

CSCE 411

Greedy Algorithms and Graphs Practice

1. For a positive integer k and a graph $G = (V, E)$, a k -coloring of G is a function $c : V \rightarrow \{1, \dots, k\}$ such that $\forall \{u, v\} \in E, c(u) \neq c(v)$. That is, a k -coloring assigns each vertex one of k colors such that no two adjacent vertices receive the same color. We say that G is k -colorable if it has a k -coloring.

Let d be a positive integer, and let G be a graph with maximum degree d . Prove that G is $(d + 1)$ -colorable [Hint: Give a greedy algorithm to color G].

Solution: We give the following algorithm to $(d + 1)$ -color G :

GREEDYCOLOR($G = (V, E)$):

For each vertex $v \in V$:

Color v with the first color not already used by one of v 's neighbors.

Correctness: We note that since each vertex v has at most d neighbors, whenever we color a vertex v , there will always be a color in $\{1, \dots, d + 1\}$ which is not used by any of its neighbors. Therefore the algorithm will use at most $(d + 1)$ colors to color the vertices. Furthermore, for any edge $e \in E$, the algorithm will color e 's endpoints different colors. To see this, let $u \in e$ be the endpoint that the algorithm colors first and $v \in e$ be the endpoint that the algorithm colors second. When the algorithm is coloring v , u has already been colored, so the algorithm will choose a color for v which was not used by u .

Since this algorithm correctly finds a $(d + 1)$ -coloring of G , G is $(d + 1)$ -colorable.

2. For a graph $G = (V, E)$ we define the *complement* of G , denoted $\overline{G}$ as

$$\overline{G} := (V, \{\{x, y\} \subseteq V : x \neq y \wedge \{x, y\} \notin E\}).$$

That is, $\overline{G}$ is a graph on the same vertex set with all of the edges which are not in G . Prove that if G is disconnected then $\overline{G}$ is connected.

Solution: We note that a disconnected graph has at least 2 different connected components. Let $u, v \in V$ be arbitrary distinct vertices. We need to show that there exists a path between u and v in $\overline{G}$. We consider two cases:

- Suppose u and v are in different components of G . Then there is an edge between them in $\overline{G}$, so they are connected by the path u, v .
- Suppose u and v are in the same component of G . G has at least one other component, let $w \in V$ be a vertex in a different component than u and v . Then there are edges $\{u, w\}$ and $\{v, w\}$ in $\overline{G}$, so u and v are connected there by the path u, w, v .

3. Prove that for every even positive integer n and every non-negative integer $k < n$, there exists a k -regular graph on n vertices.

Solution: Let n be an even positive integer and k be a non-negative integer less than n . We construct a graph on n vertices as follows:

- Our vertex set is $V = \{0, \dots, n-1\}$.
- Our edge set E includes all edges of the form $\{i, i + \lfloor k/2 \rfloor \bmod n\}$.
- If k is odd, our edge set also includes all edges of the form $\{i, i + n/2 \bmod n\}$.

We need to prove that G is k -regular. We note that for every vertex $x \in V$, for every $1 \leq i \leq \lfloor k/2 \rfloor$, x has an edge for the form $\{x, x + i \bmod n\}$, and an edge of the form $\{x, x - i \bmod n\}$, for $2\lfloor k/2 \rfloor$ edges in total.

If k is even, these are all of x 's edges, and so x has degree $2\lfloor k/2 \rfloor = 2(k/2) = k$, as desired. Otherwise, k is odd, so x has one additional edge, $\{x, x + n/2 \bmod n\}$ (note that since n is even, $x - n/2 \bmod n = x + n/2 \bmod n$, so there is not a separate edge $\{x, x - n/2 \bmod n\}$). Therefore if k is odd, x has degree $2\lfloor k/2 \rfloor + 1 = 2((k-1)/2) + 1 = k$.

4. Recall the job scheduling problem: We are given a set of n jobs J which our machine needs to run. Each job j has two attributes:

- $j.time$ is the time that j takes our machine to complete (measured e.g. in minutes)
- $j.deadline$ is the time by which we are supposed to complete j (measured in minutes since the time when we start running jobs).

We need to select an order for the jobs to run in. In the notes we saw a greedy algorithm for this problem which minimizes the maximum lateness. Suppose that instead we want to maximize the number of jobs completed on-time (i.e. at or before their deadline). Give an efficient algorithm to find an optimal ordering for this objective. Your algorithm should run in $O(n^2)$ time or better.

