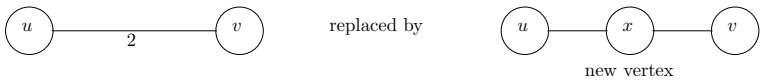
CS:3330 Exam 2, Fall 2015

Thursday, Oct 29 2015, 6:30 pm to 8:30 pm

---

1. You are given a graph G in which each edge has weight 1 or 2. You would like to find shortest paths in this graph and know that you can use Dijkstra's shortest path algorithm to do so. However, you wonder if it may be more efficient to turn G into an unweighted graph (i.e., a graph with no edge weights) and then use BFS instead of Dijkstra's shortest path algorithm. Your plan for transforming G into an unweighted graph consists of replacing each edge $\{u, v\}$ in G with weight 2 by a path of length 2 (as shown below). Note that this results is a new vertex (called x in the example below) being added to the graph for every weight-2 edge in G.



   (a) Let H be the graph obtained from the given graph G by the above-described transformation. If G has n vertices and m edges, what is the maximum number of vertices that H can have (as a function of m and n)? Similarly, what is the maximum number of edges H can have?

   (b) Suppose G is represented as an adjacency list. Describe an efficient algorithm that constructs an adjacency list representation of H. (Recall from part (a) that H is the unweighted graph obtained from G by using the above-described transformation.)

(c) What is the running time of the above algorithm? Express your answer in asymptotic notation, as a function of m and n.

(d) Suppose you wanted to solve the Single Source Shortest Path (SSSP) problem on G. Based on your answer to (c), which one of the two options would you choose: (i) transform G into H and use BFS on H or (ii) use Dijkstra's shortest path algorithm, as it is, on G?

2. Below I provide a version of the *depth-first search (DFS)* algorithm on a directed graph in which each vertex is assigned a "discovery" time (using the array `D`) and a "finish" time (using the array `F`). I show not only the function `DFS`, but also the "main program" further below that performs initializations and also calls `DFS` repeatedly until the entire graph is explored. Note that if some of the vertices in the graph are not reachable from the first source vertex, then subsequent calls to `DFS` starting from as-yet unexplored vertices are required in order to explore the entire graph.

```
DFS(u):
  Explored[u] ← True
  time ← time + 1; D[u] ← time
  for each out-neighbor v of u do
       if not Explored[v] then
            DFS(v)
  time ← time + 1; F[u] ← time

Comment: Below is the main program that calls DFS repeatedly until the entire graph is explored.
time ← 0
for each vertex u ∈ V do
  Explored[u] ← False

for each vertex u ∈ V do
  if not Explored[u] then
       DFS(u)
```

(a) Execute the above algorithm on the directed graph given below. Assume that vertices are considered in alphabetical order in all three **for**-loops you see in the code. Thus the first call to DFS is initiated at vertex A. Draw the resulting collection of DFS trees. For each vertex, show their `D`-value and `F`-value also.

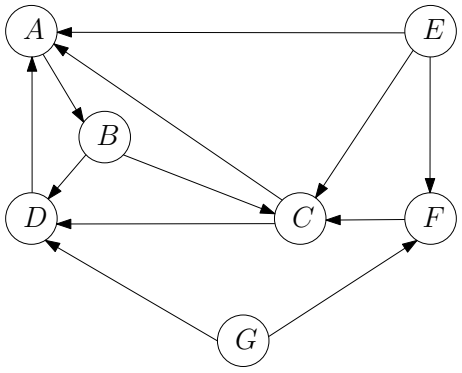


Figure 1: Directed graph to use as input for DFS.

(b) Suppose that G is a DAG. If we execute the DFS algorithm described above on G and examine the `F`-values of G, we see that the `F`-values tell us how to topologically order the vertices of G. Discover and state this connection precisely.

(c) Describe the resulting algorithm (based on computing `F`-values using DFS) for topological sorting in 2-3 sentences. What is the running time of this algorithm, stated in asymptotic notation, as a function of m and n?

3. You are given a set $X = \{x_1, x_2, \ldots, x_n\}$ of points on the real line. Your task is to design a *greedy* algorithm that finds a smallest set of intervals, each of length 2, that contains all the given points.

   Example: Suppose that $X = \{1.5, 2.0, 2.1, 5.7, 8.8, 9.1, 10.2\}$. Then the three intervals $[1.5, 3.5]$, $[4, 6]$, and $[8.7, 10.7]$ are length-2 intervals such that every $x \in X$ is contained in one of the intervals. Note that 3 is the minimum possible number of intervals because points 1.5, 5.7, and 8.8 are far enough from each other that they have to be covered by 3 distinct intervals. Also, note that my solution is not unique – for example, I can shift the middle interval $[4, 6]$ to the right, say to $[5.7, 7.7]$, without disturbing the other intervals, and we would still have an optimal solution.

