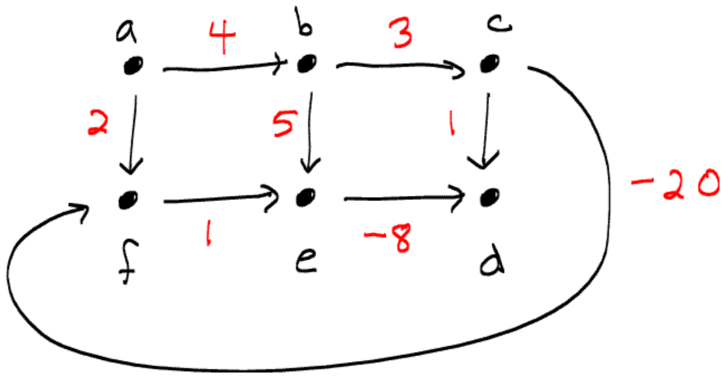
Exercises: shortest paths revisited (Bellman-Ford, Floyd-Warshall)

Questions

1. In the lecture, we saw that we did't need to use O(n^2) space for Bellman Ford by keeping track of the previous node in a shortest path. Could we apply similar idea to the problems we have seen in the past few lectures, namely can we simplify the backtracking step by storing some useful information as we build up the Opt[] table ? For each case, say whether this helps performance in the O( ) sense, either in space or time.
   - weighted interval scheduling
   - segmented least squares
   - sum of subsets, knapsack

   In the solutions, I am not going to re-explain how those particular techniques work, e.g. what the recurrences are and how they are used.

2. Run Bellman-Ford on the following graph starting at vertex **a**, namely find the shortest path to each vertex and find the prev[] value which allows you to reconstruct the paths.
   Note I have added some Bellman-Ford python code next to the lecture notes. If someone feels like converting it to Java for non-python folks, that would be great.
   The tester has this particular example. You can play with it if you want more practice.



3. How could backtracking be done for Floyd-Warshall. That is, suppose you've computed the cost of minimal cost paths for all pairs i, j which is represented by Opt[N][i][j]. How could you use the Opt[ ][ ][ ] matrix to reconstruct the path itself?

.

Answers

1.

- For interval scheduling, you iterate over i = 1 to N and you need to choose one of two options. You then build up a 1D table to store the Opt[] values. You could write down, for each I, whether you used interval i or not (use a boolean 1D array). This would allow you to avoid having to do that check when backtracking, though you would still need to check the Boolean array so it wouldn't save you much. In a O() sense it saves you nothing to do this.
- For segmented least squares, we build up a 1D Opt[j] table by considering all i in 0 <= i <= j and taking a minimum over i. If we stored that chosen value of i in some table (called which_i_gives_min[j]) , then when we backtrack we wouldn't need to recompute it. Unlike with Bellman Ford where we could avoid storing the 2D Opt[][] table, here we only have a 1D Opt[] table to begin. So adding another 1D table to keep track of the "which_i_gives_min[j] " costs us nothing in the O() sense. The backtracking step itself would become O(n) rather than O(n^2), but the computation would remains O(n^2) because we need to build up the e_ij table.
- For sum of subsets, I don't see how you avoid representing the whole Opt[][] table, or some other representation that would be O(N W). [ADDED March 11.] The reason is that you need the whole table in order to do the backtracking, namely given Opt[N][W] go backwards through the table and figure out which of the w_i could be used to get that value. If you have an idea, though, please post on the discussion board.

2.

#edges	a	b	c	d	e	f
0	0	Infinity	Infinity	Infinity	Infinity	infinity
1	0	4	Infinity	Infinity	Infinity	2
2	0	4	7	Infinity	3	2
3	0	4	7	-5	3	-13
4	0	4	7	-5	-12	-13
5	0	4	7	-20	-12	-13

vertex	a	b	c	d	e	f
Prev[vertex]	-	a	b	e	f	c

3. Here I sketch the basic idea. For any specific (j, k) pair, we need to find the intermediate vertices on the path (v_j, …. v_k). To backtrack, we first ask whether vertex v_N was used as an intermediate vertex. If j or k happens to be N, then the answer is no and we do nothing. Otherwise, we consider the minimum cost paths (v_j, …, v_N) and (v_N, …, v_k), noting that these paths don't use v_N as an intermediate vertex (since that would create a cycle - see

lecture for why we ignore cycles).   We then look at the sum of the costs of these two paths and see if this sum is less than Opt[N-1][j][k]. If it is, then the minimum cost path from v_j to v_k does indeed have v_N as an intermediate vertex.   So this path is (v_j, … v_N, … v_k).   We would then recursively call this procedure, asking for the intermediate vertices on the path from v_j to v_N which uses only v_1 to v_{N-1} as intermediate vertices. We also ask for the intermediate vertices on the path from v_N to v_k which uses only v_1 to v_{N-1} as intermediate vertices.

The trick is how to represent the paths, since that's what we're interested in here.   Let's say the recursive call findpath(v_j, v_k, N) should return a list, namely the path (v_j, …, v_k) which consists of v_j and v_k at the front and end of the list, and the intermediate vertices in the middle.   Let's say that Opt[][][] was consulted as described above and it was concluded that v_N is indeed in the minimum cost path from v_j to v_k.   Then two recursive calls, findpath(v_j, v_N, N-1) and findpath(v_N, v_k, N-1), are made.   These return lists (v_j,…, v_N) and (v_N, …, v_k) which can be concatenated – after removing one of the v_N to avoid two copies of it.

What I just described was the procedure for findpath(v_j, v_k, N).   But a similar procedure occurs for any findpath(v_j, v_k, i) call during the backtracking, namely it returns the vertices on a path (v_j, …, v_k) that only uses vertices up to v_{i-1} as intermediates.