Homework 9

蘇俊杰、劉韋成、曾守瑜

Problem1

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

(b) Add the edge $(6, 8)$ to the graph and discuss the changes this makes to the execution of the algorithm.

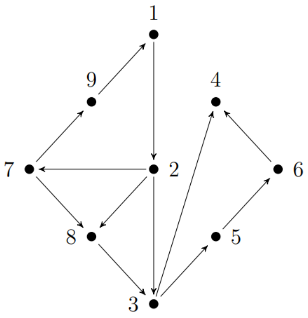


Figure 1: A directed graph with DFS numbers

Problem1(a)

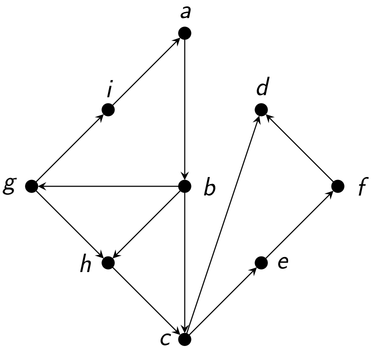
Algorithm $\text{STRONGLY\_CONNECTED\_COMPONENTS}(G, n)$
  for every vertex v of G **do**
    $v.DFS\_Number := 0;$
    $v.Component := 0;$
  $Current\_Component := 0;$ $DFS\_N := n;$
  while $v.DFS\_Number = 0$ for some v **do**
    $SCC(v);$

procedure $\text{SCC}(v)$
  $v.DFS\_Number := DFS\_N;$
  $DFS\_N := DFS\_N - 1;$
  insert v into $Stack$;
  $v.High := v.DFS\_Number;$
  for all edges (v, w) **do**

Problem1(a)

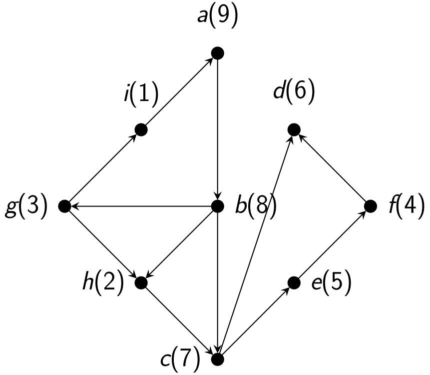
if $w.DFS\_Number = 0$ **then**
    $SCC(w)$;
    $v.High := \max(v.High, w.High)$;
else if $w.DFS\_Number > v.DFS\_Number$ and $w.Component = 0$ **then**
    $v.High := \max(v.High,\ w.DFS\_Number)$;
if $v.High = v.DFS\_Number$ **then**
    $Current\_Component := Current\_Component + 1$;
    repeat
        remove x from the top of $Stack$;
        $x.Component := Current\_Component$;
    until $x = v$

Problem1(a)

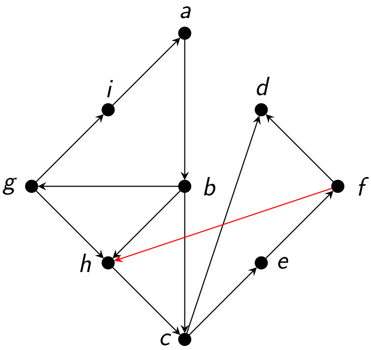


Problem1 (a)

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

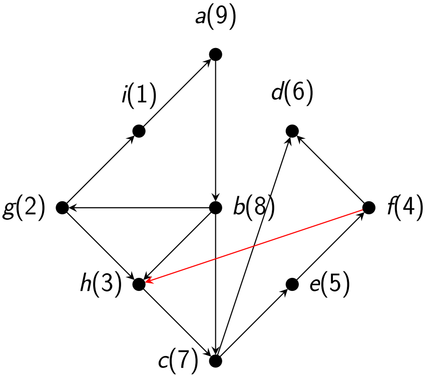


Problem1 (b)



Problem1 (b)

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

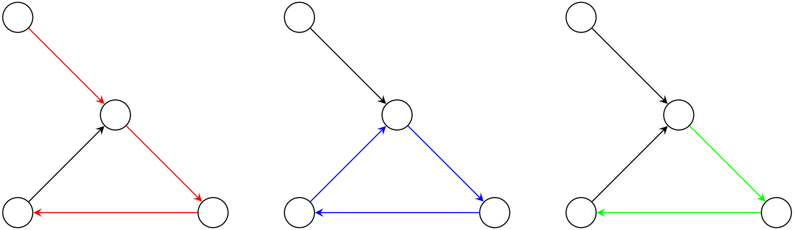


Problem2

(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.

