SIENAcollege
Computer Science

Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2024

Topic Notes: Dynamic Programming

We next consider *dynamic programming*, a technique for designing algorithms to solve problems by setting up recurrences with overlapping subproblems (smaller instances), solving those smaller instances, remembering their solutions in a table to avoid recomputation, then using the subproblem solutions from the table to obtain a solution to the original problem.

The idea comes from mathematics, where "programming" means "planning".

Simple Example: Fibonacci Numbers

You have certainly seen the Fibonacci sequence before, defined by:

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0$$
$$F(1) = 1$$

A direct computation using this formula would involve recomputation of many Fibonacci numbers before $F(n)$.

But if instead, we store the answers to the subproblems in a table (in this case, just an array), we can avoid that recomputation. For example, when we compute $F(n)$, we first need $F(n-1)$, then $F(n-2)$. But the computation of $F(n-1)$ will also have computed $F(n-2)$. With a dynamic programming technique, that answer will have been stored, so $F(n-2)$ is immediately available once we have computed it once.

With the Fibonacci sequence, we can take a complete "bottom up" approach, realizing that we will need answers to all smaller subproblems ($F(0)$ up to $F(n-1)$) in the process of computing $F(n)$, we can populate an array with 0 in element 0, 1 in element 1, and all successive elements with the sum of the previous two. This makes the problem solvable in linear time, but uses linear space.

Moreover, with this approach, we need only keep the most recent two numbers in the sequence, not the entire array. So we can still get a linear time solution constant space.

The Change Maker Problem

Levitin has an example problem we will discuss extensively in class where we are to find the smallest number of coins needed to add up to a specific total, given a set of coin denominations. We do this all the time when making change with our standard set of coins: 37 cents is most efficiently denominated with a quarter, a dime. and two pennies.

With our coin system, we would be able to use a *greedy* technique (which is our next major category of algorithms after dynamic programming). That means you would always take as many of the largest denomination possible without exceeding your total, then move down to lower denominations, until you have the desired total.

In the general case, this approach will not work. If you had coins worth 10, 7, 2, and 1, and you wanted to make 14 cents, the greedy approach would yield 1 10-cent piece and 2 2-cent pieces, while the optimal combination is 2 7-cent pieces.

One way to solve this problem in the general case is through an exhaustive search: try all possible legal combinations and take the one that used the fewest coins. The algorithm below accomplishes this.

ALGORITHM CHANGEMAKER(D, amt)
//Input: $D[0..d-1]$ array of coin denominations
//Input: amt desired coin total value
//Output: the minimum number of coins in D to sum to amt
if $amt = 0$ **then**
return 0
$min \leftarrow \infty$
for $i \leftarrow 0..d-1$ **do**
if $amt \geq D[i]$ **then**
$x \leftarrow 1 + ChangeMaker(D, amt - D[i])$
if $x < min$ **then**
$min \leftarrow x$
return min

Following this algorithm, we can create an exhaustive search trace by drawing the recursive call tree.

In class, we will build such a tree for `D = [1, 3, 5], amt = 8`.

How many times is `ChangeMaker` called?

More importantly, how many times does it compute each subproblem?

Dynamic programming aims to avoid all of those repeated computations.

What if we could keep track of the answers for subproblems we have already completed, so we could just look them up rather than recompute them each time?

ALGORITHM CHANGEMAKER($D, amt, sols$)
//Input: $D[0..d-1]$ array of coin denominations
//Input: amt desired coin total value
//Input: $sols[0..maxAmt]$ saved solutions, initialized to all -1
//Output: the minimum number of coins in D to sum to amt
if $amt = 0$ **then**
return 0
// did we already compute this amount's result?
if $sols[amt] \geq 0$ **then**

return $sols[amt]$
// we need to compute this amount's result
$min \leftarrow \infty$
for $i \leftarrow 0..d-1$ **do**
if $amt \geq D[i]$ **then**
$x \leftarrow 1 + ChangeMaker(D, amt - D[i])$
if $x < min$ **then**
$min \leftarrow x$
// save this result before returning in case we need it again
$sols[amt] \leftarrow min$
return min

The approach uses a technique called *memory functions*. These work just like regular functions, but do what we described above. At the start of the function, it looks to see if the requested instance has already been solved. If so, we just return the previously-computed result. If not, we compute it, save it in the table of answers in case it's needed again, then return the answer.

