Dynamic Programming

James Lee
jameslee@gwu.edu

Siddharth Chaubal
schaubal@gwmail.gwu.edu

Tariq Kamal
tariq@gwmail.gwu.edu

October 13, 2009

Abstract

Dynamic programming is a strategy for solving problems that can be viewed as a sequence of decisions where each choice can affect future decisions. In this case, selecting locally optimal choices may lead to a globally non-optimal sequence. These algorithms can efficiently enumerate all optimal sequences and then choose the best. This paper looks at three examples of how dynamic programming can be applied. The first, in a string function where the goal is to find the difference between two strings. The second, for determining order of matrix multiplications which minimizes the number of operations. The third, in game playing where you need to find the cheapest way across a checkerboard when each space has a different cost.

1 Introduction

When the greedy method is applied to problems, it breaks them down into subproblems and always selects the best choice for each subproblem to optimize some objective. For example, when filling a knapsack, it will always select as much of the most valuable good (with respect to weight) as it can at the moment without considering future or past decisions. Now let's modify the knapsack problem to say that you can either take an item or leave it. You can't take a little bit of something and a little bit of another. A greedy algorithm will again naively take as much of the valuable stuff as possible. But when it gets to the end and the knapsack is almost full, there might be more items left, but they don't fit. How do you know there isn't a better combination of items that more completely fills the knapsack and has more value as a result? The greedy algorithm will not know any better, and will return to you a non-optimal solution.

Dynamic programming is similar to the greedy method in that it also breaks problems into subproblems. The difference is that it chooses the best sequence of decisions to optimize some objective. Dynamic programming algorithms generate multiple decision sequences and select the best based on some metric. They remain efficient by not generating sequences that can't possibly be optimal.

Dynamic programming algorithms typically define a cost function which looks at all the choices that can be made for any particular subproblem and how those choices affect the overall cost. It will then usually try to make the choice which minimizes or maximizes cost, depending on the problem.

The term "dynamic programming" does not refer to computer programming, but rather mathematical programming or mathematical optimization.

2 String Editing By James Lee

Consider a spell-checker for a moment. When you misspell a word, it must look through a whole lexicon and find the closest word to what you were trying to spell. How does it determine when two strings are close to another? One way is to look at the *edit distance* between the two, that is, the number of operations it takes to transform one to the other, where an operation is the addition, deletion, or substitution of a single character. Each operation can also have an associated cost. For example, it might cost more to add and remove characters than to simply change them.

One approach to finding the edit distance might be to apply a greedy algorithm. A greedy algorithm would divide the problem into a series of decisions and always take the best at each step. In this case, it would start at the first character of the two strings and decide whether to add, delete, or change them, then continue to the next characters in the strings. This will give you a solution, but it will not be optimal. Suppose you wanted to see how many operations it would take to transform "abcde" to "bcde". The greedy algorithm would look at the first characters and change the 'a' to a 'b' assuming that operation cost the least. It would then proceed to change every other character in the string and delete the last one to match the length.

The problem is that locally optimal solutions aren't always globally optimal. In this case, it would be better to just delete the 'a' from the string, even if it had a slightly higher cost. Dynamic programming allows us to look at a sequence of operations, and then choose the best. This works on the principal of optimality which says that if you have an optimal—in this case minimum-cost—sequence that transformed string A to string B, and you undo the last operation leaving you with string A', then that subsequence is also a minimum-cost solution for transforming A' to B. Thus, eventually you'll work your way down to a sequence of one operation which will be optimal for beginning the transformation. It sounds recursive, and it is.

Let's consider a function $cost(i, j)$ which gives us the minimum cost for transforming the first i characters of string A into the first j characters of string B. Using the example above, if $i = 2$ and $j = 3$, then $cost(2, 3)$ would give us the minimum cost of transforming "ab" to "bcd".

If i and j are 0, then each string is empty and the cost of transforming an empty string to another empty string is 0. This is the base case.

