Dynamic Programming: Odds & Ends

Introduction In this set of notes we will see how to

- Find max-cost paths in a DAG in $O(|V|+|E|)$ time
- Construct Longest Common Subsequences in $O(mn)$ time using only $O(m+n)$ space.

Max-Cost Paths in DAGS Let $G = (V, E)$ be a weighted directed graph. Dijkstra's algorithm constructs a shortest path tree rooted at (any vertex) s in $O(|E| \log |V|)$ time (or $O(|E|+|V| \log |V|)$ with more sophisticated data structures) as long as the algorithm does not contain any negative cycles.

Surprisingly, the corresponding max-cost problem, i.e., finding a maximum cost simple path tree rooted at s, is not known to be solvable in polynomial time. As we will now see, though, this problem is solvable using a dynamic programming approach in $O(E| + |V|)$ time if G is a Directed Acyclic Graph (DAG).

First preprocess the graph by performing an $O(|V| + |E|)$ topological sort. When running the sort let s, the tree root, be the first item outputted by the sort (this is possible since, by asking for it to be the root, we are requiring that it have no incoming edges). In $O(|V|)$ time relable the vertices as $v_1, v_2, \ldots, v_n$ where $v_1 = s$ and if $i < j$ then v_i preceeds v_j in the topological order.

Note that we usually implicitly assume that directed digraphs are represented via *out*-adjacency lists, i.e., for each vertex, a list of the edges leaving that vertex. For our solution we will need *in*-adjacency lists that, for each vertex, lists the edges *leaving* that vertex. These can be built from the out-list in $O(|E| + |V|)$ time (how?).

Now consider the max-cost path from v_1 to any vertex v. Suppose this path has length $t-1$. Label the path vertices as

$$v_1 = u_1,\, u_2,\, u_3, \ldots, u_{t-1},\, u_t = v.$$

Because G is a DAG, no path from v_1 to u_{t-1} can contain v. Furthermore, $u_1,\, u_2,\, u_3, \ldots, u_{t-1}$ must be a max cost path from s to u_{t-1} (since otherwise we could replace that subpath with a more expensive subpath to u_{t-1}, building an even more expensive path to $u_t = v$).

Setting d_i to be the cost of the max path from s to v_i we find that $d_1 = 0$ and for $i > i$

$$d_i = \max_{j\,:\,(v_j,v_i)\in E} (d_j + w_{j,i}) \tag{1}$$

where $w_{i,j}$ is the weight of the edge from v_j to v_i. By the properties of the topological ordering we can rewrite this as

$$d_i = \max_{j<i\,:\,(v_j,v_i)\in E} (d_j + w_{j,i})\,. \tag{2}$$

This immediately gives us a dynamic programming algorithm

1. Set $d[i] = 0$ for all i
2. For $i = 2$ to n do
3. Set $d[i] = \max_{j\,:(v_j,v_i)\in E} (d[j] + w_{j,i})$

Note that in line 3, when calculating $d[j] + w_{j,i}$, we have $j < i$. S,o by induction, the value in $d[j]$ is already the correct max-cost path value. Thus, $d[i]$ is set to be the correct max-cost path value.

The running time of this algorithm is $O(|V| + |E|)$ since line 3 can be implemented in time equal to the indegree of node v_j.

Note that to actually find the path we will need to keep another array, $pred[i]$, that stores the value of j that maximizes the expression in (1).

Return to Longest Common Subsequence Recall that the LCS problem is to find the LCS of two strings X and Y, respectively of length n and m. The DP algorithm calculates $d_{i,j}$, the length of the longest common subsequence of the first i characters in X and the first j characters in Y with $d_{n,m}$ being the actual length sought.

The DP equation was $d_{i,0} = d_{0,j} = 0$ for all i, j and, when i or j not 0,

$$d_{i,j} = \begin{cases} d_{i-1,j-1} + 1 & \text{if } x[i] = y[j] \\ \max(d_{i-1,j}, d_{i,j-1}) & \text{if } x[i] \neq y[j] \end{cases}$$

We can replace this with the following recurrence (why?)

