Homework 9

Yu-Ju Teng Ling-Hsuan Chen

Problem 1

1. (7.16 modified)

   (a) Run the strongly connected components algorithm (the original version) on the directed graph shown in Figure 1. When traversing the graph, the algorithm should follow the given DFS numbers (from 9 down to 1). Show the *High* values as computed by the algorithm in each step.

   (b) Add the edge $(6, 5)$ to the graph and discuss the changes this makes to the execution of the algorithm.

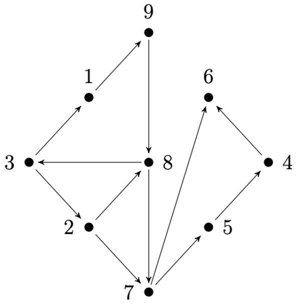
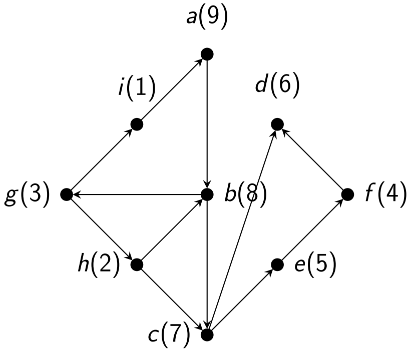


Figure 1: A directed graph with DFS numbers (from 9 down to 1)

Problem 1 (a)

Change the DFS numbers to the corresponding alphabetical order to avoid confusion.



Problem 1 (a)

When running SCC(v), we do as the following:

1. When encountering an unvisited vertex w, we run SCC(w) and keep exploring.

2. When encountering a visited vertex w' and w' does not belong to any component, then we should consider whether w' has a higher leader than v's. If so, update $v.high$ to $w'.DFS\_Number$.

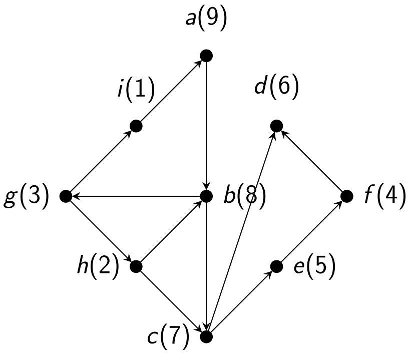
3. When exhausting all edges, we first check whether v is the leader of itself.

   3.1 If so, a SCC was found.

   3.2 Otherwise we backtrack and assign v's leader to vertices along the way if v's leader is higher.

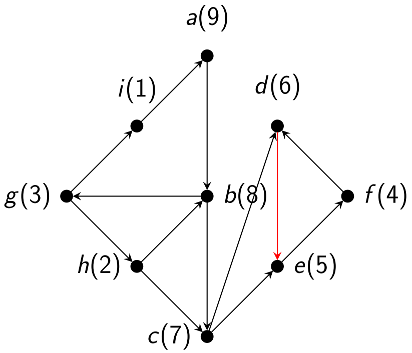
Problem 1 (a)

Vertex	a	b	c	d	e	f	g	h	i
DFS_Number	9	8	7	6	5	4	3	2	1
a	9	-	-	-	-	-	-	-	-
b	9	8	-	-	-	-	-	-	-
c	9	8	7	-	-	-	-	-	-
d	9	8	7	6	-	-	-	-	-
c	9	8	7	6	-	-	-	-	-
e	9	8	7	6	5	-	-	-	-
f	9	8	7	6	5	4	-	-	-
e	9	8	7	6	5	4	-	-	-
c	9	8	7	6	5	4	-	-	-
b	9	8	7	6	5	4	-	-	-
g	9	8	7	6	5	4	3	-	-
h	9	8	7	6	5	4	3	8	-
g	9	8	7	6	5	4	8	8	-
i	9	8	7	6	5	4	8	8	9
g	9	8	7	6	5	4	9	8	9
b	9	9	7	6	5	4	9	8	9
a	9	9	7	6	5	4	9	8	9
Component_Num	5	5	4	1	3	2	5	5	5



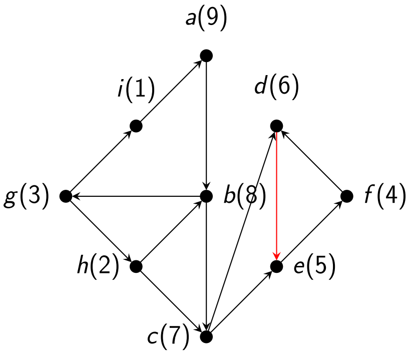
Problem 1 (b)

Add the edge (6, 5) (=(d,e)) to the graph and discuss the changes.



