Just like divide and conquer algorithms, dynamic programming also attempts to break a problem into several smaller subproblems, solve these subproblems recursively, and then combine their solutions to get a final solution. However, sometimes there is no single clearcut way of breaking up a problem into smaller problems. One might need to try several possibilities. The problem with doing this is that now we have many more recursive calls to handle, and the running time of the algorithm can easily become exponential. Dynamic programs overcome this issue by storing the answer to any subproblem they solve, and reusing it if that subproblem needs to be solved again in the course of the algorithm. The crucial observation is that there are only a polynomial number of different subproblems that ever need to be solved. We illustrate this approach through three different examples, two of which are variants of problems that we discussed in the first lecture – weighted interval scheduling and shortest paths.

3.1 Weighted Interval Scheduling Problem

In the weighted interval scheduling problem, we want to find the maximum-weight subset of non-overlapping jobs, given a set J of jobs that have weights associated with them. Job $i \in J$ has a start time s_i, a finish time f_i, and a weight w_i. We seek to find an optimal schedule—a subset O of non-overlapping jobs in J with the maximum possible sum of weights. Formally,

$$O = \text{argmax}_{O \subseteq J; \forall i,j \in O, \text{either} f_i \leq s_j \text{or} f_j \leq s_i} \sum_{i \in O} w_i$$

When the weights are all 1, this problem is identical to the interval scheduling problem we discussed in lecture 1, and for that, we know that a greedy algorithm that chooses jobs in order of earliest finish time first gives an optimal schedule. A natural question is whether the greedy algorithm works in the weighted case too. Unfortunately it does not and can in fact give a really bad (suboptimal) solution. Here is a simple example. Suppose there are two jobs: the first has start time 1, finish time 3 and weight 1; the second has start time 2, finish time 4 and weight 100. The greedy algorithm schedules the first job, whereas the optimal one schedules the second. Other greedy approaches run into similar issues.

Can we somehow break up this problem into smaller subproblems? Suppose we knew that a particular job i was in the optimal schedule, then we could consider the following two subsets of jobs: $J_1 = \{j : f_j \leq s_i\}$ and $J_2 = \{j : s_j \geq f_i\}$ and solve the problem separately over these two subsets. These subsets are completely non-overlapping, so if we can solve the problem optimally over each of them separately, it is trivial to put the solution together and obtain an overall optimal solution. The problem is that we don't know any job i in the optimal schedule.

Here is where dynamic programming comes in. We try out all the different possibilities for i and then pick the best of them. If we were to do this naïvely, we would be making $2n$ recursive calls at the first level itself, and the running time of the algorithm would explode. The savings is in noting

that the number of different subproblems we ever need to solve are only quadratic in number: if we list out all of the start times and all of the end times, each subproblem is characterized by one start time and one end time, and so there are only n^2 different ones that we need to consider.

We will now elaborate on a simpler version of this approach. We sort the jobs in order of increasing starting time. Our algorithm considers two possibilities at each step: either the first job is scheduled or not. If it is scheduled, we remove all jobs that overlap with it and recursively solve the problem on the remaining jobs. If it is not scheduled, we remove the first job and recursively solve the problem on the remaining jobs. We try both possibilities and pick the best of the two. Naïvely, the running time of the algorithm is given by the recurrence $T(n) = 2T(n-1) + \mathcal{O}(1)$ which solves to $T(n) = 2^{\mathcal{O}(n)}$.

In order to obtain a better running time we note that the only kinds of subproblems our algorithm looks at consist of finding the optimum schedule for jobs i through n for some i. Furthermore, each subproblem further depends on subproblems of smaller size, that is, corresponding to $j > i$. Therefore, if we remember solutions to subproblems of smaller sizes that have already been solved, we can eliminate many redundant calculations the divide and conquer algorithm performs.

