Solutions to Homework 1: DFS, Paths and Greedy

Solution 1:

We have omitted the drawing of G^R. For parts (b)–(d), see Fig. 1. The final strong components are $\{f, g\}$, $\{i, j, h\}$, $\{c, d, e\}$, and $\{a, b\}$.

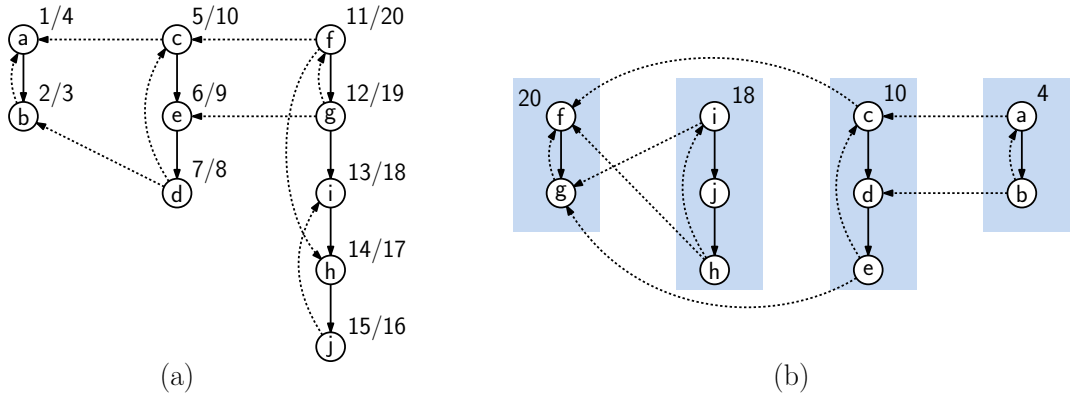

Figure 1: Strong components: (a) DFS on G^R and (b) final DFS with finish times shown for roots.

Solution 2:

(a) The fact that the digraph is acyclic follows immediately from the fact that edges are directed in increasing order of depth.

(b) To assign depths, first find all vertices with in-degree 0. This can be done in $O(n+m)$ time as follows. Associate an in-degree counter with each vertex, which is initialized to zero. Traverse the adjacency list, and for each edge (u, v) seen, increment v's in-degree counter.

Next, perform a DFS on the digraph starting each tree at a vertex with in-degree zero. Assign its depth to be 1. Whenever a vertex u discovers a new vertex v, set $\text{depth}[v] \leftarrow \text{depth}[u] + 1$. (As an additional check on the correctness of the graph, whenever a forward or cross edge is seen, we can check that the depth increases by one.) Since this is DFS, it can be done in $O(n + m)$ time.

(c) Let V_1 be the set of vertices for regions of even depth and let V_2 be the vertices of regions of odd depth. Since each edge connects two vertices whose depths differ by one, each edge will have one endpoint in V_1 and one in V_2.

Solution 3:

(a) An example is shown in Fig. 2, where s is the source vertex.

- The algorithm first processes s, which sets $d[a] = 2$ and $d[b] = 4$.

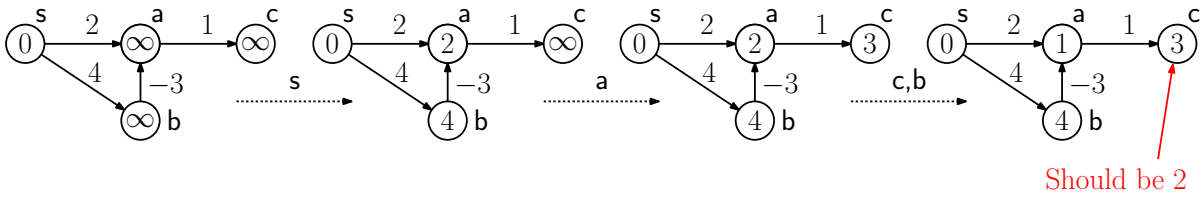

Figure 2: Why Dijkstra fails on graphs with negative edge weights.

- Next is a at distance $d[a] = 2$. This performs relax(a, c), which sets $d[c] = 3$.
- Next is c at distance $d[c] = 3$, which does nothing, since c has no outgoing edges.
- Finally, is b at distance $d[b] = 4$. This performs relax(b, a), which sets $d[a] = 1$.

