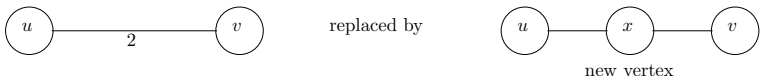
CS:3330 Exam 2 Solution, Fall 2015

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?

   Solution: The maximum number of vertices H can have is $m + n$ and the maximum number of edges H can have is $2m$.

   (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.)

   Solution: In the following code n is the number of vertices in G and m is the number of edges of G. Also, $weight(v, w)$ is just a shorthand to refer to the weight of edge $\{v, w\}$, which is stored in two places – in `HAdjList`$[v]$, along with w and in `HAdjList`$[w]$, along with v.

```
c ← n + 1; create HAdjList, an array of length m + n
for each vertex v in G do
    Create HAdjList[v].neighbors, a linked list to hold neighbors of v
    for each neighbors w of v do
        if weight(v, w) = 1 then
            HAdjList[v].neighbors.insert(w)
        else if weight(v, w) = 2 and w > v then
            HAdjList[v].neighbors.insert(c)
            HAdjList[w].neighbors.insert(c)
            Create HAdjList[c].neighbors
            HAdjList[c].neighbors.insert(v)
            HAdjList[c].neighbors.insert(w)
            c ← c + 1
```

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

   Solution: $\Theta(m + n)$ in the worst case.

   (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?

Solution: (i). The worst case running time of the transformation algorithm in (b) is $\Theta(m+n)$. Also, BFS runs in $\Theta(m+n)$ time. Therefore, using the transformation algorithm followed by BFS yields a solution to SSSP tha runs in $\Theta(m+n)$ time. The min-heap based implementation of Dijkstra's shortest path algorithm runs in $\Theta((m+n)\log n)$ time in the worst case, which is asymptotically worse than option (i).

Remarks: Some students were a bit careless about avoiding adding two new vertices for each weight 2 edges. Also, a few students had trouble with the running time analysis of the pseudocode in (b).

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.

Solution: Next to each vertex v in the figure below, I show the tuple $[\texttt{D}(v), \texttt{F}(v)]$.

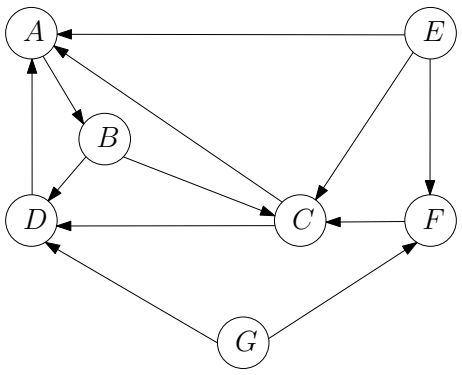
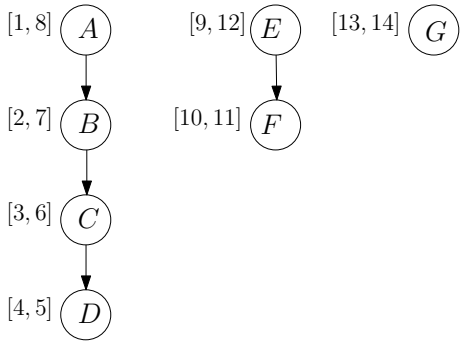


Figure 1: Directed graph to use as input for DFS.



(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.

Solution: The sequence of vertices of G in decreasing order of their `F`-value is a topologically sorted order of vertices in G.

(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?

Solution: When `DFS` completes processing a vertex u, append it to a list L. Thus L will contain vertices in increasing order of "finish" time. Reversing this list in $\Theta(n)$ time will yield a topologically sorted order of vertices. The running time of this algorithm is dominated by the running time of depth-first search and is thus $\Theta(m+n)$.

Remarks: Most students solved (a) correctly. A number of students made the correct connection between `F`-values and topological sort in part (b), but decided to solve (c) by running DFS followed by sorting. This is of course not incorrect, but unnecssarily inefficient. Even, if one wants to sort `F`-values after DFS has been completed, it can be done in $\Theta(n)$ by noting the fact that `F`-values are all distinct and come from the set $\{1, 2, \ldots, n\}$. Unfortunately, most students did not notice this feature of the `F`-values that makes it rather easy to sort, without using a full-fledged sorting algorithm.

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!

Solution: Here is the pseudocode.

```
G ← ∅; R ← −∞
for i ← 1 to n do
    if x_i > R then
        G ← G ∪ {[x_i, x_i + 2]}
        R ← x_i + 2
return G
```

(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.

Solution: Here is the proof. For any interval I, let $s(I)$ denote its left endpoint and let $f(I)$ denote its right endpoint. Let $O = \{o_1, o_2, \ldots, o_p\}$ be an optimal solution to the problem. We label the intervals in O such that $f(o_1) < f(o_2) < \cdots < f(o_p)$. Also, let $G = \{g_1, g_2, \ldots, g_q\}$ be the solution produced by the greedy algorithm described above. Label the intervals in G in the order in which they are selected by the algorithm. This implies that $f(g_1) < f(g_2) < \cdots < f(g_q)$. By definition of an optimal solution, we know that $p \leq q$. We will now show that $q \leq p$ and therefore p and q are identical and therefore G is also an optimal solution.

