Dynamic Programming

Megan M. Sheets, Eric Larson, Matt Boettger

Edited by Chuck Cusack

These notes are based on chapter 16 of [1] and lectures from CSCE423/823, Spring 2001. For a more basic introduction to dynamic programming, see the lecture notes from CSCE310, Fall 2000.

Many divide-and-conquer algorithms, like `Mergesort`, solve problems by combining the solutions of subproblems. However, in some cases, a divide-and-conquer algorithm solves each subproblems many times, which is clearly not efficient. In these cases, a technique called **dynamic programming** solves such problems so that each subproblem is solved only once. Dynamic programming usually applies to optimization problems.

1 Dynamic Programming

Definition 1 *An* **optimization problem** *is one in which we assign a value to each solution, and wish to find a solution with minimal or maximal value.*

Definition 2 **Dynamic programming** *is a technique to solve optimization problems that exhibit two properties: optimal substructure and overlapping subproblems.*

- *A problem exhibits* **optimal substructure** *if an optimal solution to a problem consists of optimal solutions to subproblems (or subsolutions).*
- *A problem exhibits* **overlapping subproblems** *if the standard recursive algorithm to solve the problem would solve subproblems many times.*

Dynamic programming solves problems by combining the solutions to subproblems. In order for the technique to work, a problem must exhibit optimal substructure, since without it, there is no basis for defining subproblems. In order for the technique to provide a more efficient solution than a standard recursive algorithm, a problem should exhibit overlapping subproblems.

Dynamic programming provides a more efficient solution than a standard recursive algorithm by saving the solutions to subproblems in a table. To develop a dynamic programming algorithm, the following steps should be followed.

1. Define what is meant by optimal solution.
2. Define the value of an optimal solution based on values of subsolutions. In other words, develop a recursive definition for the value of an optimal solution.

3. Compute the values of all subsolutions.

4. Construct the optimal solution from computed information.

There are two ways of accomplishing step 3 above: bottom-up or memoization. Bottom-up simply means to compute the values in the table in a systematic way, usually row by row. **Memoization** uses the regular recursive algorithm with one modification. Whenever the value of a subproblem is computed, it is stored in a table, and every time a a recursive call is made, the table is checked first, and if the value is available, it is used instead of being re-computed.

Problems which can be solved with dynamic programming include

- Matrix Chain Multiplication
- 0-1 Knapsack Problem
- Longest Common Subsequence
- Optimal Polygon Triangulation

We now turn to a few examples.

2 Longest Common Subsequence (LCS)

Let $X = \langle x_1, x_2, \ldots, x_n \rangle$. Then $Z = \langle z_1, z_2, \ldots, z_n \rangle$ is a **subsequence** of X if there exists a strictly increasing sequence $\langle i_1, \ldots, i_k \rangle$ such that $x_{i_j} = z_j$ for $j = 1, \ldots, k$.

Example: $X = \langle A, F, X, Q, R, N \rangle$. Then $Z = \langle A, Q, R, N \rangle$ is a subsequence of X, and $Z = \langle A, Q, N, R \rangle$ is not a subsequence of X.

Definition 3 *If X and Y are sequences, Z is a* **common subsequence** *of X and Y if Z is a subsequence of both X and Y.*

Definition 4 *A* **longest common subsequence (LCS)** *of X and Y is a common sequence of maximal length.*

Example: Let $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$ The sequence $Z = \langle B, C, A \rangle$ is a common subsequence of X and Y. But this sequence is not a longest common subsequence of X and Y because their exists a longer sequence ($Z = \langle B, C, B, A \rangle$, for example).

Now, we will develop an algorithm to compute an LCS of two strings.

Theorem 1 (Optimal Substructure) *Let $X = \langle x_1, \ldots, x_n \rangle$, $Y = \langle y_1, \ldots, y_m \rangle$, and let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y. Then:*

1. *If $x_n = y_m$, then $z_k = x_n = y_m$ and Z_{k-1} is and LCS of X_{n-1} and Y_{m-1}.*

2. *If $x_n \neq y_m$ and $z_k \neq x_n$ then Z is an LCS of X_{n-1} and Y.*

3. *If $x_n \neq y_m$ and $z_k \neq y_m$ then Z is an LCS of X and Y_{m-1}.*

Proof:

1. If Z_{k-1} is not an LCS of X_{n-1} or Y_{m-1} then there exists a sequence Z' longer than Z_{k-1} that is an LCS of X_{n-1} and Y_{m-1} (Z' has length of at least k). Then $Z' \cup x_n$ is a common subsequence for X and Y of length at least $k+1$, which contradicts Z being and LCS of X and Y. Therefore Z_{k-1} is an LCS of X_{n-1} and Y_{m-1}.