For $1 \leq i \leq n$, let value(i) be the maximum total weight of non-overlapping jobs with indices at least i. From our earlier discussion, we see that

$$\text{value(i)} = \max\{w_i + \text{value}(j), \text{value}(i+1)\} \tag{3.1.1}$$

where j is equal to the index of the first job that starts after job i ends.

We now use the following algorithm to find the weight of the optimum schedule.

1. Sort jobs in ascending order of starting time.
2. For $1 \leq i \leq n$, find the smallest $j > i$ such that job j doesn't overlap with job i.
3. For $i = n$ down to 1, compute value(i) using Equation (3.1.1) and store it in memory.
4. The weight of the optimum schedule is value(1).

If we want to recover the optimum schedule, we would remember the values of value(i) as well as the option that led to it.

The reader shall convince himself that once the list of jobs is sorted in ascending order of starting time, it takes time $\mathcal{O}(\log n)$ to carry out step 2 of our algorithm per job, for a total of $\mathcal{O}(n \log n)$. Hence, the running time of the algorithm is

$$T(n) = \mathcal{O}(n \log n) + \mathcal{O}(n \log n) + n\mathcal{O}(1),$$

where the two $\mathcal{O}(n \log n)$ terms come from sorting and from finding the first job that starts after job i for all jobs, and $\mathcal{O}(n)\mathcal{O}(1)$ appears because there are $\mathcal{O}(n)$ subproblems, each of which takes constant time to solve. Simplifying the expression for $T(n)$ yields

$$T(n) = \mathcal{O}(n \log n).$$

Note that we can implement this same algorithm in a slightly different top-down fashion. We start with the original problem and then use recursion along the lines of Equation (3.1.1) to find value(1). However, once again we maintain an array of all values. The first time we solve a subproblem, we store its value in the array (also called "memoization"). Whenever we make a recursive call, we first check to see if the value of the subproblem has already been stored. If so, we use the stored value, else we compute the subproblem. The reader should convince himself that the running time of this recursive algorithm is also $\mathcal{O}(n \log n)$.

3.2 Sequence Alignment

The Sequence Alignment problem is motivated in part by computational biology. One of the goals of scientists in this field is to determine an evolutionary tree of species by examining how close the DNA is between two potential evolutionary relatives. Doing so involves answering questions of the form "how hard would it be for a species with DNA 'AATCAGCTT' to mutate into a species with DNA 'ATCTGCCAT'?"

Formally stated, the Sequence Alignment problem is as follows:

Given: two sequences over some alphabet and costs for addition of a new element to a sequence, deletion of an element, or exchanging one element for another.

Goal: find the minimum cost to transform one sequence into the other.

(There is a disclaimer here. We assume that only one transformation is done at each position. In other words, we assume that 'A' doesn't mutate to 'C' before mutating again to 'T'. This can be ensured either by explicit edict or by arranging the costs so that the transformation from 'A' to 'T' is cheaper than the above chain; this is a triangle inequality for the cost function.)

As an example, we will use the following sequences:

$$\begin{aligned} S &= \text{ATCAGCT} \\ T &= \text{TCTGCCA} \end{aligned}$$

Our goal is to find how to reduce this problem into a simpler version of itself.

The main insight into how to produce an algorithm comes from noting that once we decide how to transform the first letter of S (in our example, 'A'), the problem reduces to a simple smaller version of itself. So, once again we try out all possibilities for tranforming the first letter and pick the best one. There are three things that we might do:

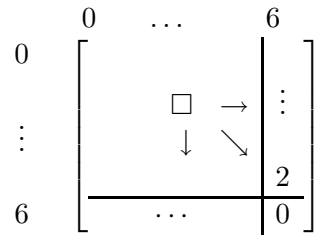
- We could remove 'A' from the sequence. In this case, we would need to transform 'TCAGCT' into 'TCTGCCA'.
- We could assume that the 'T' at the beginning of sequence T is in some sense "new", so we add a 'T'. We would need to transform 'ATCAGCT' into 'CTGCCA', then prepend the 'T'.
- We could just assume that the 'A' mutated *into* the 'T' that starts the second string, and exchange 'A' for 'T'. We would then need to transform 'TCAGCT' into 'CTGCCA'.