Observe that on termination, $d[c] = 3$, but in fact $\delta(s, b) = 2$ by the path $\langle s, b, a, c\rangle$. (Note that this assumes the version of Dijkstra's algorithm presented in class, which performs the relax operator on all outgoing edges, not just the edges to unprocessed vertices. Different variants of Dijkstra's algorithm will have different counterexamples.)

(b) We will do this by recalling the proof of Dijkstra's algorithm given in the lecture, and showing that these negative edges do not affect the correctness proof.

The crux of the correctness proof for Dijkstra is showing that, whenever a vertex u is processed, $d[u]$ is equal to the true distance $\delta(s, u)$. Suppose to the contrary that this is not true, and consider the *first* finished vertex u for which this fails to hold, which implies that $d[u] > \delta(s, u)$.

Let S denote the set of processed vertices, just prior to processing u. The true shortest path from s to u must exit S along some edge (x, y), where $x \in S$ and $y \notin S$ (see Fig. 3).

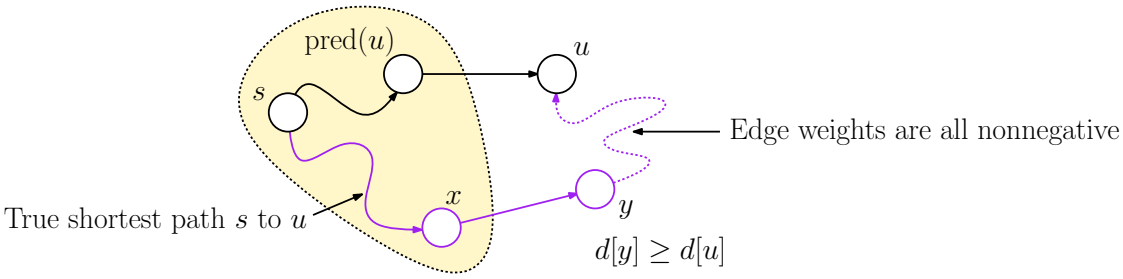

Figure 3: Correctness of Dijkstra's Algorithm even with negative edges from the source.

Because u is the first vertex where we made a mistake, and since x was already processed, we have $d[x] = \delta(s, x)$. When the algorithm processes x, it performs relax(x, y), which implies that

$$d[y] \;=\; d[x] + w(x, y) \;=\; \delta(s, x) + w(x, y) \;=\; \delta(s, y).$$

Observe that the remainder of the path from y to u will not revisit s, since if that were to happen, it would mean that there is a negative cost cycle from s through y and back to s. Since we do not revisit s, all the edges along the path from y to x are nonnegative. Therefore,

$\delta(s, y) \leq \delta(s, u)$. Because u was chosen before y for processing, we have $d[u] \leq d[y]$. Putting this together, we have

$$\delta(s, u) \ < \ d[u] \ \leq \ d[y] \ = \ \delta(s, y) \ \leq \ \delta(s, u).$$

Clearly we cannot have $\delta(s, u) < \delta(s, u)$, which establishes the desired contradiction. Therefore, Dijkstra's algorithm is correct.

An alternative argument is based on the idea that we can transform G into a graph that has no negative weight edges, but behaves exactly in the same manner from the perspective of Dijkstra's algorithm. First, let's assume that there is at least one negative weight edge emanating from s, for otherwise, all the edge weights are nonnegative, and Dijkstra's algorithm is correct. Let $-W < 0$ denote the smallest weight among all the edge emanating from s. Consider a modified graph G' which results by increasing the weights of all edges emanating from s by $+W$ (see Fig. 4). Let G' denote the resulting graph. Now, when we initialize Dijkstra's algorithm, we compensate for the change by setting $d[s] = -W$. Observe that the costs of all the paths in G' are exactly the same as in G. Furthermore, since all the edges now have nonnegative weights, Dijkstra's algorithm operates on a graph with nonnegative edge weights, and hence is correct.

