CS 787: Advanced Algorithms

Dynamic Programming

Instructor: Dieter van Melkebeek

We continue our review of some topics that are usually covered in an undergraduate algorithms course. Today's topic is dynamic programming.

1 Paradigm

Dynamic programming can be viewed as a refinement of divide-and-conquer. Just like in the latter, there is an underlying recursion that reduces an instance to easier instances of the same problem, but we make sure that:

(a) the number of distinct subproblems in the recursion tree remains small, and

(b) that each subproblem is solved no more than once.

To illustrate the second point, consider the straightforward recursive algorithm to compute the nth Fibonacci number F_n, based on the recurrence

$$F_n = F_{n-1} + F_{n-2} \tag{1}$$

with the base cases $F_1 = 1$ and $F_2 = 1$. The recursion tree is a binary tree of depth n, and is complete up to level $n/2$. Thus, the number of nodes in the tree and the running time of the algorithm are at least $2^{n/2}$. However, note that there are only n distinct subproblems, namely computing F_i for $i \in [n]$. This means that many of the subproblems are solved over and over, and a lot of work can be saved by ensuring (b).

There are two ways to ensure (b):

- Memoization. We create a table that contains an entry for every problem instance that can occur as the input for one of the calls in the recursion tree. We initially mark each entry in the table as unsolved. Each time we make a recursive call, we first check the table. If the entry is marked as unsolved, we make the recursive call, and when it ends we mark the entry as solved and store the result of the recursive call there. If the entry is marked as solved, we do not make the recursive call but simply read the result from the table and return it right away.

  Memoization can be applied generically, but the table may require a lot of memory space.

- Solving instances bottom-up. The recursion implicitly induces a partial order on the problem instances. This is because the inputs to the recursive calls are "easier" than the input itself. We can build up the table of instances from easiest to harder up to the point where we reach the input instance. Each time we use the recurrence to fill in the next entry in the table, the results of the recursive calls are already present in the table.

  This approach is less generic as one needs to figure out the underlying partial order, and may result in solving subproblems that don't occur in the recursion tree for the given input. However, it sometimes allow to save on memory space because there is no need to keep the entire table in memory.

In the case of computing the Fibonacci numbers based on the recurrence (1), the subproblems are "computing F_i" for $i \in [n]$, so the memoization table is of size n. In this case "easier" means "smaller i". When we compute the solutions to the subproblems in a bottom-up fashion, we only need to keep track of the last two values computed, so we only need to keep 2 entries in memory.

In our Fibonacci example it is straightforward to keep the number of subproblems small. Typically, this step requires more ingenuity. Often the key insight consists of discerning/ensuring some structure in the subproblems that occur in the recursion tree, and the crux of a dynamic programming approach can often be captured by precisely specifying the subproblems.

2 Weighted Interval Scheduling

Given: n jobs specified by intervals and weights w_i for $i \in [n]$.

Goal: Schedule a subset S of these jobs on a single machine in a non-overlapping way such that $\sum_{i \in S} w_i$ is maximized.

Note that this problem generalizes the interval scheduling problem from the previous lecture, which corresponds to $w \equiv 1$.

For the general weighted case, no greedy approach is known. As for divide-and-conquer, it appears difficult to imagine a correct scheme that would reduce to instances of half the size by splitting the set of jobs in half. In fact, our approach will reduce to instances that can be just one job smaller.

Consider the first job under some ordering. In some optimal solution S this job will either be scheduled, or not scheduled. If it is not scheduled, then S coincides with an optimal schedule S' for the $n-1$ remaining jobs. If the first job is scheduled, then S consists of the first job and an optimal schedule S'' for the jobs that do not overlap with the first job. We recursively compute S and S'', and see which of the two yields the better schedule S.

What structure do the subproblems that occur in the resulting recursion tree have? They are all specified by a subset of the original n jobs. As there are 2^n such subsets, this observation is not good enough to achieve a polynomial running time. However, if we consider the jobs in the order of earliest start-time first, then the subsets exhibit some additional structure: they are all suffixes of this ordering, i.e., they consist of all jobs starting from some index i in the ordering. As there are only n suffixes, we have realized (a) in the description of the paradigm.

Thus, we order the jobs earliest start-time first. We define $\mathrm{OPT}(i)$ for $i \in [n]$ as the maximum value of a valid schedule for jobs i through n. We are interested in $\mathrm{OPT}(1)$, and have the recurrence

$$\mathrm{OPT}(i) = \max(\mathrm{OPT}(i+1), w_i + \mathrm{OPT}(\mathrm{next}(i))), \tag{2}$$

where $\mathrm{next}(i)$ denotes the index of the first job after i that does not overlap with i; if there is no such job, we define $\mathrm{next}(i)$ to be $n+1$, and set $\mathrm{OPT}(n+1)$ to 0. We compute the one-dimensional table $\mathrm{OPT}(i)$ from the back ($i = n+1$) to the front ($i = 1$) using the recurrence (2), and return $\mathrm{OPT}(1)$.