To analyze how much better this approach is, let's define a *non-trivial* call here as one that computes a new entry in $sols$ by executing the for loop and making recursive calls. Now we can re-draw the call tree (in class).

How many non-trivial calls are needed here? What is the largest number for a given value of amt?

We can estimate the running time here: each non-trivial call for a problem size with d coin denominations does a small constant amount of work in addition to the recursive call.

In the worst case, we will perform $d \cdot amt$ iterations of the for loop.

This is what is called a *top-down dynamic programming solution*. The memory function idea is used to compute results as needed, storing them in the array of solutions. In this case, we filled in all possible values. This does not always happen – it's possible some subproblems will never need to be solved.

We can simplify the whole process and in some cases improve efficiency with a *bottom-up dynamic programming solution*. Here, we instead loop over subproblems from smaller to larger size and compute each. This way, we know that all needed subproblem solutions would have already been computed each step of the way.

ALGORITHM CHANGEMAKER(D, amt)
//Input: $D[0..d-1]$ array of coin denominations
//Input: amt desired coin total value
//Output: the minimum number of coins in D to sum to amt
$sols[0..amt] \leftarrow [-1, -1, ..., -1]$
$sols[0] \leftarrow 0$
for $a \leftarrow 1..amt$ **do**
// determine $sols[a]$
$min \leftarrow \infty$
for $i \leftarrow 0..d-1$ **do**

if $a \geq D[i]$ **then**
$x \leftarrow 1 + sols[a - D[i]]$
if $x < min$ **then**
$min \leftarrow x$
$sols[a] \leftarrow min$
return $sols[amt]$

Here, the inner loop will execute $d \cdot amt$ times.

In class exercise: complete the bottom-up solution for
`D=[1, 5, 7]` and $amt = 24$.

Binomial Coefficients

Another example you've probably seen before is the computation of *binomial coefficients*: the values $C(n, k)$ in the binomial formula:

$$(a + b)^n = C(n, 0)a^n + \cdots + C(n, k)a^{n-k}b^k + \cdots + C(0, n)b^n.$$

These numbers are also the n^{th} row of Pascal's triangle.

The recurrences that will allow us to compute C(n,k):

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k) \quad \text{for n > k > 0}$$
$$C(n, 0) = 1$$
$$C(n, n) = 1$$

As with the Fibonacci recurrence, the subproblems are overlapping – the computation of $C(n - 1, k - 1)$ and $C(n - 1, k)$ will involve the computation of some of the same subproblems. So a dynamic programming approach is appropriate.

The following algorithm will fill in the table of coefficients needed to compute $C(n, k)$:

ALGORITHM BINOMIAL(n, k)
// Input: n, the power for $(a + b)^n$
// Input: k, the exponent of b in the desired term in the expanded polynomial
// $C[0..n, 0..k]$ is a dynamic programming array
// Output: the coefficient of the $a^{n-k}b^k$ term in the expanded polynomial
for $i \leftarrow 0..n$ **do**
for $j \leftarrow 0..min(i, k)$ **do**
if $j = 0$ **or** $j = i$ **then**
$C[i, j] \leftarrow 1$
else
$C[i, j] \leftarrow C[i - 1, j - i] + C[i - 1, j]$

return $C[i, k]$

It's been a while since we did a more formal analysis of an algorithm's efficiency.

The basic operation will be the addition in the `else`. This occurs once per element that we compute. In the first $k + 1$ rows, the inner loop executes i times (the first double summation in the formula below). For the remaining rows, the inner loop executes $k + 1$ times (the second double summation). There are no differences in best, average, and worst cases: we go through the loops in their entirety regardless of the input.

So we can compute the number of additions $A(n, k)$ as:

$$\begin{aligned} A(n,k) &= \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1 \\ &= \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k \\ &= \frac{(k-1)k}{2} + k(n-k) \in \Theta(nk). \end{aligned}$$

Knapsack Problem

We now revisit the *knapsack problem*, which we first considered with a brute-force approach.

