21120491 Advanced Data Structure and Algorithm Analysis **Spring-Summer 2022**

Lecture 8: Dynamic Programming

Lecturer: Deshi Ye *Scribe: D. Ye*

8.1 Overview

If a recursive solution can be used to get a solution very inefficiently (by repeatedly searching through the same subproblems), then memorization and dynamic programming can be used to make it more efficient.

Recall that divide-and-conquer is good for subproblems that are independent and solved separately, in which the instance size is degraded proportionally, e.g., like the general form of $T(n) = aT(n/b) + f(n)$.

Dynamic programming is to break up a problem into many overlapping sub-problems, then build solutions for larger and larger sub-problems, and store solutions for sub-problems for reuse.

There are two different ways of solving Dynamic programming problems:

- **Memorization:** Top Down. This is a modified version of the **recursive approach** where we are storing the solution of sub-problems in an extra memory or look-up table to avoid the re-computation.
- **Tabulation:** Bottom Up. This is an **iterative approach** to build the solution of the sub-problems and store the solution in an extra memory.

Example: Calculate Fibonacci Numbers: $F(N) = F(N-1) + F(N-2)$.

Memorization approach:

```
Initialize an array F[N+1] with negative values.

int fib(N) {
   if( F[N] < 0 ) {
     if( N <= 1 ) F[N] = 1
    else F[N] = fib(N-1) + fib(N-2)
  }
return F[N]
```

Tabulation approach:

```
int  Fibonacci ( int N )
{   int  i, Last, NextToLast, Answer;
    if ( N <= 1 )  return  1;
    Last = NextToLast = 1;    /* F(0) = F(1) = 1 */
    for ( i = 2; i <= N; i++ ) {
        Answer = Last + NextToLast;   /* F(i) = F(i-1) + F(i-2) */
```

```
7           NextToLast = Last; Last = Answer;  /* update F(i-1) and F(i-2) */
8       }  /* end-for */
9       return  Answer;
10  }
```

Memorization is easy to code i.e. you just add an array or a lookup table to store down the results and never recomputes the result of the same problem. With memorization, if the recursion tree is very deep, you will run out of stack space, which will crash your program. The advantage of the Tabulation method is smart to remember subproblems. But, usually, the method is harder. We need to understand the subproblem dependencies and solve subproblems in order which never require solutions to subproblems not solved before.

If a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have **optimal substructure**.

When shall we use dynamic programming? If the problem has overlapping sub-problems and optimal substructure, then it is good to use dynamic programming to solve it.

Steps to design a dynamic programming:

1. Characterize an optimal solution
2. Recursively define the optimal values
3. Compute the values in some order
4. Reconstruct the solving strategy

8.2 All-Pairs Shortest Path

Given a directed graph $G = (V, E)$ with real-valued edge lengths and no negative cycles, our goal is to find the shortest path for all pairs of $v_i \in V$ and $v_j \in V$ ($i \neq j$). Let $l(e)$ be the edge length for the edge $e \in E$.

Characterize an optimal solution: There are many ways to define optimal solutions[1].

In this lecture, we introduce the Floyd-Warshall Algorithm [1]. To define the subproblems, consider an input graph $G = (V, E)$ and arbitrarily assign its vertices the names $1, 2, \ldots, n$ (where $n = |V|$). Subproblems are then indexed by prefixes $\{1, 2, \ldots, k\}$ of the vertices, with k serving as the measure of subproblem size, as well as an origin v_i and destination v_j.

Denotes $D^k[i][j]$ to be the shortest path (if any) from node v_i to node v_j that passes through only vertices numbered at most k, *i.e.*, use vertices $v_1, v_2, \ldots, v_k$ as intermediate vertices, and does not contain a directed cycle. Then the length of the shortest path from v_i to v_j is $D^n[i][j]$. If no such path exists, $D^n[i][j] = +\infty$.

There are $(n + 1) \cdot n \cdot n = O(n^3)$ subproblems. For a fixed origin v_i and destination v_j, the set of allowable paths grows with k, and so $D^k[i][j]$ can only decrease as k increases.

Recursively define the optimal values: The base case $D^0[i][j] = l(i, j)$ if (v_i, v_j) is an edge. Further, $D^0[i][i] = 0$, and $D^0[i][j] = +\infty$ if (v_i, v_j) is not an edge.

[1]The Bellman-Ford Algorithm is to denote $D(i, j, k)$ to be the shortest path from node v_i to v_j using at most k edges. Then the length of the shortest path from v_i to v_j is $D(i, j, n - 1)$.

We will recursively define the optimal values and show that the problem has an optimal substructure. Suppose P is a $v_i - v_j$ path with no cycles and all internal vertices in $\{1, 2, \ldots, k\}$, and it is the shortest path. Ask yourself a question, what must the shortest path P look like? There are only two cases. Either the vertex k is an internal vertex of P or the vertex k is not an internal vertex of P.