Problem 1 (b)

Vertex	a	b	c	d	e	f	g	h	i
DFS_Number	9	8	7	6	5	4	3	2	1
a	9	-	-	-	-	-	-	-	-
b	9	8	-	-	-	-	-	-	-
c	9	8	7	-	-	-	-	-	-
d	9	8	7	6	-	-	-	-	-
e	9	8	7	6	5	-	-	-	-
f	9	8	7	6	5	6	-	-	-
e	9	8	7	6	6	6	-	-	-
ⓓ	9	8	7	6	6	6	-	-	-
ⓒ	9	8	7	6	6	6	-	-	-
b	9	8	7	6	6	6	-	-	-
g	9	8	7	6	6	6	3	-	-
h	9	8	7	6	6	6	3	8	-
g	9	8	7	6	6	6	8	8	-
i	9	8	7	6	6	6	8	8	9
g	9	8	7	6	6	6	9	8	9
b	9	9	7	6	6	6	9	8	9
ⓐ	9	9	7	6	6	6	9	8	9
Component_Num	3	3	2	1	1	1	3	3	3

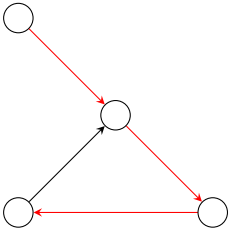


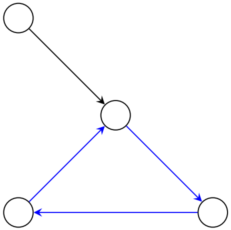
Problem 2

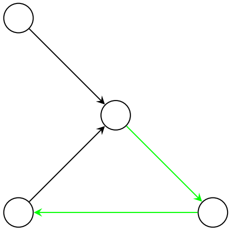
2. (7.88) Let $G = (V, E)$ be a directed graph, and let T be a DFS tree of G. Prove that the intersection of the edges of T with the edges of any strongly connected component of G form a subtree of T.

Problem 2

Simple example:


(a) DFS tree


(b) SCC edges


(c) intersection

Problem 2

Proof by contradiction:

Suppose the intersection are two subtrees T_1 and T_2. Because T_1 and T_2 are in the same strongly connected component, according to the property of SCC, there must be an edge from T_1 to T_2 and an edge from T_2 to T_1.
No matter which subtree the DFS procedure reaches first, it will finally go through the edge which connects T_1 and T_2 and visit the other subtree. Then the DFS tree T must contain that edge but it clearly doesn't. Contradiction.

Problem 3

3. We have discussed in class the idea of using DFS to find an augmenting path (if one exists) in a network with some given flow. Please present the algorithm in suitable pseudocode.

Problem 3

```
**Algorithm AugmentingPath**$(G(V, E), s, t)$
  **begin**
    $Temp\_S$, $Result\_S$ := empty stack;
    $found$ := false;
    **AugmentingPathDFS**$(G, s)$;
    **if** $found$ **then**
      $ResultPath\_S$ := empty stack;
      $post\_v$ := $Result\_S$.pop();
      **while** $Result\_S \neq$ empty **do**
        $current\_v$ := $Result\_S$.pop();
        $ResultPath\_S$.push(edge($current\_v$, $post\_v$));
        $post\_v$ := $current\_v$;
      **while** $ResultPath\_S \neq$ empty **do**
        print $ResultPath\_S$.pop();
    **else**
      print "Augmenting path does not exist.";
  **end**
```

Problem 3

```
Algorithm AugmentingPathDFS($G(V, E), v$)
    begin
        mark $v$;
        $Temp\_S$.push($v$);
        if $v = t$ then
            // reach destination, meaning find the path
            $ResultS := TempS$;
            $found := true$;
        for each edge $(v, w) \in E$ do
            if $found$ then
                break;
            if $f(v, w) < c(v, w)$ or $f(w, v) > 0$ then
                if $w$ is unmarked then
                    AugmentingPathDFS($G, w$);
        $Temp\_S$.pop();
    end
```

Problem 4

4. Consider designing an algorithm by dynamic programming to determine the length of a longest common subsequence of two strings (sequences of letters). For example, "abbcc" is a longest common subsequence of "abcabcabc" and "aaabbbccc", and so is "abccc".