Problem2

Simple example



(a) DFS tree **(b)** SCC edges **(c)** intersection

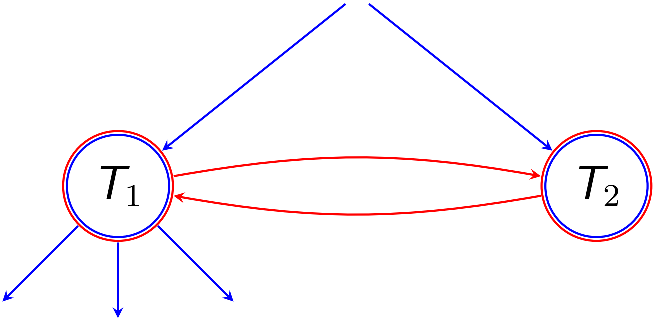
Problem2

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 a path from T_1 to T_2 and a path from T_2 to T_1.
No matter which subtree the DFS procedure reaches first, it will finally go through the path which connects T_1 and T_2 and visit the other subtree. Then the DFS tree T must contain that path but it clearly doesn't. Contradiction.

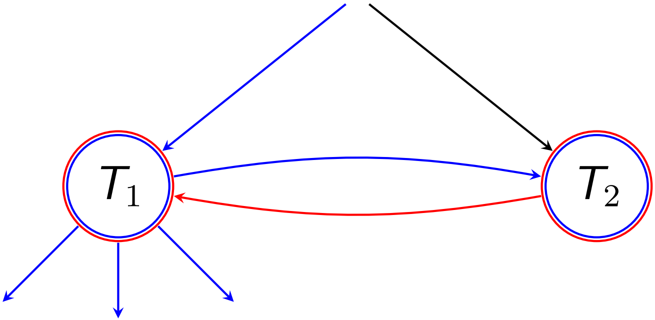
Problem2

SCC
DFS tree T



Problem2

SCC
DFS tree T



Problem3

Consider the algorithm discussed in class for determining the strongly connected components of a directed graph. Is the algorithm still correct if we replace the line "$v.high := \max(v.high, w.DFS\_Number)$" by "$v.high := \max(v.high, w.high)$"? Why? Please explain.

Problem3(cont'd)

Algorithm

function STRONGLY_CONNECTED_COMPONENTS(G, n)
  for every vertex v of G **do**
    $v.DFS\_Number := 0;$
    $v.Component := 0;$
  $Current\_Component := 0; DFS\_N := 0;$
  while $v.DFS\_Number = 0$ for some v **do**
    $SCC(v)$

Problem3(cont'd)

Algorithm

```
procedure SCC(v)
    v.DFS_Number := DFS_N;
    DFS_N := DFS_N−1;
    insert v into Stack;
    v.High := v.DFS_Number;
    for all edges (v, w) do
        if w.DFS_Number = 0 then
            SCC(w);
            v.High :=max(v.High, w.High)
        else if w.DFS_Number > v.DFS_Number and
    w.Component = 0 then
            v.High := max(v.High, w.DFS_Number)
    if v.High = v.DFS_Number then
        CurrentComponent := CurrentComponent + 1;
        repeat
            remove x from the top of Stack;
            x.component := CurrentComponent
        until x = v
```

Problem3(cont'd)

Still correct.
Only when line 10 is True(We look at a vertex w that we have reached before and it does not belong to any SCC yet), we can reach line 11.
At this moment, if $w.DFS\_Number$ and $w.High$ are the same then this case has no impact. If they are different, the only case is $w.DFS\_Number < w.High$, indicating that v and w are in the same SCC. Since we will finally return to the vertex that is the leader of this SCC, its $High$ will set to $\max(v.High, w.High)$, which is exactly its DFS_Number(Because the propagation of the $High$ value of the SCC leader).
Hence when we reach line 11, we can argue that the algorithm is still correct if we replace line 11 by "$v.high := \max(v.high, w.high)$" .
If you have trouble understanding this, draw a figure and trace the code!

