The Magic of Dynamic Programming

Author:
M. Ashraf Iqbal and Atif Alvi, Department of Computer Science, Lahore University of Management Sciences, D.H.A., Lahore-54792, Pakistan {aiqbal, alvi}@lums.edu.pk

Abstract — Around the world, students of mathematics, engineering and science struggle every day to learn the difficult art of dynamic programming, an important algorithm design technique. The otherwise successful teaching model of getting the students started on a path of exciting discovery and then letting them find their own way with minimal help fails while teaching dynamic programming. We contend that the primary reason of this failure is the inadequate treatment of this topic in popular algorithm textbooks. In this paper, we provide vital pedagogical aids whose inclusion in the teaching mix would make the learning curve a lot gentler for many students. We propose that teachers should highlight multiple recursive formulations of the same problem followed by their recursion trees and DAGs (directed acyclic graphs). It then becomes relatively easy for students to visualize the problem and to devise a computationally cheap dynamic programming solution. We use typical optimization problems from textbooks and research literature to demonstrate our technique. Overall, we strive to demystify dynamic programming in order to make it accessible to all students.

Index Terms — DAG, dynamic programming, optimal substructure, recursion tree

INTRODUCTION

Reference [1] advocates the teaching methodology of encouraging students to discover almost everything by themselves. This technique works fine for a number of divide & conquer and greedy algorithms but fails while teaching the intricacies of dynamic programming, which is no doubt a more advanced concept. Our teaching experience indicates that it is quite simple to show the actual working of this technique on a specific example but it is extremely difficult to make the students discover it by themselves. It seems as if there is some magic behind this approach or there is something missing. Until and unless we find that missing thing, it will not be possible to effectively teach this otherwise very simple but powerful approach. According to [2]: “After you understand it, dynamic programming is probably the easiest algorithm design technique to apply in practice...Until you understand it, however, dynamic programming seems like magic.”

We contend that some very popular textbooks on algorithms have not done full justice with this important topic and that is why we have serious problems teaching it. In this paper, we shall provide the missing links needed to apply this technique in solving complex problems. We shall describe the inadequacies of the conventional approach of some very popular textbooks on algorithms. We illustrate through two typical dynamic programming problems how our approach improves upon the conventional way of teaching. We also take a brief look at some researchers’ use of dynamic programming and compare it with our methodical approach.

MISSING LINKS OF DYNAMIC PROGRAMMING

We propose that an instructor should go through the following sequence of steps while teaching dynamic programming:

- Make multiple recursive formulations of the optimization problem.
- Make a recursion tree corresponding to the top-down recursive formulation of the problem.
- Convert the recursion tree into a directed acyclic graph (DAG) by merging similar vertices into a single vertex.
- Compute the optimal solution by moving bottom-up in the DAG.

The italicized phrases highlight what is missing from the traditional approach. It is important to have multiple recursive formulations for the same problem in order to make a comparison between various approaches. It is very much possible that one recursive formulation is more efficient in solving one problem while another is more effective in another situation. The students should have multiple choices instead of relying on just one formulation. The DAG is also a very useful tool. It not only helps us design the bottom-up dynamic programming algorithm but is also meaningful otherwise: the directed edges represent the dependencies in the problem and its solution. In fact, the number of edges in the DAG will determine the time...
complexity of the dynamic programming solution (remember that the number of edges in the recursion tree is an exact
measure of the time complexity of the simple recursive implementation of the problem).

We shall now apply our technique to some typical dynamic programming problems.

THE MATRIX-CHAIN MULTIPLICATION PROBLEM

If we have a chain of $A_1, A_2, \ldots, A_n$ of $n$ matrices to be multiplied, where for $i = 1, 2, \ldots, n$, matrix $A_i$ has dimensions $p_{i-1} \times p_i$, we can minimize the number of scalar multiplications if we can discover an optimal parenthesization of the chain of matrices. This is a classic dynamic-programming-amenable problem as it exhibits an optimal substructure: optimal solutions to its subproblems can be combined to give an optimal solution to the whole problem. An optimal parenthesization of $A_1 A_2 \ldots A_j$ that splits the product between $A_k$ and $A_{k+1}$, for some $i \leq k < j$, contains within it optimal parenthesizations of $A_1 A_2 \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$. We should set up a recursion that gives the cost of the optimal solution in terms of optimal solutions to subproblems. For a chain of $n$ matrices, our subproblems are the problems of determining the minimum cost of a parenthesization of $A_i A_{i+1} \ldots A_j$ for $1 \leq i \leq j \leq n$. Let $M[i, j]$ be the minimum number of scalar multiplications needed to compute the product $A_i A_{i+1} \ldots A_j$. The optimal solution to the whole problem would thus cost $M[1, n]$.

