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:

Algorithm 1 SCC building algorithm

```
1: function DFS(v, do_label, smallest_node, node_remaining, node_removed, SCC_graph)
2:     v.visited ← 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 ← node_remaining.nextSmallest()
8:     for {i|(v, i) ∈ 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 ∈ 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 ← label(v)
18:        if id < smallest_node then
19:            smallest_node ← v
20: function GETSCC(G)
21:    smallest_node ← nil
22:    DFS(G.firstNode(), true, smallest_node, G.nodes, [], nil)
23:    smallest_node_copy ← smallest_node.copy()
24:    G.resetVisited()
25:    SCC_graph ← nil
26:    while G.node_count > 0 do
27:        G.resetVisited()
28:        node_removed ← []
29:        DFS(smallest_node, false, smallest_node_copy, G.nodes, node_removed, SCC_graph)
30:        smallest_node ← smallest_node_copy
31:        SCC_graph.addNode(node_removed)
32:    return SCC_graph
```

Algorithm 2 Longest path in an unweighted DAG

```
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:     length ← 0
8:     while nodeCount > 0 do
9:         newSourceNodes ← []
10:        for {i|i ∈ sourceNodes} do
11:            for {v|(v, i) ∈ E} do
12:                v.inDegree ← v.inDegree − 1
13:                if v.inDegree = 0 then
14:                    newSourceNodes.add(v)
15:            G.remove(i)
16:            nodeCount ← nodeCount − 1
17:        sourceNodes ← newSourceNodes
18:        length ← length + 1
19:    return length
```

Algorithm 3 Longest path in a weighted DAG

```
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.length ← 0
8:     maxLength ← min_real
9:     while nodeCount > 0 do
10:        newSourceNodes ← []
11:        for {i|i ∈ sourceNodes} do
12:            for {v|(v, i) ∈ E} do
13:                v.inDegree ← v.inDegree − 1
14:                v.weight ← max(v.weight, i.weight + w(v, i))
15:                if maxLength < v.weight then
16:                    maxLength ← v.weight
17:                if v.inDegree = 0 then
18:                    newSourceNodes.add(v)
19:            G.remove(i)
20:            nodeCount ← nodeCount − 1
21:        sourceNodes ← newSourceNodes
22:    return maxLength
```

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.length ← 0
8:     while nodeCount > 0 do
9:         newSourceNodes ← []
10:        for {i|i ∈ sourceNodes} do
11:            for {v|(v, i) ∈ E} do
12:                v.inDegree ← v.inDegree − 1
13:                v.weight ← max(v.weight, i.weight + w(v, i))
14:                v.length
15:                if v.inDegree = 0 then
16:                    newSourceNodes.add(v)
17:            G.remove(i)
18:            nodeCount ← nodeCount − 1
19:        sourceNodes ← newSourceNodes
       return sort(v.weight)
```