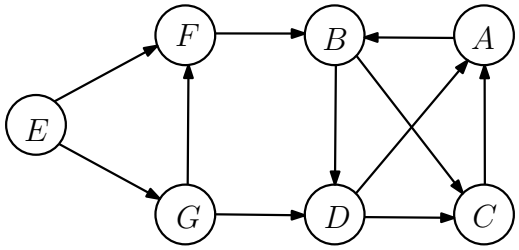
# COMP 3804 — Solutions Tutorial March 15

**Question 1:** Consider the following directed graph:




**(1.1)** Draw the *DFS*-forest obtained by running algorithm DFS; the pseudocode is given on the last page. Algorithm DFS uses algorithm EXPLORE as a subroutine. The pseudocode for this subroutine is also given on the last page.

Classify each edge as a tree edge, forward edge, back edge, or cross edge. In the *DFS*-forest, give the *pre*- and *post*-number of each vertex. Whenever there is a choice of vertices, pick the one that is alphabetically first.

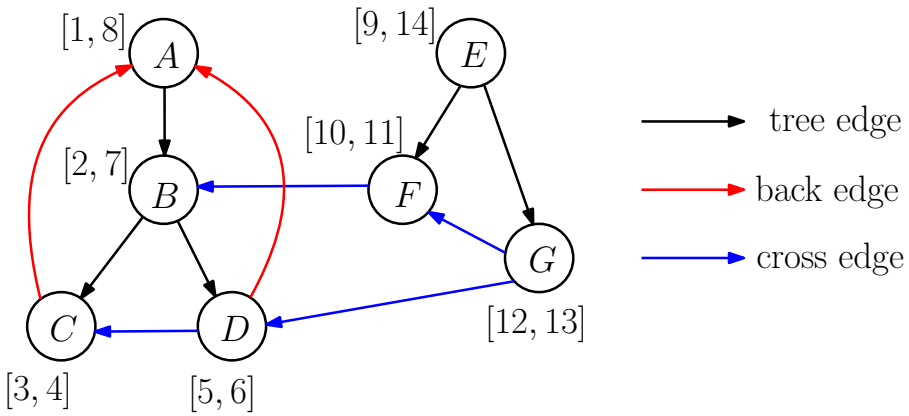
**(1.2)** Draw the *DFS*-forest obtained by running algorithm DFS. Classify each edge as a tree edge, forward edge, back edge, or cross edge. In the *DFS*-forest, give the *pre*- and *post*-number of each vertex. Whenever there is a choice of vertices, pick the one that is alphabetically last.

**Solution:** We start by recalling algorithms DFS and EXPLORE:

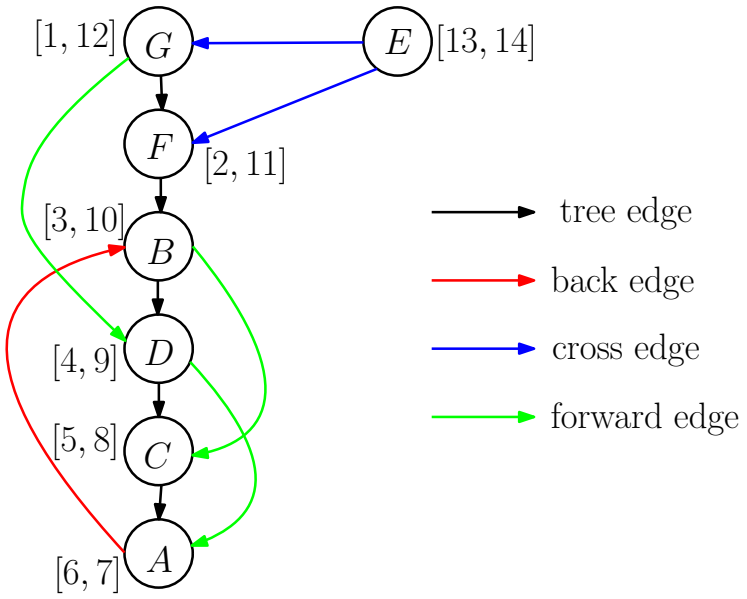
**Algorithm** DFS($G$):
**for each** vertex $v$
**do** $visited(v) = false$
**endfor**;
$clock = 1$;
**for each** vertex $v$
**do if** $visited(v) = false$
  **then** EXPLORE($v$)
  **endif**
**endfor**