   (a) Suppose that elements of X are presented in increasing order. Describe (using pseudocode) a greedy algorithm, running in $O(n)$ time, for this problem.
   Note: The space below is much larger than the space you need to answer this question!

(b) Using the "greedy stays ahead" approach that we used for the *Interval Scheduling* greedy algorithm proof, prove that your algorithm indeed produces an optimal solution. Your proof needs to be clear and precise, in addition to being correct.

4. Here are two scheduling problems that are variants of problems familiar to you. However, the greedy algorithms that worked the original problems don't work for these variants. Your task is to devise counterexamples to show this.

   (a) Recall the problem of *Scheduling to Minimize Lateness* for which we designed a greedy algorithm. Now consider a variant of this problem in which each job i has an associated deadline d_i, an associated execution time t_i, and a *release time* r_i. The release time r_i of a job imposes the constraint that job i can only be scheduled at or later than r_i because the job is only available for execution at time r_i. The original problem did not have release times and it was assumed that all jobs are available from the very beginning, i.e., at time 0.

   The greedy algorithm *Earliest Deadline First (EDF)* that worked for the original problem of Scheduling to Minimize Lateness, no longer works when we have release times. Construct a simple input for the problem (by specifying $\{d_i, t_i, r_i\}$ for all i) for which the EDF algorithm does not produce an optimal solution. Show the solution produced by EDF and compare it to the optimal solution.

   (b) A variant of the *Interval Scheduling* problem is one in which each interval has an associated non-negative weight. In this problem (called the *Weighted Interval Scheduling* problem), we want to find a set of mutually non-overlapping intervals that have the maximum total weight. For example, consider intervals $I_1 = [1, 3]$, $I_2 = [2, 4]$, and $I_3 = [3.5, 4.5]$ and suppose that $w(I_1) = w(I_3) = 1$ and $w(I_2) = 10$. Then, the optimal solution to this problem would be $\{I_2\}$ and not $\{I_1, I_3\}$ because the weight of I_2 is 10 whereas the weight of $\{I_1, I_2\}$ is $1 + 1 = 2$.

   The greedy algorithm that we used to solve the Interval Scheduling problem repeatedly picked an interval with earliest finish time and deleted other intervals that overlapped the selected interval. Show that this algorithm does not produce an optimal solution to the Weighted Interval Scheduling problem.

5. Consider the undirected edge-weighted graph shown below.
(Downloaded from `https://en.wikibooks.org/wiki/A-level_Mathematics/MEI/D1/Networks`)

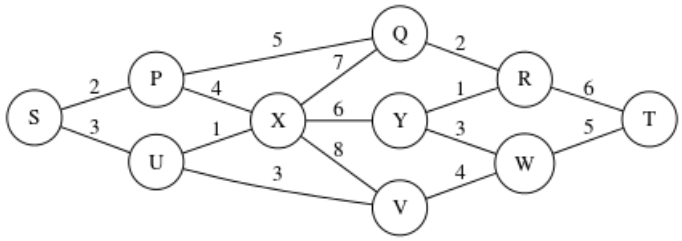


Figure 2: Undirected Edge-weighted graph to use for Dijkstra's Shortest Path algorithm.

(a) Show the execution of Dijkstra's shortest path algorithm (pseudocode given below) for solving the Single Source Shortest Path (SSSP) problem on this graph. Use the vertex `S` as the source. For each iteration of the **while**-loop show (i) the vertices in S, the set of vertices to which we know the correct distances (ii) the d'-values assigned to the vertices in $V \setminus S$ during that iteration, and (iii) the vertex v^* selected in that iteration.

$S \leftarrow \{s\}$; $d[s] \leftarrow 0$
while $S \neq V$ **do**
  for each vertex $u \in V \setminus S$ **do**
    $d'[u] \leftarrow \infty$

  for each vertex $u \in V \setminus S$ **do**
    $d'[u] \leftarrow \min_{(v,u)\in E, v\in S}\{d[v] + w(v,u)\}$

  Select a vertex $v^* \in V \setminus S$ with smallest d'-value
  $d[v^*] \leftarrow d'[v^*]$
  $S \leftarrow S \cup \{v^*\}$

(b) Dijkstra's shortest path algorithm only works for graphs with non-negaive edge weights. To prove this, construct an appropriate simple directed graph G, some of whose edge weights are negative. Show the execution of Dijkstra's shortest path algorithm on this graph and show that when Dijkstra's algorithm terminates, at least one of the d-values is wrong, i.e., does not represent the distance from the source vertex.