Recall the problem:

Given n items with weights $w_1, w_2, ..., w_n$ and values $v_1, v_2, ..., v_n$, what is the most vaulable subset of items that can fit into a knapsack that has a total weight capacity of W.

When we first considered the problem, we generated all possible subsets and selected the one that resulted in the largest total value where the sums of the weights were less than or equal to W.

To generate a more efficient approach, we use a dynamic programming approach. We can break the problem down into overlapping subproblems as follows:

Consider the instance of the problem defined by first i items and a capacity j ($j \leq W$).

Let $V[i, j]$ be optimal value of such an instance, which is the value of an optimal solution considering only the first i items that fit in a knapsack of capacity j.

We can then solve it by considering two cases: the subproblem that includes the i^{th} item, and the subproblem that does not.

If we are not going to include the i^{th} item, the optimal subset's value (considering only items 0 through $i - 1$) would be $V[i - 1, j]$.

If we do include item i (which is only possible if $j - w_i \geq 0$), its value is obtained from the optimal subset of the first $i - 1$ items that can fit within a capacity of $j - w_i$, as $v_i + V[i - 1, j - w_i]$.

It's also possible that the i^{th} item does not fit (where $j - w_1 < 0$), in which case the optimal subset is $V[i-1, j]$.

We can formulate these into a recurrence:

$$V[i,j] = \begin{cases} \max\{V[i-1,j], v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

Combine with some initial conditions:

$V[0, j] = 0$ for $j \geq 0$, $V[i, 0] = 0$ for $i \geq 0$.

The solution to our problem is the value $V[n, W]$.

The text has an example in Figure 8.5, where the solution is obtained in a "bottom up" fashion, filling in the table row by row or column by column, until we get our ultimate answer at in the lower right corner. This guarantees we will compute each subproblem exactly once and is clearly $\Theta(nW)$ in time and space.

One problem with that approach is the fact that we are computing some subproblems that will never be used in a recurrence starting from $V[n, W]$.

An alternate approach, this time working from the top (the solution) down (the subproblems), is to solve each subproblem as it's needed. But...to avoid recomputing subproblems, we remember the answers to subproblems we have computed and just use them when they exist.

The pseudocode on p. 295 of Levitin implements the memory function idea for the knapsack problem.

Applying this approach to the text's example results in only 11 of the 20 values that were not initial conditions are computed, but only one value is reused (Figure 8.6). Larger instances would result in more subproblems being unneeded and more being reused.

Warshall's and Floyd's Algorithms

Levitin presents these two graph algorithms as dynamic programming algorithms, although they do not fit the pattern of the algorithms we have seen so far.

Reachability

As a simple example of something we can do with a graph, we determine the subset of the vertices of a graph $G = (V, E)$ which are *reachable* from a given vertex s by traversing existing edges.

A possible application of this is to answer the question "where can we fly to from ALB?". Given a directed graph where vertices represent airports and edges connect cities which have a regularly-scheduled flight from one to the next, we compute which other airports you can fly to from the starting airport. To make it a little more realistic, perhaps we restrict to flights on a specific airline.

Reachability can be determined using the graph traversal algorithms we considered earlier (breadth-

first, depth-first), or in fact any spanning tree algorithm.

The cost of this procedure will involve at most $\Theta(|V|+|E|)$ operations if all vertices are reachable, which is around $\Theta(|V|^2)$ if the graph is dense.

We can think about how to extend this to find reasonable flight plans, perhaps requiring that all travel takes place in the same day and that there is a minimum of 30 minutes to transfer.

Transitive Closure

Taking the *transitive closure* of a graph involves adding an edge directly all other vertices that are reachable from each vertex. We could do this by computing the reachability for each vertex, in turn, with, for example, the graph traversal algorithms. This would cost a total of $\Theta(|V|^3)$.

A more direct approach is due to Warshall.

We modify the graph so that when we're done, for every pair of vertices u and v such that v is reachable from u, there is a direct edge from u to v.

Note that this is a destructive process! We modify our starting graph.

The idea is that we build the transitive closure iteratively. When we start, we know that edges exist between any vertices that are connected by a path of length 1.