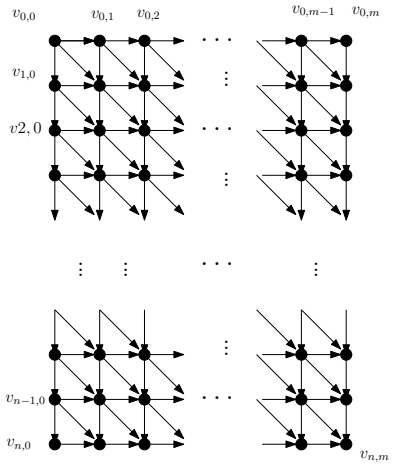
$$d_{i,j} = \max(d_{i-1,j},\, d_{i,j-1},\, d_{i,j} + \delta_{i,j})$$

where

$$\delta_{i,j} = \begin{cases} 1 & \text{if } x[i] = y[j] \\ 0 & \text{if } x[i] \neq y[j]. \end{cases}$$

Now consider the directed grid graph with $(m+1)(n+1)$ vertices $v_{i,j}$, $0 \leq i \leq n$ and $0 \leq j \leq m$.

- For $i \neq 0$, there is an edge of cost zero from $v_{i-1,j} \rightarrow v_{i,j}$.
- For $j \neq 0$, there is an edge of cost zero from $v_{i,j-1} \rightarrow v_{i,j}$.
- For $i \neq 0, j \neq 0$ there is an edge of cost $\delta_{i,j}$ from $v_{i-1,j-1} \rightarrow v_{i,j}$.



$d_{i,j}$ is now exactly the cost of a max-cost path from $v_{0,0}$ to $v_{i,j}$. The LCS problem is now the problem of finding a max cost path from $v_{0,0}$ to $v_{n,m}$. The cost of that path ($d_{n,m}$) is the LCS length and the actual LCS can be constructed by walking through the path edges consecutively, finding, in order, the edges $d_{i-1,j-1} \rightarrow v_{i,j}$ with weight 1 and returning their corresponding $x[i]$.

Note that the graph that we built is a DAG with "source" $v_{0,0}$ and the order in which we processed the $d_{i,j}$ in the LCS algorithms is actually a topological order on the vertices. The LCS algorithm is thus just implementing the DAG max-cost path algorithm we developed above, restricted to a particular type of graph with a known topological order.

Saving Space in the LCS algorithm In class we noted that calculating $d_{n,m}$ required $O(mn)$ time but only $O(n+m)$ space. Finding the actual LCS, though, required keeping an $O(mn)$ space predecessor table, where $pred[i,j]$ was the predecessor (arrow) pointing to the item that achieved the max value of $d_{i,j}$. To find the LCS we walked through the predecssor arrow backwords from $d_{n,m}$, reporting the diagonal arrows.

We will now see how to find the LCS using only $O(m+n)$ space.

We actually solve a slightly more general problem. Let $G = (V, E)$ be the $(m+1) \times (n+1)$ grid graph described above

- For $i \neq 0$, there is an edge from $v_{i-1,j} \rightarrow v_{i,j}$.
- For $j \neq 0$, there is an edge from $v_{i,j-1} \rightarrow v_{i,j}$.
- For $i \neq 0, j \neq 0$ there is an edge from $v_{i-1,j-1} \rightarrow v_{i,j}$.

We are only told that the graph exists but we don't actually build it. We are also given an $O(1)$ way of calculating the cost of any edge on the fly. In the LCS case, we can do this by setting all edge costs equal to zero except for $v_{i-1,j-1} \rightarrow v_{i,j}$ which has cost $\delta_{i,j}$.

