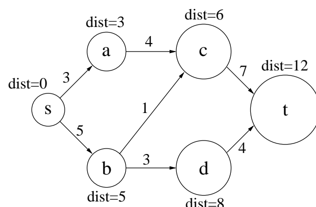
Figure 1.1 Calculating distances from s in a dag.



Dynamic Programming

So far we have seen several examples of algorithmic techniques: Divide-and-conquer, greedy decision-making, graph exploration — to name a few. It is wonderful when these specialized tools succeed in solving a problem — and we have seen many occasions of this.

In this chapter and the next we shall see two very general algorithmic techniques: *dynamic programming and linear programming.* These are the sledgehammers of the algorithms craft. They succceed in solving problems in which other techniques fail; but the resulting algorithms are typically slower — occasionally not even polynomial.

1 Another look at shortest paths in dags

In the conclusion of our study of shortest paths in graphs we noticed that the problem becomes especially easy when the graph is a dag. What makes the shortest path algorithm run so fast on dags is that each edge is updated only once. Hence, the `dist` of a node v can be found by just comparing the $\texttt{dist}(u) + l(u, v)$ of all its predecessors. The algorithm can be recast as follows:

for each $v \in V$, in topological order:
if $v = s$: $\texttt{dist}(s) = 0$
else: $\texttt{dist}(v) = \min_{(u,v)\in E}\{\texttt{dist}(u) + l(u, v)\}$

All we do is march from node to node, further and further away from s, and calculate the distance from s of each node by examining the distances of its predecessors. (If a node v is not reachable from v, its `dist` is set, quite naturally, to ∞, the minimum over the empty set.)

There are several ways of looking at this simple algorithm. Here is perhaps the most suggestive one: We are solving a problem (find the shortest path from s to t) by solving *larger and larger subproblems* (notice the size of the nodes in the figure). Indeed, finding the shortest path from s to a is a subproblem whose solution is needed in order to find the shortest path from s to t. And these subproblems can be thought of as becoming larger (harder to solve, requiring the solution of more subproblems) as we venture further down in the topological order. For s, the subproblem is so simple that we immediately know its answer: zero.

Dynamic programming is a very powerful algorithmic style in which, in order to solve a problem, we identify subproblems and solve them one-by-one, smaller first, using the solutions for the small problems to solve the larger ones until the whole problem is solved. The algorithm for shortest paths in a dag is in some sense the archetypical dynamic programming algorithm.

Figure 1.2 The same problem seen as filling a table.

node v	`dist`(v)
s	0
a	$\min\{\texttt{dist}(s)+3\}=3$
b	$\min\{\texttt{dist}(s)+5\}=5$
c	$\min\{\texttt{dist}(a)+4, \texttt{dist}(b)+1,\}=6$
d	$\min\{\texttt{dist}(b)+3\}=8$
t	$\min\{\texttt{dist}(c)+7, \texttt{dist}(d)+4,\}=12$

But there is another way of looking at this algorithm. Fundamentally it entails filling a table, calculating a function, `dist`, of the nodes. The value of this function at a node depends on the values at the node's predecessors in the dag, and certain local data (namely, the lengths of the edges).

It so happens that the function we are calculating involves taking the minimum of sums, but this is almost irrelevant: We might as well calculate the maximum. Or we could have $\cdot$ instead of $+$ inside the brackets (and initialize $\texttt{dist}(s)$ to 1), thus calculating the path with the largest *product* of lengths. Or we could be computing at each entry the quantity $\texttt{dist}(v) = \sum_{(u,v)\in E}\{\text{dist}(u)\}$ (again initializing `dist|`(s) `to 1). It is not hard to see that this variant of the algorithm would compute` *`the number of paths`* `from` s `to` t (see Problem ???) The point is, the same basic algorithm would accomplish all these tasks.

boxed: The principle of optimality

boxed: Pascal's triangle: the oldest dynamic program

boxed: The risks of recursion

boxed: Programming?

The origin of the term "dynamic programming" has very little to do with writing code. In the 1950s, when it was first tossed by Richard Bellman, computer programming was an esoteric activity practiced by so few people that it did not yet merit a name. Back then "programming" was synomymous with "planning." "Dynamic programming" originally applied to the task of finding the best way to plan a multi-stage process: Think of the dag of Figure 1.1 as such a decision process. Time advances from left to right (four time units in this example), while the vertices at each vertical level stand for different states. The edges leaving a node would represent different possible actions, leading to different states in the next time period.