If i is 0 and $j > 0$, then you can just add characters to A until it matches B. Intuitively, this cost is just j times the cost of insertions, I. But instead,

let's think of this recursively, so we insert the jth character of B into the jth position of A. Then the jth character is correct and the cost only needs to be calculated for the characters before it, thus $cost(0, j) = cost(0, j-1) + I(B_j)$.

Similarly, if $i > 0$ and $j = 0$, then you can just delete characters from A until it is empty, so $cost(i, 0) = cost(i-1, 0) + D(A_i)$, where $D(A_i)$ is the cost of deleting the A_i character.

If both i and j are greater than 0, then the decision is not so easy, but again, we're only looking at the ith and jth characters. If those characters are the same, then there is no need to apply an operation to them, so the total cost is simply the cost of transforming the characters before them. That would look like $cost(i, j) = cost(i-1, j-1)$.

Otherwise, when the characters are not the same, we could transform the first i characters of A into the first $j-1$ characters of B, then insert the jth character of B into the jth position of A. That would look like $cost(i, j) = cost(i, j-1) + I(B_j)$.

Likewise, we could transform the first $i-1$ characters of A into the first j characters of B, then delete the ith character from A. That would look like $cost(i, j) = cost(i-1, j) + D(A_i)$.

Finally, we could transform the first $i-1$ characters of A into the first $j-1$ characters of B and change A_i into B_j. That would look like $cost(i, j) = cost(i-1, j-1) + C(A_i, B_j)$, where $C(A_i, B_j)$ is the cost of changing A_i into B_j.

Those are the three options for when i and j are both greater than 0 and the characters at those positions are not equal. Simply choose the one with the least cost. Altogether, the cost function looks like:

$$cost(i, j) = \begin{cases} 0 & \text{if } i = 0, j = 0 \\ cost(0, j-1) + I(B_j) & \text{if } i = 0, j > 0 \\ cost(i-1, j) + D(A_i) & \text{if } i > 0, j = 0 \\ cost(i-1, j-1) & \text{if } i > 0, j > 0, A_i = B_j \\ cost'(i, j) & \text{otherwise} \end{cases}$$

where

$$cost'(i, j) = \min \left\{ \begin{array}{l} cost(i, j-1) + I(B_j), \\ cost(i-1, j) + D(A_i), \\ cost(i-1, j-1) + C(A_i, B_j) \end{array} \right\}$$

Now the minimum cost of transforming string A with length m to string B with length n can be calculated by $cost(m, n)$. Additionally, if the chosen operation (insertion, deletion, change, do nothing) is stored for each call to *cost*, then you can also derive the optimum sequence of operations to arrive at that minimum cost.

You may notice that calling $cost(m, n)$ will ultimately invoke call for every combination of $i < m$ and $j < n$, including many redundant calls. Efficiency can be gained by storing the results of each call to *cost* in a table, then checking

if the value has already been computed by looking it up in the table for each subsequent call to *cost*. This technique is called *memoization*.

In fact, because $cost(m, n)$ will ultimately call *cost* for every combination of i and j recursively, stack space can be saved by precomputing every value of *cost* iteratively, starting at the base case. Then, the last value in the table will be the final result. An algorithm for this looks like:

Algorithm STRING-EDIT-DISTANCE($A[1..m], B[1..n]$)

```
cost[0..m, 0..n]
for i ← 0 to m do
  for j ← 0 to n do
    if i = 0 and j = 0 then
      cost[0, 0] ← 0
    else if i = 0 and j > 0 then
      cost[0, j] ← cost[0, j − 1] + I(B[j])
    else if i > 0 and j = 0 then
      cost[i, 0] ← cost[i − 1, 0] + D(A[i])
    else if A[i] = B[j] then
      cost[i, j] ← cost[i − 1, j − 1]
    else
      cost[i, j] ← min ( cost[i, j − 1] + I(B[j]),
                         cost[i − 1, j] + D(A[i]),
                         cost[i − 1, j − 1] + C(A[i], B[j]) )
    end if
  end for
end for
return cost[m, n]
```