It is obvious that $M[i, j] = 0$ for $i = 1, 2, \ldots, n$. For $i < j$, let us assume that an optimal parenthesization splits the product $A_i A_{i+1} \ldots A_j$ between $A_k$ and $A_{k+1}$, where $i \leq k < j$. Keeping in mind that each matrix $A_i$ has dimensions $p_{i-1} \times p_i$, we see that computing the matrix product of the two subproblems entails $p_{i-1} p_k p_j$ scalar multiplications. Thus, we can set up the following recurrence:

$$M[i, j] = \min_{i \leq k < j} \left( M[i, k] + M[k+1, j] + p_{i-1} p_k p_j \right)$$

(1)

Conventional Approach

If recurrence (1) is implemented directly, we end up doing an exponential amount of work. Instead, a dynamic programming solution that computes in a bottom-up manner, storing intermediate results and using them whenever required, does only a polynomial amount of work. This is exactly how [3] attempts to solve the problem. Reference [3] also claims that a recursive implementation of the recurrence takes exponential time, but this claim is substantiated only in a later section. Reference [3] then computes the optimal cost in a bottom-up dynamic programming way. The pseudocode of the dynamic programming algorithm, given on page 336 of [3], operates on a triangular table whose sudden appearance puzzles students. The issue of arriving logically at this table becomes all the more important when one finds that some problems require a rectangular table while others need a three dimensional table for a dynamic programming solution.

Our Approach

Our teaching experience shows that it is best to ask students to draw a recursion tree for a reasonable size matrix-chain immediately after arriving at (1). An example is shown in Figure 1 for a chain of size four. As soon as they draw a recursion tree, they can actually see that it grows exponentially at each layer downwards from the root. They are therefore convinced that a direct recursive algorithm in this case would be expensive, in fact exponential. The number of edges in the tree is exactly $3^{n-1} - 1$. The recursion tree not only captures the essence of the problem in all its complexity, but also holds the key to its solution: the students can see that many nodes (subproblems) are repeated at various layers. Hence, both the exponential nature of the problem and the existence of only a polynomial number of subproblems are evident even to an average student.

As the next step, the class should be asked to transform the recursion tree into a DAG. The DAG (Figure 2) removes all unnecessary repetitions and gives a clear picture of the reduced amount of work that would have to be done using a dynamic programming approach. The drawings of the recursion tree (with exponential growth) and the DAG (with constrained growth) are indeed powerful tools for understanding the magic behind dynamic programming. The DAG essentially tells us about the various dependencies and naturally leads to the bottom-up dynamic programming algorithm. The triangular table of [3] is in fact a simplified version of the DAG where edges are removed and it is implied that a vertex on top will depend upon all the vertices below it. It is only a natural, and even mechanical, step to draw the triangular tables based on the DAG. It is important to note that a triangular matrix does not convey all the information contained in the DAG.

As the number of distinct subproblems is exactly equal to $n(n+1)/2$, which is only $O(n^2)$, we can use the bottom-up technique of dynamic programming to avoid the numerous recomputations of the top-down recursive formulation. This cuts
down the complexity to the number of edges in the DAG of Figure 2, which grows as \( \sum_{n=1}^{n} n(n-1) \). This sums to \( n(n^2-1) \), yielding an only \( O(n^3) \) solution. For example, the number of edges in Figure 2 is exactly 20.

**The Partition Problem**

We next look at another interesting and seemingly unrelated problem. Given a fixed sequence \( \{s_1, s_2, ..., s_n\} \) of \( n \) nonnegative numbers, we want to divide the set into \( p \) partitions such that the maximum sum over all the partitions is minimum.

We can find an optimal solution by exhaustive recursive search. If we let \( M_p[i..j] \) denote the optimal solution to dividing a sequence of numbers with indices from \( i \) to \( j \) for some \( i \leq j \leq n \) into \( p \) partitions, we can form the following recurrence:

\[
M_p[i..j] = \min_{i \leq k < j} \max(M_p[i..k], M_q[k+1..j])
\]

Below, we discuss two interesting cases of the above recurrence.

**Case 1:** We can set \( p-q = q \), which means \( q = p/2 \). The recurrence in (2) is reproduced below with this condition:

\[
M_p[i..j] = \min_{i \leq k < j} \max(M_p[i..k], M_p[k+1..j])
\]

Equation (3) represents the so-called binary solution. We put a divider in the middle of the sequence and try to optimally partition the subsequences formed to the left and right into \( p/2 \) partitions each. This process can be repeated recursively until the base case is reached. The recursion tree is shown in Figure 3 for a sequence of five numbers and four partitions. It is interesting to note that the recursion in (3) as well as the recursion tree for the partition problem are exactly the same as those for the matrix-chain multiplication problem ((1) and Figure 1), except for one difference: the recursion tree for matrix multiplication problem has a depth proportional to \( n \), while the depth of the partition problem’s tree is proportional to \( \log_2(n) \).