Once we have the above, it becomes a simple matter to produce a recurrence that describes the solution. Given two sequences S and T, we can define a function $D(i_s, j_s, i_t, j_t)$ that describes the distance between the subsequences $S[i_s \dots j_s]$ and $T[i_t \dots j_t]$.

$$D(i_s, j_s, i_t, j_t) = \min \begin{cases} D(i_s + 1, j_s, i_t, j_t) + c \text{ (delete } S[i_s]) \\ D(i_s, j_s, i_t + 1, j_t) + c \text{ (add } T[i_t]) \\ D(i_s + 1, j_s, i_t + 1, j_t) + c \text{ (exchange } S[i_s] \text{ to } T[i_t]) \end{cases}$$

There are two ways to implement the above using dynamic programming. The first approach, top-down, creates a recursive function as above but adds memoization so that if a call is made with the same arguments as a previous invocation, the function merely looks up the previous result instead of recomputing it. The second approach, bottom-up, creates a four dimensional array corresponding to the values of D. This array is then populated in a particular traversal, looking up the recursive values in the table. The array is of size m^2n^2, where $n = |S|$ and $m = |T|$, and each cell in the array takes constant time to compute. So the overall time complexity of the algorithm is $\mathcal{O}(m^2n^2)$.

However, we can do better at the bottom-up approach than $\mathcal{O}(m^2n^2)$. Note that only the starting indices (i_s and i_t) are changed. In other words, we only ever compute $D(\dots)$ with $j_s = |S|$ and $j_t = |T|$. Thus we don't need to store those values in the table, and we only need a two-dimensional array with coordinates i_s, i_t. For our example above, this is a 7x7 matrix:



Each element D_{ij} in this matrix denotes the minimum cost of transforming $S[i \dots n]$ to $T[j \dots m]$. Thus the final solution to the problem will be in D_{00}.

When the algorithm is computing the boxed element of the matrix, it need only look at the squares adjacent to it in each of the three directions marked with arrows, taking the minimum of each of the three as adjusted for the cost of the transformation. The matrix can be traversed in any fashion that ensures that the three squares the box is dependent on are filled in before the boxed square is. (Keeping in mind whether the array is stored in column- or row-major format could help cache performance should the whole array not fit.)

Each square takes constant time to compute, and there are $|S| \cdot |T|$ squares, so the overall running time of the algorithm is $\mathcal{O}(|S| \cdot |T|)$.

3.3 Shortest Paths and the Bellman-Ford Algorithm

Recall the shortest paths problem:

Given: Weighted graph $G = (V, E)$ with cost function $c : E \to \mathbb{R}$, and a distinguished vertex $s \in V$.

Goal: Find the shortest path from node s to all other vertices.

In the first lecture we discussed Dijkstra's algorithm for this problem. The algorithm can be easily implemented in $\mathcal{O}(|V|^2)$ time but there are also better $\mathcal{O}(|E| + |V| \log |V|)$ implementations using Fibonacci heaps. Unfortunately the algorithm fails to give the correct answer if the graph contains any negative cost edges. (Negative costs might not seem natural in the context of computing distances, but shortest path problems come up in a variety of situations where negative costs may be present. We will discuss one such example in the context of network flow.)

Can we compute shortest path in graphs with negative cost edges? Dynamic programming gives an answer. Recall that in a dynamic program our main goal is to try to express the optimal solution in the form of a solution to a simpler subproblem. What could such a problem be in the context of shortest paths? Let us focus on the simpler case where we want to find the shortest path between s and t alone. One approach may be to guess the vertex on the shortest path immediately preceding t. Suppose this vertex is u. Then the problem reduces to finding the shortest path between s and u. However, this doesn't immediately seem simpler than the original problem itself. Can we attribute some property to the path between s and u that shows that we have made some progress towards making the problem simpler? Here are two ways:

1. The path between s and u is shorter (in terms of the number of "hops" or edges in the path) than the one between s and t.