Let $i \leq i'$ and $j \leq j'$, i.e., $v_{i,j}$ is to the upper left of $v_{i',j'}$. We are going to design an algorithm $BP(i,j,i',j')$ which returns a max-cost path from $v_{i,j}$ to $v_{i',j'}$ in $O((i'-i+1) \times (j'-j+1))$ time and $O((i'-i+1)+(j'-j+1))$ space.

Note that based on our discussion above this immediately builds the LCS in $O(mn)$ time using $O(m+n)$ space.

We need the following facts

- If $i \leq i'$ and $j \leq j'$, then, because all edges go down one step and/or right one step, *any* path between $v_{i,j}$ and $v_{i',j'}$ falls fully within the box

$$Box(i,j,i',j') = \{(u,t) \,|\, i \leq u \leq i',\ j \leq v \leq j'\}$$

- If $v_{u,t}$ is an item on some max cost path between $v_{i,j}$ and $v_{i',j'}$ then we can find a max cost path between them by concatenating (i) any max cost path between $v_{i,j}$ and $v_{u,t}$ and (ii) any max cost path between $v_{u,t}$ and $v_{i',j'}$, i.e., returning

$$BP(i,j,u,t)\ BP(u,t,i',j')$$

- If $j' = j$, $Box(i,j,i',j')$ is a vertical path walking from $v_{i,j}$ to $v_{i',j}$. This can be found and returned by $BP(i,j,i,j')$ in $O(i'-i+1)$ time
- If $i' = i$, $Box(i,j,i',j')$ is a horizontal path walking from $v_{i,j}$ to $v_{i',j'}$. This can be found and returned by $BP(i,j,i,j')$ in $O(j'-j+1)$ time

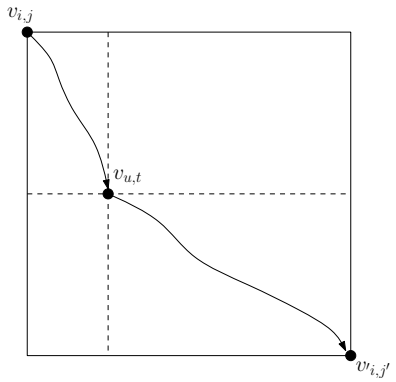


Figure 1: If $v_{u,t}$ is on a max-cost $v_{i,j}$-$v_{i',j'}$ path then the concatenation of $BP(i,j,u,t)$ and $BP(u,t,i',j')$ is a max-cost $v_{i,j}$-$v_{i',j'}$ path. In our algorithm we will ensure that $u = \lfloor (i+i')/2 \rfloor$.

- if $i' = i+1$, the associated graph is a two layer DAG with $2(j'-j+i)$ vertices and $\leq 4(j'-j+1)$ edges. It is easy to see that $BP(i,j,i{+}1,j')$ can return the associated max cost path in time $O(j'-j+1)$, e.g., by using the DAG max-cost path algorithm in the first part of these notes (or a special purpose simpler algorithm)

Note that any path between $v_{i,j}$ and $v_{i',j'}$ must contain at least one vertex (u,t) with $u = \lfloor \frac{i+i'}{2} \rfloor$. Suppose we have a procedure $Mid(i,j,i{+}1,j')$ that is guaranteed to return a vertex on some max cost path between $v_{i,j}$ and $v_{i',j'}$. Then the second bullet point above implies that we can recurse by finding a max cost path from $v_{i,j}$ to $v_{u,t}$ and a max cost path from $v_{u,v}$ to $v_{i',j'}$ and concatenate them. If $i'-i+1 \geq 3$ (and $j \neq j$) then the depths of the new graphs constructed will be less than the originals. We stop the recursion (using the third, fourth and fifth bullet points) if the new graphs are only one or two levels deep or one level wide.