Let's see how this algorithm works with the original example of changing "abcde" to "bcde", assuming every operation has a cost of 1:

i	j	**Result**	**Operation**
0	0	0	None
0	1	$0+1=1$	I('b')
0	2	$1+1=2$	I('c')
0	3	$2+1=3$	I('d')
0	4	$3+1=4$	I('e')
1	0	$0+1=1$	D('a')
1	1	$\min(2,2,1)=1$	C('a', 'b')
1	2	$\min(2,3,2)=2$	I('c') or C('a', 'c')
1	3	$\min(3,4,3)=3$	I('d') or C('a', 'd')
1	4	$\min(4,5,4)=4$	I('e') or C('a', 'e')
2	0	$1+1=2$	D('b')
2	1	1	None
2	2	$\min(2,3,2)=2$	I('c') or C('b', 'c')
2	3	$\min(3,4,3)=3$	I('d') or C('b', 'd')
2	4	$\min(4,5,4)=4$	I('e') or C('b', 'e')
3	0	$2+1=3$	D('c")
3	1	$\min(4,2,3)$	D('c")
3	2	1	None
3	3	$\min(2,4,3)=2$	I('d')
3	4	$\min(3,5,4)=3$	I('e')
4	0	$3+1=4$	D('d')
4	1	$\min(5,3,4)=3$	D('d')
4	2	$\min(4,2,3)=2$	D('d')
4	3	1	None
4	4	$\min(2,4,3)=2$	I('e')
5	0	$4+1=5$	D('e')
5	1	$\min(6,4,5)=4$	D('e')
5	2	$\min(5,3,4)=3$	D('e')
5	3	$\min(4,2,3)=2$	D('e')
5	4	1	None

Note, this table is usually represented by a matrix like:

$i\backslash j$	**0**	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	1	1	2	3	4
2	2	1	2	3	4
3	3	2	1	2	3
4	4	3	2	1	2
5	5	4	3	2	1

The value in $i = 5, j = 4$ is the final result. Changing "abcde" to "bcde" takes 1 operation. To find out the sequence of operations, work your way back from the end, decrementing i and j based on the operation in the table until you get to 0:

i	j	**Operation**
5	4	None
4	3	None
3	2	None
2	1	None
1	0	D('a')
0	0	None

As expected, the only operation is to delete the 'a' at the first position.

Memoization makes this algorithm easy to analyze: it will always apply a constant time mathematical function for each entry in a table with dimensions of the length of the first string, m, by the length of the second string, n. In other words, $O(mn)$.

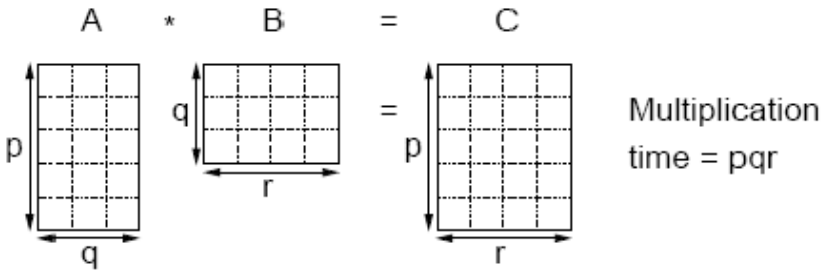
Many improvements could be made, though. For example, since *cost* only looks back at most one row, only the current and last row need to be stored. This improves space efficiency to $O(m)$.

You may also notice that the further away from the diagonal in the *cost* matrix you are, the higher the cost becomes. If you are only interested in seeing if two strings match within a certain threshold, k, you only need to calculate the values of *cost* within a stripe of width $2k + 1$ around the diagonal. Values outside that stripe can just be assigned an infinite cost. This improves the time efficiency to $O(kl)$ where l is the length of the shortest string.