2. The path between s and u visits a smaller set of vertices than the one between s and t.

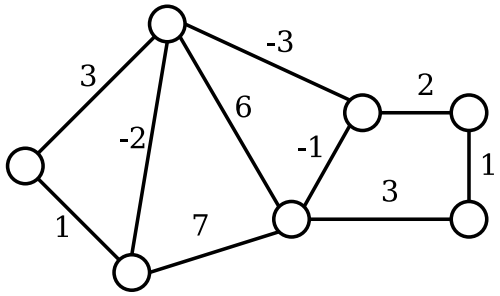
Following the first approach leads to the Bellman-Ford algorithm, while the second leads to the Floyd-Warshall algorithm. We will see that both approaches have unique strengths. We will discuss the single-source multiple-target version of the problem first and then extend it to all-pairs shortest paths. As an aside, Richard Bellman is the inventor of dynamic programming.

We first note an important assumption—in order for the two algorithms to work G must not contain cycles of negative weight. If it does, a shortest path is not well-defined as one can reduce the distance between any two nodes to negative infinity by going to the cycle, traversing it an infinite number of times and then going to the destination. One may ask, in this context, for the shortest path that visits each node at most once. The two algorithms mentioned above cannot find such paths. In fact this problem is NP-hard. Section 3.3.2 discusses the matter of negative cycles further.

We will start with the Bellman-Ford algorithm and discuss both the single-source and all-pairs variants; the Floyd-Warshall algorithm is slower in the single-source case but faster in the all-pairs case, and will be discussed later.

The key observation for the first approach comes as a result of noticing that any shortest path in G will have at most $|V| - 1$ edges. Thus if we can answer the question "what's the shortest path from s to each other vertex t that uses at most $|V| - 1$ edges," we will have solved the problem. We can answer this question if we know the neighbors of t and the shortest path to each node that uses at most $|V| - 2$ edges, and so on.

For a concrete illustration, consider the following graph:



Computing the shortest path from s to t with at most 1 edge is easy: if there is an edge from s to t, that's the shortest path; otherwise, there isn't one. (In the recurrence below, such weights are recorded as ∞.)

Computing the shortest path from s to t with at most 2 edges is not much harder. If u is a neighbor of s and t, there is a path from s to t with two edges; all we have to do is take the minimum sum. If we consider missing edges to have weight ∞ (which we are), then we needn't figure out exactly what us are neighbors of s and can just take the minimum over all nodes adjacent to t. (Or even over all of V.) We must also consider the possibility that the shortest path is still just of length one.

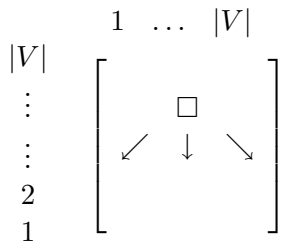
A similar process is repeated for 3 edges; we look at each neighbor u of t, add the weight of going from s to u with at most 2 hops to the weight of going from u to t, then taking the minimum such weight. Again, we must consider the possibility that the best path doesn't change when adding the 3rd edge.

Formally, we define $A(i,k)$ to be the length of the shortest path from the source node to node i that uses at most k edges, or ∞ if such a path does not exist. Then the following recurrence will compute the weights of the shortest paths, where $N(i)$ is the set of nodes adjacent to i and min returns ∞ if $N(i)$ is empty:

$$A(i,k) = \min \begin{cases} \min_{j \in N(i)} \left(A(j,k-1) + c(j,i) \right) \\ A(i,k-1) \end{cases}$$

The first option corresponds to the case where the path described by $A(i,k)$ actually involves all k edges; the second option is if allowing a kth edge doesn't change the solution. The second option can be removed if we consider an implicit, zero-weight loop at each node (that is, $c(v,v) = 0$ for all v).

