# 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. It follows the bullets described in the hint: we start by calling $getSCC(G)$, the algorithm does a DFS starting from any node 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 see which nodes it can reach, and build a SCC.

**Time complexity:** $O(|E|)$

**Correctness:**

**2** Longest path in DAG

(a) algorithm is given in alg **??**.

**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 + 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.

(b) algorithm is given in alg **??**. 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.

**Correctness:**

**Lemma:** during the topological sort, every time a node $p$ becomes source, its label will be the 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.

(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.

**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 $f_{t_1}...f_{t_n}$, 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

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

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.

**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.

Suppose we have a smallest element $d_k$ in $S$, $d_k \leq d_i, 1 \leq i \leq n$. Let the total lateness of $S$ be $t(S)$. Consider the sequence $S'$ with $d_k$ and $d_{k-1}$ swapped, its difference in lateness with $t(S)$ is $t(S) - t(S') = \sigma_{i=1}^{k-1} d_i - \sigma_{i=1}^{k-2} d_i - d_k = d_{k-1} - dk \geq 0$, and we have $t(S) \geq t(S')$. Since $t(S)$ is given back by $SC$, and satisfies $t(S) \leq t(S_i)$, where $S_i$ is any schedule, we have $t(S) = t(S')$ and $d_{k-1} = dk$.

Similarly, starting from $S'$, we repeatedly switch $d_k$ with its previous element, and can prove $d_1 = ... = d_{k-1} = d_k$. 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 $i < j$, $d_i \leq d_j$.

**Algorithm 1** SCC building algorithm

```
function DFS(v, do_label, smallest_node, node_remaining, node_removed, SCC_graph)
    v.visited ← true
    if do_label = false then
        node_remaining.remove(v)
        node_removed.add(v)
        if v = smallest_node then
            smallest_node ← node_remaining.nextSmallest()
    for {i|(v, i) ∈ E, i.visited = false} do
        if do_label = true then
            DFS(i, do_label, smallest_node, node_remaining, node_removed, SCC_graph)
        else
            if i ∈ SCC_graph then
                SCC_graph.addEdge(v, i)
            else
                DFS(i, do_label, smallest_node, node_remaining, node_removed, SCC_graph)
    if do_label then
        id ← label(v)
        if id < smallest_node then
            smallest_node ← v
function getSCC(G)
    smallest_node ← nil
    DFS(G.firstNode(), true, smallest_node, G.nodes, [], nil)
    smallest_node_copy ← smallest_node.copy()
    G.resetVisited()
    SCC_graph ← nil
    while G.node_count > 0 do
        G.resetVisited()
        node_removed ← []
        DFS(smallest_node, false, smallest_node_copy, G.nodes, node_removed, SCC_graph)
        smallest_node ← smallest_node_copy
        SCC_graph.addNode(node_removed)
    return SCC_graph
```

**Algorithm 2** Longest path in an unweighted DAG

```
function longestPath(G)
    nodeCount ← len(V)
    sourceNodes ← []
    for {i|i ∈ V} do
        if i.inDegree = 0 then
            sourceNodes.add(i)
    length ← 0
    while nodeCount > 0 do
        newSourceNodes ← []
        for {i|i ∈ sourceNodes} do
            for {v|(i, v) ∈ E} do
                v.inDegree ← v.inDegree − 1
                if v.inDegree = 0 then
                    newSourceNodes.add(v)
            G.remove(i)
            nodeCount ← nodeCount − 1
        sourceNodes ← newSourceNodes
        length ← length + 1
    return length
```

**Algorithm 3** Longest path in a weighted DAG

```
function longestPath(G)
    nodeCount ← len(V)
    sourceNodes ← []
    for {i|i ∈ V} do
        if i.inDegree = 0 then
            sourceNodes.add(i)
            i.label ← 0
    maxLength ← 0
    while nodeCount > 0 do
        newSourceNodes ← []
        for {i|i ∈ sourceNodes} do
            for {v|(i, v) ∈ E} do
                v.inDegree ← v.inDegree − 1
                v.label ← max(v.label, i.label + l(i, v))
                if maxLength < v.label then
                    maxLength ← v.label
                if v.inDegree = 0 then
                    newSourceNodes.add(v)
            G.remove(i)
            nodeCount ← nodeCount − 1
        sourceNodes ← newSourceNodes
    return maxLength
```

**Algorithm 4** Weighted DAG job scheduling

```
function LONGESTPATH(G)
    nodeCount ← len(V)
    sourceNodes ← []
    for {i|i ∈ V} do
        if i.inDegree = 0 then
            sourceNodes.add(i)
            i.label ← 0
    while nodeCount > 0 do
        newSourceNodes ← []
        for {i|i ∈ sourceNodes} do
            for {v|(i, v) ∈ E} do
                v.inDegree ← v.inDegree − 1
                v.label ← max(v.label, i.label + l(i))
                if v.inDegree = 0 then
                    newSourceNodes.add(v)
            G.remove(i)
            nodeCount ← nodeCount − 1
        sourceNodes ← newSourceNodes
    return sort(v.label)
```

**Algorithm 5** Optimal order of files

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

**Algorithm 6** Sorting using SC

```
function SORT(array)
    times, deadlines ← SC(array, array)
    return deadlines
```