**Algorithm** EXPLORE($v$):
$visited(v) = true$;
$pre(v) = clock$;
$clock = clock + 1$;
**for each** edge $(v, u)$
**do if** $visited(u) = false$
  **then** EXPLORE($u$)
  **endif**
**endfor**;
$post(v) = clock$;
$clock = clock + 1$

We start with **(1.1)**. In case there is more than one choice, we pick the alphabetically smallest one. Thus, algorithm DFS($G$) starts by calling EXPLORE($A$). Here is the resulting *DFS*-forest:




Next we do **(1.2)**. In case there is more than one choice, we pick the alphabetically largest one. Thus, algorithm DFS($G$) starts by calling EXPLORE($G$). Here is the resulting *DFS*-forest:




**Question 2:** Let $G = (V, E)$ be a directed graph, in which each edge $(u, v)$ has a positive weight $wt(u, v)$, and let $s$ be a source vertex in $V$. Let $n = |V|$ and $m = |E|$.

Recall that Dijkstra's algorithm computes for each vertex $v$, the length $\delta(s, v)$ of a shortest path from $s$ to $v$, in total time $O((n + m) \log n)$. The pseudocode for this algorithm is given on the last page.

Assume that each edge weight $wt(u, v)$ is an integer in the set $\{1, 2, \ldots, 17\}$. Prove that Dijkstra's algorithm can be implemented such that the running time is $O(m + n)$.

*Hint:* A *priority queue* is a data structure that stores any finite sequence of numbers and supports the operations INSERT, EXTRACT_MIN and DECREASE_KEY. An example of a priority queue is a min-heap.

A priority queue is called *monotone* if, during any sequence of operations, the smallest element never decreases. Thus, if at some moment, the smallest element is 45, then later on, the smallest value is always at least 45.

Assume that at any moment, any number stored in a monotone priority queue is an integer belonging to the set $\{0, 1, 2, \ldots, k\}$. Start by showing that any sequence consisting of $n$ INSERT-operations, $n$ EXTRACT_MIN-operations and $m$ DECREASE_KEY-operations (these operations may appear in any order) can be processed in total time $O(m + n + k)$.

**Solution:** As the hint suggests, we start by describing a data structure that implements a monotone priority queue.

- We have a set $Q$ of items, where each item $v$ has a key $key(v)$, which is an integer belonging to the set $\{0, 1, 2, \ldots, k\}$.
  - We store the set $Q$ in an array $A[0, \ldots, k]$. Each entry $A[i]$ is a doubly-linked list storing all items $v$ in $Q$ for which $key(v) = i$.

- We have a variable *current_min*, whose value is the smallest integer $i$ for which the list $A[i]$ is non-empty.

  Note: Since the priority queue is monotone, this variable never decreases.

- INSERT($v$): Add item $v$ at the end of the list $A[key(v)]$.
  - This takes $O(1)$ time.
- DECREASE_KEY($v, x$): This operation gets a pointer to the node in the list $A[key(v)]$ that stores the item $v$. The real number $x$ satisfies $x < key(v)$. Note that since the priority queue is monotone, we have $x \geq key(w)$, where $w$ is an arbitrary item in the list $A[current\_min]$.

  To process this operation DECREASE_KEY($v, x$), we do the following:
  - Delete $v$ from the list $A[key(v)]$.
  - Set $key(v) = x$.
  - Add item $v$ at the end of the list $A[key(v)]$.
  - Note: We do not have to update the variable *current_min*.
  - The entire operation takes $O(1)$ time.
- EXTRACT_MIN: We do the following:
  - Let $v$ be an arbitrary item in the list $A[current\_min]$.
  - Delete $v$ from the list $A[current\_min]$.
  - If the list $A[current\_min]$ is empty: keep on increasing *current_min* until the list $A[current\_min]$ is non-empty.
  - Return $v$.
  - The entire operation takes time proportional to 1 plus the number of times the variable *current_min* is increased.

Consider an arbitrary sequence consisting of $n$ INSERT-operations, $n$ EXTRACT_MIN-operations and $m$ DECREASE_KEY-operations (these operations may appear in any order).

- It takes $O(k)$ time to initialize the array $A[0, \ldots, k]$.
- The total time for the $n$ INSERT-operations is $O(n)$.
- The total time for the $m$ DECREASE_KEY-operations is $O(m)$
- What is the total time for the $n$ EXTRACT_MIN-operations:
  - It is $O(n)$ plus the total number of times the variable *current_min* is increased.
  - The variable *current_min* can be increased only $k$ times.

