CSC263 Assignment 6

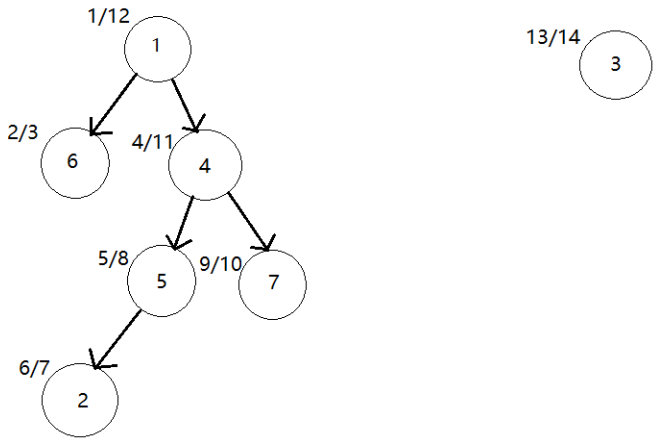
Angela Zhu, Xinyi Ji, Zhuozi Zou

March 28, 2019

1 Problem 1

(Written by Zhuozi Zou, read by Angela Zhu and Xinyi Ji)

a.



b. Back edges: 0
Forward edges: 2
Cross-edges: 5

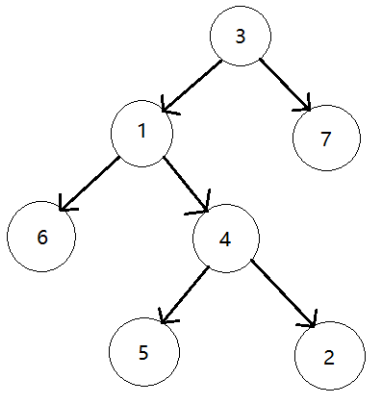
c. The theorem I am using: the White-Path Theorem.

From part b) we know that the DFS of the directed graph G does not have a back edge. Then by the application of the White-Path Theorem (from lec18 slides page 95), G has no cycle. Thus G is a Directed Acyclic Graph (DAG).
According to week 9 tutorial slides page 6, since G is a DAG, it is possible to execute all the tasks in G in an order that respects all the precedence requirements given by the graph edges. Since the graph G represents seven courses and their prerequisites, it is possible to take all the courses in a sequential order that satisfies all the prerequisite requirements.

d. The algorithm I am using: Topological_Sort(G) from week 9 tutorial.

This algorithm is applicable since Topological_Sort(G) produces a topological sort of the DAG G. Topological_Sort(G) returns a linked list of the nodes of G in the decreasing order of finish times, and for every edge (u, v) of G, u appears before v in this list.
The list of the courses in order they can be taken without violating any prerequisite is: [3, 1, 4, 7, 5, 2, 6].

e.



2 Problem 2

(Written by Angela Zhu, read by Xinyi Ji and Zhuozi Zou)

Algorithm Description:

The algorithm I designed stores the equality constraints into an adjacency list, where the nodes of the graph are the n variables, and each equality constraint is taken as an edge of an undirected graph.
After creating the graph G (stored as the adjacency list, with $|V| = n$ and $|E| = m$), BFS(G, x_1) is performed. This graph G may have disconnected parts, therefore, after one BFS, the algorithm finds the next white node x_i in G, and calls BFS(G, x_i). This is repeated until all nodes in G are discovered and explored.
For each node x_i where $1 \leq i \leq n$, let $val[x_i]$ denote the current value of x_i, which is assigned an initial value of i (i.e. $val[x_1] = 1$, $val[x_2] = 2$, ..., $val[x_n] = n$). Upon discovery of x_i during BFS, if x_i has a parent (i.e. it is not the beginning node of this BFS), $val[x_i]$ is then assigned the value $val[p[x_i]]$. If not, $val[x_i]$ remains unchanged. Therefore, after each BFS, all connected nodes should have the same val.
After all the nodes are discovered and explored, loop through all the m constraints. If it is an inequality constraint $x_i \neq x_j$, compare $val[x_i]$ and $val[x_j]$.

- If $val[x_i] = val[x_j]$, this means that x_i is equal to x_j. This contradicts with the inequality constraint $x_i \neq x_j$, therefore there does not exist such an assignment that does not violate any of the constraints, and the algorithm terminates immediately.
- If $val[x_i] \neq val[x_j]$, this means that x_i and x_j are not connected in G, therefore they do not have the same values. The inequality constraint $x_i \neq x_j$ is satisfied, and the algorithm continues.