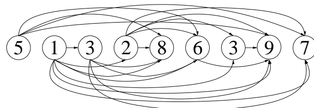
The etymology of *linear programming*, the subject of the next chapter, is similar.

boxed: The mother of all algorithms?

1.1 Constructing the dag: longest increasing subsequences

Often solving a problem by dynamic programming involves finding shortest paths (of some sort) in a dag — *except that we are not given the dag;* the dag is constructed in a way that depends crucially on the problem we wish solved. Constructing the right dag is then the main

Figure 1.3 The dag of increasing subsequences.



difficulty in solving such a problem.

For example, consider this problem: We are given an array of n integers $a_1, \ldots, a_n$, and we are asked to find the *longest increasing subsequence*, that is, the maximum m for which there are indices $1 \leq i_1 < \cdots < i_m \leq n$ such that $a_{i_1} < a_{i_2} < \cdots < a_{i_m}$.

For example, if the array is $5, 1, 3, 2, 8, 6, 3, 9, 7$, then the longest increasing subsequence is $1, 3, 6, 9$. How do we find it efficiently?

It is natural to construct the following dag (V, E): It has nodes $V = \{1, 2, \ldots, n\}$, and there is an edge from i to j in E if and only if (a) $i < j$ and (b) $a_i < a_j$. That is, we draw an edge from each number to all subsequent larger numbers. It *is* a dag, as evidenced by the topological ordering $1, 2, \ldots, n$ (see Figure 1.3).

Furthermore, any path $i_1, \ldots, i_n$ in this dag is an increasing subsequence, since the presence of the edges $(i_1, i_2), (i_2, i_3), \ldots, (i_{m-1}, i_m)$ implies that $i_1 < i_2 \cdots < i_m$ and $a_{i_1} < a_{i_2} < \cdots < a_{i_m}$. So, finding the longest increasing subsequence is tantamount to finding *the longest path in this dag!* Our shortest-paths algorithm can accomplish this, of course, by setting all edge lengths to -1. But here is a more direct version of the same algorithm:

for $j = 1, 2, \ldots, n$:
set $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$
return the largest value of L

That is, the longest increasing subsequence ending at j is one plus the maximum $L(i)$ over all predecessors i of j in the dag. (If there are no edges into j, we take the maximum over the empty set to be zero.) Needless to say, in the end we return the *largest* of all $L(i)$'s observed. Notice that the second line of the algorithm requires a loop over all predecessors of j to be calculated; the adjacency lists of the reverse graph (recall ???) would be handy here. The algorithm is linear in E, which however is about n^2 in the worst case — the worst case being when the given array is sorted in increasing order. Hence what we have described is an $O(n^2)$ algorithm for the longest increasing subsequence problem.

For a faster $O(n \log n)$ algorithm for this problem, see???.

1.2 Constructing the dag: edit distance

Given two strings, how close are they to each other? For example, of the three last names of the authors of this book, which two are the closest?

There is a very natural measure of distance between strings x and y: The smallest number of *edits* — insertions, delections, and overwritings of characters — that would transform x into y (or y into x, since these edit operations are invertible, this distance measure is symmetric). For example, the edit distance between the string `to` and the string `fro` is 2: To turn `to` into `fro`, we overwrite `t` with `f`, and we insert `r` right after that, a total of two edits. And there is no way to achieve the same effect with one.

boxed: The story of BLAST

But how does one compute this distance? The answer is, by dynamic programming. Because, rather surprisingly, the edit distance of two strings is the shortest path in a particular dag that can be constructed from the two strings.

In designing a dynamic programming algorithm for a problem, the most crucial question is, *what is a subproblem?* In other words, what are the vertices of the underlying dag? Once the appropriate notion of a subproblem has been defined (one that allows us to use solutions of small problems to solve larger ones), then it is usually an easy matter to write the algorithm: Iterate solving one subproblem after the other, in order of increasing size.

Suppose that our problem is to find the edit distance of two strings, say `PAPADIMITRIOU` and `VAZIRANI`. What is a subproblem? One idea comes to mind: A subproblem should be something that goes part of the way in converting one string to the other; say, the smallest number of edits for converting some *prefix* of the first, say `PAPAD`, to some *prefix* of the second, say `VAZ`.