– Thus, the total time for the $n$ EXTRACT_MIN-operations is $O(n+k)$.

- We conclude that the total time to process the sequence of $n$ INSERT-operations, $n$ EXTRACT_MIN-operations and $m$ DECREASE_KEY-operations is $O(m+n+k)$.

Now we are going to use this result to implement Dijkstra's algorithm for the case when all edge weights are integers in the set $\{1, 2, \ldots, 17\}$. Here is the algorithm that we have seen in class:

**Algorithm** DIJKSTRA$(G, s)$:
**for each** $v \in V$
**do** $d(v) = \infty$
**endfor**;
$d(s) = 0$;
$S = \emptyset$;
$Q = V$;
**while** $Q \neq \emptyset$
**do** $u =$ vertex in $Q$ for which $d(u)$ is minimum;
delete $u$ from $Q$;
insert $u$ into $S$;
**for each** edge $(u, v)$
**do if** $d(u) + wt(u, v) < d(v)$
**then** $d(v) = d(u) + wt(u, v)$
**endif**
**endfor**
**endwhile**

- To implement Dijkstra's algorithm, it is enough to have a monotone priority queue; see property 4 on page 104 of my handwritten notes.

- Any path between two vertices has at most $n-1$ edges and, thus, length at most $17(n-1)$.

- Thus, any value $d(v)$ is either $\infty$ or an element of the set $\{0, 1, \ldots, 17(n-1)\}$.

- We store the set $Q$ in our monotone priority queue, where $k = 17n$ plays the role of $\infty$.

- The running time of Dijkstra's algorithm is equal to the total time to process a sequence of $n$ INSERT-operations, $n$ EXTRACT_MIN-operations and $m$ DECREASE_KEY-operations. We have seen above that this is $O(m+n+k)$, which is $O(m+n)$.

**Question 3:** Lionel Messi[1] is having a difficult time. Not only has his team been kicked out of the Champions League, he is even booed by his own fans. Lionel decides to hang up his boots and become a software developer[2].

[1] When I made this question, Messi was still playing in Paris.
[2] "We offer you a salary of half a million". "That's reasonable. I assume this is per week?"

On the first day of his new job, Lionel is asked to implement a sorting algorithm, i.e., an algorithm that takes as input an arbitrary sequence of numbers and returns these numbers in sorted order.

Since Lionel has no clue about algorithms, he looks at the cheater website `chegg.com`. Unfortunately, this website does not have implementations of sorting algorithms. However, Lionel does find a highly optimized implementation of Dijkstra's algorithm. For any directed input graph $G = (V, E)$, in which each directed edge $(u, v)$ has a weight $wt(u, v) > 0$, and for any given source vertex $s$, this algorithm computes for each vertex $v$, the length $\delta(s, v)$ of a shortest path from $s$ to $v$, in total time $O((|V| + |E|) \log |V|)$.

Prove that Lionel can use this implementation of Dijkstra's algorithm to sort any sequence of $n$ numbers in $O(n \log n)$ time.

*Hint:* Given a sequence $x_1, x_2, \ldots, x_n$ of numbers, define a (very simple!) directed graph $G$ with positive edge weights. Run Dijkstra's algorithm on this graph.

**Solution:** When given as input a sequence $x_1, x_2, \ldots, x_n$ of numbers, Professor Messi's algorithm does the following:

1. By scanning the sequence, find the smallest number in the sequence and denote it by $M$. This takes $O(n)$ time.

2. Construct a directed "star" graph $G = (V, E)$:

   (a) $V = \{s, v_1, v_2, \ldots, v_n\}$

   (b) $E = \{(s, v_1), (s, v_2), \ldots, (s, v_n)\}$

   (c) For each $i = 1, 2, \ldots, n$, the edge $(s, v_i)$ gets weight $x_i - M + 1$.
   Note: All edge weights are positive, even if $M < 0$.

   (d) Constructing this graph takes $O(n)$ time.

3. Run Dijkstra's algorithm on this graph $G$.

   This takes time $O((|V|+|E|) \log |V|)$. Since $|V| = n+1$ and $|E| = n$, this is $O(n \log n)$.

4. It is clear that for each $i = 1, 2, \ldots, n$, $\delta(s, v_i)$ is equal to $x_i - M + 1$, because there is ony one path from $s$ to $v_i$. It follows from property 4 on page 104 of my handwritten notes that Dijkstra computes all values $\delta(s, v_i)$ in sorted order.

