Michael Inoue
405171527

Ling
Discussion 1C 2-3:50PM

CS 180

Homework 3

23 October 2019

1 Solution

Algorithm 1: Number of shortest paths between two nodes

```
Input  : Graph $G$ with vertices $V$ and edges $E$, contains nodes $v$ and $w$
Output: Number of shortest paths $v - w$ from $v$ to $w$
1  Initially every node is unlabeled
2  Initially every node's number of shortest paths, $n_s$ is 0
3  Initially, array of layers, $L$ is empty
4  Initially, layer counter $i = 0$
5  Label $v$ as 0
6  Set $v$'s number of paths, $n_s$, to 1
7  foreach Node $s$ in layer $i$ do
8      Consider each edge from $s$: $e = (s, t)$
9      if $t$ is unlabeled then
10         Add $t$ to layer $L[i + 1]$
11         Set $t$'s label to $s$'s label + 1
12         Add $s$'s number of shortest paths, $s_{n_s}$ to $t$'s number of shortest paths, $t_{n_s}$
13     else
14         if $s$'s label + 1 = $t$'s label then
15             Add $s$'s number of shortest paths, $s_{n_s}$ to $t$'s number of shortest paths, $t_{n_s}$
16         end
17     end
18     Increment $i$ by 1
19 end
20 return $w$'s number of shortest paths, $w_{n_s}$
```

We first prove the correctness of the algorithm. We perform a modified version of BFS, starting at our initial node, v, labeling each subsequent node by the minimum edges necessary to reach such a node from v. For each new unlabeled node t discovered by its incident node s, we accomplish this by simply setting its label equal to t's label + 1 (as we traverse one more edge to reach t). We also add the number of paths from s to t to t's number of paths (as every different path that reaches s subsequently should reach t). Now, if such a node t has been discovered by s but is already labeled (meaning it was already discovered), then we only add the number of paths from s to the total number of paths to t if t's label is exactly one more than s's (i.e. its exactly one layer below). This is by the property of the BFS tree we have been constructing: the shortest path between a node at layer i from the initial layer 0 should be a distance i long. Thus, a path through s in layer i and t in layer $i + 1$ should be exactly the distance from our initial node to s + 1, or $i + 1$. This prevents us from counting any edges that would connect to t but yield a longer path, as that would violate the requirement of being a shortest path by definition. Finally, we return the number of shortest paths in w, w_{n_s}. As each shortest path to every node should have been accumulated up to this point, so should have the shortest path to w, w_{n_s}.

As we have accounted for every shortest path to w, we are finished, and our algorithm must yield correct results.

Complexity Analysis

Our algorithm simply performs a modified version of BFS, which runs in $O(m + n)$ time. In addition to the standard procedure of BFS in labeling each node as necessary, and adding each new node to the next layer, we also update the number of shortest paths to both a newly discovered (unlabeled) node as well as a discovered (labeled) node if its label meets a certain criteria. We then return the number of shortest paths from our node w. However, all of these additional operations are $O(1)$, so this algorithm still runs in $O(m + n)$ time.

2 Solution

We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

We use the following theorem, proved in the book on page 86:

Theorem 3.7
Let T be a depth-first search tree, let x and y be nodes in T, and let (x,y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other.

Now, we begin our proof that $G = T$.

Proof. By assumption in the problem statement, T contains every node in G. Thus, we must show that T contains every edge in G to prove that $T = G$.

Suppose, towards a contradiction, that there is some edge $e = (t, v)$ such that $e \in G$ and $e \notin T$. Then by Theorem 3.7, because T is a DFS tree, either t or v are descendants of each other. Suppose, without loss of generality, that v is a descendant of t. Because T is a BFS tree, that means that either t and v were not connected because v was discovered and connected to some node w beforehand. By properties of a BFS tree (as proved in class), this node w must be exactly one layer above t, and w and v must be within the same layer. Because T is a DFS tree, w and v must have been explored before t, as there is some edge between w and v, but not between u and v. But then, as t is an undiscovered, it will be discovered by some node in the same layer as v, x. But as T is a BFS tree, this implies that t is some layer below x, which is a contradiction, as t is an ancestor of v, and thus should be above x and v.

