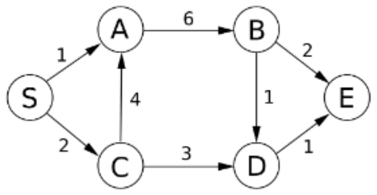
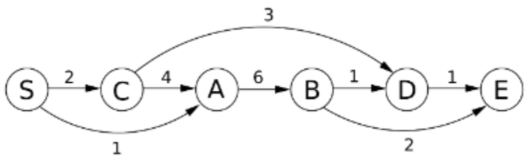
CS 170 DIS 06

Released on 2018-10-08

1 Shortest Paths with Dynamic Programming!





In this problem we will design a dynamic programming algorithm for finding the shortest $S-E$ path in a DAG like the one above.

(a) What are the subproblems you need to solve?

(b) Can you define the optimal solution to any subproblem as a function of the solutions to other subproblems?

(c) The dependency graph of a DP is a directed graph in which each subproblem becomes a vertex, and edge (u, v) denotes that subproblem u requires the solution to subproblem v in order to be solved. What does the dependency graph look like for this problem? What property of the dependency graph allows us to solve the problem using DP?

(d) If we proceed iteratively, what would be the best order to solve these subproblems?

(e) How many subproblems are there, and how long does it take to solve each? What is the overall running time of this algorithm?

(f) Run your algorithm on the example graph. What is the shortest path?

Solution:

(a) We need to know the minimum length paths from the start node to the nodes on possible paths from S to E. This will allow us to evaluate which nodes we should include on our path.

(b) Yes. We know that out of all direct ancestors A_i of E, at least one of them must be on the shortest path from S to E. Let $dist(V)$ denote the subproblem of finding the shortest path from S to some vertex V. For any U in the graph, we can argue that shortest distance from S to U is $min\{dist(A_1) + l(A_1, U), ..., dist(A_N) + l(A_N, U)\}$.

(c) Let's think about the dependency graph this gives us. We know $dist(V)$ is dependent on the $dist(A_i)$ for all direct ancestors of V. Thus the dependency graph will look like the reverse of the given graph! And since our given graph is a DAG, we know the dependency graph will also be a DAG. The fact that the dependency graph is a DAG is quite significant. By construction the dependency graph is directed. But more importantly, if the dependency graph is acyclic, we know that we don't have some set of subproblems that are all mutually dependent on each other (and thus impossible to solve). When the dependency graph of a problem is a DAG, we sometimes say that this problem possesses an "optimal substructure" that makes it solvable via DP.

(d) When we get to a subproblem, we want to have computed the $dist$ value for all its direct ancestors. Thus it makes the most sense to solve subproblems in topological order, starting from S, since all direct ancestors of a vertex must come before it in topological order.

(e) There are $|V|$ subproblems total. This gives us $|V|$ work to look at each subproblem once. Then, for each subproblem, we need to iterate through all the ancestors of the given vertex (this is precisely equal to the vertex's in-degree). In any directed graph, the sum of the indegrees of each node is precisely equal to the sum of outdegrees, which is precisely equal to $|E|$ (since each edge contributes to the indegree of exactly one node, and the outdegree of exactly one node).

(f) SCDE

2 String Shuffling

Let x, y, and z be strings. We want to know if z can be obtained only from x and y by interleaving the characters from x and y such that the characters in x appear in order and the characters in y appear in order. For example, if $x =$ **efficient** and $y =$ **ALGORITHM**, then it is true for $z =$ **effALGiORciIenTHMt**, but false for $z =$ **efficientALGORITHMS** (extra characters), $z =$ **effALGORITHMicien** (missing the final t), and $z =$ **effOALGRicieITHMnt** (out of order). How can we answer this query efficiently? Your answer must be able to efficiently deal with strings with lots of overlap, such as $x =$ **aaaaaaaaaab** and $y =$ **aaaaaaaac**.

1. Design an efficient algorithm to solve the above problem and state its runtime.

2. Consider an iterative implementation of our DP algorithm in part (a). Naively if we want to keep track of every solved sub-problem, this requires $O(|x||y|)$ space (double check to see if you understand why this is the case). How can we reduce the amount of space our algorithm uses?

