Solutions to Practice Problems 3

Solution 1:

(a) The vertices are extracted in the order s, a, b, c. The d-values immediately after each extraction are shown below.

extracted	$d[s]$	$d[a]$	$d[b]$	$d[c]$
(initial)	0	∞	∞	∞
s	0	2	4	∞
a	0	2	4	5
b	0	2	4	5
c	0	2	4	5

Note: When b is extracted, the edge (b, a) is examined and $d[b] + w_{ba} = 4 - 4 = 0 < 2 = d[a]$. But a has already been extracted, and the version of Dijkstra given in class never revisits a vertex once it has been extracted, so $d[a]$ is not updated.

The true shortest distances are $\delta(s, a) = 0$ (along s, b, a), $\delta(s, b) = 4$, and $\delta(s, c) = 3$ (along s, b, a, c). The algorithm reports $d[a] = 2$ and $d[c] = 5$, so the d-values of *both* a and c are incorrect.

(b) Yes. Let (u, v) be the unique negative edge and let $G' = G - (u, v)$ be the digraph with that negative edge deleted. Every edge of G' has nonnegative weight, so Dijkstra's algorithm is valid on G'. Run it twice:

- from s, obtaining $d_s(x)$ for all x, and
- from v, obtaining $d_v(x)$ for all x.

Then report, for every vertex x,

$$\delta(s, x) \;=\; \min\big\{\, d_s(x),\;\; d_s(u) + w + d_v(x) \,\big\}.$$

Correctness. Since G has no negative-cost cycle, some shortest path from s to x is simple (no vertex repeated), and therefore uses the edge (u, v) at most once. There are two cases:

1) The shortest path does not use (u, v) at all. Then it is a path of G', so its weight is $d_s(x)$.

2) The shortest path uses (u, v) exactly once. It splits as a path from s to u, the edge (u, v), and a path from v to x. Both $s \to u$ and $v \to x$ paths lie in G' and the total weight is $d_s(u) + w + d_v(x)$.

Running time. Two calls to Dijkstra's algorithm take $O((n + m) \log n)$ time with the heap-based implementation, and combining the results takes $O(n)$ time, so the total is asymptotically the same as a single call.

Solution 2:

(a) Relaxing the edges in the order $(s,a),(a,b),(c,d),(d,c),(c,s)$ gives the following.

after round	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$
(initial)	0	∞	∞	∞	∞
1	0	3	5	∞	∞
$2,3,4$	0	3	5	∞	∞

The values stop changing after the first round. On the additional pass, the edges (s,a) and (a,b) hold with equality, and each of (c,d), (d,c) and (c,s) has $d[u]=\infty$ at its tail, so no relaxation is possible. DETECT therefore reports that there is *no* negative-cost cycle.

This answer is wrong: the cycle $c \to d \to c$ has weight $1+(-4)=-3<0$. The reason is that c and d are not reachable from s, so their d-values remain ∞ forever and the test $d[u]+w_{uv}<d[v]$ never fires on the edges of the cycle.

(b) (i) Let v be any vertex. Since v is reachable from s, there is a simple path from s to v, say $s=x_0,x_1,\ldots,x_k=v$. Its vertices are distinct, so $k\le n-1$.
We show by induction on i that $d[x_i]$ is finite after i rounds of relaxation. For $i=0$ this holds because initialization sets $d[s]=0$. For the inductive step, suppose $d[x_{i-1}]$ is finite after round $i-1$. Round i relaxes every edge of G, including the edge (x_{i-1},x_i). At the moment that edge is relaxed, $d[x_{i-1}]$ is finite. Hence after that relaxation $d[x_i]\le d[x_{i-1}]+w_{x_{i-1}x_i}<\infty$.
Therefore $d[v]$ is finite after $n-1$ rounds, and it remains finite for the rest of the execution. Since v was arbitrary, every d-value is finite after $n-1$ rounds.