Big Claim: $q \leq p$.
In order to prove the "Big Claim" we first show the following "Little Claim."
Little Claim: For all i, $1 \leq i \leq p$, $f(g_i) \geq f(o_i)$. (This is the claim that establishes that "greedy stays ahead.")
Proof: We prove this claim by induction. For the **base case** consider intervals o_1 and g_1. Both of these intervals contain x_1 and therefore $f(o_1) \leq x_1 + 2$. However, our greedy algorithm picks g_1 such that $s(g_1) = x_1$ and therefore $f(g_1) = x_1 + 2$. From this, we conclude that $f(g_1) \geq f(o_1)$.

Induction hypothesis: Suppose that $f(g_i) \geq f(o_i)$ for all i, $1 \leq i \leq r < p$.
Now consider the leftmost point $x \in X$ such that $f(g_r) < x$. Note that such a point x exists because g_r is not the last interval chosen by the greedy algorithm. Our greedy algorithm will choose $[x, x + 2]$ as the next interval and therefore $g_{r+1} = [x, x + 2]$. According to the induction hypothesis, $f(o_r) \leq f(g_r)$ and therefore $f(o_r) < x$ also. This means that $s(o_{r+1}) \leq x$ and therefore $f(o_{r+1}) \leq x + 2 = f(g_{r+1})$.
End of Proof of "Little Claim."

In order to now prove the "Big Claim" we use "proof by contradiction" and suppose that $p < q$. Consider interval g_{p+1} and a point $x \in X$ that belongs to g_{p+1}. According to the claim proved above, $f(o_p) \leq f(g_p)$ and therefore x is not in any of the intervals $o_1, o_2, \ldots, o_p$, contradicting the fact that O is an optimal solution. Thus, it must be the case that $p \geq q$.
End of Proof of "Big Claim."

Remarks: Students had a lot of trouble expressing the above proof precisely. Somewhat surprisingly, some students also had trouble with expressing the pseudocode in part (a) precisely.

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.

   Solution: Consider two jobs $J_1 = (3, 1, 2)$ and $J_2 = (4, 2, 0)$. The EDF algorithm outputs the schedule (J_1, J_2). In this schedule J_1 starts at time 2 because that is when it is released. J_1 completes at time 3 and is not late. J_2 starts at time 3, completes at time 5, and is 1 unit late.

   Now consider the schedule (J_2, J_1). In this schedule J_2 starts at time 0, completes at time 2, and is not late. Then, J_1 starts at time 2, completes at time 3, and is also not late.

   Thus, there is a schedule that has smaller lateness than he schedule produced by EDF.

   (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.

   Solution: Consider the example presented in the problem description above. For this example, the greedy algorithm that picks intervals by earliest finish times will produce the solution $\{I_1, I_3\}$, with weight 2. But, the solution $\{I_2\}$ has weight 10 and therefore this greedy 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`)

   (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.

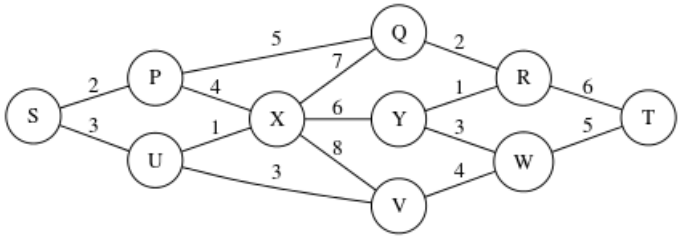


Figure 2: Undirected Edge-weighted graph to use for Dijkstra's Shortest Path algorithm.

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

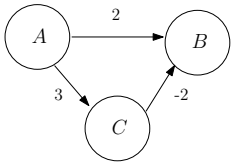
  for each vertex $u \in V \setminus S$ **do**
    $d'[u] \leftarrow \min_{(v,u)\in E, v\in S}\{d[v] + w(v,u)\}$

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

Solution: The following table of d'-values after each iteration shows the progress of Dijkstra's algorithm on the given graph. The asterisk next to a number indicates that the corresponding vertex had the smallest d'-value in that iteration and was therefore chosen to join S. The - symbols in each row indicate vertices that have already joined S.

P	Q	R	S	T	U	V	W	X	Y
2*	∞	∞	-	∞	3	∞	∞	∞	∞
-	7	∞	-	∞	3*	∞	∞	6	∞
-	7	∞	-	∞	-	6	∞	4*	∞
-	7	∞	-	∞	-	6*	∞	-	10
-	7*	∞	-	∞	-	-	10	-	10
-	-	9*	-	∞	-	-	10	-	10
-	-	-	-	15	-	-	10	-	10*
-	-	-	-	15	-	-	10*	-	-
-	-	-	-	15*	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-

(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.



Solution: Suppose we use Dijkstra's shortest path algorithm to solve the SSSP problem from vertex A in the 3-node graph above. The algorithm will report that the distance from A to B is 2. However, there is a shorter path A-C-B that has length 1.