   By scanning this sorted order, we obtain the sorted order of the input numbers $x_1, x_2, \ldots, x_n$ in $O(n)$time.

5. The total running time is $O(n) + O(n) + O(n \log n) + O(n)$, which is $O(n \log n)$.

**Question 4:** Let $G = (V, E)$ be a connected undirected graph, in which each edge has a weight. An edge $\{u, v\}$ is called *annoying* if the graph $G' = (V, E \backslash \{\{u, v\}\})$ is not connected.

Assume there is a unique edge in $E$ with largest weight; denote this edge by $e$.

Prove that $e$ is an edge in every minimum spanning tree of $G$ if and only the edge $e$ is annoying.

**Solution:** Let $u$ and $v$ be the two vertices of $e$.

We first assume that $e$ is an edge in every minimum spanning tree of $G$. We have to prove that $e$ is an annoying edge.

Let $T$ be an arbitrary minimum spanning tree of $G$. By removing the edge $e$ from $T$, we obtain two trees $T_1$ and $T_2$, where $u$ is a vertex of $T_1$ and $v$ is a vertex of $T_2$. Let $A$ be the vertex set of $T_1$ and let $B$ be the vertex set of $T_2$. Let $e' = \{u', v'\}$ be an edge in $G$ of minimum weight with $u' \in A$ and $v' \in B$. Let $T'$ be the spanning tree obtained from $T$ by replacing $e$ by $e'$. We have seen in class that the weight of $T'$ is equal to the weight of $T$. This implies that $e$ and $e'$ have the same weight. Since the largest edge weight in $G$ is unique, it follows that $e = e'$. Thus, there is only edge in $G$ between the sets $A$ and $B$. Therefore, the edge $e$ is annoying.

To prove the converse, assume that the edge $e$ is annoying. If we remove $e$ from $G$, we obtain two connected subgraphs, say $G_1$ containing $u$, and $G_2$ containing $v$. Let $A$ be the vertex set of $G_1$ and let $B$ be the vertex set of $G_2$. Observe that $e$ is the only edge between $A$ and $B$. As a result, every spanning tree of $G$ contains the edge $e$. In particular, this is trie for every minimum spanning tree.

**Question 5:** In class, we have seen a data structure for the UNION-FIND problem that stores each set in a linked list, with the header of the list storing the name and size of the set, and each node storing a back pointer to the header. If we start with $n$ sets, each having size one, then we have seen in class that, using this data structure, any sequence of $n-1$ UNION-operations can be processed in $O(n \log n)$ time.

Give an example of a sequence of $n-1$ UNION-operations, for which the algorithm takes $\Omega(n \log n)$ time.

**Solution:** We have seen in class that the operation UNION$(A, B)$ takes time

$$\Theta\left(\min(|A|, |B|)\right).$$

We assume for simplicity that $n$ is a power of two, say $n = 2^k$. We start with $n$ sets, each of size 1.

- We do $n/2$ UNION-operations, each one on two sets of size 1. Afterwards, we have $n/2$ sets, each of size 2. The total time for these $n/2$ UNION-operations is $\Theta(n)$.
- We do $n/2^2$ UNION-operations, each one on two sets of size 2. Afterwards, we have $n/2^2$ sets, each of size $2^2$. The total time for these $n/2^2$ UNION-operations is $\Theta(n)$.
- We do $n/2^3$ UNION-operations, each one on two sets of size $2^2$. Afterwards, we have $n/2^3$ sets, each of size $2^3$. The total time for these $n/2^3$ UNION-operations is $\Theta(n)$.
- We do $n/2^4$ UNION-operations, each one on two sets of size $2^3$. Afterwards, we have $n/2^4$ sets, each of size $2^4$. The total time for these $n/2^4$ UNION-operations is $\Theta(n)$.