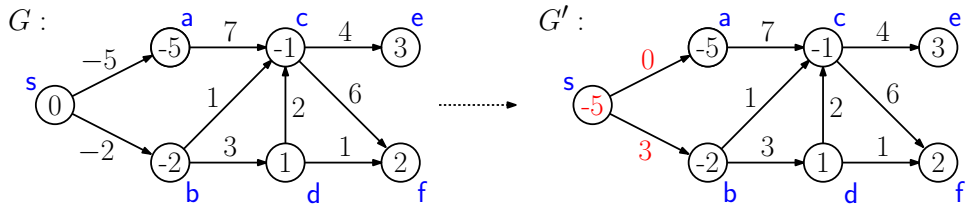

Figure 4: Transforming a graph into an equivalent one with nonnegative edge weights.

Solution 4:

(a) Yes, the greedy segmentation is optimal for minimizing the total penalty. To prove this we will use a standard exchange argument to show that it is possible to convert an optimal solution into the greedy solution without increasing the sum of penalties. Let G denote a greedy layout and let O denote any optimal layout. If $G = O$, then we are done. Otherwise, let w_i denote the first word where the two layouts differ. (In Fig. 5(a) this is w_3.)

Let k be the index of the current line. Since greedy always puts as many words on a line as it can fit, it must be that greedy puts word w_i on line k and opt places it on a later line. We may assume that there are no empty lines in the optimal segmentation, and therefore this word appears at the start of line $k + 1$.

Now, consider a new layout O' that is equivalent to O, but in which we move w_i from line $k + 1$ to line k (see Fig. 5(a)). Clearly, O' is also a valid layout. (The current line had space for w_i, since greedy put it there, and the next line has even more space since it now has one fewer word.)

We assert that this exchange does not increase the sum of penalties. After the exchange, the penalty for line k decreases by w_i and the penalty for line $k + 1$ increases by w_i, so the

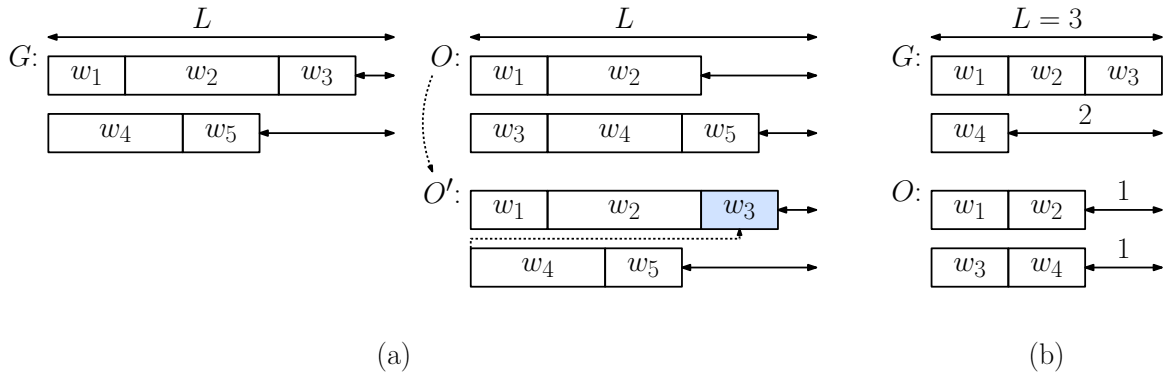

Figure 5: (a) Proof that the greedy segmentation optimizes the total penalty and (b) a counterexample showing that it does not minimize the max penalty.

net penalty change is zero. Thus, the overall sum of penalties remains unchanged. (This only shows that the two penalties are equal. However, if w_i had been only word on its line, after the change the entire line is empty, and we could strictly reduce the total penalty by eliminating the line. However, this would imply that there is a segmentation of lower penalty than O, which yields a contradiction.)

By repeating this operation, an induction argument shows that we eventually convert O into G without ever increasing the sum of penalties, which implies that greedy itself is optimal.

(b) No, the greedy heuristic is not optimal for minimizing the maximum line penalty. A counterexample arises by setting $L = 3$, and having $w_1 = w_2 = w_3 = w_4 = 1$ (see Fig. 5(b)). The greedy algorithm generates a layout with the first three words on the first line (penalty 0) and the last word on the second line (penalty 2), for a maximum penalty of 2. However, putting two words on each line (penalties 1 and 1) yields a lower maximum penalty of 1.