Problem4

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?

Problem4

給定兩個 sequence，求出這兩個 sequence 的最長共同子序列

Problem4

A = A' + "x"
B = B' + "y"
LCS(A, B) = 0 if A or B are empty
LCS(A, B) = LCS(A', B') if x = y;

Problem4

A = A' + "x"
B = B' + "y"
LCS(A, B) = 0 if A or B are empty
LCS(A, B) = LCS(A', B') if x = y

otherwise...
LCS(A, B) = max(LCS(A', B), LCS(A, B'), LCS(A', B')) if x $\neq$ y;

Problem4

開一個二維陣列
每格代表兩個子字串的最長共同子序列長度

	-	a	b	c	a	b	c	a	b	c
-	0	0	0	0	0	0	0	0	0	0
a	0									
a	0									
a	0									
b	0									
b	0									
b	0									
c	0									
c	0									
c	0									

Problem4

首先先看 abcabcabc 與 a 能夠迸出什麼火花

	-	a	b	c	a	b	c	a	b	c
-	0	0	0	0	0	0	0	0	0	0
a	0									

從最左邊開始看

	-	a	b	c	a	b	c	a	b	c
-	0	0	0	0	0	0	0	0	0	0
a	0									

相當於找 a 與 a 的最長共同子序列

Problem4

若有兩個字串 A 與 B，兩個字串的結尾相同，比如都是 x
那麼很明顯，A 與 B 的最長共同子序列應該是 A-1 與 B-1（去掉結尾字元）這兩個子字串的最長共同子序列，再加上原本被去掉的 x

在剛剛的例子，a 與 a 都有相同結尾
所以這格應該填上 (空字串) 與 (空字串) 的結果 +1
也就是 1

	-	a	b	c	a	b	c	a	b	c
-	0	0	0	0	0	0	0	0	0	0
a	0	1								

Problem4

若有兩個字串 A 與 B，兩個字串的結尾不同，比如 x 與 y
最長共同子序列不可能同時包含這兩個，因為這兩個 x y 都在結尾
所以就去檢查 A-1 與 B 的，以及 A 與 B-1 的，以及 A-1 與 B-1

	-	a	b	c	a	b	c	a	b	c
-	0	0	0	0	0	0	0	0	0	0
a	0	1								

此時表格上方是 0 (ab 與空字串)、左方是 1 (a 與 a)、左上方是 0 (a 與空字串)。取最大的那個，也就是 1

	-	a	b	c	a	b	c	a	b	c
-	0	0	0	0	0	0	0	0	0	0
a	0	1	1							

其中檢查左上方的程序可以省略 (why?)
時間複雜度是 $O(mn)$

Problem4

	-	a	b	c	a	b	c	a	b	c
-	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1
a	0	1	1	1	2	2	2	2	2	2
a	0	1	1	1	2	2	2	3	3	3
b	0	1	2	2	2	3	3	3	4	4
b	0	1	2	2	2	3	3	3	4	4
b	0	1	2	2	2	3	3	3	4	4
c	0	1	2	3	3	3	4	4	4	5
c	0	1	2	3	3	3	4	4	4	5
c	0	1	2	3	3	3	4	4	4	5