The correctness of the OPT table follows from the above analysis. As for the running time, the initial sorting takes $O(n \log n)$ time. We leave it as an exercise to show how to compute the

next table in $O(n \log n)$ time. The rest of the algorithm takes time $O(n)$. Hence, the total time to compute the OPT table is $O(n \log n)$.

Once we have computed the array of OPT values, we can retrieve a schedule S that realizes OPT(1) in a subsequent forward sweep over the array. We start at position $i = 1$. If the first term on the right-hand side of (2) yields the maximum, we do not take up job i in S, and move to position $i + 1$; otherwise, we take up job i in S, and move to position next(i). We do so until we reach position $n + 1$. In order to facilitate this process, we can store in location i of our table not only OPT(i) but also which of the two expressions on the right-hand side of (2) realizes the maximum. This phase runs in time $O(n)$, so the overall running time is $O(n \log n)$.

3 Sequence Alignment

> Given: two sequences $A[1..n]$ and $B[1..m]$ over an alphabet Σ.
>
> Goal: align A and B such that the sum of the number of skipped symbols and misaligned symbols is as small as possible.

As an example, let A denote the string "occurrence" and B the string "ocurrance" over the Roman alphabet. An optimal alignment would skip one of the first two "c"'s in A, and misalign the first "e" in A with the "a" in B, for a total penalty of 2.

Sequence alignment arises in word processing (where n and m are relatively small and Σ is relatively large), and in computation biology (where n and m are typically huge but Σ is small). For example, in order to determine the evolutionary tree of species biologists examine how well the DNA of two potential evolutionary relatives aligns.

The first decision we need to make is whether to (a) skip the first symbol in A, or (b) skip the first symbol in B, or (c) align the first symbols of A and B. In all three cases, what is left is to optimally align the remainders of A and B. All resulting subproblems are specified by the suffixes of A and B that need to be aligned. This leads to the following specification of the subproblems: For $i \in [n]$ and $j \in [m]$, let OPT(i, j) denote the minimum penalty for aligning $A[i..n]$ and $B[j..m]$. We are interested in OPT(1,1), and have the recurrence

$$\text{OPT}(i, j) = \min(1 + \text{OPT}(i + 1, j), 1 + \text{OPT}(i, j + 1), \delta_{A[i],B[j]} + \text{OPT}(i + 1, j + 1)), \qquad (3)$$

where we set $\text{OPT}(n + 1, j) = \text{OPT}(i, m + 1) = \text{OPT}(n + 1, m + 1) = 0$, and $\delta_{a,b}$ denotes the indicator of the condition $a = b$.

Our OPT table is now two-dimensional. In what order do we compute the entries? Note that the right-hand side of (3) only uses the neighboring cells to the bottom, right, and bottom-right of cell (i, j). Thus, we could organize the computation diagonal-wise (where the order within the diagonal does not matter, or column-wise (where the order within a column is bottom-up), or row-wise (where the order within a row if left-right). In either case the time to compute OPT(1,1) is $O(nm)$. The memory requirements differ: $O(n + m)$ for the diagonal-wise ordering, $O(n)$ for the column-wise ordering, and $O(m)$ for the row-wise ordering. This is because we only need to keep track of the current and previous diagonal/column/row.

If we use the method from the previous section to retrieve an actual alignment that realizes OPT(1,1), we need to store the entire table, in which case the memory requirement increases to $O(nm)$. For some applications in computational biology that amount may be too large. By

exploiting the fact that the OPT values can be computed using only $O(n+m)$ space, one can reduce the space complexity to $(n+m)$ while keeping the running time at $O(nm)$. We leave this as an exercise (hint: divide-and-conquer).

Note that the process of finding an optimal alignment can be thought of as finding a shortest path from position (1,1) to position $(n+1, m+1)$ in the $(n+1) \times (m+1)$ grid graph where the edges only go from left to right (length 1), top to bottom (length 1), or diagonally (length given by the δ function).

4 Iterated Matrix Multiplication

Given: n integer matrices M_i for $i \in [n]$, where M_i has dimension $d_{i-1} \times d_i$.

Goal: compute the iterated product $M_1 M_2 \dots M_n$.

Using the straightforward algorithm for multiplying two matrices, there are two ways to organize the computation for $n = 3$: As $(M_1 M_2)M_3$ or as $M_1(M_2 M_3)$. The number of required integer additions and multiplications can differ vastly. For example, for $(d_0, d_1, d_2, d_3) = (10, 100, 10, 1000)$, the first option only takes $d_0 d_1 d_2 + d_0 d_2 d_3 = 110,000$ multiplications, whereas the second one takes $d_1 d_2 d_3 + d_0 d_1 d_3 = 2,000,000$ multiplications.

In general, our aim is to break up the computation of the iterated matrix product into a sequence of pairwise matrix multiplications so as to minimize the number of integer multiplications involved.