Formally, the algorithm is

1. $BP(i,j,i,j')$
2. if $j' = j$ then
3. return $BP(i,j,i',j)$ by writing the verticle path in $O(i'-i+1)$ time
4. if $i' = i$ then
5. return $BP(i,j,i,j')$ by writing the horizontal path in $O(j'-j+1)$ time
6. else if $i' = i+1$
7. return $BP(i,j,i,j')$ by running the DAG algorithm in $O(j'-j+1)$ time
8. else
9. Run $Mid(i,j,i+1,j')$ to find (u,v) with $u = \lfloor \frac{i+i'}{2} \rfloor$
10. return $BP(i,j,u,v)$ $BP(u,v,i',j')$

Note the following

1. The correctness of the algorithm follows from the discussion above.
2. If $Mid(i,j,i',j')$ uses $O((i'-i+1)+(j'-j+1))$ space then $BP(i,j,i,j')$ also uses $O((i'-i+1)+(j'-j+1))$ space
3. If $Mid(i,j,i',j')$ uses $O((i'-i+1)\times(j'-j+1))$ time then $BP(i,j,i,j')$ also uses $O((i'-i+1)\times(j'-j+1))$ time

(1) and (2) are obvious. We will see a proof of (3) in the final section of these notes.

Given the above, all that is left to finish our algorithm is to is show how to implement $Mid(i,j,i',j')$ in $O((i'-i+1)\times(j'-j+1))$ time and $O((i'-i+1)+(j'-j+1))$ space

Implementing $Mid(i,j,i',j')$

Let G' be the induced subgraph of G containing all of the

$$\{v_{s,t} \,|\, i \leq s \leq v',\ j \leq t \leq j'\}$$

and $u = \lfloor \frac{i+i'}{2} \rfloor$. $Mid(i,j,i',j')$ has to find a point (u,v) on a max cost path in G' from $v_{i,j}$ to $v_{i',j'}$ in G'

Start by creating two new subgraphs G_1 and $\bar{G}_2$.

G_1 is the induced subgraph of G' containing all $v_{s,t}$ with $s \leq u$. Let $d^1_{s,t}$ be the cost of a max cost path from $v_{i,j}$ to $v_{s,t}$ in G_1. Using the same approach used for the LCS algorithm we can walk down row by row calculating $d^1_{s,*}$ (i.e., the entire s row's values) from $d^1_{s-1,*}$. This uses

$$O\left(\frac{1}{2}(i'-i+1)\times(j'-j+1)\right) \text{ time}$$

and

$$O\left(\frac{1}{2}(i'-i+1)+(j'-j+1)\right) \text{ space}$$

and, when finished, we have stored (only) all of the values $d^1_{u,*}$

G_2 is the induced subgraph of G' containing all $v_{s,t}$ with $s \geq u$. For $v_{s,t}$ in G_2 let $d^2_{s,t}$ be the cost in G_2 of the max cost path from $v_{s,t}$ to $v_{i',j'}$

Now define $\bar{G}_2$ be G_2 with all edges reversed in direction (keeping the same costs) and $\bar{d}^2_{s,t}$ to be the cost in $\bar{G}_2$ of a max cost path from $v_{i',j'}$ to $v_{s,t}$. Note that, because of the edge reversals, every path in G_2 corresponds to a reversed path in $\bar{G}_2$ with the same cost. In particular $\bar{d}^2_{s,t} = d^2_{s,t}$ for all s, t.

Using the same approach as for G_1 we can start from row j' and walk *up*, calculating $\bar{d}^2_{s,*}$ from $\bar{d}^2_{s+1,*}$. This uses

$$O\left(\frac{1}{2}(i'-i+1)\times(j'-j+1)\right) \text{ time}$$

and

$$O\left(\frac{1}{2}(i'-i+1)+(j'-j+1)\right) \text{ space}$$

and, when finished, we have stored (only) all of the values $\bar{d}^2_{u,*}$ which is equivalent to having the values $d^2_{u,*}$.