Solution:

1. First, we note that we must have $|z| = |x| + |y|$, so we can assume this. Let $S(i, j)$ be true if and only if the first i characters of x and the first j characters of y can be interleaved to make the first $i + j$ characters of z. Then x and y can be interleaved to make z if and only if $S(|x|, |y|)$ is true.

   For the recurrence, if $S(i, j)$ is true then either $z_{i+j} = x_i$, $z_{i+j} = y_j$, or both. In the first case it must be that the first $i - 1$ characters of x and the first j characters of y can be interleaved to make the first $i+j-1$ characters of z; that is, $S(i-1, j)$ must be true. In the second case $S(i, j - 1)$ must be true. In the third case we can have either $S(i - 1, j)$ or $S(i, j - 1)$ or both being true. This yields the recurrence:

$$S(i, j) = (S(i - 1, j) \wedge (x_i = z_{i+j})) \vee (S(i, j - 1) \wedge (y_j = z_{i+j}))$$

   The base case is $S(0, 0) = T$; we also set $S(0, -1) = S(-1, 0) = F$ for convenience. The running time is $O(|x||y|)$.

   Somewhat naively if we'd like an iterative solution, we can keep track of the solutions to all subproblems with a 2D array where the entry at row i, column j is $S(i, j)$. If we iterate over this array row by row, going left to right, we'll always be able to fill in the next entry using values we've already computed.

   Notice, however, that to compute any entry, we only really need the infomration in the previous row, and the current row we're filling out. Thus, rather than holding onto the entire table, we only need to store the current and previous row, reducing us from $O(m * n)$ space to $O(m)$ space.

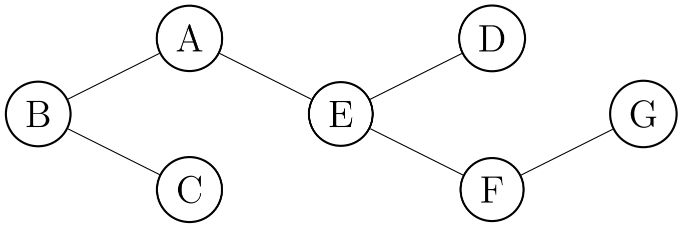
2. We can keep track of the solutions to all subproblems with a 2D array of size $|x||y|$ where the entry at row i, column j is $S(i, j)$. If we iterate over this array row by row, going left to right, we'll always be able to fill in the next entry using values we've already computed.

   Notice, however, that to compute any entry, we only really need the infomration in the previous row, and the current row we're filling out. Thus, rather than holding onto the entire table, we only need to store the current and previous row, reducing from $O(|x||y|)$ space to $O(\min(|x|, |y|))$ space.

3 Vertex cover

A *vertex cover* of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ that includes *at least* one endpoint of (covers) every edge $e \in E$. Give a dynamic programming algorithm which finds a vertex cover of a **tree** T.

For instance, in the following tree, possible vertex covers include $\{A, B, C, D, E, F, G\}$ and $\{A, C, D, F\}$ but not $\{C, E, F\}$. The smallest vertex cover has size 3: $\{B, E, G\}$.



(Bonus: can you find a greedy algorithm for this problem?)

Solution: The subproblem $V(u)$ will be defined to be the size of the minimum vertex cover for the subtree rooted at node u. We have $V(u) = 0$ if u is a leaf, as the subtree rooted at u has no edges to cover. The crucial observation is that if a vertex cover does not use a node it has to use all its neighboring nodes. Hence, for any internal node i, we can either use node i, or we do not use node i, and instead use its children:

$$V(i) = \min \left\{ 1 + \sum_{j:(i,j)\in E} V(j), \sum_{j:(i,j)\in E} \left(1 + \sum_{k:(j,k)\in E} V(k) \right) \right\}$$

Note that because T is a tree, the solutions to the subproblems are independent. The algorithm can then solve all the subproblems in order of decreasing depth in the tree and output $V(n)$. Note that any edge (i, j) is used at most twice - once for calculating $V(i)$ and once for calculating $V(k)$ for k the parent of i. Hence the running time is $O(|V| + |E|) = O(|V|)$.

Bonus: the greedy algorithm arises from the observation that if we have a vertex cover S where some leaf v of T is in S, we can turn it into a vertex cover which is no larger where $v \notin S$ by replacing v with its parent. Hence there is a minimum vertex cover S with no leaves; then S must contain the parents of all leaves. The greedy algorithm works by iteratively applying this observation.