The next question is, how does one go from one pair of prefixes to another? Let us think.

If we know the edit distance between `PAPAD` and `VAZ`, call it $E(5, 3)$, we can easily calculate the edit distance between certain other prifixes: The distance between `PAPADI` and `VAZ` is $E(5, 3) + 1$: convert `PAPAD` to `VAZ`, and delete the `I`. The distance between `PAPAD` and `VAZI` is also $E(5, 3) + 1$: convert `PAPAD` to `VAZ`, and then insert an `I`. Finally, the distance between `PAPADI` and `VAZI` is $E(5, 3)$, simply because the same letter (`I`) comes next in both words — otherwise it would be $E(5, 3) + 1$.

In general, to compute the edit distance between two words, word x with m letters and word y with n, we define $E(i, j)$ to be the edit distance between $x[i]$ and $y[j]$, where by $x[i]$ we denote the first i letters of x and similarly for $y[j]$. Obviously, our final objective is to compute $E(m, n)$.

But what is $E(i, j)$? To answer, suppose first that the ith letter of x is different from the jth letter of y. But then, the last step in the conversion of $x[i]$ to $y[j]$ must be one of these three:

- a deletion, in which case $E(i, j) = E(i - 1, j) + 1$; or
- an insertion, implying $E(i, j) = E(i, j - 1) + 1$; or
- an overwrite, implying $E(i, j) = E(i - 1, j - 1) + 1$.

On the other hand, if the two last letters of the two prefixes are the same, then there is no increase in the edit distance, and the last option is replaced by:

- $E(i, j) = E(i - 1, j - 1)$.

Therefore, we can calculate $E(i, j)$ as the minimum of these three possibilities!

Notice that a principle of optimality is at work here: An optimal sequence of edits converting x to y, always working at the earliest letter at which the two strings still differ, must consist of optimal sequences of edits converting various prefixes of x to prefixes of y.

Finally, in dynamic programming we need a place to start, usually a place where the indices have zero values. This is easy in the present problem: What is $E(0, j)$? It is the fastest way to convert the 0-length prefix of x — the empty string — to the first j letters of y. And we know the fastest way to do this: j insertions. Similarly, $E(i, 0) = i$ — i deletions.

Figure 1.4 The distance between `PAPADIMITRIOU` and `VAZIRANI` is 10.

		V	A	Z	I	R	A	N	I
	0	1	2	3	4	5	6	7	8
P	1	1	2	3	4	5	6	7	8
A	2	2	1	2	3	4	5	6	7
P	3	3	2	2	3	4	5	6	7
A	4	4	3	3	3	4	4	5	6
D	5	5	4	4	4	4	5	6	7
I	6	6	5	5	4	5	5	6	6
M	7	7	6	6	5	6	6	6	7
I	8	8	7	7	6	6	7	7	6
T	9	9	8	8	7	7	7	8	7
R	10	10	9	9	8	8	8	8	8
I	11	11	10	10	9	9	9	9	8
O	12	12	11	11	10	10	10	10	9
U	13	13	12	12	11	11	11	11	10

And this leads directly to a dynamic programming algorithm for computing the edit distance. Let $\texttt{diff}(i, j)$ be 1 if the ith letter of x and the jth letter of y are different, and 0 otherwise:

```
for all i and j:  set E(i,0) = i and E(0,j)j
for i = 1,2,...,m:
for j = 1,2,...,n:
set E(i,j) equal to the minimum of these three:
E(i,j) = E(i-1,j) + 1
E(i,j) = E(i-1,j) + 1
E(i,j) = E(i-1,j) + diff(i,j)
return E(m,n)
```

This dynamic programming algorithm entails filling a two-dimensional table, going through the entries in any order, as long as the subproblems become larger and larger —that is to say, indices never decrease: Row after row as in the algorithm above, column after column (the two loops would nest the other way around), or in diagonals, incrementing $i + j$.