Now note that, for every $j \leq t \leq j'$, $d^1_{u,t} + d^2_{u,t}$ is the cost of max cost path from $v_{i,j}$ to $v_{i',j'}$ *passing through* $v_{u,t}$. Since a max cost path from $v_{i,j}$ to $v_{i',j'}$ must pass through at least one node $v_{u,t}$ the value

$$\max_t \left(d^1_{u,t} + d^2_{u,t}\right)$$

is the cost of a max-cost path from from $v_{i,j}$ to $v_{i',j'}$. If t' is the index t at which the maximum occurs then all our procedure needs to do is return the node (u, t').

Note that this last step only used $O\left(j'-j+1\right)$ time and $O\left(j'-j+1\right)$ space so the entire procedure for $Mid(i,j,i',j')$ only used

$$O\left(\frac{1}{2}(i'-i+1)\times(j'-j+1)\right) \text{ time}$$

and

$$O\left(\frac{1}{2}(i'-i+1)+(j'-j+1)\right) \text{ space.}$$

Running time of $BP(i,j,i',j')$

We now conclude by explaining how to calculate the running time of $BP(..)$ given the running time of $Mid(..)$.

When we write below "the work **** done recursively by procedure $BP(i,j,i',j')$" we mean the work done by the procedure at its top level *and* the work done recursively by all the procedures it and it's recusively called subroutines call.

Now note that the output of $BP(i,j,i',j')$ is a path of length $\Theta(i'-i+1)+(j'-j+1))$. Lines 3, 5, and 7 of the algorithm end the recursion by printing out some edges of the solution path. Note that every edge of the solution path is only printed out once. Therefore, the total amount of work at lines 3,5 and 7 done recursively by procedure $BP(i,j,i',j')$ is $\Theta(i'-i+1)+(j'-j+1))$. So, when calculating the total work done recursively by procedure $BP(i,j,i',j')$, it only remains to calculate the work done in the $Mid(...)$ calls recursively. For the purposes of the analysis we can rewrite the procedure as

1. $BP(i,j,i',j')$
2. if $(i' > i+1)$ and $j' \neq j$ then
3. Run $Mid(i,j,i+1,j')$ to find (u,v) with $u = \lfloor \frac{i+i'}{2} \rfloor$
4. Call $BP(i,j,u,v)$ $BP(u,v,i',j')$

Set $w = j'-j$, $k = i'-i$ be the width and height of the grid graph defined by $BP(i,j,i,j')$ Define $T(w,h)$ to be the running time of $BP(...)$ when the graph it defines has width w and height h, i.e., when we call $BP(i,j,i+h,j+w)$. Our goal is to show that $T(w,h) = O(wh)$ since that implies $BP(i,j,i,j')$ in $O((i'-i+1)\times(j'-j+1))$ time. The reason we write it this way is that wh is the *area* of the grid graph defined by $BP(i,j,i,j')$ and it will help our intuition to think of the running time of the algorithm as being linear in the area of the graph.

Note that we have already seen that $Mid(i,j,i+h,j+w)$ requires $O(hw)$ time. Let $c > 0$ be a constant such that the running time of $Mid(i,j,i+h,j+w)$ is $\leq chw$. Our work recurrence relation is then

$$T(w,h) \leq T(x,\frac{h}{2}) + T(w-x,\frac{h}{2}) + chw$$

where $x = v - i$ and we have initial conditions $T(0,h) = T(w,0) = T(w,1) = 0$.

Let $c' = 2c$. We prove by induction that $T(w,h) \leq c'wh$. This is obviously true if $h = 1$. Suppose we have proven the statement by induction for $h = 1, 2, 3, \ldots, k-1$. Note that $\frac{k}{2} \leq k-1$. So we know by induction that, for any value of x

$$\begin{aligned} T(w,h) &\leq T(x, \frac{h}{2}) + T(w-x, \frac{h}{2}) + cwh \\ &\leq c'x\frac{h}{2} + c'(w-x)\frac{h}{2} + cwh \\ &\leq c'\frac{1}{2}wh + cwh \\ &= (\frac{c'}{2} + c)wh = c'wh \end{aligned}$$

and we are done.