We can find all pairs of vertices which are connected by a path of length 2 (2 edges) by looking at each pair of vertices u and v and checking, for each other vertex, whether there is another vertex w such that u is connected to w and w is connected to v. If so, we add a direct edge u to v.

If we repeat this, we will then find pairs of vertices that were connected by paths of length 3 in the original graph. If we do this $|V|$ times, we will have all possible paths added.

The outermost loop is over the "intermediate" vertices (the w's), and inner loops are over u and v.

This is still a $\Theta(|V|^3)$ algorithm, though efficiency improvements are possible.

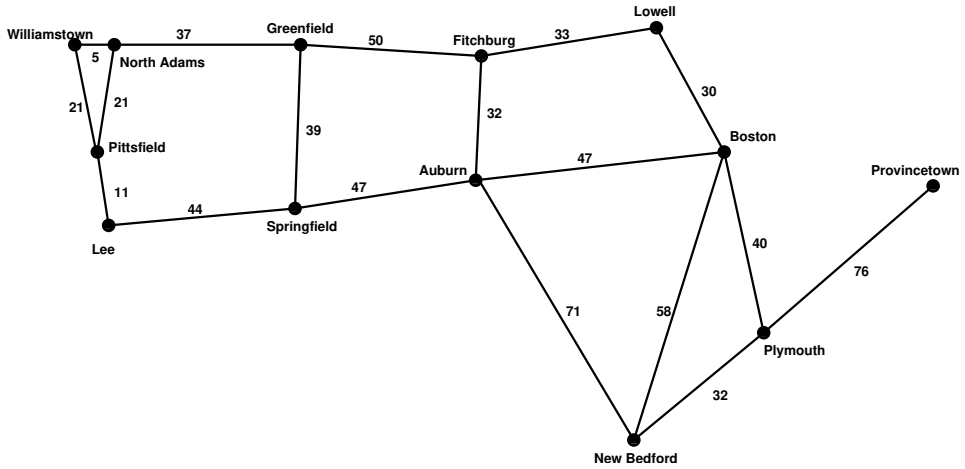
Here is psuedocode for Warshall's algorithm as an operation directly on an adjacency matrix representation of a graph, where each entry is a boolean value indicating whether an edge exists.

ALGORITHM WARSHALL(A)
  //Input: $A[1..n][1..n]$ an adjacency matrix representation of a graph
  // where $A[i][j]$ is **true** iff a edge exists from i to j
  // Each $R^{(i)}[1..n][1..n]$ is an iteration toward the closure
  $R^{(0)} \leftarrow A$
  for $k \leftarrow 1..n$ **do**
    for $i \leftarrow 1..n$ **do**
      for $j \leftarrow 1..n$ **do**
        $R^{(k)}[i][j] \leftarrow R^{(k-1)}[i][j]$ **or** $(R^{(k-1)}[i][k]$ **and** $R^{(k-1)}[k][j])$
  return $R^{(n)}$

All Pairs Minimum Distance

We can expand just a bit on the idea of Warshall's Algorithm to get *Floyd's Algorithm* for computing minimum distances between all pairs of (reachable) vertices – the *all sources shortest path problem*.

For the example graph:



We can use the same procedure (three nested loops over vertices) as we did for Warshall's Algorithm, but instead of just adding edges where they may not have existed, we will add or modify edges to have the minimum cost path (we know of) between each pair.

The pseudocode of this algorithm below again works directly on the adjacency matrix, now with weights representing the edges in the matrix.

ALGORITHM FLOYD(W)
  //Input: $W[1..n][1..n]$ an adjacency matrix representation of graph edge weights
  // nonexistent edges have a weight of ∞
  $D \leftarrow W$ // a matrix copy
  for $k \leftarrow 1..n$ **do**
    for $i \leftarrow 1..n$ **do**
      for $j \leftarrow 1..n$ **do**
        $D[i][j] \leftarrow min(D[i][j], D[i][k] + D[k][j])$
  return D

Like Warshall's Algorithm, this algorithm's efficiency class is $\Theta(|V|^3)$.

Notice that at each iteration, we are overwriting the adjacency matrix, so again we have a destructive algorithm.

And we can see Floyd's Algorithm in action using the above simple graph with the "MassFloyd" program distributed in GitHub.