Thus, we conclude that no edge e can exist, and thus, $G = T$ □

3 Solution

Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least $\frac{n}{2}$, then G is connected. Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

Proof. We will prove the negation of the claim. Suppose we have some unconnected graph G. We wish to prove that not all nodes have degree of at least $\frac{n}{2}$. If our graph is unconnected, then we have at least two components, C_1 and C_2. Suppose, towards a contradiction, that every node in each component has degree of $\frac{n}{2}$ or greater. Now, a node $v \in C_1$ or C_2 is connected to at least $\frac{n}{2}$ nodes in C_1 or C_2, respectively. Thus, including v, there are at least $\frac{n}{2} + 1$ nodes in each component. Thus, adding the number of nodes total in each component, we have $\frac{n}{2} + 1 + \frac{n}{2} + 1 = \text{n} + 2$ nodes total, which is a contradiction, as G only has n nodes. Thus, we conclude that not every vertex in G has degree of at least $\frac{n}{2}$.

□

4 Solution

Lemma 4.1. *Let D_i be the difference between the weight carried by truck i and its weight capacity, W in our solution, A. Let D'_i be the difference between the weight carried by truck i and its weight capacity, W in the optimal solution, O. We claim that our solution has the same total difference, or smaller, for each truck than the optimal solution; that is*

$$\sum_{i=1}^{k} D_i \leq \sum_{i=1}^{k} D'_i$$

for all trucks 1 through k if there are remaining packages after the first k trucks have left.

We prove such a claim by induction.

Proof. We prove by induction the following equation:

$$\sum_{i=1}^{k} D_i \leq \sum_{i=1}^{k} D'_i \tag{1}$$

for all $k \in N$. We first check to see if our base case holds, namely when $k = 1$. We see that when $k = 1$, no trucks have left yet. Based on our algorithm, our first truck is packed until the next package doesn't fit (exceeding the weight capacity of the truck). Thus, by definition, the total difference D_1 between the weight capacity of W truck 1 and its filled weight is minimum. Thus, $D_1 \leq D'_1$, and our base case holds.

Now we take equation (1) to be our inductive hypothesis, for some $(k-1) \in \mathbb{N}$. We wish to prove that equation (1) holds true for $k \in \mathbb{N}$. Now, we note that the left-hand side of equation (1) can be rewritten as:

$$\sum_{i=1}^{k-1} D_i + D_k \tag{2}$$

Applying the inductive hypothesis, we know that $\sum_{i=1}^{k-1} D_i \leq \sum_{i=1}^{k-1} D'_i$. Thus, it remains to be shown that $D_k \leq D'_k$ in order to prove equation (1) to be true. For the *kth* truck, as long as there are boxes remaining after its departure, our algorithm packs as many boxes as possible in the truck. Thus, by definition, $D_k \leq D'_k$.

Thus, we conclude that equation (1) is true for all k, so long as there are boxes remaining after the departure of the *kth* truck.

□

Now, we wish to prove that our solution uses the same number of trucks as the optimal solution.

Proof. Let the number of trucks of our solution A be m and let the number of trucks of the optimal solution O be n. We wish to prove that $m = n$. Suppose, for a contradiction, that $m > n$. We note that because there is an *nth* truck that leaves the packaging center, after the $(n-1)$th truck has left, there are packages remaining. Thus, we can apply Lemma 4.1 for the first $(n-1)$ trucks that have left in our solution and the optimal solution. By Lemma 4.1,

$$\sum_{i=1}^{n-1} D_i \leq \sum_{i=1}^{n-1} D'_i. \tag{3}$$

Hence, the total difference between each truck's capacity and the weight its carrying is minimal in the first $(n-1)$ trucks. This implies that in the first $(n-1)$ trucks, the total weight carried in our solution, A, is maximal, and because each package must be delivered in the order it was received at the packaging center, the total number of packages delivered must thereby also be maximal. Now, when the nth truck leaves, we have no more packages remaining at the packaging center. But, as our algorithm A has ensured we have delivered the maximal number of packages in the first $(n-1)$ trucks, if the optimal solution O runs out of packages after the departure of the nth truck, then our algorithm, too, must run out of packages after the departure of the nth truck, as it packs the maximal number of packages remaining in said truck. However, this would mean that the next $(n+1)$th truck that leaves in A would leave with no packages remaining, which is a contradiction, as each truck only leaves when it has a package to deliver. Thus, we have that $m \leq n$. As O is the optimal solution, this means that $m = n$.

