

CS180 Homework 3

Zhehao Wang 404380075 (Dis 1B)

Apr 16, 2016

1 Strongly connected components in a directed graph

(a) Prove SCC graph is a DAG.

Proof by contradiction: assume that a path s_i, \dots, s_j, s_i exists in the SCC graph, where s_k ($i \leq k \leq j$) are the newly created distinct SCC nodes.

Consider a node p_i in SCC node s_i , there exists a path from p_i to any node p_{i+1} in SCC s_{i+1} , since there exists a node q_i in SCC s_i that has a directed edge to a node q_{i+1} in SCC s_{i+1} , and there exists a path from p_i to q_i and from q_{i+1} to p_{i+1} .

Starting from p_i , apply the above conclusion repeatedly till we reach a node p_j in SCC s_j , and we have that there exists a path from p_i to p_j . Similarly, we have there exists a path from p_j to p_i . Thus the nodes p_i and p_j should belong to the same SCC node which is the combination of SCC nodes s_i and s_j . This contradicts with the nodes being distinct in the assumption. Thus we have the directed SCC graph is acyclic, which makes it a DAG by definition.

(b) The algorithm is given in alg 1.

It follows the hint: we start by calling $getSCC(G)$, the algorithm does a DFS starting from the node v who can reach all nodes in G , and labels the nodes by *reverse* post-order number (which is why we have smallest labeled node rather than largest as suggested by the hint). The node with smallest label is remembered, and we then start from that node, do another DFS to find the nodes that it can reach, build a SCC with those node, then remove them, and start again with the smallest labeled node. The *DFS* function takes additional parameters to tell if this run is doing the initial labeling, or this run is getting a SCC and removing it from the original graph, in case of the latter, several additional parameters are added so that the state is kept during each recursive call.

Algorithm 1 SCC building algorithm

```
1: function DFS( $v$ ,  $do\_label$ ,  $smallest\_node$ ,  $node\_remaining$ ,  $node\_removed$ ,  $SCC\_graph$ )
2:    $v.visited \leftarrow true$ 
3:   if  $do\_label = false$  then
4:      $node\_remaining.remove(v)$ 
5:      $node\_removed.add(v)$ 
6:     if  $v = smallest\_node$  then
7:        $smallest\_node \leftarrow node\_remaining.nextSmallest()$ 
8:   for  $\{i | (v, i) \in E, i.visited = false\}$  do
9:     if  $do\_label = true$  then
10:      DFS( $i$ ,  $do\_label$ ,  $smallest\_node$ ,  $node\_remaining$ ,  $node\_removed$ ,  $SCC\_graph$ )
11:    else
12:      if  $i \in SCC\_graph$  then
13:         $SCC\_graph.addEdge(v, i)$ 
14:      else
15:        DFS( $i$ ,  $do\_label$ ,  $smallest\_node$ ,  $node\_remaining$ ,  $node\_removed$ ,  $SCC\_graph$ )
16:   if  $do\_label$  then
17:      $id \leftarrow label(v)$ 
18:     if  $id < smallest\_node$  then
19:        $smallest\_node \leftarrow v$ 
20: function GETSCC( $G$ )
21:    $smallest\_node \leftarrow nil$ 
22:   DFS( $v$ ,  $true$ ,  $smallest\_node$ ,  $G.nodes$ ,  $[], nil$ )
23:    $smallest\_node\_copy \leftarrow smallest\_node.copy()$ 
24:    $G.resetVisited()$ 
25:    $SCC\_graph \leftarrow nil$ 
26:   while  $G.node\_count > 0$  do
27:      $G.resetVisited()$ 
28:      $node\_removed \leftarrow []$ 
29:     DFS( $smallest\_node$ ,  $false$ ,  $smallest\_node\_copy$ ,  $G.nodes$ ,  $node\_removed$ ,  $SCC\_graph$ )
30:      $smallest\_node \leftarrow smallest\_node\_copy$ 
31:      $SCC\_graph.addNode(node\_removed)$ 
32:   return  $SCC\_graph$ 
```

Time complexity: this algorithm is $O(|E|)$. The initial DFS traversal visits each edge exactly once and is $O(|E|)$. All the subsequent DFSs combined also visits each edge exactly once, and is $O(|E|)$ (This makes the assumption that calls such as deciding if a node's in a SCC (line 12), and adding edges in the resulting SCC graph (line 13) are $O(1)$, which can be achieved by hashing). So the overall complexity is $O(|E|)$.