   (a) Formulate the solution using recurrence relations.

   (b) Present the algorithm in suitable pseudocode, based on the previous recursive formulation. What is the time complexity of your algorithm?

Problem 4 (a)

Let $LCS(A, B)$ be the length of a longest common subsequence of strings A and B.

Let string $A = A' + X$ (X is a single character), string $B = B' + Y$ (Y is a single character). Therefore,

$$LCS(A, B) =$$

$$\begin{cases} 0 & \text{if } A \text{ or } B \text{ is an empty string} \\ LCS(A', B') + 1 & \text{if } X = Y \\ \max(LCS(A', B), LCS(A, B')) & \text{if } X \neq Y \end{cases}$$

Problem 4 (b)

Assume string A and B start the index from 1:

```
Algorithm calculateLCS($A, B$)
    begin
        $m, n :=$ len($A$), len($B$);
        initialize $LCS[m+1][n+1]$ with 0; // base case
        for $i := 1$ to $m$ do
            for $j := 1$ to $n$ do
                if $A[i] = B[j]$ then
                    $LCS[i][j] := LCS[i-1][j-1] + 1$;
                else
                    $LCS[i][j] := \max(LCS[i-1][j], LCS[i][j-1])$;
        return $LCS[m][n]$;
    end
```

Time complexity: $O(mn)$.

Problem 5

5. The cost of finding a key value in a binary search tree is linearly proportional to the depth/level of the node where the key value is stored, with the root considered to be at level 0. Obviously, for a key value that is known to be looked up more frequently, it is better stored in a node at a smaller level.

   Consider designing by dynamic programming an algorithm that, given the look-up frequencies of n key values, constructs an optimal binary search tree that will incur the least total cost for performing all the look-ups.

   (a) Formulate the solution using recurrence relations; let $F[1..n]$ be the look-up frequencies of the n key values $K[1..n]$, which are in sorted order.

   (b) Present the algorithm in suitable pseudocode, based on the previous recursive formulation. What is the time complexity of your algorithm?

Problem 5 (a)

Among all possible subtrees, find the one with minimal cost.

For each subtree, its cost is the sum of (1) left child, (2) right child, and (3) the frequency of all nodes in the subtree since every level of each point is increasing by 1 due to the new added root.

Let $OPTcost(i, j)$ be the minimal cost of the subtree containing items from i to j.

Making each node as root r, try to find the one which can provide the minimal cost.

$$OPTcost(i,j) =$$

$$\begin{cases} 0 & \text{if } i > j \\ F[i] & \text{if } i = j \\ \min_{i \le r \le j} \{optcost(i, r-1) + optcost(r+1, j) + \sum_{k=i}^{j} F[k]\} & \text{otherwise} \end{cases}$$

Problem 5 (b)

```
function $\textsc{ConstructOptimalBST}(K, F, n)$
    // initialization
    initialize $cost[n+1][n+1]$ with $\infty$;
    initialize $root[n+1][n+1]$ with 0;
    for $i = 1$ to $n$ do
        $cost[i][i] := F[i]$
        $root[i][i] := i$
// continue ...
```

Problem 5 (b)

```
// $L$ = number of children in tree
for $L = 2$ to $n$ do
    for $i = 1$ to $n - (L - 1)$ do
        $j := i + L - 1$
        $freqSum := FreqSum(F, i, j)$ // calculate $\sum_{k=i}^{j} F[k]$
        for r = i to j do
            $c := freqSum$
            if $r > i$ then
                $c := c + cost[i][r-1]$
            if $r < j$ then
                $c := c + cost[r+1][j]$
            if $c < cost[i][j]$ then
                $cost[i][j] := c$
                $root[i][j] := r$
return $cost[1][n]$ , $BuildTree(K, root, 1, n)$
```

Problem 5 (b)

```
function FreqSum($F, i, j$)
    $freq\_sum := 0$
    for k = i to j do
        $freq\_sum := freq\_sum + F[k]$
    return $freq\_sum$
function BuildTree($K, R, i, j$)
    if $i > j$ then
        return null;
    $root := K[R[i][j]]$
    $root.left := BuildTree(K, R, i, R[i][j] - 1)$
    $root.right := BuildTree(K, R, R[i][j] + 1, j)$
    return $root$
```

Time complexity: $O(n^3)$.