You might protest that it is not reasonable to include the last line in the penalty. We can adjust the counterexample to work even in this case. Place a word w_5 after w_4 that is almost as long as L. In either case, this word will need to placed on a line by itself, and w_4 will not be on the last line.

Solution to the Challenge Problem: I know of no "elegant" solution to this problem. My solution involves a number of primitive operations, each of which takes $O(n)$ time to compute. This is followed by a process of elimination to first find the head and then classify all the vertices.

Observe that feet and head have degree one, the neck has degree two, and the shoulder and hip have degree three or higher. We will maintain a set of vertex marks. To start, we initialize mark$[v] = 0$, for all vertices v. Here are some useful utilities, which run in $O(n)$ time. Given any vertex u:

Compute deg(u): Sum the 1's in this row u.
Select any neighbor of u: Return the smallest v such that $A[u, v] = 1$.
Mark the neighbors of vertex u: Set mark$[v] = 1$, for each v such that $A[u, v] = 1$.
Select any (unmarked) vertex: Return the smallest v such that mark$[v] = 0$.

Here are some useful primitives:

Is u the head? Check that u's degree is 1, and that its neighbor has degree 2. If so, u is the head and otherwise it is not.

Is u the neck? First check that u's degree is 2. If not, then it is not the neck. Otherwise, let v and w be its two neighbors. Compute the degrees of v and w. If either has degree 1, then u is the neck, and otherwise it is not. Observe that if we determine that u is the neck, *we find the head as side effect* (namely the neighbor that has degree 1).

Given that u is the shoulder or hip, find the head: Mark u and all its neighbors. Note that this marks both the shoulder and hip. Take any unmarked vertex x, and let v be any neighbor of x. Vertex x might be a foot, the neck, or the head. Apply the above two tests to determine whether x is either the head or neck, in which case we will have found the head. Otherwise, x is a foot and v is either the hip or shoulder, but not the same as u. Without reseting the marks, mark all of v's neighbors. Now, the only unmarked vertex is the head, which we return.

Observe that each of the above utilities can be performed in $O(n)$ time, since they involve a constant number of the basic primitives. Our first objective is to find the head. We'll see below why this suffices.

(1) Take any vertex w. Check whether w is the neck. If so, we have found the head as a side effect, and we are done.
(2) Otherwise, compute $\deg(w)$. If $\deg(w) = 1$:
    (a) Let u be w's only neighbor. (Observe that w is either the head or a foot, and so u is either the neck, shoulder, or hip.)
    (b) By the above utility, check whether u is the neck. If so, w is the head, and we are done.
    (c) If not, we know that u is either the shoulder or hip. Apply the above utility to find the head, and we are done.
(3) Otherwise, ($\deg(w) > 1$ and w is not the neck). It follows that w is either the shoulder or hip. Apply the above utility to find the head, and we are done.

Observe that we have applied our utilities a constant number of times, so the total time to find the head is $O(n)$. Once the head is known we proceed as follows to classify the remaining vertices:

(1) Reset all the marks.
(2) Letting u denote the head, find its only neighbor w, which we label as the neck.
(3) Letting s denote w's other neighbor, we know that s is the shoulder.
(4) Mark u, s, and all of s's neighbors. This includes the neck w, all the front-feet, and the hip. The only unmarked vertices are the rear feet.
(5) Select any unmarked vertex (a rear foot). Let h denote its only neighbor, which must be the hip.
(6) Knowing the head u, neck w, shoulder s, and hip h, all that remains is to classify the feet. The front feet consists of all of s's neighbors (excluding w and h). The rear feet consists of all of h's neighbors (excluding s). We have now classified all the vertices.

Again, this has involved only a constant number of utilities, so the final classification takes $O(n)$ time. Clearly, the entire algorithm takes $O(n)$ time. Whew!

Note that this algorithm uses $O(n)$ additional space for storing the marks. The problem can be solved in $O(n)$ time, using only $O(1)$ additional storage. (Thanks to Shuhao for pointing this out to me.)