The bottom-up dynamic programming approach to this problem uses an $|V| \times |V|$ matrix.



Rows represent the maximum number of edges that are allowed for a given instance; the rows are ordered so that the final answer (with a maximum of $|V| - 1$ edges) is at the top of the graph. Columns represent vertices in the graph.

When computing $A(i, k)$, represented by the boxed square, the algorithm looks at all of the boxes in the next row down for nodes that are adjacent to i, as well as i itself. If the degree of the nodes in the graph are not bounded other than by $|V|$, then each square takes $\mathcal{O}(|V|)$ time. Since there are $|V|^2$ squares, this immediately gives us an upper bound of $\mathcal{O}(|V|^3)$ for the running time. However, we can do better.

Consider a traversal of one entire row in the above matrix. During this pass, each edge will be considered twice: once from node s to node t, and once from t to s. (To see this, note that the diagonal arrows in the schematic above correspond to edges.) Thus the work done in each *row* is $\mathcal{O}(|E|)$. There are $|V|$ rows, so this gives a total bound of $\mathcal{O}(|V| \cdot |E|)$ for the algorithm.

This presentation of these shortest-path algorithms only explains how to determine the weight of the shortest path, not the path itself, but it is easy to adapt the algorithm to find the path as well. This is left as an exercise for the reader.

3.3.1 All Pairs Shortest Paths

The all pairs shortest path problem constitutes a natural extension of the single source shortest path problem. Unsurprisingly, the only change required is that rather than fixing a particular start node and looking for shortest paths to all possible end nodes, we instead look for shortest paths between all possible pairs of a start node and end node. Formally, we may define the problem as follows:

Given: A graph $G = (V, E)$. A cost function $c : E \to \mathbb{R}$.

Goal: For every possible pair of nodes $s, t \in V$, find the shortest path from s to t.

Just as the problem itself can be phrased as a simple extension of the single source shortest paths problem, we may construct a solution to it with a simple extension to the Bellman-Ford algorithm we used for that problem. Specifically, we need to add another dimension (corresponding to the start node) to our table. This will change the definition of our recurrence to

$$A[i, j, k] = \min \begin{cases} A[i, j, k-1] \\ \min_l \{A[i, l, k-1] + c(l, j)\} \end{cases} .$$

If we take $c(l, l) = 0$ for all $l \in V$, we may write this even more simply as

$$A[i, j, k] = \min_l \{A[i, l, k-1] + c(l, j)\}.$$

It is easy to see how this modification will impact the running time of our algorithm — since we have added a dimension of size $|V|$ to our table, our running time will increase from being $\mathcal{O}(|V| \cdot |E|)$ to being $\mathcal{O}(|V|^2 \cdot |E|)$.

Can we do any better? In the single-source case since we were only computing paths from s to other destinations, it was reasonable to branch on the last but one node along the shortest path. In the

all-pairs case, we can in fact branch on any intermediate point, and we will choose the mid-point of the path to branch on, to minimize the number of levels of recursion. Essentially, we are leveraging the fact that while accommodating any possible start for our path increases the complexity of our algorithm, it also gives us more information to work with at each step. Previously, at step k, only paths of length less than k starting at s and paths of length 1 were inexpensive to check; now, at step k, we have that any path of length less than k is inexpensive to check. So, where before we would have split a path of length k into a path of length $k-1$ and a path of length 1, we now split such a path into two paths of length $k/2$. Assuming that $|V|-1$ is a power of 2 (and so the values of k we encounter will all be powers of 2), our recurrence relation becomes

$$A[i,j,k] = \min_l \{A[i,l,k/2] + A[l,j,k/2]\}, \text{ where } A[i,j,1] = c(i,j).$$