**Case 2:** If we set \( q = 1 \), which means \( p - q = p - 1 \), we get the following recurrence:

\[
M_p[i..j] = \min_{i \leq k < j} \max(M_{p-1}[i..k], M_1[k+1..j])
\]

It is easy to see that \( M_p[k+1..j] \) is the sum of the numbers with indices from \( k+1 \) to \( j \). This leads to the so-called incremental partitioning strategy as we form the \( p \)th partition by placing the last divider right after the \( k \)th element for some \( i \leq k < j \). The maximum sum over all the partitions would then be the larger of the sums to the left and right of this last divider. The sum to the right of this last divider, \( M[k+1..j] \), is simply \( \sum_{a=k+1}^{j} s_a \). The recursion tree for (4) is shown in Figure 5 for a sequence of six numbers and four partitions. Its DAG is shown in Figure 6.

**Conventional Approach**

Reference [2] includes the partition problem on pp.56-59. Its treatment of this problem has the same defects as those of [3] for the matrix multiplication problem. No recursion tree or DAG is drawn and there is no mention of the binary way of solving it. In addition, the dynamic programming rectangular tables that are shown (p.59) neither give any clue about the various dependencies that exist among their entries, nor are they any help in understanding the complexity of the algorithm. The common form of (1) to (4) is also not found as such in textbooks. There, the students are introduced to each problem in isolation and the common thread that runs through all is never explored.

**Our Approach**

We believe in showing the students at least two recursive formulations for some of the dynamic programming problems. With this experience, the students are able to pick the better recursive approach.

The directed acyclic graph of Figure 6 is essentially a rectangular table similar to the one given in [2]. In fact, the DAG also paints a vivid picture of the dependencies. For example, in order to calculate \( M_4[1..6] \) we need to calculate all the vertices connected to it. In the classroom, we also stress (1) to (4) encouraging students to write them on their own after they have understood the problems. Their facility in this indicates their level of understanding.
The recursion tree and DAG are visual tools that correctly depict the complexity of the problem: the number of edges in them is a measure of the complexity of the respective solution. For the binary solution (Case 1) of the partition problem, the number of vertices in the DAG in a typical layer in Figure 4 is $O(n^2)$, which makes the number of edges $O(n^3)$. The overall complexity thus turns out to be $O(n^3 \log p)$. For the incremental solution (Case 2), the number of edges per layer in Figure 6 is $O(n^2)$ and thus the overall complexity is $O(n^2 p)$. For comparison, the complexity of the recursive solution can be determined by counting the edges in the recursion tree of Figure 5, which grows as $1 + 2 + 4 + 7 + 11 + 16 + \ldots$, i.e. $O(n^2)$. The interesting thing to note here is that if we increase the number of partitions then the number of edges becomes the summation $1 + 2 + 4 + 8 + 16 + 32 + \ldots 2^n$, which is $O(2^n)$. Thus, the number of partitions to be formed drastically affects the complexity of the brute-force recursive solution.

**Layered Graph Approach of Researchers**

References [4] and [5] present a layered graph solution to the partitioning problem, which is inherently a dynamic programming solution. The only difference is that the DAG approach is derived in a formal manner after making a recursive formulation, whereas the layered graph approach seems like an elegant trick as it directly arrives at the layered graph. Moreover, it fails to model the binary version of recursive formulations and works only for the incremental one. Hence, the layered graph is just a special case of dynamic programming and does not always guarantee an efficient solution. Without following the steps that we prescribe, the researchers were forced to rely on their ingenuity to arrive at a solution, which, although being beautiful, is neither straightforward nor easily reproducible.

**Conclusions**

In this paper, we have proposed an enhanced methodology for teaching dynamic programming. We felt the need to do so after finding the treatment of popular textbooks inadequate. We have identified certain ingredients missing from the mix of teaching methods whose inclusion takes the magic out of dynamic programming. These are: the recursion tree, the directed acyclic graph, the different recursive formulations of the same problem, and the discovery of similarities and differences of the various problems chosen to demonstrate dynamic programming.

Reference [6] presents a comprehensive and detailed treatment of the topics discussed in this paper. Therein we also discuss the all-pairs shortest path problem and show that its binary recursive formulation is more efficient than the corresponding incremental formulation.

**Acknowledgement**

We wish to thank R. Khan, N. Malik, S. Mahkari, R. Jaferi, S. Rahman, F. Lodhi, and S. Abbasi for their help and encouragement. We also thank the Lahore University of Management Sciences for supporting this research.

**References**


FIGURES AND TABLES

FIGURE 1
RECURSION TREE FOR THE MATRIX-CHAIN MULTIPLICATION PROBLEM

FIGURE 2
DAG FOR THE MATRIX-CHAIN MULTIPLICATION PROBLEM
FIGURE 3
RECURSION TREE FOR THE BINARY SOLUTION OF THE PARTITION PROBLEM (CASE 1)

FIGURE 4
DAG FOR THE BINARY SOLUTION OF THE PARTITION PROBLEM (CASE 1)
FIGURE 5
RECURSION TREE FOR THE INCREMENTAL SOLUTION OF THE PARTITION PROBLEM (CASE 2)

FIGURE 6
DIRECTED ACYCLIC GRAPH FOR THE INCREMENTAL SOLUTION OF THE PARTITION PROBLEM (CASE 2)