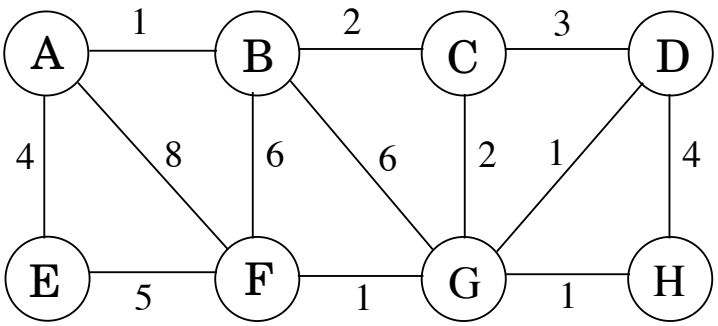
   Solution: The algorithm is described below.

```
   Input: An undirected graph G = (V, E) with m edges
   Result: A coloring of G where at least 2m/3 edges are properly colored
1  for v ∈ V do
2  |  Set count[color1]← 0, count[color2]← 0, count[color3]← 0
3  |  for v′ neighbor of v do
4  |  |  count[color of v′] + +
5  |  end
6  |  Set v_color ← argmin{count[color1], count[color2], count[color3]}
7  end
```

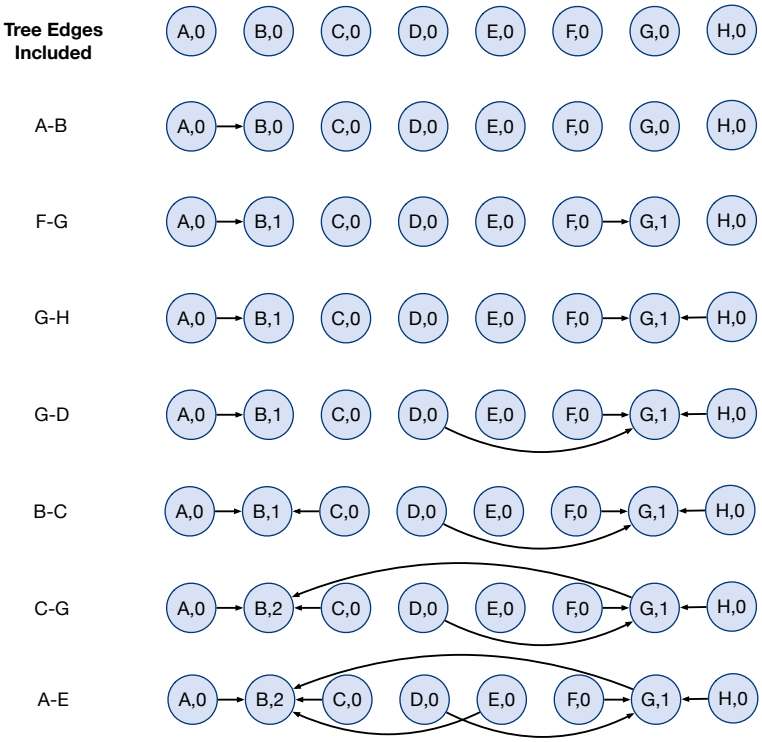
   Runtime: The first loop involves going through every vertex v in the graph, and the second involves going through every neighbor of v. There are polynomially many vertices and each vertex has a polynomial number of neighbors, thus the algorithm runs in polynomial time.

   Proof of Correctness: Every time we color a vertex v, we determine the fate of some number of edges (i.e. whether those edges will be properly colored or not at the end): these edges are exactly the edges (v, v') where v' has already been colored. But every time we color a vertex v, we color it in such a way to minimize the number of edges (v, v') that are improperly colored. There are 3 colors, so every time we color a vertex v, at most $1/3$ of the edges whose fate we decide will be improperly colored. Since this is true for every step of the algorithm (i.e. every time we color a vertex), we will have improperly colored $\leq 1/3$ of the edges by the end of the algorithm.

2. Show an execution of Kruskal's algorithm to compute the minimum spanning tree of the following graph. Show the state of the connected components (union find) data structure at each step:



Solution: The algorithm will first sort the edges (breaking ties arbitrarily) to give this order: A-B, F-G, G-H, G-D, B-C, C-G, C-D, A-E, D-H, E-F, B-F, B-G, A-F. Each edges is added to the tree only if it connects two new connected components. The following diagram shows the tree edges added and the state of the union-find data-structure after each edge addition.



3. Suppose you have a processor that can operate 24 hours a day, every day. People can submit jobs to the processor by giving a start and finish time during a day. The processor can only run one job at a time. If a job is accepted, it must run continuously between the start and finish times. For example, if the start time is 10 pm and the finish time is 3 am, then the job must run from 10 pm to 3 am every day. Give a polynomial time algorithm that on input such a list of such jobs outputs a set of compatible jobs of maximal size. Prove that your algorithm works.