Case 1: Vertex k is not an internal vertex of P. In this case, the path P can immediately be interpreted as a solution to a smaller subproblem with prefix length $k-1$, still with origin v_i and destination v_j. The path P must be an optimal solution to the smaller subproblem. That means $D^k[i][j] = D^{k-1}[i][j]$.

Case 2: Vertex k is an internal vertex of P. Let P_1 be the prefix of P that travels from v_i to k, and P_2 the suffix of P that travels from k to v_j. In this case, we can view P_1 is the solution of the subproblem $D^{k-1}[i][k]$ and P_2 the solution of the subproblem $D^{k-1}[k][j]$.

To show the property of optimal substruture. We need to prove that P_1 is the optimal solution of the subproblem $D^{k-1}[i][k]$, and P_2 is the optimal solution $D^{k-1}[i][k]$, respectively.

We prove it by contradiction. Without loss of generality, we assume P_1^* is the optimal solution of the subproblem $D^{k-1}[i][k]$, while P_1 is not. If the concatenation of P_1^* and P_2 would be a cycle-free path P^* from v_i to v_j with internal vertices in $\{1, 2, \ldots, k\}$.

If the concatenation P^* of P_1^* and P_2 contains a cycle, we can repeatedly splicing out cycles such that v_i is still the origin and v_j is the destination. Note that if the input graph has **no negative cycles**, cycle-splicing can only shorten a path.

Up to now, if P_1 is not an optimal solution, we can construct a new path P^* with a smaller length, which is a contradiction that P is the shortest path, and then the property of optimal substructure is satisfied.

Remark: The Floyd-Warshall Algorithm works when there is no negative cycle.

In sum, for every $k \in \{1, 2, \ldots, n\}$, v_i and v_j, the recurrence form is

$$D^k[i][j] = \min \begin{cases} D^{k-1}[i][j] & \text{Case 1} \\ D^{k-1}[i][k] + D^{k-1}[k][j] & \text{Case 2} \end{cases} \tag{8.1}$$

Compute the values in some order We can solve all subproblems by for loops. The algorithm uses a corresponding triple loop because subproblems are indexed by three parameters (an origin, a destination, and a prefix of vertices). It's important that the outer loop is indexed by the subproblem size k, so that all of the relevant terms d_{ij}^k are available for constant-time lookup in each inner loop iteration.

```
1 for k = 1 to n      // subproblem size
2         let D^k = (d_ij^k) be a new n × n matrix
3         for i = 1 to n          // origin
4             for j = 1 to n      // destination
5                 d_ij^k = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
```

The running time is $O(n^3)$.

Reconstruct the solving strategy In the above, we already get the value of an optimal solution. To construct the shortest path for each origin v_i and destination v_j, we can use an array helper$[i, j]$ to story the nodes that the path P travels from v_i to v_j.

In Case 2, when the path P travels through k, then helper$[i, j] = k$, otherwise, let helper$[i, j]$ = null. To find the shortest path from v_i to v_j, we consult helper$[i, j]$. If it is null, then the shortest path is just the edge

(i, j). Otherwise, we recursively compute the shortest path from v_i to $k = \text{helper}[i, j]$ and concatenate this with the shortest path from $k = \text{helper}[i, j]$ to v_j.

Final remark: we can use the Floyd-Warshall Algorithm to check whether a graph contains a negative cycle.

Lemma 8.1 *The input graph $G = (V, E)$ has a negative cycle if and only if, at the conclusion of the Floyd-Warshall algorithm, $D^n[i][i] < 0$ for some node $v_i \in V$.*

Proof: If the graph G does not contain a negative cycle, the Floyd-Warshall Algorithm correctly computes all shortest-path distances. and there is no path from a vertex v to itself shorter than the empty path (which has length 0). Thus $D^n[i][i] = 0$ for all $i \in V$.

If the graph G contains a negative cycle. We leave it as an exercise to show that $D^n[i][i] < 0$ for some vertex $v_i \in V$. ■

8.3 Matrix chain multiplication

Given n matrices, we would like to compute $M_1 M_2 \cdots M_n$, in which the matrix M_i has size of $r_{i-1} \times r_i$. Our goal is to determine the order of multiplications such that the number of multiplications is minimized.

Characterize an optimal solution: subproblems Note that matrices cannot be reordered, we only need to consider sequences of matrices. Let $M_{i,j}$ be the result matrix of multiplying matrices M_i through M_j.

It is easy to check that $M_{i,j}$ is a matrix of size $r_{i-1} \times r_j$.

Fact: Multiplying two matrices of size $r \times k$ and $k \times c$ takes rkc multiplications.

Optimal substructure Assume the last step is $M_{i,l} M_{l+1,j}$, where $i \le l < j$ that means we need to compute the subproblem $M_{i,l}$ and the subproblem $M_{l+1,j}$. It costs $r_{i-1} r_l r_j$ multiplications for the last step.