其中紅字代表這格是同字元結尾
也就是「左上 +1」步驟發生的地點

Problem4

```
function LCS(A, B)
    m := len(A), n := len(B);
    initialize Table with type int[m + 1][n + 1];
    for i from 0 to m do
        for j from 0 to n do
            if i = 0 or j = 0 then
                Table[i][j] := 0;
            else if A[i - 1] = B[j - 1] then
                Table[i][j] := Table[i - 1][j - 1] + 1;
            else
                Table[i][j] := max(Table[i][j - 1], Table[i - 1][j]);
    return Table[m][n];
```

Problem5

Consider designing by dynamic programming an algorithm that, given as input a sequence of distinct numbers, determines the length of a longest increasing subsequence in the input sequence. For instance, if the input sequence is $1, 3, 11, 5, 12, 14, 7$, $9, 15$, then a longest subsequence is $1, 3, 5, 7, 9, 15$ whose length is 6 (another longest subsequence is $1, 3, 11, 12, 14, 15$).

(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?

Problem5

給定一個 sequence S，求出最長遞增子序列 (LIS) 的長度

Problem5

首先求 recurrence relations
概念上可以先想，對前 n 個元素，我有前 n 個元素的 LIS 長度加上第 n+1 個元素時，LIS 長度會不會跟著 +1？
那就得看原本的 LIS 最右方的元素是否比第 n+1 個元素還小，這樣才能保持遞增
所以我們得知道原先的 LIS 長怎樣

Problem5

我們也不能只記得一個遞增子序列
例如：4 5 0 1 2
看到 0 時只在意前頭的 4 5，而看到 1 時也只在意前頭的 4 5，就會忽略掉此時已經一樣長的 0 1
從而在看到 2 時不會發現已經有更長的遞增子序列 0 1 2 出現了

Problem5

解決方法是，在元素被加到陣列時，我們要去記的是：**在所有將此元素放在最右邊的子序列當中**最長的遞增子序列長度
延續上面的例子 4 5 0 1 2，在看到 0 時我記住的是"0" 這個序列，而看到 1 時我記住的是"0 1" 這個序列
看到 2 時，前面出現過兩個最長的遞增子序列"4 5" 與"0 1"，而 2 可以接在"0 1" 的右方，所以記住"0 1 2"

要取出整個陣列的 LIS 長度，我們需要從頭到尾看誰記住的遞增子序列最長，而不是只看最右邊的元素存了什麼就好，複雜度會是 $O(n)$

Problem5

於是 recurrence relations 可以這樣寫

$$length(i) = \begin{cases} 1 & i = 1 \\ 1 & \forall 1 \leq j < x . S[i] >= S[j] \\ \max_{j \in T}(length(j)) + 1 & \text{otherwise} \end{cases}$$

$$(T = \{j \mid 1 \leq j < x \wedge S[i] < S[j]\})$$

$length(x)$ 代表從 $S[1]$ 到 $S[x]$ 為止，以 $S[x]$ 為尾的子序列當中最長的遞增子序列長度是多長
所以必須先知道有誰是可以讓 $S[x]$ 接在後面的
因為前面記住的就是「以 $S[i]$ 為尾的最長遞增子序列」，所以只需要比對 $S[i]$ 與 $S[x]$ 即可
找到前面最長的再 +1 即為所求
如果 $S[x]$ 不巧正是最小的元素，無法接在任何人後面，那就記住長度為 1 的子序列 (只包含自己)

Problem5

轉換成 pseudocode
思考另一個例子 1 3 11 5 12
$\mathit{length}(1) = 1$
$\mathit{length}(2) = \mathit{max}(1, \mathit{length}(1) + 1)$ (max 後方的 +1 推到裡面)
$\mathit{length}(3) = \mathit{max}(1, \mathit{length}(1) + 1, \mathit{length}(2) + 1)$
$\mathit{length}(4) = \mathit{max}(1, \mathit{length}(1) + 1, \mathit{length}(2) + 1)$
$\mathit{length}(5) =$
$\mathit{max}(1, \mathit{length}(1) + 1, \mathit{length}(2) + 1, \mathit{length}(3) + 1, \mathit{length}(4) + 1)$