2. Since X_{n-1} is a subsequence of X, then the length of an LCS for X_{n-1} and Y is no longer than an LCS for X and Y. Since $x_n \neq z_{n-1}$, Z_k is a common substring of X_{n-1} and Y, so it is an LCS of X_{n-1} and Y.

3. Symmetric to 2. □

Define $c[i,j]$ to be the length of an LCS of X_i and Y_j. Then by Theorem 1, we have:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j, \end{cases} \tag{1}$$

Based on the definition of $c[i,j]$, The following algorithm computes the length of an LCS.

```
LCSLength(x,n,y,m)
{
  if(n==0) OR (m==0)
      return 0;
  else if (x[n]==y[m])
      return LCSLength(x,n-1,y,m-1)+1;
  else
      return max{ LCSLength(x,n-1,y,m), LCSLength(x,n,y,m-1) };
}
```

Let $T(n,m)$ be the complexity of `LCSlength(x,n,y,m)`. Then it is not hard to see that

$$T(n,m) = T(n,m-1) + T(n-1,m) + O(1),$$

which can be shown to be exponential. However, the values $c[i,j]$ can be computed row by row, starting with the first row. The value of the optimal solution is $c[n,m]$, and the LCS can be found by tracing backwards through the table. See Section 16.3 of [1] for the details of the algorithm. Figure 1 illustrates the method.

3 Optimal Polygon Triangulation

A convex polygon can be represented by listing the vertices in counterclockwise order. That is, we represent a polygon as $P = \langle v_0, \ldots, v_{n-1} \rangle$, where $\overline{v_0 v_1}$, $\overline{v_1 v_2}$, $\ldots \overline{v_{n-2} v_{n-1}}$, and $\overline{v_{n-1} v_0}$ are the edges.

		A	B	C	A	D	A
	0	**0**	0	0	0	0	0
B	0	0	**1**	1	1	1	1
A	0	1	**1**	1	2	2	2
C	0	1	1	**2**	2	2	2
A	0	1	1	2	**3**	3	3
D	0	1	1	3	3	**4**	**4**
C	0	1	1	2	3	4	**4**

Figure 1: Computing an LCS for $X = \langle A, B, C, A, D, A \rangle$ and $Y = \langle B, A, C, A, D, C \rangle$. We compute the table starting with the top row, left to right. The bold numbers trace back to find the LCS. The LCS is $\langle B, C, A, D \rangle$.

A **triangulation** T of a polygon is a set of chords that divide the polygon into triangles. A triangulation of polygon with n vertices has $n - 3$ chords and $n - 2$ triangles. The weight of a triangle $\triangle v_i v_j v_k$ can be defined in several ways. Two possibilities are the area of the triangle, or the sum of the lengths of the edges:

$$W(\triangle v_i v_j v_k) = |\overline{v_i v_j}| + |\overline{v_j v_k}| + |\overline{v_i v_k}|$$

where $|\overline{v_i v_j}|$ is the Euclidean distance from v_i and v_j. Given a weight function, the weight of a triangulation, $W_T(P)$ is the sum of the weights of the triangles. We define an optimal triangulation to be one which minimizes the sums of the weights of triangles in the triangulation.

We will develop a method of computing an optimal triangulation that will work no matter what weight function is chosen. We begin by showing that an optimal triangulation is composed of optimal triangulations of subproblems.

Theorem 2 *The optimal triangulation exhibits optimal substructure.*

Proof: Let $P = \langle v_0, \ldots, v_{n-1} \rangle$ be a convex polygon, T an optimal triangulation of P with weight $W_T(P)$, and $\triangle v_0 v_k v_{n-1}$ be a triangle in T. (Notice that, for some k, $\triangle v_0 v_k v_{n-1}$ is in T). Let $P_1 = \langle v_0, \ldots, v_k \rangle$, and $P_2 = \langle v_k, \ldots, v_{n-1} \rangle$. Then $T = T_1 \cup T_2 \cup \triangle v_0 v_k v_{n-1}$, where T_1 is the triangulation of P_1 and T_2 is the triangulation of P_2. Notice that

$$W_T(P) = W_{T_1}(P) + W_{T_2}(P) + W(\triangle v_0 v_k v_{n-1}).$$

Assume T_1 is not optimal for P_1 (similar for T_2 and P_2). Then there is a triangulation T_1 for P_1 such that $W_{T_1'}(P_1) < W_{T_1}(P_1)$. But we can construct a triangulation $T' = T_1' \cup T_2 \cup \triangle v_0 v_k v_{n-1}$ for P with

$$\begin{aligned} W_{T'}(P) &= W_{T_1'}(P_1) + W_{T_2}(P_2) + W(\triangle v_0 v_k v_{n-1}) \\ &< W_{T_1}(P_1) + W_{T_2}(P_2) + W(\triangle v_0 v_k v_{n-1}) \\ &= W_T(P) \end{aligned}$$