(ii) Recall the invariant of Bellman-Ford: after k rounds of relaxation, $d[v]$ is at most the minimum weight of a path from s to v that uses at most k edges.
Let v be any vertex and let P be a shortest path from s to v. Since G has no negative-cost cycle, we may take P to be simple, so it uses at most $n-1$ edges. Taking $k=n-1$, the invariant gives $d[v]\le w(P)=\delta(s,v)$. On the other hand, $d[v]$ is always the weight of some actual path from s to v, so $d[v]\ge\delta(s,v)$. Therefore $d[v]=\delta(s,v)$ for every vertex v after $n-1$ rounds.
Now consider any edge (u,v). A shortest path from s to u followed by the edge (u,v) is a path from s to v, so $\delta(s,v)\le\delta(s,u)+w_{uv}$, that is, $d[v]\le d[u]+w_{uv}$. Hence no edge can be relaxed.

(iii) Suppose no edge can be relaxed, so that

$$d[v] \;\le\; d[u]+w_{uv} \qquad \text{for every edge } (u,v)\in E.$$

Let $C=(x_0,x_1,\ldots,x_k)$ with $x_k=x_0$ be an arbitrary cycle of G. Applying the inequality to each of its edges and summing,

$$\sum_{i=1}^{k} d[x_i] \;\le\; \sum_{i=1}^{k} d[x_{i-1}] \;+\; \sum_{i=1}^{k} w_{x_{i-1}x_i} \;=\; \sum_{i=1}^{k} d[x_{i-1}] \;+\; w(C).$$

Because C is a cycle, $x_0=x_k$, so the sets $\{x_1,\ldots,x_k\}$ and $\{x_0,\ldots,x_{k-1}\}$ are the same and the two sums of d-values are equal. Since every d-value is finite, we may cancel them, leaving $0\le w(C)$. Therefore, any cycle has non-negative weight.

(c) Add a new vertex s' to G, together with an edge (s', v) of weight 0 for every $v \in V$, and call the result G'. Run DETECT(G', s').

Correctness. Every vertex of G' is reachable from s' in one edge, so every d-value is finite and part (b) applies: DETECT correctly decides whether G' has a negative-cost cycle. Now we want to show G' has a negative-cost cycle if and only if G has one. Since G is a subgraph of G', every cycle of G is a cycle of G'. Conversely, the modification of the graph will not introduce new cycles. s' has no incoming edges, so no cycle of G' passes through s', and every cycle of G' therefore uses only original edges and is a cycle of G. Thus G' has a negative-cost cycle if and only if G does, and in particular the modification creates no new negative-cost cycle.

Running time. G' has $n + 1$ vertices and $m + n$ edges, so the cost is $O\big((n + 1)(m + n)\big) = O(nm + n^2)$, which is $O(nm)$ whenever $m \geq n$.

Solution 3:

(a) *Algorithm.*

1. Traverse T from s (by BFS or DFS), computing for each vertex the weight of its tree path: set $d[s] = 0$, and when the traversal moves from u to a child v along the tree edge (u, v), set $d[v] = d[u] + w_{uv}$.
2. For every edge $(u, v) \in E$ of the whole digraph, test whether $d[u] + w_{uv} \geq d[v]$. If some edge fails this test, reject.
3. Otherwise accept.

Step 1 visits each of the $n - 1$ tree edges once, so it takes $O(n)$ time. Step 2 scans each adjacency list once, so it takes $O(n + m)$ time. The total is $O(n + m)$.

(b) (i) Suppose the algorithm rejects, because some edge (u, v) has $d[u] + w_{uv} < d[v]$. Follow the tree path from s to u, which is a path of weight $d[u]$, and then take the edge (u, v). This is a path from s to v in G of weight $d[u] + w_{uv} < d[v]$. Since $d[v]$ is the weight of T's path to v, that tree path is not a shortest path, and so T is not a shortest-path tree.