3 Chain Matrix Multiplication By Siddharth Chaubal

Suppose that we wish to multiply a series of matrices $A_1 A_2 \dots A_n$. Our goal is to find the mots efficient way to do a "scalar" multiplication of these matrices together. Meaning that we are free to parenthesize the multiplication however we like without changing its order.

Example Calculate $M = A \times B \times C \times D$, where the dimensions of the matrices are $A : 10, 20 \quad B : 20, 50 \quad C : 50, 1 \quad D : 1, 100$.



Calculating $M = A \times (B \times (C \times D))$ requires 125,000 operations. Calculating $M = (A \times (B \times C) \times D)$ requires 2,200 operations.

So the problem is not to perform the multiplication, but merely to decide the order of the parentheses. Clearly the second method is more efficient. Now that we have identified the problem, how do we determine the optimal parenthesization of a product of n matrices?

Brute Force We could go through each parenthesis deciding which gives the minimum cost. This will take a long time for a large set of matrices. In general, it is $O(2^n)$.

Recursion We could find the cost of finding each of the possible combinations by making recursive calls to find the minimum cost to compute each possibility. We then choose the best one. But then we see it gives us the same time complexity as brute force, $O(2^n)$.

The problem is that we do a lot of redundancy while trying out all of the possibilities (even recursion doesn't do much). The solution is to omit the redundant work by using the answers we've already computed. For example, to find the minimum cost of $A \times B \times C$, we have to find the minimum cost of $A \times B$ as well. As this recursion grows more, the redundancy also grows more and more.

Dynamic Programming Approach Dynamic programming solves this problem. It is better because we solve the sub-problems only once, as opposed to many times with recursion.

To solve a problem using dynamic programming, the problem needs to fulfill certain properties. It should have an "optimal substructure" and "overlapping sub-problems."

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems.

Proof. Chain matrix multiplication exhibits optimal substructure. Consider a sequence of matrices to be represented by a binary (infix) tree where the leaves are the matrices and the internal nodes are intermediary products. Let T be the tree corresponding to the optimal way of multiplying $A_i \dots A_j$.

T has a left subtree L and a right subtree R. L corresponds to multiplying $B = A_i \dots A_k$ and R to multiplying $C = A_{k+1} \dots A_j$ for some integer $i \leq k \leq j-1$.

The cost corresponding to T is $cost(T) = cost(R) + cost(L) + cost(BC)$.

For the principal of optimality to hold, we need to show that L is the best tree of $A_i \dots A_k$ and R is the best tree for $A_{k+1} \dots A_j$. It suffices to show it for L.

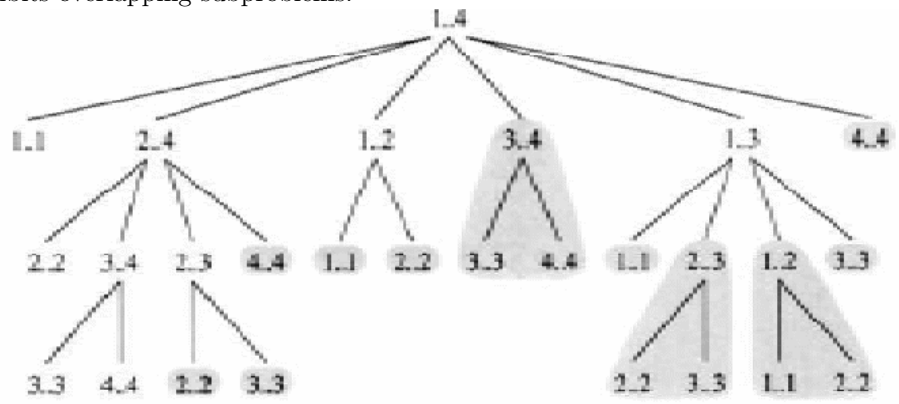
The proof is by contradiction. If L were not optimal, then there would be a better tree L' for $A_i \dots A_k$: $cost(L') < cost(L)$. Then, take the tree T' whose left subtree is L' and whose right tree is R:

$$\begin{aligned} cost(T') &= cost(L') + cost(R) + cost(BC) \\ &< cost(L) + cost(R) + cost(BC) = cost(T) \end{aligned}$$

Thus, $cost(T') < cost(T)$ which contradicts the assumption that T was the best tree for its matrices. Therefore, L must be optimal. □

Overlapping subproblems means that the space of subproblems must be small. That is, any recursive algorithm solving the problem should solve the same subproblems over and over, rather than generating new subproblems.

