Data Structure and Algorithm

Homework #4

=== Homework Reference Solution ===

***Problem* 2.**

(1)

Array *extracted*: 4 1 3 7 8 2

(2)

```
I ← { }              // record the value of the inserted numbers
L ← 0          // record the number of elements of INSERT array I
num ← 0              // record the number of elements of extracted
for i = 1 to n+m
   if sequence[i] == 'E'
        tmp ← ∞
        for j = 1 to L
             if tmp > I[j]
                  tmp ← I[j]
        extracted[num] ← tmp
        num ← num + 1
   else
        I[L] ← sequence[i]
        L ← L + 1
```

Time complexity = O(*mn*)

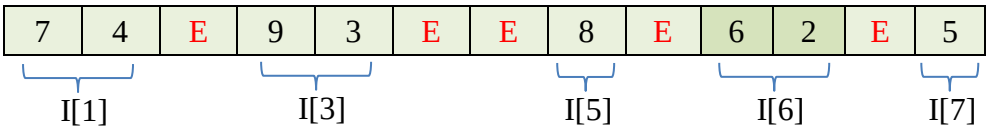
Space complexity = O(*n*)

(3)

We want to show that the array *extracted* returned by OFF-LINE-MINIMUM is correct, meaning that for $i = 1, 2, \ldots, m$, *extracted*[j] is the key returned by the jth EXTRACT-MIN call.

7	4	E	1	E	9	3	E	E	8	E	6	2	E	5
I[1]			**I[2]**		**I[3]**				**I[5]**		**I[6]**			**I[7]**

We start with n INSERT operations and m EXTRACT-MIN operations. The smallest value of all the elements will be extracted in the first EXTRACT-MIN after its insertion. So we find j such that the minimum element is in $K[j]$, and put the minimum element in *extracted*[j].



Now we reduce to a similar problem with $n - 1$ INSERT operations and $m - 1$ EXTRACT-MIN operations in the following way: the INSERT operations are the same but without the insertion of the smallest that was extracted, and the EXTRACT-MIN operations are the same but without the extraction that extracted the smallest element.

Conceptually, we unite subsequence I[j] and I[j+1], removing the extraction between them and also removing the insertion of the minimum element from I[j] ∪ I[j+1]. Uniting I[j] and I[j+1] is accomplished by line 6. We need to determine which set is $K[l]$, rather than just using $K[j+1]$ unconditionally, because $K[j+1]$ may have been destroyed when it was united into a higher-indexed set by a previous execution of line 6.

Because we process extractions in increasing order of the minimum value found, the remaining iterations of the **for** loop correspond to solving the reduced problem.

Time complexity = O(mn)

(4)

To implement this algorithm, we place each element in a disjoint-set forest. Each root has a pointer to its *Ki* set, and each *Ki* set has a pointer to the root of the tree representing it. All the valid sets *Ki* are in a linked list.

Before OFF-LINE-MINIMUM, there is initialization that builds the initial sets $K[i]$ according to the I[i] sequences.

- Line 2 turns into $j \leftarrow$ FIND-SET(i).
- Line 5 turns into $K[l] \leftarrow next[K[j]]$.
- Line 6 turns into $l \leftarrow$ LINK(j, l) and remove $K[j]$ from the linked list.

To analyze the running time, we note that there are n elements and that we have the following disjoint-set operations:

- n MAKE-SET operations
- at most $n - 1$ UNION operations before starting
- n FIND-SET operations
- at most n LINK operations

Thus the number m of overall operations is $O(n)$.

The total running time is $O(m\,\alpha(n)) = O(n\,\alpha(n))$.

Problem 3.

(1)

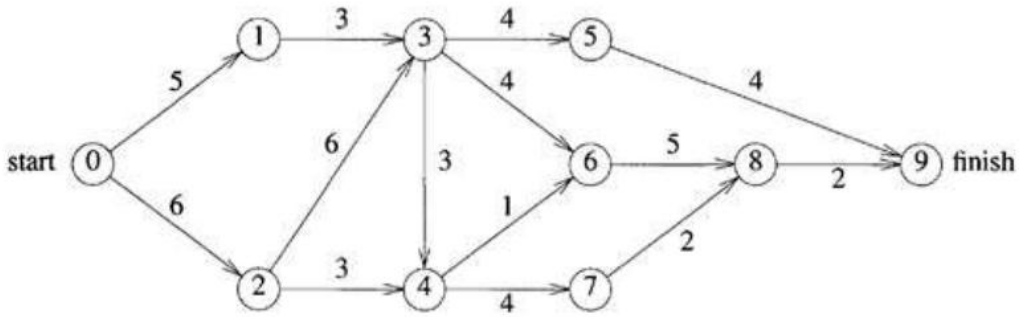
If an activity is critical, it means that the early and late start times are the same.

This, Implies It's possible the whole time needed to finish the project will also be affected if we could shorten the time needed to finish it (by the same amount of time).

(2)

No. Because the critical path constructed by critical activities specifies the total least amount of time needed to finish the project, which means if we couldn't shorten time needed to finish these activities there is no way we could shorten the total time finishing the project.