(ii) Suppose the algorithm accepts, so $d[v] \leq d[u] + w_{uv}$ for every edge $(u, v) \in E$. Let v be any vertex and let P be any path from s to v, say $s = x_0, x_1, \ldots, x_k = v$. Applying the inequality to each edge of P gives $d[x_i] - d[x_{i-1}] \leq w_{x_{i-1}x_i}$, and summing over $i = 1, \ldots, k$ the left-hand side collapses to end terms only:

$$LHS = \sum_{i=1}^{k} \big(d[x_i] - d[x_{i-1}]\big) = d[v] - d[s]$$

$$RHS = \sum_{i=1}^{k} w_{x_{i-1}x_i} \;=\; w(P)$$

$$d[v] - d[s] \leq w(P)$$

Since $d[s] = 0$ we get $d[v] \leq w(P)$. As P was an arbitrary path from s to v, this gives $d[v] \leq \delta(s, v)$. Conversely $d[v]$ is the weight of an actual path from s to v, so $d[v] \geq \delta(s, v)$. Hence $d[v] = \delta(s, v)$ for every v, which says exactly that the path of T from s to v is a shortest path. So T is a shortest-path tree.

Solution 4:

(a) Let t^* be a time at which the depth equals d, and let L be the set of d lectures whose intervals contain t^*. Any two lectures of L share the time t^*, so they overlap and hence conflict. In any valid schedule, conflicting lectures occupy different classrooms, so the d lectures of L occupy d distinct classrooms. Therefore every valid schedule uses at least d classrooms.

(b) Algorithm: Sort the lectures by start time, so that $s_1 \leq s_2 \leq \cdots \leq s_n$. Maintain a min-heap containing one entry per open classroom, keyed by the finish time of the last lecture assigned to that classroom. For each lecture i in sorted order, look at the minimum key f of the heap. If the heap is nonempty and $f \leq s_i$, that classroom is free: remove its entry, assign lecture i to it, and reinsert it with key f_i. Otherwise open a new classroom for lecture i and insert an entry with key f_i.

The schedule produced is valid: a classroom is reused only when the last lecture in it finishes at or before s_i, and since lectures are processed in order of start time every earlier lecture in that classroom finishes no later, so lecture i conflicts with none of them.

(c) Suppose the algorithm opens the k-th classroom while processing lecture i. At that moment the other $k-1$ classrooms were all busy, that is, each contains a lecture j with $f_j > s_i$. Also $s_j \leq s_i$, because j was processed earlier and the lectures are sorted by start time. So each such lecture j satisfies $s_j \leq s_i < f_j$, meaning its interval contains the time s_i. Together with lecture i itself, that is k lectures whose intervals contain s_i, so the depth at time s_i is at least k, giving $k \leq d$. Hence the algorithm never opens more than d classrooms, and by part (a) it never uses fewer, so it uses exactly d.

Running time. Sorting takes $O(n \log n)$. Each lecture causes $O(1)$ heap operations on a heap of at most n entries, for $O(n \log n)$ total. The overall running time is $O(n \log n)$.

(d) The proposed variant can use more than d classrooms. Consider the four lectures

$$[0, 1), \qquad [0, 2), \qquad [1, 4), \qquad [2, 3).$$

The depth of this instance is 2: at time 0 only $[0, 1)$ and $[0, 2)$ are running, at time 1 only $[0, 2)$ and $[1, 4)$, and at time 2 only $[1, 4)$ and $[2, 3)$. Sorted by finish time the order is $[0, 1), [0, 2), [2, 3), [1, 4)$, and the variant assigns

lecture	classroom
$[0, 1)$	R_1 (first classroom opened)
$[0, 2)$	R_2 (conflicts with $[0, 1)$ in R_1)
$[2, 3)$	R_1 (free, since $[0, 1)$ has finished)
$[1, 4)$	R_3 (conflicts with $[2, 3)$ in R_1 and with $[0, 2)$ in R_2)

so it uses 3 classrooms where 2 suffice. The point is that sorting by finish time destroys the property used in part (c): the lectures blocking the existing classrooms need no longer all be running at a common time.

Solution 5: *Algorithm.* Drive as far as possible before each stop. Let i be the index of the current position, starting at $i = 0$ (so the car is at $x_0 = 0$ with a full charge). Repeatedly let j be the largest index with $x_j - x_i \le R$. If $j = n$, the car can reach the destination without another stop and the algorithm halts. Otherwise recharge at x_j, increment the counter, and set $i = j$. Since consecutive gaps never exceed R we always have $j > i$, so the process makes progress and terminates.