Solution. The algorithm is given below.

```
   Input: set of jobs with start times and finishing times
   Result: A 24 hour processor that can handle most jobs
 1 Let J_1, J_2, ... ,J_n be the jobs.
 2 Let J be an empty set
 3 for i from 1 through n do
 4 |   s = J_i's start time
 5 |   e = s + 24 hours
 6 |   Sort all jobs by their finishing time with s as the start time, and consider only
   |     jobs whose finish time is within e.
 7 |   Run the greedy algorithm for interval scheduling from class on the sorted jobs,
   |     and call the solution J'
 8 |   if |J'| > |J| then
 9 |   |   J = J'
10 |   end
11 end
12 return J
```

(a) **Run time:** In each iteration of the For loop, we first sort the jobs and then run the greedy algorithm from class. Since the time for for each iteration is $O(n \log n)$ and the number of iterations is n, the overall runtime is $O(n^2 \log n)$.

(b) **Proof of correctness:** It is important to note that there is no start time that could be taken as a reference. Even if we take $00:00$ to be the reference, there are jobs that overlap $00:00$ defeating the purpose. Apriori it is not clear what the start time of the optimal solutions is, but note that the possible start times can only be one of the start times of the jobs. In class, we proved that the greedy interval scheduling algorithm will give the optimal solution for a given set of jobs and starting time. So applying it for all possible start times and picking a solution with largest number of jobs is guaranteed to find the optimum solution.

4. In class we discussed an algorithm to color the vertices of an undirected n vertex graph with 2 colors so that every edge gets exactly 2 colors (assuming such a coloring exists). We know of no such algorithm for finding 3-colorings in polynomial time. Here we'll figure out how to color a 3-colorable graph with $O(\sqrt{n})$ colors.

   (a) Give a greedy polynomial time algorithm that can properly color the vertices with $\Delta+1$ colors, as long as every vertex of the graph has degree at most Δ.

   Solution. Pseudo-Code version:

```
   Input: An undirected graph G = (V, E) with max degree Δ + 1
   Result: Color G using atmost Δ + 1 colors.
1  for v ∈ V do
2  |  Let C be {1, 2, ..., Δ + 1}
3  |  for v' ∈ neighbors of v do
4  |  |  Remove the v'_color from C
5  |  end
6  |  Set v_color to be an arbitrary color from C
7  end
```

   Runtime: The first loop involves going through every vertex v in the graph, and the second involves going through every neighbor of v. There are polynomially many vertices and each vertex has a polynomial number of neighbors, thus the algorithm runs in polynomial time.

   Proof of Correctness: Regardless of how the neighbors of a node are colored, it is always possible to color a node with one of the $\Delta + 1$ colors. Each node has at most Δ neighbors, so there are at most Δ colors that a node *cannot* be colored with, yet $\Delta + 1$ colors are available. Thus we will never run out of color, and thus greedy coloring of the nodes work.

(b) Give a polynomial time algorithm that can properly color the graph with $O(\sqrt{n})$ colors, as long as the input graph is promised to be 3-colorable. HINT: If a vertex v has more than $\sqrt{n}$ neighbors, then argue that the subgraph of the neighbors of v must be bipartite, and use the algorithm from class to color v and its neighbors with 3 new colors. Continue this process until every vertex has less than $\sqrt{n}$ neighbors, and then use the algorithm from part (a).

Solution.

Input: An undirected 3-colorable graph $G = (V, E)$.
Result: Color G using at most $O(\sqrt{|V|}) = O(\sqrt{n})$ colors.

```
for v ∈ V do
    if v has at least √n uncolored neighbors then
        Pick 3 new colors, c1, c2, c3;
        Color v with c1;
        Color the induced subgraph of v's uncolored neighbors with c2, c3 by
          traversing the subgraph and assign alternating color on the path;
    end
end
Let G' be the induced subgraph of remaining uncolored nodes;
Color G' with algorithm 3(a) using √n new colors;
```

i. **Runtime:** The first for loop involves inspecting each node, and counting the number of colored neighbors, which requires inspecting every edge twice and every node once. Thus, for the first loop overall runs in $O(|E|+|V|)$ (or $O(m+n)$) time. Creating the induced subgraph involves, at most, copying over the original graph, which requires work proportional to the length of the input, followed by $O(m+n)$ to restrict the graph to the pertinent edges and vertices. Coloring that takes $O(m+n)$ time, which follows from the analysis in part (a). Overall, the runtime of this algorithm ss $O(m+n)$.

ii. **Proof of Correctness:** In a 3-colorable graph, the neighbors of a single node must be 2-colorable, because they all share a neighbor of the same color. Hence, every vertex with more than $\sqrt{n}$ (uncolored) neighbors with its neighbours, gets assigned 3 new colors each iteration. The outer loop iterates at most $\sqrt{n}$ times, because each time it does so, it colors at least $\sqrt{n}$ uncolored nodes, and there are only n nodes total. Each iteration uses 3 colors, and thus the first loop uses at most $3\sqrt{n}$ colors, which is $O(\sqrt{n})$. The second part correctly colors the induced subgraph with at most $\sqrt{n}$ colors, as the first loop reduces the maximum degree in the induced subgraph to at most $\sqrt{n}-1$. Therefore, the algorithm uses at most $O(\sqrt{n})$ colors overall, as desired.