Proof. We will take an example to demonstrate that chain matrix multiplication exhibits overlapping subproblems.



The figure above illustrates the binary tree representation of the problem (as explained earlier). From this figure we can see that there are several overlapping sub-problems (shaded in gray). And as the number of matrices grows, the overlapping of its subproblems grows. □

Top-down Approach (Memoization) This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its subproblems, and if its subproblems are overlapping, then one can easily *memoize*, or store, the solutions to the subproblems in a table. Whenever we attempt to solve a new subproblem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the subproblem and add its solution to the table. A pseudocode example of this problem using this approach follows:

Algorithm MEMOIZED-MATRIX-CHAIN(p)
  INITIALIZATION()
  LOOKUP-CHAIN($p, 1, n$)

First we set all of the values $m[i, j] = \infty$ to indicate that they have not been computed yet.

Algorithm INITIALIZATION()
  for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to n **do**
      $m[i, j] = \infty$

```
    end for
  end for

Algorithm LOOKUP-CHAIN(p, i, j)
  if m[i, j] < ∞ then
    return m[i,j];
  end if
  if i = j then
    m[i, j] ← 0
  else
    for k ← i to j − 1 do
      q ←LOOKUP-CHAIN(p, i, k)+LOOKUP-
      CHAIN(p, k + 1, j) + p[i − 1] · p[k] · p[j]
      if q < m[i, j] then
        m[i, j] ← q
        return m[i, j]
      end if
    end for
  end if
```

Bottom-up Approach This is the more interesting case. Once we formulate the solution to a problem recursively in terms of its subproblems, we can try reformulating the problem in a bottom-up fashion: try solving the subproblems first and use their solutions to build on and arrive at solutions to bigger subproblems. This is usually done in a tabular form by iteratively generating solutions to bigger and bigger subproblems by using the solutions to small subproblems. The following pseudocode demonstrates this:

```
Algorithm MATRIX-CHAIN-ORDER(p)
  for i ← 1 to n do
    m[i, i] ← 0
  end for
  for l ← 2 to n do
    for i ← 1 to n − l + 1 do
      j ← i + l − 1
      m[i, j] ← min_{i≤k<j}(m[i, k] + m[k + 1, j] + p[i − 1] · p[k] · p[j])
    end for
  end for
```

The three nested for-loops in this algorithm create a time complexity of $O(n^3)$.

Conclusion Dynamic programming reduces the time complexity of the matrix multiplication problem from $O(2^n)$ to $O(n^3)$. Out of two approaches, bottom-up is better as it is more practical in the sense that it requires no recursion, no

testing if a value has already been computed, and we can save space by throwing away some of the subresults that are no longer needed.

There is also a new, even more efficient algorithm developed in 1984 by Hu and Shing which achieves a time complexity of $O(n \lg n)$ They showed how the matrix chain multiplication problem can be transformed into the problem of partitioning a convex polygon into non-intersecting triangles. But it is a more generalized approach to this problem.

4 Checkerboard By Tariq Kamal

A checkerboard is a collection of $m \times n$ squares as shown in the figure below. Each square is denoted by i, j where i is the row number and j is the column number.

	1	**2**	**3**	**4**	**5**	...	***n***
1	5	4	8	8	1	2	3
2	5	6	8	2	1	4	5
3	9	2	6	1	7	8	6
...	1	3	5	8	2	1	2
m	4	6	5	3	1	8	9

The task is to find the least cost path from the lowest rank (first row) to the last rank (last row). The rule is that the checker could only move diagonally down left, straight down, or diagonally down right.