Solution: By the analysis in the notes of the greedy algorithm for minimizing maximum lateness, we know that if it is possible to order a set of jobs such that they all complete on time (i.e. such that the maximum lateness is 0), then ordering the jobs by increasing deadline will complete them all on time. Thus, finding a solution for this problem can be viewed as choosing a set of jobs to complete on time, sequencing those jobs from earliest to latest deadline, and then sticking the rest of the jobs at the end, where they will not finish on time. We give an algorithm based on this idea which uses a list R to keep track of the jobs it will complete on-time.

MAXIMIZEONTIME(J)

```

 $A \leftarrow$  an array of the jobs sorted from earliest deadline to latest
 $R \leftarrow$  an empty list of jobs
For  $i$  from 1 to  $n$ :
    Add  $A[i]$  to the end of  $R$ 
    If  $\sum_{j \in R} j.time > A[i].deadline$ :
         $x \leftarrow$  the job in  $R$  with the maximum time
        Remove  $x$  from  $R$ 
Add all jobs from  $A$  that are not currently in  $R$  to the end of  $R$ 

```

Let d be the number of jobs that an optimal schedule (one with a maximum number of on-time jobs) for J fails to complete on time. We will prove the correctness of our algorithm by induction on d .

Base Case: If $d = 0$, then it is possible to complete all jobs on time. As noted above, whenever this is possible, ordering by increasing deadline will accomplish it, so in this case, our algorithm will add all jobs to R and return the optimal schedule.

Induction Hypothesis: Suppose that for any set of jobs for which there exists a schedule which completes at most $d - 1$ jobs late, our algorithm finds an optimal schedule.

Induction Step: Let the i 'th iteration of the for loop be the first one where our algorithm removes a job from R and let $A[m]$ be the job it removes. We claim that there is an optimal schedule which completes $A[m]$ late. Let OPT be a schedule with only d jobs completed late. Since this was the first iteration where a job was removed, we know that $\sum_{k=1}^i A[k] > A[k].deadline$. Since the jobs were sorted by deadline, this means that $\sum_{k=1}^i A[k] > A[j].deadline$ for any $j \leq k$. Thus, it is impossible to complete all i of these jobs on-time, so OPT must complete at least one of them late.

Let $A[\ell]$ be a job from among $A[1 : i]$ which OPT completes late. By our choice of m , we know that $A[m].time \geq A[\ell].time$. Thus, if we add $A[\ell]$ to the set of jobs that OPT completes on-time and remove $A[m]$, we get another optimal schedule, OPT' which finishes the same number of jobs on time (because $A[m]$ was the first job our algorithm removed, we know that it's possible to complete all the jobs in $A[1 : m - 1]$ on-time, and because $A[m].time \geq A[\ell].time$, OPT' finishes the jobs it attempts to complete from $A[m : n]$ at or before OPT does). We note that since OPT' does not complete $A[m]$ on time, it is still an optimal schedule for $J \setminus \{A[m]\}$, so the optimal number of jobs to complete on-time for J is the same as the optimal number for $J \setminus \{A[m]\}$. By our IH, our algorithm finds an optimal schedule for $J \setminus \{A[m]\}$ (and after throwing out $A[m]$, running our algorithm on J performs the same steps as running it on $J \setminus \{A[m]\}$), so our algorithm finds an optimal schedule for J .

Finally, we note that the for loop runs $O(n)$ times, and each iteration can be performed in $O(n)$ time (we need to add up to the times of the $O(n)$ jobs in R , and potentially iterate through them to find the job with the max time and remove it. Thus, the overall runtime is $O(n^2)$.

Sidenote: We can improve the runtime to $O(n \log n)$ if we use a priority queue for R instead of just a list. This algorithm is called the Moore-Hodgson algorithm.

5. Let G be a graph with n vertices and at least $n^2/4$ edges. Prove that G has a subgraph G' with minimum degree at least $n/4$ (i.e. $\min_{v \in V(G')} \deg_{G'}(v) \geq n/4$).

Solution: Repeatedly remove a vertex of degree less than $n/4$ until none remain. We remove fewer than $n(n/4)$ edges in this way, so the remaining graph still has at least one edge, meaning we cannot have removed all of the vertices, and every vertex that remains must have degree greater than $n/4$.