(3) Early Activity times



	0	1	2	3	4	5	6	7	8	9	stack
Initial	0	0	0	0	0	0	0	0	0	0	0
0	0	5	6	0	0	0	0	0	0	0	1,2
1	0	5	6	8	0	0	0	0	0	0	2
2	0	5	6	12	9	0	0	0	0	0	3
3	0	5	6	12	15	16	16	0	0	0	4,5
4	0	5	6	12	15	16	16	19	0	0	6,7,5
6	0	5	6	12	15	16	16	19	21	0	7,5
7	0	5	6	12	15	16	16	19	21	0	8,5
8	0	5	6	12	15	16	16	19	21	23	5
5	0	5	6	12	15	16	16	19	21	23	null

Late Activity times

Le[9] = ee[9] = 23;

Le[8] = le[9] – <8, 9> = 23 – 2 = 21;

Le[5] = le[9] - <5, 9> = 23 – 4 = 19;

Le[6] = le[8] - <6, 8> = 21 – 5 = 16;

Le[7] = le[8] - <7, 8> = 21 – 2 = 19;

Le[4] = le[6] - <4, 6> = le[7] - <4, 7> = 15;

Le[3] = le[4] - <3, 4> = le[6] - <3, 6> = 12;

Le[2] = le[4] - <2, 4> = le[3] - <2, 3> = 6;

Le[1] = le[3] - <1, 3> = 12 – 3 = 9;

Le[0] = le[2] - <0, 2> = 6 – 6 = 0;

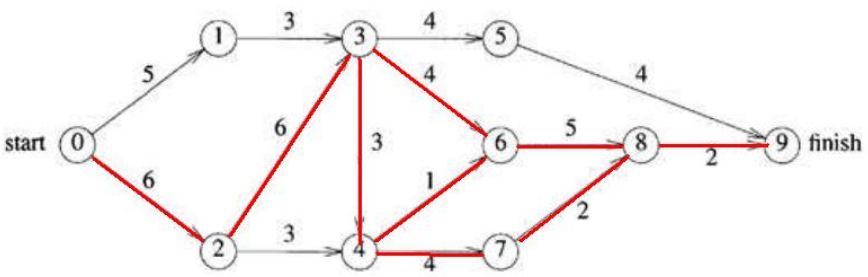
i	0	1	2	3	4	5	6	7	8	9
Le[i]	0	9	6	12	15	19	16	19	21	23

Edge	early starting time	late starting time
⟨0; 1⟩	0	4
⟨0; 2⟩	0	0
⟨1; 3⟩	5	9
⟨2; 3⟩	6	6
⟨2; 4⟩	6	12
⟨3; 4⟩	12	12
⟨3; 5⟩	12	15
⟨3; 6⟩	12	12
⟨4; 6⟩	15	15
⟨4; 7⟩	15	15
⟨5; 9⟩	16	19
⟨6; 8⟩	16	16
⟨7; 8⟩	19	19
⟨8; 9⟩	21	21

(4)

According to the early and late start times of all the vertices, we can conclude the earliest time the project can finish is ee[9], which is 23.

(5)



(6)

Yes. In this example, we can reduce the time cost of <0, 2>, <2, 3>, <8, 9> to speed up the project.

Problem 4.

(1)

If a graph G is represented by an adjacency matrix A, we need to look up every entry in A in the BFS algorithm. Due to the fact that A is and |V|-by-|V| matrix, the time complexity of BFS is $\Theta(|V|^2)$.

(2)

The diameter of an undirected tree can be computed within two steps :

- Step 1. Choose an arbitrary source in the tree. Run BFS and select any vertex X at maximum distance from source. X must be a leaf of this tree.
- Step 2. Use X as source and run BFS again. X will be at one end of a diameter and any vertex at maximum distance from X can be the other end of the diameter. Both ends of the diameter must be leaves.

The running time of this algorithm is $\Theta (V + E)$.

(3)

$u.\pi$: the predecessor of u

u.d: discovery event

u.f: finishing event

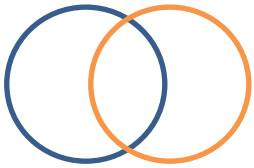
```
DFS(G)
   for each vertex u ∈ G.V
         u.color ← WHITE
         u.π ← NIL
      time = 0
      for each vertex u ∈ G.V
         if u.color == WHITE
            DFS-Traverse(G, u)


DFS-Traverse(G, u)
      Stack S ← ∅
   push(S, u)
   while empty(S) == false do
      x ← pop(S)
      if x.color == WHITE do
```

time ← *time* + 1
x.d ← *time*
x.color ← GRAY
push(*S*, *x*)
for each *v* ∈ G.*Adj*[*x*] do
if *v.color* == WHITE do
v. *π* ← *x*
push(*S*, *v*)
else if *x.color* = GRAY do
time ← *time* + 1
x.f ← *time*
x.color ← BLACK

(4)

■ "->"

An Euler tour can be divided into a set of edge-disjoint simple cycles. (See the example below) In each simple cycle, each vertex v in the cycle has only one edge coming in and one edge leading out; therefore, in-degree(v)=out-degree(v) for every vertex v in a simple cycle. Since G has an Euler tour, it contains some edge-disjoint simple cycles. We can make sure that for each vertex v in the graph, in-degree(v) = out-degree(v).



■ "<-"

Assume that we start from a vertex u, traverse the graph randomly of some edges, and create a cycle back to u (since it has one edge going in, it must have some unvisited edge going out, and G is strongly connected). We continue the process to find more cycles until all edges are visited. Due to the fact that in-degree(v) = out-degree(v) for every vertex v in the graph, we can sure that we will find some cycles in the graph. When we visit every edge, we have found an Euler tour in the graph.

Problem 5.

If vertex i is a universal sink, the i_{th} row of the adjacency-matrix will be all 0, and the i_{th} column will be all 1 except the a_{ii} entry.

The algorithm begins from a_{11}. If the entry $a_{ij} = 0$ then j = j + 1 (move right); if aij = 1 then i = i +1 (move down). It will finally stop at an entry a_{kn} of the last row or a_{nk} of the last column (n = |V|, $1 \leqq k \leqq |V|$). After the process, we should check if vertex k satisfies the definition of universal sink.

Since we always make a step right or down, it runs in O(V). Also, the checking process of a universal sink can be done in O(V).