Correctness: Let $G = (V, E)$ be the original graph, $G' = (V, E')$ and $E = \{\text{edges picked by the initial DFS post order traversal of } G\}$. G' is a DAG. The smallest labeled node r from reverse post order traversal is a sink in G' .

We want to prove that for all $n \in V$, if there exists a path p_i from r to n in G , then there exists a path p_j from n to r in G .

2 Longest path in DAG

(a) algorithm is given in alg 2.

Algorithm 2 Longest path in an unweighted DAG

```

1: function LONGESTPATH( $G$ )
2:    $nodeCount \leftarrow len(V)$ 
3:    $sourceNodes \leftarrow []$ 
4:    $lastNode \leftarrow nil$ 
5:   for  $\{i | i \in V\}$  do
6:     if  $i.inDegree = 0$  then
7:        $sourceNodes.add(i)$ 
8:    $length \leftarrow 0$ 
9:   while  $nodeCount > 0$  do
10:     $newSourceNodes \leftarrow []$ 
11:    for  $\{i | i \in sourceNodes\}$  do
12:      for  $\{v | (i, v) \in E\}$  do
13:         $v.inDegree \leftarrow v.inDegree - 1$ 
14:        if  $v.inDegree = 0$  then
15:           $newSourceNodes.add(v)$ 
16:           $v.from \leftarrow i$ 
17:       $G.remove(i)$ 
18:       $lastNode \leftarrow i$ 
19:       $nodeCount \leftarrow nodeCount - 1$ 
20:       $sourceNodes \leftarrow newSourceNodes$ 
21:       $length \leftarrow length + 1$ 
22:    $node \leftarrow lastNode$ 
23:    $path \leftarrow [lastNode]$ 
24:   while  $node.from \neq nil$  do
25:      $path.push(node.from)$ 
26:      $node = node.from$ 
27:   return  $length, path$ 

```

Time complexity: this iterative algorithm is $O(|E|)$, as it's using an $O(|E|)$ topological sorting algorithm with number of phases it takes remembered.

Correctness: this algorithm does a similar thing as topological sort, except that it counts the phases needed to remove all nodes. The correctness is based on the conclusion that length of longest path in a DAG G is $1 + \text{length of longest path in } G'$, where G' is the remaining graph after all sources in G are removed; this conclusion can be proved by induction.

And as we find new sources, we update the *from* field of the node to keep track of the nodes on the longest path.

(b) algorithm is given in alg 3. Similar with the idea of problem (a), we base the algorithm on topological sort, and label each node. Each time the algorithm removes a source node i , the nodes j that it connects to will have new label $w(j) = \max(w(j), l(i, j) + w(i))$. The largest label in the DAG will be returned as the length of the longest path.

Time complexity: similar with a topological sort, this algorithm is $O(|E|)$, as each edge in the graph will be used exactly once.

Algorithm 3 Longest path in a weighted DAG

```
1: function LONGESTPATH( $G$ )
2:    $nodeCount \leftarrow len(V)$ 
3:    $sourceNodes \leftarrow []$ 
4:    $maxNode \leftarrow nil$ 
5:   for  $\{i | i \in V\}$  do
6:     if  $i.inDegree = 0$  then
7:        $sourceNodes.add(i)$ 
8:        $i.label \leftarrow 0$ 
9:    $maxLength \leftarrow 0$ 
10:  while  $nodeCount > 0$  do
11:     $newSourceNodes \leftarrow []$ 
12:    for  $\{i | i \in sourceNodes\}$  do
13:      for  $\{v | (i, v) \in E\}$  do
14:         $v.inDegree \leftarrow v.inDegree - 1$ 
15:        if  $v.label < i.label + l(i, v)$  then
16:           $v.label \leftarrow i.label + l(i, v)$ 
17:           $v.from \leftarrow i$ 
18:        if  $maxLength < v.label$  then
19:           $maxLength \leftarrow v.label$ 
20:           $maxNode \leftarrow v$ 
21:        if  $v.inDegree = 0$  then
22:           $newSourceNodes.add(v)$ 
23:       $G.remove(i)$ 
24:       $nodeCount \leftarrow nodeCount - 1$ 
25:     $sourceNodes \leftarrow newSourceNodes$ 
26:   $node \leftarrow maxNode$ 
27:   $path \leftarrow [maxNode]$ 
28:  while  $node.from \neq nil$  do
29:     $path.push(node.from)$ 
30:     $node = node.from$ 
31:  return  $maxLength, path$ 
```