If this step is reached, then there exists an assignment that does not violate any of the constraints. Indeed, the val attributes of the node is a satisfying assignment. Loop through all the n nodes, and for each node x_i, output the value $val[x_i]$. After the iteration ends, the algorithm is terminated.

Time Complexity:

- Creating the adjacency list takes $\mathcal{O}(|V| + |E|)$ time, as learned in class. Since $|V| = n$ and $|E| = m$, the worst-case running time is $\mathcal{O}(n + m)$.
- We learned in class that the worst-case time complexity of performing BFS on G is $\mathcal{O}(|V| + |E|)$, which is $\mathcal{O}(n + m)$. Even though the graph may be disconnected and BFS may be performed multiple times, all the n nodes and m edges are reached and explored only once, therefore this time complexity still holds.

- In the worst-case, we loop through all the m constraints, and compare the two variables. Since each comparison takes constant time, which is $\mathcal{O}(1)$, the total time of this step is $m \cdot \mathcal{O}(1) = \mathcal{O}(m)$.
- We loop through all the n variables, and output the value for each variable. Since outputting takes constant time, which is $\mathcal{O}(1)$, the total time of this step is $n \cdot \mathcal{O}(1) = \mathcal{O}(n)$.

Thus the total time complexity of this algorithm is $\mathcal{O}((n+m)+(n+m)+m+n) = \mathcal{O}(3m+3n) = \mathcal{O}(3(m+n))$, which is of $\mathcal{O}(m+n)$.
Therefore, the worst-case running time of our algorithm is $\mathcal{O}(m+n)$.

3 Problem 3

(Written by Xinyi Ji, read by Angela Zhu and Zhuozi Zou)

WTS: no minimum spanning tree of G contains e_{max}.

Proof. I am going to prove this by contradiction.
Assume, for the sake of contradiction, the negation of what we are proving, that there is a minimum spanning tree of G containing e_{max}, call it T.
Since T is a tree, e_{max} has divided this tree into 2 parts, as it is the only edge that connects these parts together (if there is another tree in T that connected these 2 parts, then we would have a cycle in the tree). Let these two parts be v_1 and v_2.

From the question we know that for every edge $e \in E$, there is a cycle in G that contains e, hence there must exist another edge e_0 in G that connected v_1 and v_2. Since the one of the two nodes that e_{max} connects belongs to v_1 and the other belongs to v_2, if such e_0 does not exist, we cannot form a cycle in G that contains e_{max}. Let's add this e_0 to the tree T, and now we have a cycle in T that contains e_{max}. By the fact that removing any edge from this cycle results in a tree again, let's remove e_{max}. Hence we result in a spanning tree of G, call it T'. The only difference between T and T' is that T contains e_{max} but not e_0, and T' contains e_0 but not e_{max}. Both of them are spanning trees of G.

Let $E[T]$ denote the set of edges in T and let $E[T']$ denote the set of edges in T'.
The weight of T is $\sum_{i \in \{E[T] \setminus e_{max}\}}(w(i) + w(e_{max})$,

and the weight of T' is $\sum_{i \in \{E[T'] \setminus e_0\}}(w(i) + w(e_0)$, hence:

$$\text{weight}(T) \text{ - } \text{weight}(T') = \sum_{i \in \{E[T] \setminus e_{max}\}}(w(i) + w(e_{max}) - \sum_{i \in \{E[T'] \setminus e_0\}}(w(i) + w(e_0)$$

By the difference between T and T':

$$\sum_{i \in \{E[T] \setminus e_{max}\}}(w(i)) = \sum_{i \in \{E[T'] \setminus e_0\}}(w(i))$$

Hence weight(T) - weight(T')= $w(e_{max}) - w(e_0) > 0$, since e_{max} is the edge with maximum weight in G, and every edge of G has distinct weight. So the weight of T' is less than the weight of T, but T is a minimum spanning tree of G by our assumption, which means that there is no other spanning tree of G whose weight is less than T. Here we get a contradiction.

Since assuming that there is a minimum spanning tree of G containing e_{max} leads to a contradiction, the assumption must not be true. This proves that there is no minimum spanning tree of G that contains e_{max}.

■