But this contradicts the fact that T is optimal. Therefore T_1 (and T_2) is optimal, and the polygon triangulation has optimal substructure. $\square$

Theorem 2 is the basis of a recursive definition for the optimal cost of a triangulation. Let $t[i, j]$ be the weight of an optimal triangulation of the polygon $\langle v_{i-1}, v_i, \ldots, v_j \rangle$, for $1 \leq i \leq j \leq n$. Since the polygon $\langle v_{i-1}, v_i \rangle$ is only a line, we define the weight to be zero in this case, so $t[i, i] = 0$ for $i = 0, \ldots n-1$. Then we can define $t[i, j]$ as follows

$$t[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k \leq j-1}(t[i,k] + t[k+1,j] + W(\triangle v_{i-1} v_k v_j) & \text{if } i < j \end{cases}$$

In addition if we store in an array $K[i, j]$ the the value of k that gave optimal triangulation $t[i, j]$, we can reconstruct the optimal triangulation. As with the LCS problem, the recursive algorithm would take exponential time. However, a bottom-up implementation has complexity $O(n^3)$.

It turns out that the problem of computing an optimal parenthesization of a matrix-chain product is a special case of finding an optimal triangulation of a convex polygon. Notice that if we associate each pair of vertices (v_{i-1}, v_i) with matrix A_i, and define the weight to be $W(\triangle v_i v_j v_k) = p_i p_j p_k$, then the recursive definition for $t[i, j]$ is the same as $m[i, j]$.

Figure 2 demonstrates that each triangulation of a polygon has a corresponding parse tree (a full binary tree), as does a full parenthesization of a sequence of $n-1$ matrices.

For more details about the matrix chain multiplication problem, or the correspondence of these two problems, see chapter 16 of [1].

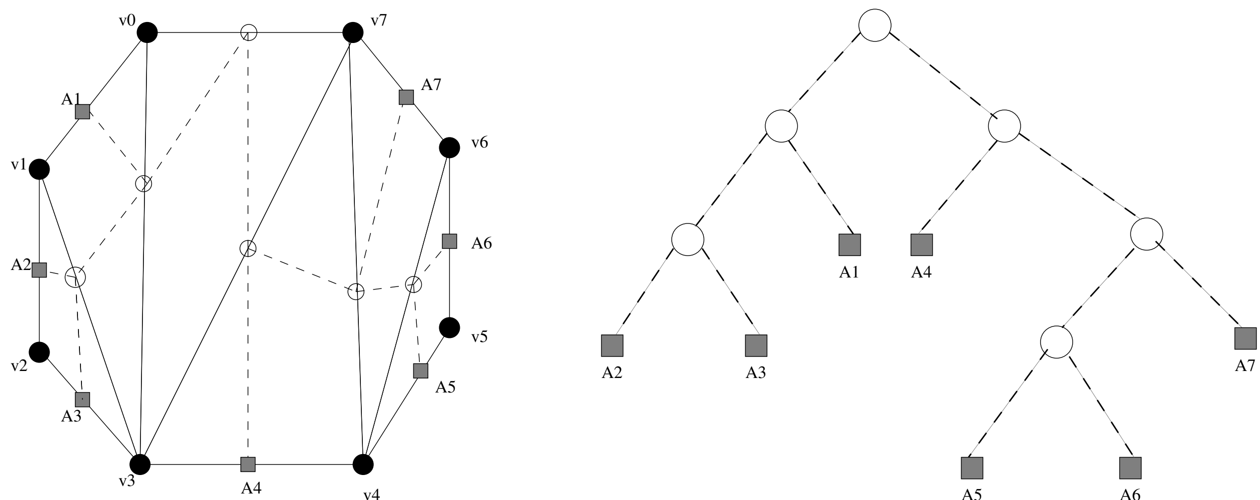

Figure 2: **A triangulation and corresponding parse tree.** The parse tree also corresponds to the parenthesization $(((A_1 A_2) A_3)(A_4((A_5 A_6) A_7)))$

References

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.