Correctness:

Lemma: during the topological sort, every time a node p becomes source, its label will be the length of the longest path starting from any node and ending at p .

This lemma is easily proved by contradiction, as when the node becomes source there won't be any path going to it, so the label of the source will be the length of maximum path that ends at the node. (The labels of all nodes are initialized as 0, so that negative labels won't be kept, and in case of negative labels, we could just ignore the earlier paths that lead to this node.)

The algorithm returns the maximum label of nodes in the graph after all nodes become sources, given the lemma, the algorithm will return the length of the longest path in the graph. And as we update the label with larger values, we update the *from* field of the node to keep track of the longest nodes on the path.

(c) algorithm is given in alg 4. Similar idea as in (b). The algorithm returns a list of jobs sorted by

their labels (defined in the algorithm as the longest time it takes to get the prerequisites of that job done), and a schedule that starts each job at the labeled time minimizes the total time needed to finish all jobs.

Algorithm 4 Weighted DAG job scheduling

```

1: function LONGESTPATH( $G$ )
2:    $nodeCount \leftarrow len(V)$ 
3:    $sourceNodes \leftarrow []$ 
4:   for  $\{i | i \in V\}$  do
5:     if  $i.inDegree = 0$  then
6:        $sourceNodes.add(i)$ 
7:        $i.label \leftarrow 0$ 
8:   while  $nodeCount > 0$  do
9:      $newSourceNodes \leftarrow []$ 
10:    for  $\{i | i \in sourceNodes\}$  do
11:      for  $\{v | (i, v) \in E\}$  do
12:         $v.inDegree \leftarrow v.inDegree - 1$ 
13:         $v.label \leftarrow \max(v.label, i.label + l(i))$ 
14:        if  $v.inDegree = 0$  then
15:           $newSourceNodes.add(v)$ 
16:       $G.remove(i)$ 
17:       $nodeCount \leftarrow nodeCount - 1$ 
18:       $sourceNodes \leftarrow newSourceNodes$ 
19:   return  $sort(v.label)$ 

```

Time complexity: this algorithm is $O(\max(|E|, |V| \log |V|))$. $O(|E|)$ comes from the topological sort and labeling, which will visit each edge exactly once; and $|V| \log |V|$ comes from the sorting by label in the end.

Correctness: the proof is similar as that of (b) except that we don't need to consider the negative label case. A brief description below:

Lemma: during the topological sort, when a node becomes source, its label is the earliest time when the corresponding job can start. This can be easily proved by induction given the label definition in alg 4.

By the lemma, we have that the algorithm provides a schedule that starts a job as soon as it can be started. This schedule is optimal, which can be easily proved by contradiction.

3 Optimal order of files

Algorithm is given in alg 5, which is a greedy algorithm that orders files as $f_{t_1} \dots f_{t_n}$, such that for each $t_i > t_j$, $\frac{p_{t_i}}{l_{t_i}} \geq \frac{p_{t_j}}{l_{t_j}}$.

Time complexity: this algorithm is $O(n \log n)$, where n is the number of files. The calculation of $\frac{p_i}{l_i}$ requires a traversal of array, which is $O(n)$, and the sorting afterwards is $O(n \log n)$, which makes the entire algorithm $O(n \log n)$.

Correctness: proof by the algorithm's greedy choice property and optimal substructure property.