- Etc. etc.
- We do $n/2^{k-1} = 2$ UNION-operations, each one on two sets of size $2^{k-2} = n/4$. Afterwards, we have $n/2^{k-1} = 2$ sets, each of size $2^{k-1} = n/2$. The total time for these $n/2^{k-1}$ UNION-operations is $\Theta(n)$.
- We do $n/2^k = 1$ UNION-operation, on two sets of size $2^{k-1} = n/2$. Afterwards, we have $n/2^k = 1$ set of size $n$. The time for this UNION-operation is $\Theta(n)$.
- Overall, during all these $k$ stages, the amount of time is

  $$\Theta(kn) = \Theta(n \log n),$$

  which is

  $$\Omega(n \log n).$$

**Question 6:** You are given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$ over some finite alphabet. We consider the problem of converting $X$ to $Y$, using the following operations:

1. Substitution: replace one symbol by another one.
2. Insertion: insert one symbol.
3. Deletion: delete one symbol.

For example, if $X =$ "logarithm" and $Y =$ "algorithm", we can convert $X$ to $Y$ in the following way:

1. Start with "logarithm".
2. Inserting "a" at the front gives "alogarithm".
3. Deleting "o" gives "algarithm".
4. Replacing the second "a" by "o" gives "algorithm".

The *edit distance* between the strings $X$ and $Y$ is defined to be the minimum number of operations needed to convert $X$ to $Y$. For example, the edit distance between $X =$ "logarithm" and $Y =$ "algorithm" is three, because $X$ can be converted to $Y$ using three operations, but not using two operations. If the string $X$ has length $m$ and the string $Y$ is empty, then the edit distance between $X$ and $Y$ is equal to $m$.

Give a dynamic programming algorithm (in pseudocode) that computes, in $O(mn)$ time, the edit distance between the strings $X$ and $Y$. Argue why your algorithm is correct.

**Solution:** We want to apply dynamic programming, so we have to go through the three steps, as we did in class.

**Step 1: Structure of the optimal solution.**

Consider the optimal way to convert $X = x_1 x_2 \dots x_m$ to $Y = y_1 y_2 \dots y_n$. There are three possibilities:

1. $y_n$ has been inserted. Then, the optimal conversion of $X$ to $Y$ consists of the optimal conversion of $x_1x_2\ldots x_m$ to $y_1y_2\ldots y_{n-1}$, followed by the insertion of $y_n$.

2. $x_m$ has been deleted. Then, the optimal conversion of $X$ to $Y$ consists of the optimal conversion of $x_1x_2\ldots x_{m-1}$ to $y_1y_2\ldots y_n$, followed by the deletion of $x_m$.

3. Neither $y_n$ has been inserted, nor $x_m$ has been deleted. There are two possibilities:

   (a) If $x_m = y_n$: In this case, the optimal conversion of $X$ to $Y$ is just the optimal conversion of $x_1x_2\ldots x_{m-1}$ to $y_1y_2\ldots y_{n-1}$.

   (b) If $x_m \neq y_n$: In this case, the optimal conversion of $X$ to $Y$ consists of the optimal conversion of $x_1x_2\ldots x_{m-1}$ to $y_1y_2\ldots y_{n-1}$, followed by replacing $x_m$ by $y_n$.

Thus, the optimal solution contains optimal solutions to subproblems.

**Step 2: Set up a recurrence relation for the optimal solution.**

We define, for $0 \leq i \leq m$ and $0 \leq j \leq n$:

$$S(i,j) = \text{ the edit distance between } x_1\ldots x_i \text{ and } y_1\ldots y_j.$$

Thus, we have to compute the value of $S(m,n)$. We obtain the following recurrences:

$$S(i,0) = i \text{ for } 0 \leq i \leq m,$$

$$S(0,j) = j \text{ for } 0 \leq j \leq n,$$

$$S(i,j) = \min(S(i,j-1)+1, S(i-1,j)+1, S(i-1,j-1)) \text{ if } i>0,\ j>0, \text{ and } x_i = y_j,$$

$$S(i,j) = \min(S(i,j-1)+1, S(i-1,j)+1, S(i-1,j-1)+1) \text{ if } i>0,\ j>0, \text{ and } x_i \neq y_j.$$

**Step 3: Solve the recurrence, in a bottom-up order.**

**for** $i = 0$ **to** $m$ **do** $S(i,0) = i$ **endfor**;
**for** $j = 1$ **to** $n$ **do** $S(0,j) = j$ **endfor**;
**for** $i = 1$ **to** $m$
**do for** $j = 1$ **to** $n$
  **do if** $x_i = y_j$
    **then** $S(i,j) = \min(S(i,j-1)+1, S(i-1,j)+1, S(i-1,j-1))$
    **else** $S(i,j) = \min(S(i,j-1)+1, S(i-1,j)+1, S(i-1,j-1)+1)$
    **endif**;
  **endfor**;
**endfor**;
return $S(m,n)$

It is clear that the overall running time is $O(mn)$.