The running time for this version of the algorithm may be seen to be $\mathcal{O}(|V|^3 \log |V|)$. In each iteration, in order to compute each of the $|V|^2$ elements in our array, we need to check each possible midpoint, which leads to each iteration taking $\mathcal{O}(|V|^3)$ time. Since we double the value of k at each step, we need only $\lceil \log |V| \rceil$ iterations to reach our final array. Thus, it is clear that the overall running time must be $\mathcal{O}(|V|^3 \log |V|)$, a definite improvement over our previous running time of $\mathcal{O}(|V|^2 \cdot |E|)$ when run upon a dense graph.

3.3.2 Negative Cycles

Throughout the preceding discussion on how to solve the shortest path problem — whether the single source variant or the all pairs variant — we have assumed that the graph we are given will not include any negative cost cycles. This assumption is critical to the problem itself, since the notion of a shortest path becomes ill-defined when a negative cost cycle is introduced into the graph. After all, once we have a negative cost cycle in the graph, we may better any proposed minimum cost for traveling between a given start node and end node simply by including enough trips around the negative cost cycle in the path we choose. While the shortest path problem is only well-defined in the absence of negative cost cycles, it would still be desirable for our algorithm to remain robust in the face of such issues.

In fact, the Bellman-Ford algorithm does behave predictably when given a graph containing one or more negative cycles as input. Once we have run the algorithm for long enough to determine the the best cost for all paths of length less than or equal $|V|-1$, running it for one more iteration will tell us whether or not the graph has a negative cost cycle: our table will change if and only if a negative cost cycle is present in our input graph. In fact, we only need to look at the costs for a particular source to determine whether or not we have a negative cycle, and hence may make use of this test in the single source version of the algorithm as well. If we wish to find such a cycle after determining that one exists, we need only consider any pair of a start node and an end node which saw a decrease in minimum cost when paths of length $|V|$ were allowed. If we look at the first duplicate appearance of a node, the portion of the path from its first occurrence to its second occurrence will constitute a negative cost cycle.

3.4 Floyd-Warshall algorithm for shortest paths

A second way of decomposing a shortest path problem into subproblems is to reduce along the intermediate nodes used in the path. In order to do so, we impose a labeling upon its nodes from 1 to n.

When we reduce the shortest paths problem in this fashion, we end up with the table

$$A[i, j, k] = \text{ shortest path from } i \text{ to } j \text{ using only intermediate nodes from } 1, \ldots, k,$$

which is governed by the recurrence relation

$$A[i, j, k] = \min \begin{cases} A[i, j, k-1] \\ A[i, k, k-1] + A[k, j, k-1] \end{cases}, \text{ where } A[i, j, 0] = c(i, j).$$

It is reasonably straightforward to see that this will indeed yield a correct solution to the all pairs shortest path problem. Assuming once again that our graph is free of negative cost cycles, we can see that a minimum path using the first k nodes will use node k either 0 or 1 times; these cases correspond directly to the elements we take the minimum over in our definition of $A[i, j, k]$. An inductive proof of the correctness of the Floyd-Warshall algorithm follows directly from this observation.

If we wish to know the running time of this algorithm, we may once again consider the size of the table used, and the time required to compute each entry in it. In this case, we need a table that is of size $|V|^3$, and each entry in the table can be computed in $\mathcal{O}(1)$ time; hence, we can see that the Floyd-Warshall algorithm will require $\mathcal{O}(|V|^3)$ time in total.

Although this algorithm gives a better bound on the running time in the all-pairs case, it fails to scale down nicely to the single-source version of the problem, and the Bellman-Ford algorithm provides the better bound for that version. The Floyd-Warshall algorithm can also be used to detect negative cycles – namely by determining whether $A[i, i, n]$ is negative for any node i. Moreover, it can be implement in quadratic space by reusing space from previous iterations.