Greedy choice property: there exists an optimal solution $S = f_{t_1} \dots f_{t_n}$, such that $\frac{p_{t_1}}{l_{t_1}}$ is the largest among all jobs.

Algorithm 5 Optimal order of files

```
1: function OPTIMALORDER(files)
2:   weightedFiles  $\leftarrow []$ 
3:   for  $\{i | i \in \text{files}\}$  do
4:     weightedFiles.add( $\frac{p_i}{l_i}$ )
5:   return sort(weightedFiles)
```

Proof: suppose S' is an optimal solution with the sequence $f_{t'_1} f_{t'_2} \dots f_{t'_{k-1}} f_{t_1} f_{t_{k+1}} \dots f_{t_n}$ where $t'_1 \neq t_1$. The average access time of S' is

$$t(S') = \sum_{i=2}^n ((\sum_{j=1}^{i-1} l_{t'_j}) \cdot p_{t'_i})$$

Consider the solution S'' with f_{t_1} and $f_{t_{k-1}}$ swapped, the difference between average access time of S'' and that of S' is

$$t(S') - t(S'') = p_{t_1} * l_{t_{k-1}} - p_{t_{k-1}} * l_{t_1} = (\frac{p_{t_1}}{l_{t_1}} - \frac{p_{t_{k-1}}}{l_{t_{k-1}}}) \cdot l_{t_1} \cdot l_{t_{k-1}} \geq 0$$

Similarly, starting from $t(S'')$, each time we swap f_{t_1} with its previous file, we'll have a smaller or equal access time than before. Thus the optimal solution should contain f_{t_1} as its first element, whose $\frac{p_{t_1}}{l_{t_1}}$ is the largest.

Optimal substructure property: let $S = f_{t_1} \dots f_{t_n}$ be an optimal solution, then $S_1 = f_{t_2} \dots f_{t_n}$ is the optimal solution for the sub problem without f_{t_1} .

Proof by contradiction: assume there's a better solution $S'_1 = f_{t'_2} \dots f_{t'_n}$, $t(S'_1) < t(S_1)$ for the sub problem without f_{t_1} . Then $S' = f_{t_1} S'_1$ has the average access time of $t(S') = t(S'_1) + l_{t_1} * (1 - p_{t_1}) < t(S_1) + l_{t_1} * (1 - p_{t_1}) = t(S)$, which contradicts with S being the optimal solution for the original problem.

With both properties, the greedy algorithm in question is correct.

4 Sorting from SC

Suppose that $SC(p_i, d_i)$ takes in n jobs, and each has a p_i and d_i . SC returns the optimal list of jobs expressed in two arrays *times*, *deadlines*. The sorting algorithm is described in alg 6.

Algorithm 6 Sorting using SC

```
1: function SORT(array)
2:   times, deadlines  $\leftarrow SC(\text{array}, \text{array})$ 
3:   return deadlines
```

Time complexity: SC is $o(n \log n)$, and other calls are $O(1)$, so the overall *sort* is $o(n \log n)$.

Correctness: suppose that the algorithm gives back a deadline array of $S = d_1 \dots d_n$, we want to prove for each $i < j$, $d_i \leq d_j$.

We start by proving that the first element d_1 is a smallest element by contradiction.

Assume we have a smaller element $d_k < d_1$ in S , $1 \leq i \leq n$. Let the maximum lateness of S be $t(S) = \max((\sigma_{i=1}^n p_i) - d_n)$. Consider the sequence S' with d_k and d_{k-1} swapped, let its maximum lateness be $t(S')$. With given input where $p_i = d_i$, $t(S) = \sigma_{i=1}^{n-1} d_i$, $t(S') =$, and either way $t(S') \leq t(S)$. Similarly, starting from S' , we repeatedly switch d_k with its previous element, and can prove $d_1 = \dots = d_{k-1} = d_k$. Thus d_1 is a smallest element.

Consider the array S_1 which is S with d_1 removed, using the above described process we can prove that d_2 is a smallest element in S_1 . We continue until the array S is exhausted, and we have for each $i < j$, $d_i \leq d_j$.