But the same algorithm can also be seen as shortest paths in a dag. Namely, the dag that has as nodes $V = \{(i, j) : 0 \leq i \leq j, 0 \leq j \leq n\}$ and as edges $E = \{((i-1, j), (i, j)), ((i, j-1), (i, j)), ((i-1, j-1), (i, j)) : 1 \leq i \leq j, 1 \leq j \leq n\}$, all of length 1 — except for the edges $((i-1, j-1), (i, j))$ when the ith letter of x is the same as the jth letter of y, in which case the length of the edge is zero (see Figure ???). We are seeking the distance between nodes $s = (0, 0)$ and $t = (m, n)$.

1.3 Subset sum

The subset sum problem is this: We are given integers $a_1, \ldots, a_n$, and another integer K; we wish to determine whether we can make up K as the sum of integers from among the given ones. Think of the a_i's as denominations of coins, and K as the amount that we want to form. For example, the list of integers could be $3, 5, 10$ and K might be 14.

There are two versions of the subset sum problem: We could allow repetitions (we have an infinite supply of each coin, in which case the answer to the simple example above is "yes," *viz.* $14 = 3 + 3 + 3 + 5$), or insist that each integer be used only once (the answer in this case would be "no").

As we shall see in Chapter ???, the subset sum problem (in both its versions) is unlikely to have a polynomial time algorithm. However, there is an algorithm that solves it in time $O(nK)$. (As we know very well from Chapter 1, such algorithm is *not* polynomial in the input size, since the expression for its running time involves the input integer K, and not its logarithm.)

Both versions of the problem can be cast as finding paths in appropriate dags. Let us consider first the version with repetitions first. The dag corresponding to the input $a_1, \ldots, a_n, K$ has a set of nodes $V = \{0, 1, \ldots, K\}$, and the following edges: $E = \{(j, j + a_i) : j, j + a_i \in V, 1 \leq i \leq n\}$. See Figure **??** for the gad corresponding to the instance above.

It is easy to see that vertices reachable from vertex 0 in this dag are precisely the integers that can be expressed as the sum of the given denominations: The path from 0 spells the denominations that are used.

As usual, the algorithm can also be seen as filling a table, namely, computing a Boolean $\mathtt{sum}(j)$ which is 1 if j can be expressed as the sum of a_i's:

```
sum(0) = 1
for j = 1,...,K :
sum(j) = max{sum(j − a_i) : 1 ≤ i ≤ n and a_i ≤ j}
return sum(K).
```

The convention is that max over the empty set is, quite naturally, 0. We return $\mathtt{sum}(K)$. Since the dag has $O(nK)$ edges (each node has at most n incoming edges, one per coin), this is also the worst-case running time of the algorithm.

What if repetitions are not allowed? The dag is now a little more complicated: The vertices are $V = \{(i, j) : 0 \leq i \leq n, 0 \leq j \leq K\}$, and the edges are $E = \{((i-1, j), (i, j + a_i)) : i = 1, \ldots, n, j = a_i, a_i + 1, \ldots, K\} \cup = \{((i-1, j), (i, j)) : i = 1, \ldots, n, j = 0, 1, \ldots, K\}$That is, for each coin a_i we have a "layer" of nodes (i, j), one for each possible sum j, and we go from layer $i-1$ to layer i either by adding a_i to j or by keeping the same j. We are seeking a path from vertex $(0, 0)$ to (n, K). (See Figure **??**; notice that the dag on the left can be seen as the dag on the right with all nodes in each column "collapsed" to a single vertex.)

It is easy to see that a node (i, j) is reachable from $(0, 0)$ in this graph if and only if j can be written as a sum of a subset of the integers $a_1, \ldots, a_i$. This is by induction on i: It is true when $i = 0$, and if it holds up to $i - 1$ it also holds for i, since the only way to reach (i, j) is either via $(i - 1, j)$ — j can be expressed as a sum of $a_1, \ldots, a_{i-1}$ — or via $(i - 1, j - a_i)$ — a_i is needed.

The table version of the algorithm is this:

```
initialize:  set sum(0,0) = 1, all other sum(i,j) = 0
```

```
for j = 1, ..., n :
for j = 0, ..., K :
if j < a_i : sum(i, j) = sum(i − 1, j),
else:  sum(i, j) = max{sum(i − 1, j), sum(i − 1, j − a_i)}
return sum(n, K).
```

1.4 One-dimensional dynamic programming

1.5 All-pairs shortest paths

1.6 The traveling salesman problem

1.7 Dynamic programming in trees