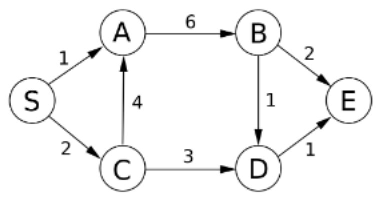
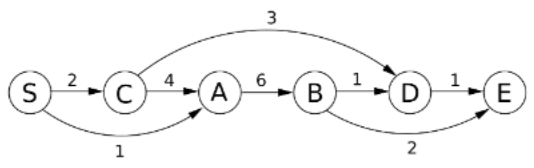
CS 170 DIS 06

Released on 2018-02-28

1 Shortest Paths with Dynamic Programming!





Write a dynamic programming algorithm to determine the minimum length path from the start (S) to the end (E).

1. What are the subproblems you need to solve to find the minimum length from S to E?

2. Can you define the optimal solution to any subproblem as a function of the solution to its own subproblems? What does the dependency graph look like? The dependency graph is a 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 about the given graph makes this particular DP solution convenient here?

3. If we were to proceed iteratively, what would be the best order to solve these subproblems in?

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

5. Run your algorithm on the graph. What is the shortest graph?

Solution:

1. 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.

2. 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 $U = min\{dist(A_1) + l(A_1, U), ..., dist(A_N) + l(A_N, U)\}$.

   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.

3. 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.

4. 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).

5. 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 $z =$ **efficientALGORITHMextraCHARS** (miscellaneous characters), $z =$ **effALGORITHMicien** (missing the final t), and $z =$ **awoefnawolnef** (obviously wrong). How can we answer this query efficiently? Your answer much be able to efficiently deal with strings 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 $O(m * n)$ 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

$$S(i,j) = \begin{cases} 1 & \text{if } x[0:i] \text{ and } y[0:j] \text{ can be interleaved to make } z[0:i+j] \\ 0 & \text{otherwise} \end{cases}$$

Notice that if x up to index i can be interleaved with y up to index j to get z up to index $i+j$, then we can say the following. $z[i+j]$ must equal either $x[i]$ or $y[j]$. This is equivalent to saying that any valid interleaving must end on either the last character of $x[0:i]$ or the last character of $y[0:j]$.

If $x[i] == z[i+j]$ (the interleaving ends on the last character of $x[0:i]$), then for us to have a valid interleaving of x and y up to indices i, and j respectively, we need S(i-1, j) to be 1.

Similarly, if $y[j] == z[i+j]$ (the interleaving ends on the last character of $y[0:j]$), then we need S(i, j-1) to be 1.

Let $\Delta_x(i,j)$ be an indicator function that is 1 if $x[i] == z[i+j]$ and 0 otherwise.

Let $\Delta_y(i,j)$ be an indicator function that is 1 if $y[j] == z[i+j]$ and 0 otherwise.

Then

$$S(i,j) = \max\{\Delta_x(i,j) * S(i-1,j), \Delta_y(i,j) * S(i,j-1)\}$$

In other words, this recursive relationship tells us that $S(i,j)$ is 1 if $x[i] == z[i+j]$ AND $S(i-1,j) == 1)$, OR if $y[i] == z[i+j]$ AND $S(i,j-1) == 1$.

Our base cases are $S(i,0) = 1$ if the first i characters of x are the first i characters of z and similarly for y.

Let m be the length of x and n be the length of y We have one subproblem for each pair consisting of an index in x and an index in y, giving us $O(m * n)$ subproblems total. Each subproblem is solvable in constant time, giving us $O(m * n)$ runtime overall.

2. 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.