Problem5

開一個陣列 *length*，初始值皆為 1

1	3	11	5	12
1	1	1	1	1

接著逐一將 $length(1)+1$、$length(2)+1$、$length(3)+1$ 與 $length(4)+1$ 考慮進去
由於 $length(1)$ 就是 1，所以在初始化時等同於已經確定 $length(1)$ 的值，我們就可以往後看 $length(2..n)$ 的值是否要更新
如果 $S[1] < S[x]$，則 $length(x)$ 就需要考慮 $length(1)$ 的值，如果比較大就更新

1	3	11	5	12
1	2	2	2	2

Problem5

此時 $length(2)$ 的值已經被確定了，照這個作法做下去

1	3	11	5	12
1	2	3	3	3

當我們固定 $length(3)$ 時，出現了不能接在 11 後面的元素
也就是說 $length(4)$ 的數值不會考慮 $length(3)$ 的大小，因為
$S[3] \geq S[4]$

1	3	11	5	12
1	2	3	3	4

Problem5

接下來固定 *length*(4)，此時 *length*(5) 的值跟 *length*$(4)+1$ 一樣，所以不會更動

1	3	11	5	12
1	2	3	3	4

Recall: length(5) = max(0, length(1)+1, length(2)+1, length(3)+1, length(4)+1)

Problem5

最終結果

1	3	11	5	12
1	2	3	3	4

答案取 4
不是因為它在最右邊，而是因為它是 *length* 當中的最大值

Problem5

```
function LIS(S)
    n := len(S);
    initialize length with type int[n];
    for i from 1 to n do
        length[i] := 1;
    for i from 1 to n do
        for j from (i+1) to n do
            if S[i] < S[j] then
                length[j] := max(length[i]+1, length[j]);
    return max(length);
```

複雜度為 $O(n^2)$

Problem5

有另外一種演算法可以降低複雜度 ?!
依然是基於同樣的遞迴關係

Problem5

開一個 array

1	2	3	4	5
-	-	-	-	-

一開始先把開頭元素放進去

1	2	3	4	5
1	-	-	-	-

這象徵了 1 這個元素對應到的 length 值是 1

Problem5

接下來看 3 要放哪？
由關係式我們知道要往前看比 3 小的元素的 length 值再 +1
在 array 當中的操作是：找到比自己**略小**的元素，往後方加上去

1	2	3	4	5
1	3	-	-	-

加入 11 也同理。那麼加入 5 的時候呢？
由於 11 比 5 大，所以 11 對應到的 length 不納入考量

1	2	3	4	5
1	3	11	-	-

此時略小於 5 的元素是 3，於是在 3 的後面加上 5

1	2	3	4	5
1	3	11	-	-
		5		

Problem5

放進 12
在 12 前面，11 與 5 都有最大的 length

1	2	3	4	5
1	3	11	-	-
		5	12	

我們可以只看 5 就好，因為可能出現介於 5 與 11 的數字，他可以放在 5 後面變成最長
也就是，上頭的 11 是可以忽略的

1	2	3	4	5
1	3	5	12	-

這是演算法運行時，array 實際的內容
array[k] 的數字代表，所有符合 $length(i) = k$ 的 i 當中，最小的 S[i] 值。長度為 k 的遞增子序列當中最小的結尾

Problem5

又因為我們是挑正好略小的元素，並塞數字到右邊
所以被取代的數字一定比塞進去的數字大，而其右方的數字也是
所以這個 array 總會是有序的

二元搜尋！
利用二元搜尋就能在 log 時間級找到**略小**的元素在哪！
時間複雜度變成 $O(n\log n)$

Problem5

function LIS2(S)
    n := len(S), max_length := 1;
    initialize Array with type int[n];
    Array[1] := S[1];
    for i from 2 to n **do**
        find the largest element in Array[1:max_length] that is smaller than S[i];
        put S[i] right after the element;
        if S[i] is put beyond Array[1:max_length] **then**
            max_length++;
    return max_length;

上頭的 Array[1:max_length] 是包含 Array[max_length] 的範圍
C++ 的 std::lower_bond 函數就提供了「給出略小的元素的右邊」在哪的功能。