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.

There exists a path from any node p_i in SCC s_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, q_{i+1} to p_{i+1}.

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

(b) The algorithm is given in alg 1.

Time complexity: $O(|E|)$

Correctness:

2 Longest path in DAG

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

Time complexity: this algorithm is $O(|E|)$.

Correctness: this algorithm does a similar thing as topological sort, except that it counts the steps needed to remove all the nodes, instead of labeling each removed nodes. The recursive idea is 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.

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

Time complexity: this algorithm is $O(|E|)$.

Correctness:

(c) algorithm is given in alg 4. Similar idea as in (b).

Time complexity: this algorithm is $O(max(|E|, |V| \log |V|))$.

Correctness:

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 $t_i > t_j$, $l_{t_i} \cot p_{t_i} \leq l_{t_j} \cot p_{t_j}$.

Time complexity: this algorithm is $O(n \log n)$, where n is the number of files.

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 $l_{t_1} \cot p_{t_1}$ 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

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

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 $l_{t_1} \cot p_{t_1}$ is the smallest.

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 takes in n jobs, and each has its associated p_i and d_i. SC returns the optimal list of jobs. The sorting algorithm is described in alg **??**.

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 an array $S = d_1...d_n$, 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 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|(v, i) ∈ 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.length ← 0
    maxLength ← min_real
    while nodeCount > 0 do
        newSourceNodes ← []
        for {i|i ∈ sourceNodes} do
            for {v|(v, i) ∈ E} do
                v.inDegree ← v.inDegree − 1
                v.weight ← max(v.weight, i.weight + w(v, i))
                if maxLength < v.weight then
                    maxLength ← v.weight
                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.length ← 0
    while nodeCount > 0 do
        newSourceNodes ← []
        for {i|i ∈ sourceNodes} do
            for {v|(v, i) ∈ E} do
                v.inDegree ← v.inDegree − 1
                v.weight ← max(v.weight, i.weight + w(v, i))
                v.length
                if v.inDegree = 0 then
                    newSourceNodes.add(v)
            G.remove(i)
            nodeCount ← nodeCount − 1
        sourceNodes ← newSourceNodes
    return sort(v.weight)
```

Algorithm 5 Optimal order of files

```
function optimalOrder(files)
    for {i|i ∈ files} do
        weightedFiles.add(i.probability · i.length)
    return sort(weightedFiles)
```

Algorithm 6 Sorting using SC

```
function sort(array)
    jobs.deadlines ← array
    jobs.processingTimes ← array
    jobs ← SC(jobs)
    return jobs.deadlines
```