Let $m(i, j)$ denote the minimum number of multiplications needed to compute $M_{i,j}$. Thus,

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le l < j} m(i,l) + m(l+1,j) + r_{i-1} r_l r_j & \text{if } j > i \end{cases} \tag{8.2}$$

Compute the values in some order: Bottom-up implementation We will store the values of $m(i, j)$ is a 2-dimensional array $M[i][j]$. We cannot just compute the matrix in the simple row-by-row order. For example, when computing $M[3][5]$, we would need to access both $M[3, 4]$ and $M[4, 5]$, but $M[4, 5]$ is in row 4, which has not yet been computed. Instead, the trick is to compute diagonal-by-diagonal working out from the middle of the array. In particular, we organize our computation according to the number of matrices in the subsequence.

The variable k denotes the length of the matrix chain, and it should be in the outside loop.

```
1 /* r contains number of columns for each of the N matrices */
2 /* r[ 0 ] is the number of rows in matrix 1 */
```

```
/* Minimum number of multiplications is left in M[ 1 ][ N ] */
void OptMatrix( const long r[ ], int N, TwoDimArray M )
{   int  i, j, k, L;
    long  ThisM;
    for( i = 1; i <= N; i++ )    M[ i ][ i ] = 0;
    for( k = 1; k < N; k++ ) /* k = j - i */
        for( i = 1; i <= N - k; i++ ) { /* For each position */
            j = i + k;     M[ i ][ j ] = Infinity;
            for( L = i; L < j; L++ ) {
                ThisM = M[ i ][ L ] + M[ L + 1 ][ j ]
                 + r[ i - 1 ] * r[ L ] * r[ j ];
                if ( ThisM < M[ i ][ j ] )  /* Update min */
                M[ i ][ j ] = ThisM;
    }  /* end for-L */
         }  /* end for-Left */
}
```

The running time is $O(n^3)$.

8.4 Product Assembly

Given two assembly lines, each with n stations. Stations of assembly-line 0 are $S_{0,0}, S_{0,1}, \ldots, S_{0,n-1}$, respectively. Stations of assembly-line 1 are $S_{1,0}, S_{1,1}, \ldots, S_{1,n-1}$, respectively.

Stations $S_{0,i}$ and $S_{1,i}$ perform the same function, but (may) have different assembly times (or process time) $t_{0,i}$ and $t_{1,i}$. Transfer time $t_{0\to1,i}$ for station $S_{0,i}$ to transfer the job to the assembly-line 1. Similarly, it takes $t_{1\to0,i}$ time for station $S_{1,i}$ to transfer the job the assembly-line 0.

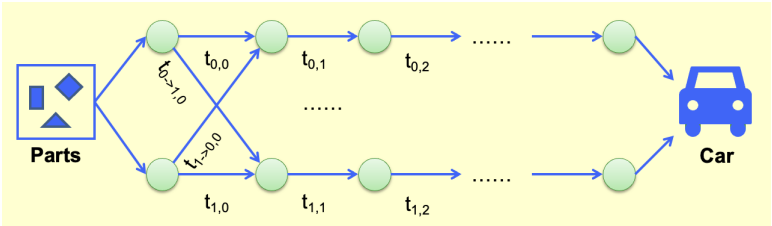

Figure 8.1: There is at most one point in each box

Our goal is to find a selection of stations from the two lines that gives the fastest way (or minimum time) through the factory.

Optimal substructure: To define suitable subproblems, we first consider what is the fastest way through station $S_{0,j}$? If $j = 1$, it is easy that it costs $t_{0,0}$. If $j > 1$, it may have reached $S_{0,j}$ in two different ways:

- Through station $S_{0,j-1}$ (on the same assembly-line).
- Through station $S_{1,j-1}$ (transfer from other assembly-line).

Suppose the fastest way is through $S_{0,j-1}$, note that we must have taken the fastest way through station $S_{0,j-1}$. Suppose the fastest way is through $S_{1,j-1}$, note that we must have taken the fastest way through station $S_{1,j-1}$.

Recursively define the optimal values: Let $f_i[j]$ be the minimum time to get through $S_{i,j}$, where $i = 0, 1$. Then, the optimal value for problem is $f^* = \min(f_0[n-1], f_1[n-1])$.

The base case is $f_0[1] = t_{0,0}$, and $f_1[1] = t_{1,0}$.

Recursive computation of $f_i[j]$:

$$f_0[j] = \min(f_0[j-1] + t_{0,j}, f_1[j-1] + t_{1\to 0,j} + t_{0,j})$$
$$f_1[j] = \min(f_1[j-1] + t_{1,j}, f_0[j-1] + t_{0\to 1,j} + t_{1,j})$$

Compute the values in some order We compute them directly in order of increasing station numbers, which is For $j = 2, \ldots, n$. It is possible since $f_i[j]$ depends only on $f_0[j-1]$ and $f_1[j-1]$. The running time is $O(n)$.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.