Correctness. Let $g_1 < g_2 < \cdots < g_p$ be the stops chosen by the greedy algorithm and let $o_1 < o_2 < \cdots < o_q$ be the stops of any optimal solution, both written as positions. We claim $g_k \ge o_k$ for every $k \le q$, that is, after the same number of stops the greedy solution is never behind. We prove this by induction. For $k = 1$, both start at x_0 with the same range, and the greedy rule chooses the farthest reachable station, so $g_1 \ge o_1$. Assume $g_{k-1} \ge o_{k-1}$. The optimal solution reaches o_k from o_{k-1}, so $o_k - o_{k-1} \le R$, hence $o_k \le o_{k-1} + R \le g_{k-1} + R$; that is, o_k is reachable from g_{k-1}. Since greedy picks the farthest station reachable from g_{k-1}, we get $g_k \ge o_k$.

Since the optimal solution uses the minimum number of stops, $p \ge q$, so g_q exists. The optimal solution reaches the destination from its last stop, so $x_n - o_q \le R$. By the claim $g_q \ge o_q$, hence $x_n - g_q \le R$ as well: after its q-th stop the greedy solution can already reach the destination, so it makes no further stops and $p = q$. Therefore the greedy algorithm is optimal.

Running time. Both i and j only ever move forward through the list of stations, so the scan examines each station a constant number of times, giving $O(n)$ time after the input is read.

Solution 6:

(a) *S is independent.* When a vertex v is added to S, every neighbor of v is deleted from the current graph at the same moment, so no vertex adjacent to v can be selected later. Hence no two vertices of S are adjacent.

S is maximal. The loop runs until the graph is empty, so every vertex of G was eventually removed, either by being selected (and placed in S) or by being deleted as a neighbor of a selected vertex. Let $u \notin S$. Then u was deleted as a neighbor of some $v \in S$, so u is adjacent to v and $S \cup \{u\}$ is not independent. Thus no vertex can be added to S, and S is maximal.

Example: The star. For $n \ge 2$ let G be the star $K_{1,n-1}$: one center vertex c adjacent to each of the $n-1$ leaves, and no other edges. If MaximalIS selects c on its first iteration, it deletes c together with all of its neighbors, which is every vertex, and halts with $S = \{c\}$ of size 1. However, all leaves are independent so the maximum independent set has size $n - 1$.

(b) Represent the current graph by an array $alive[1..n]$ of flags, all initially true, and keep a pointer that sweeps once through the vertices. To select a vertex, advance the pointer to the next v with $alive[v]$ true; add v to S, set $alive[v] \leftarrow$ false, and scan the adjacency list of v, setting $alive[u] \leftarrow$ false for every neighbor u.

The sweep costs $O(n)$ in total. An adjacency list is scanned only when its vertex is selected, and a vertex is selected at most once, so the scanning costs at most $\sum_{v \in V} \deg(v) = 2m$. The total is $O(n + m)$.

(c) Enumerate all 2^n subsets $S \subseteq V$. For each one, test independence by examining every pair of vertices of S and checking whether the pair is an edge, which takes $O(n^2)$ time per subset. Keep the largest independent subset seen. The algorithm is correct because a maximum independent set is one of the subsets examined, and the running time is $O(2^n n^2)$.

(d) By construction two vertices of the interval graph are adjacent exactly when the corresponding intervals overlap. So a set S is independent if and only if no two of its intervals overlap, which is to say the intervals of S are pairwise disjoint. A maximum independent set of G is therefore a largest set of pairwise disjoint intervals, which is exactly the interval scheduling problem solved in class by the earliest-finish-time greedy algorithm. That algorithm sorts the intervals by finish time in $O(n \log n)$ time and then makes a single linear scan, and it works directly on the list of intervals: the graph G is never built, which matters because G may have $\Theta(n^2)$ edges.

(e) The three parts compare as follows.

algorithm	running time	guarantee
MaximalIS, any graph	$O(n + m)$	maximal
brute force, any graph	$O(2^n n^2)$	maximum
earliest finish time, interval graphs	$O(n \log n)$	maximum

On a general graph we could have speed or optimality but not both. On an interval graph we get both, and the reason is structural: the intervals are totally ordered by finish time, and "finishes earliest" leaves the most room for the intervals still to be chosen. That is what makes the greedy exchange argument go through, and a general graph offers no analogue of it.