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 $s_i, ..., s_j, s_i$ 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+i} 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.

This algorithm does not follow the hint to the word (which we suspect to be misleading, as explained in the **notes** section below). Instead, the *reverse* post order labeling using DFS and the subsequent DFS traversals starting from the smallest label (we start from smallest as the algorithm does a *reverse* post order labeling) are done in one pass.

To reduce complexity, the algorithm involves bookkeeping which labels all the nodes according to the sequences they are visited (This *index* does not reflect the sequence of subsequent DFS traversals, which follows the reverse post order of the graph), and uses *lowlink* to keep track of the label of the SCC the node's in, which is the smallest index of all nodes in the SCC.

The algorithm is invoked by calling $tarjanSCC(G)$, and should output (line 25) the list of nodes in each SCC.

Algorithm 1 SCC building algorithm (Tarjan)

```
1: function TARJANSCC(G)
2:     index ← 0
3:     stack ← []
4:     for {v ∈ V} do
5:         if v.index = nil then
6:             scc(v)
7:     function SCC(v)
8:         v.index ← index
9:         v.lowlink ← index
10:        index ← index + 1
11:        SCC_nodes ← []
12:        stack.push(v)
13:        v.onStack ← true
14:        for {w|(v, w) ∈ E} do
15:            if w.index = nil then
16:                scc(w)
17:                v.lowlink ← min(v.lowlink, w.lowlink)
18:            else if w.onStack then
19:                v.lowlink ← min(v.lowlink, w.index)
20:        if v.lowlink = v.index then
21:            while w ≠ v do
22:                w ← stack.pop()
23:                w.onStack ← false
24:                SCC_nodes.addCurrent(w)
25:        SCC_nodes.output()
```

Time complexity: this algorithm is $O(|E|)$. The initial labeling DFS traversal visits each edge exactly once and is $O(|E|)$. All the subsequent DFSs combined also visits each edge exactly once, and is $O(|E|)$. So the overall complexity is $O(|E|)$.

Correctness: Let $G = (V, E)$ be the original graph, according to the algorithm description, we want to first prove that for all $r \in V$, by the time DFS finishes r, let the tree built by the DFS be $G' = (V', E')$, if there exists a path p_i in E that starts from r to any $n \in V'$, then n and r belongs to the same SCC. Then we can merge these two nodes, continue with the DFS recursively, and use the same criteria to tell which SCC the next finished node belongs to, until all nodes are finished.

Consider the case when DFS finishes the first node r, E' would contain only one path from the starting point of DFS v to the sink r. Thus any node $v' \in V'$ has a path to r. Given that there exists a path from r to any $n \in V'$ and there exists a path from n to r, r and n should be in the same SCC, and the first part of the conclusion holds. After merging n and r, we face a subproblem of the original problem with 1 less node, which we can solve by continuing with the recursion.

Note: following the hint to the word may not result in a correct algorithm. Consider the graph in 1, with a DFS starting from a, if the DFS chooses the rightmost path first, we have a post order traversal order of a, b, d, e, c; according to the hint, we then start from c, do a DFS, which will lead us to think that c, b, d, e are in an SCC, which is not the case here.

The difference between the hint and Tarjan's algorithm as given above, seems to be that Tarjan's

algorithm essentially does the first labeling DFS run and subsequent DFS runs in one pass (in this example, the nodes c, b will be taken out as a SCC before d is explored.)

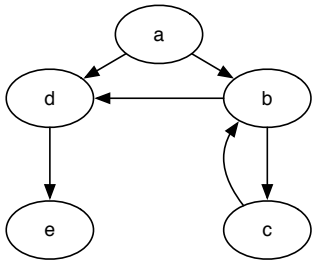

Figure 1: Problematic case with just following the hint

2 Longest path in DAG

(a) algorithm is given in alg 2. As we find new sources, we update the *from* field of the node to keep track of the nodes on the longest path.

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

Correctness: this algorithm counts the phases needed to remove all nodes in a topological sort. The correctness is based on the conclusion that length of longest path in a DAG G is 1 + 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.

Base case obviously holds for graphs with only one node, or two nodes with one directed edge between them.

Induction hypothesis: conclusion holds for any DAGs that takes n phases in a topological sort, and the longest path is $n - 1$.

Induction case holds for graphs that take $n+1$ phases. Proof by contradiction: assume there's a longer path than n (at least $n + 1$), the sub graph with all sources removed need to have length at least n, which contradicts with the induction hypothesis.

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

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

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.)

Algorithm 2 Longest path in an unweighted DAG

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

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.

(c) algorithm is given in alg 4. Similar as (b), the algorithm labels jobs, and returns a list of jobs sorted by their labels (defined as the longest time it takes to get the prerequisites of that job done). A schedule that starts each job at the labeled time minimizes the total time needed to finish all jobs.

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.

Algorithm 3 Longest path in a weighted DAG

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

3 Optimal order of files

Algorithm is given in alg 5, which is a greedy algorithm that orders files as $f_{t_1}...f_{t_n}$, 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 $S = f_{t_1}...f_{t_n}$, such that $\frac{p_{t_1}}{l_{t_1}}$ is the largest among all jobs.

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

Algorithm 4 Weighted DAG job scheduling

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

Algorithm 5 Optimal order of files

```
1: function OPTIMALORDER(files)
2:     weightedFiles ← []
3:     for {i|i ∈ files} do
4:         weightedFiles.add(p_i / l_i)
5:     return sort(weightedFiles)
```

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

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 $S = f_{t_1}...f_{t_n}$ be an optimal solution, then $S_1 = f_{t_2}...f_{t_n}$ is the optimal solution for the sub problem without f_{t_1}.

Proof by contradiction: assume there's a better solution $S'_1 = f_{t'_2}...f_{t'_n}$, $t(S'_1) < t(S_1)$ 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 ← SC(array, 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 $S = d_1...d_n$, 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.

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 $p_i = d_i$, $t(S) = \sigma_{i=1}^{k-1} d_i$, $t(S') = \sigma_{i=1}^{k-2} d_i + d_k$, and $t(S) - t(S') = d_{k-1} - d_k \geq 0$. [1]

Similarly, starting from S', we repeatedly switch d_k with its previous element, and can prove $t(S) \geq ... \geq t(S_{k-1}) \geq t(S_k)$. Thus for SC to provide the optimal solution, the first element d_1 should be 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 $i < j$, $d_i \leq d_j$.

[1] With SC being a blackbox, this given input only seems to guarantee the last element in the returned array has the largest deadline, and does not have other properties. A better solution would be to carefully craft a p_i array to give to the input. An ideal p_i array would be $p_i = d_{i+1} - d_i$, where d_i array is sorted, however this array can only be known after we have the sorting result...