Thus, our solution uses the same number of trucks as the optimal solution, and hence is optimal. $\square$

5 Solution

Algorithm 2: Number of shortest paths between two nodes

```
  Input  : List of contestants $C$ each with swimming, biking, and cycling times
  Output: List of students out in order
1 Each contestant $c$ has a swimming time $s_c$, a running time $r_c$, and a biking time $b_c$
2 Perform merge sort on list of candidates $C$ in descending order of their running and biking times
    added together $(r_c + b_c)$
3 return $C$
```

We now prove that our algorithm is optimal. Suppose that we have some optimal schedule O such that O does not necessarily abide by the rules of our algorithm, specifically, that the list or contestants in the schedule is not sorted in descending order of their running and biking times added together.

Now, in that case, we must have some sequence s followed by t such that $r_s + b_s < r_t + b_t$. Now if we swap s and t so that s now follows t in the sequence, we see that t will complete sooner than when it followed s. Moreover, t will get out of the pool first when s used to get out of the pool first. However, as $r_s + b_s < r_t + b_t$, s will finish earlier than t did with the previous unswapped schedule. Thus, we see that our particular swap does not add any extra time, and thus is optimal.

We may repeat this process for each pair (s, t) that does not abide by our ordering criteria, and will thus construct a list of contestants identical to the list we create in our algorithm: a list of contestants in descending order of combined time to run and bike. Because each swap adds no extra time for completion, this process adds no extra time.

Thus, we conclude that our algorithm is optimal.

Complexity Analysis:

Our algorithm simply performs merge sort on a list of contestants and returns said list. Because merge sort on a list with n items runs in $O(n \log n)$, our algorithm, too, runs in $O(n \log n)$.

6 Solution

(a) No (assuming the graph can be cyclical). Such a problem is NP-hard, and thus cannot be solved in polynomial time. One solution is by brute force, searching every possible path and finding the maximal path. However, such an algorithm is inefficient and has a non-polynomial, exponential complexity.

(b) Can you design an algorithm that finds the longest path in a directed acyclic graph (DAG)? (you can use an edge at most once)? If yes, describe the algorithm and analyze its time complexity.

Algorithm 3: Length of longest path in a DAG

```
Input  : DAG $G$ with vertices $V$ and edges $E$
Output: Length of longest path
Initially queue $S$ is empty
Initially every node $v$ is labeled 0
Initially the current longest path $L$ is 0
foreach node $v \in G$ do
    if $v$ has no incoming edges then
        Add $v$ to $S$
    end
end
foreach node $v \in S$ do
    Remove $v$ from $S$
    Consider each edge $e = (v, w)$
    Delete $e$
    Decrement $w$'s number of incoming edges, $e_n$, by 1
    if $v$'s label + 1 > $w$'s label then
        Update $w$'s label to be $v$'s label + 1
        if $w$'s label > $L$ then
            Set $L = w$
        end
    end
    if $w$'s number of incoming edges is 0 then
        Add $w$ to $S$
    end
end
return $L$
```

We first prove the correctness of our algorithm. We see that our algorithm simply performs a modified version of topological sort. As we traverse each edge of our graph from some node v to some other node w, we continually update the distance it takes to get to v, choosing the max between v's current label and w's label + 1. This, along with our method of topological sorting, ensures that each label will be the maximum value that it could be (as by topological sort we must traverse each source node to get to w, we account for every path prior to w and choose the largest of each path by updating w's label accordingly when necessary. We compare this to our current longest path L. The longest path must belong to at least one node. As we repeat this for every node, updating L accordingly, when we return L, it must be our longest path.

Thus, we conclude that our algorithm yields correct results.

Complexity Analysis:

We simply perform a variant of topological sort in this algorithm (which runs in $O(m+n)$, with m edges and n nodes), with the addition of comparing each label as we update it with our current longest path, L. However, this is an $O(1)$ step, so our algorithm still runs in $O(m+n)$.