# 311 Home Work 5

Ben purdy

May 6, 2020

1. Consider the interval scheduling problem studied in class. Suppose that instead of always selecting the earliest job to finish, we instead select the last job to start that is compatible with all previously selected jobs. Prove that this algorithm yields an optimal solution. What is the running time of this algorithm? Justify your answer

   *Solution.* Selecting the last activity to start that is compatible with all previously selected activities – is really the greedy algorithm but starting from the end rather than the beginning.

   Given a set S = $\{a_1, a_2, \cdots, a_n\}$ of activities. Where $a_i$ is represented as a start time and finish time $a_i = (s_i, f_i)$.

   *Proof.* Suppose we represent $a_i \in S$ as its reverse, that is we create a set S' = $\{a'_1, a'_2, \cdots, a'_n\}$ where for each $a_i \in S = a'_i = (f_i, s_i)$. So there exists optimal solution is equal to $\{a_1, a_2, \cdots, a_n\} \subset S$ which is equal to a $\{a'_1, a'_2, \cdots, a'_n\} \subset S'$. Selecting the last activity to start that is compatible with all previously selected activities, when run on S, gives the same answer as the greedy algorithm from the text. Thus an optimal solution can be obtained.

   $\square$

   $\square$

;

2. Not just any greedy approach to the interval scheduling problem produces a maximum-size set of mutually compatible jobs.

   (a) Give an example to show that the approach of selecting the job of least duration from among those that are compatible with previously selected activities does not work.

   *Solution.*

   let $\{(3,5),(1,4),(4,7)\}$ be a set then
   optimal solution is given by $\{(1,4),(4,7)\}$ but the solution derived from this approach is gives is $\{(3,5)\}$ □

   (b) Give an example to show that the approach of always selecting the compatible job that overlaps the fewest other remaining activities does not work

   *Solution.*
   As a counter example to the optimality of greedily selecting the task that conflicts with the fewest remaining activities, suppose the activity times are
   $\{(-1,1),(2,5),(0,3),(0,3),(0,3),(4,7),(6,9),(8,11),(8,11),(8,11),(10,12)\}$.
   Then, by this greedy strategy, we would first pick $(4, 7)$(4,7) since it only has a two conflicts. However, doing so would ean that we would not be able to pick the only optimal solution of
   $(-1,1)(-1,1),(2,5)(2,5),(6,9)(6,9),(10,12)(10,12)$.

   □

   (c) Give an example to show that the approach of always selecting the compatible remaining job with the earliest start time does not work.

   *Solution.*
   As a counterexample to the optimality of greedily selecting the earliest start times, suppose our activity times are $\{(1,\ 10),\ (2,\ 3),\ (4,\ 5)\}$(1,10),(2,3),(4,5). If we pick the earliest start time, we

will only have a single activity, (1, 10)(1,10), whereas the optimal solution would be to pick the two other activities.

□

*Solution.* □

3. Let $G = (V, E)$ be a directed graph where every edge $e \in E$ has a positive weight w(e) and let $s \in V$ be a specified source node of G. The bottleneck shortest path problem is to find, for every node $u \in V$ , a path P from s to u such that the weight of the minimum-weight edge in P is maximized.

   (a) Give a $O(m+n)$-time algorithm that solves the bottleneck shortest path problem in a DAG. Justify your algorithm and analyze its running time.

**Algorithm 1:** Solution

```
Input: Directed graph and starting node s
Output:Path of minimum maximum path
for each node v in G except s
$v.dist = -\infty$
$s.dist = 0$
Queue q = [s]
while q is not empty and q.peek = v do
    a = q.pop
    for each of a's neighbors do
        child.visit = true
        if child.waight ¡ a.pathwaight then
            child.waight = a.pathwaight
            child.pred = a;
        q.add(child)
    //travel down from final node
    Path = []
    curr = v
    while curr ! = s do
        Path.add(curr)
        curr = curr.pred
    return path
return path;
```

*Proof.* greedy algorithm this algorithm updates the maximum min of each node alone a path. Storing the predictor who updated this node. by updating this predictor there will only be one path from s to v which is the bottle neck shortest path, becuase the predictor waight must be equal to or greater then the the current node.

Using a queue every node is accesses once and every edge is traveled once. After this the algorithm goes back threw and obtains the path at most acessing V nodes. This the running time is O(V+E+V) = O(2V+E) = O(V+E)

□

(b) Give a $O(mlogn)$-time algorithm that solves the bottleneck shortest path problem in an arbitrary directed graph. Justify your algorithm and analyze its running time.

*Solution.*

**Algorithm 2:** Dijkstra's Solution

```
Input: Directed graph and starting node s
Output:Path of minimum maximum path
for each node v in G except s
$v.dist = -\infty$
$s.dist = 0$
S = [] - seen nodes
path = [-∞]
while $S != V$ do
    Choose v in V-S with maximum d[v]
    add v to S
    for each neighbor of v do
        path[w] = max(path[v],min(path[v],c(v,w)))
        update max heap d
return Path;
```

□

*Proof.*
Argue correctness
By choosing the maximum in d[v] we are choosing the maximum node

we have currently seen and using this node to create a path. By updating this path with the min of the maximums weighted nodes at each point we know that we are building a minimum maximum path from s to v

Running time: while accessing n nodes. we update m neighbors and call max heapify which we know runs in logn time. So by big O notation this algorithm runs in O(mlogn) time

□

4. Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets $V_1$ and $V_2$ such that $|V_1|$ and $|V_2$— differ by at most 1. Let $E_1$ be the set of edges that are incident only on vertices in $V_1$, and let $E_2$ be the set of edges that are incident only on vertices in $V_2$. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum weight edge in E that crosses the cut $(V_1, V_2)$, and use this edge to unite the resulting two minimum spanning trees into a single spanning tree. Either argue that the algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.

   *Solution.* Professor Borden algorithm fails

   Consider the graph with 4 vertices: a, b, c, and d. Let the edges be $(a, b), (b, c), (c, d), (d, a)$ with weights 1, 5, 1, and 5 respectively. Let $V_1 = \{a, d\} and V_2 = \{b, c\}$. Then there is only one edge incident on each of these, so the trees we must take on $V_1$ and $V_2$ consist of precisely the edges (a, d) and (b, c), for a total weight of 10. With the addition of the weight 1 edge that connects them, we get weight 11. However, an MST would use the two weight 1 edges and only one of the weight 5 edges, for a total weight of 7. □