At the top level, we write the iterated matrix product as

$$M_1 M_2 \dots M_n = (M_1 M_2 \dots M_k) \cdot (M_{k+1} M_{k+2} \dots M_n).$$

for some $k \in [n-1]$. Thus, the first decision to make is the choice of k. Iterating these decisions leads to subproblems specified by a subinterval $[i, j]$ of the integers $[n]$, specified by integers i and j such that $1 \le i < j \le n$. Correspondingly, we define $\mathrm{OPT}(i, j)$ as the minimum number of integer multiplications needed to compute $M_i M_{i+1} \dots M_j$. We are interested in $\mathrm{OPT}(1, n)$, and have the recurrence:

$$\mathrm{OPT}(i, j) = \min_{i \le k < j} (\mathrm{OPT}(i, k) + \mathrm{OPT}(k+1, j) + d_{i-1} d_k d_j), \tag{4}$$

where $\mathrm{OPT}(i, i)$ is set to 0 for $i \in [n]$.

Our OPT table is again 2-dimensional, but only the top-right half of the table is used. This is because the subproblems correspond to intervals. We evaluate (4) from smallest to largest interval size. The table contains $\binom{n}{2} = \Theta(n^2)$ entries. A single evaluation of (4) takes $O(n)$ times, resulting in a total running time of $O(n^3)$. Retrieving a computation schedule achieving $\mathrm{OPT}(1, n)$ can be done in the standard way. Both evaluating $\mathrm{OPT}(1, n)$ and retrieving a schedule seem to require keeping $O(n^2)$ entries in memory.

5 Shortest Paths

We revisit the shortest paths problem from lecture 2, but now allow the edge lengths to take on negative values. The problem is well-defined iff there exists no cycle with negative total length that is reachable from s and from which t can be reached.

No greedy algorithm is known in this general setting. We leave it as exercise to construct an example in which the problem is well-defined but Dijkstra's algorithm fails.

Instead, we use a dynamic programming approach known as Bellman-Ford, where the underlying recurrence is based on the decision of the last edge $e = (v, t)$ on the path from s to t. Given e, what remains is to find a shortest path from s to v. This suggests that we use the end vertex v as an index into our table. Moreover, the number of edges on the path becomes smaller under this recurrence. Thus, for $v \in V$ and non-negative integer k, we define $\mathrm{OPT}(k, v)$ as the length of a shortest path from s to v using no more than k edges, or ∞ if no such path exists. For $k \geq 1$ we have the recurrence

$$\mathrm{OPT}(k, v) = \min(\mathrm{OPT}(k-1, v), \min_{(u,v)\in E}(\mathrm{OPT}(k-1, u) + \ell(u, v))), \tag{5}$$

with $\mathrm{OPT}(0, s)$ set to 0 and $\mathrm{OPT}(0, v)$ set to ∞ for $v \neq s$.

We compute the OPT table row-by-row. We start with the base case $k = 0$. In order to evaluate (5) in a given row, we initialize the row $\mathrm{OPT}(k, \cdot)$ as a copy of the previous row $\mathrm{OPT}(k-1, \cdot)$, and then cycle over all edges $e = (u, v) \in E$ and update $\mathrm{OPT}(k, v)$ as $\min(\mathrm{OPT}(k, v), \mathrm{OPT}(k-1, u) + \ell(u, v))$. The amount of work per row is $O(n)$ for the initialization, and $O(m)$ for the updates. If the problem is well-defined, then there is a shortest path that does not repeat any vertex, which implies that the number of edges on the path is no more than $n-1$. Thus, it suffices to compute the rows up to $k = n-1$, and the overall running time is $O((n+m)n)$. Compare this to the $O((n+m)\log n)$ for Dijkstra's algorithm in the case of non-negative edge lengths, and $O(n+m)$ for breadth-first search in case all the edge lengths are the same.

If two consecutive rows in the OPT table are identical, all subsequent rows will be identical, and there is no need to increase k any further. If row $k = n$ is not identical to row $k-1$, then the digraph contains a cycle of negative total weight that is reachable from s. This property can be used to detect whether the problem is well-defined in time $O(n+m)n)$.

When computing the OPT values in a row-by-row fashion, one only needs to keep track of the previous row, so the memory requirement is $O(n)$. Retrieving the path in the standard way requires storing the entire table of size $O(n^2)$. The strategy mentioned at the end of our discussion on sequence alignment reduces the space requirement to $O(n)$ while slightly increasing the running time to $O(n+m)nlogn)$. In fact, one can do with a one-dimensional array $\mathrm{OPT}'(v)$ which is initialized as $\mathrm{OPT}(0, v)$ and keep iterating over the edges $e = (u, v) \in E$ and applying the following update rule:

$$\mathrm{OPT}'(v) = \min(\mathrm{OPT}'(v), \mathrm{OPT}'(u) + \ell(u, v)).$$

The resulting algorithm for retrieving a shortest path runs in time $O(n+m)n)$ and space $O(n)$. [1]

[1] The meaning of the intermediate values of OPT' during the execution of the algorithm is not as clean as for OPT.