Let $c(i, j)$ show the cost associated with the square i, j (where i is the row and j is the column). For example, in the figure above, $c(3, 5) = 7$.

The approach here would be to find the shortest path from each element of the first row to the last rank. As we said earlier, we can only move to three locations with each iteration. The decision to move to the next location is based on the minimum cost of the location. The following algorithm finds the minimum value out of the three possible moves (diagonally left, straight, or diagonally right):

Algorithm Minimum-Cost(i, j)
  if $i < 1$ or $i > m$ or $j < 1$ or $j > n$ **then**
    return undefined
  else
    return $\min \begin{pmatrix} \text{Minimum-Cost}(i-1, j-1), \\ \text{Minimum-Cost}(i-1, j), \\ \text{Minimum-Cost}(i-1, j+1) \end{pmatrix} + c(i, j)$
  end if

Now, to find the shortest path to the last rank, we find the shortest path from each square in row 1 to the last rank with the algorithm below. Then we print the total cost associated with the path:

Algorithm Shortest-Path()

```
p[] ← 0
for j ← 1 to n do
  p[j] ← c[1, j]
  for i ← 1 to m − 1 do
    a ←Minimum-Cost(q[i − 1, j + 1], q[1, j], q[i + 1, j + 1])
    p[i] ← p[i] + a
    if a = c[i − 1, j + 1] then
      path_array ← −1
    else if a = c[i − 1, j] then
      path_array ← 0
    else
      path_array ← 1
    end if
  end for
end for
index ← 1
min ← q[1]
for i ← 2 to n do
  if p[i] < min then
    index ← i
    min ← p[i]
  end if
end for
{Print minimum cost:}
print  min
{Print shortest path:}
print  "(i,1)"
for j ← 2 to m do
  print  "→"
  print  "i,"
  if path_array[i, j] = −1 then
    print  "j-1"
  else if path_array[i, j] = 0 then
    print  "j"
  else
    print  "j+1"
  end if
end for
```

5 Conclusion

Dynamic programming provides an efficient mechanism for solving problems that can be view as a series of overlapping subproblems. This technique exploits the principal of optimality which says that for a given optimal sequence, its subsequences are also optimal. Dynamic programming algorithms allow sub-

problems to make a poor local decision if it makes the overall sequence of decisions better. This overcomes the limitation of greedy algorithms, at the expense of additional computation time. The examples in this paper show typical dynamic programming algorithms, involving optimizing on a cost function to select the best sequence of decisions, and memoizing to reduce redundant calculations.

References

[1] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajaskeran. *Computer Algorithms*, chapter 5. Silicon Press, 2008.

[2] Wikipedia. Dynamic programming — wikipedia, the free encyclopedia, 2009. [Online; accessed 13-October-2009]. Available from: `http://en.wikipedia.org/w/index.php?title=Dynamic_programming&oldid=319601653`.

[3] Wikipedia. Levenshtein distance — wikipedia, the free encyclopedia, 2009. [Online; accessed 13-October-2009]. Available from: `http://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=315084019`.

[4] Wikipedia. Memoization — wikipedia, the free encyclopedia, 2009. [Online; accessed 13-October-2009]. Available from: `http://en.wikipedia.org/w/index.php?title=Memoization&oldid=318057681`.

[5] Wikipedia. Matrix chain multiplication — wikipedia, the free encyclopedia, 2009. [Online; accessed 13-October-2009]. Available from: `http://en.wikipedia.org/w/index.php?title=Matrix_chain_multiplication&oldid=315867706`.

[6] Dan Hirschberg. Dynamic programming, 2005. [Online; accessed 13-October-2009]. Available from: `http://www.ics.uci.edu/~dan/class/161/notes/6/Dynamic.html`.

[7] Abdou Youssef. Dynamic programming, 2006. [Online; accessed 13-October-2009]. Available from: `http://www.seas.gwu.edu/~ayoussef/cs212/dynamicprog.html`.

[8] Razvan Bunescu. Matrix chain multiplication, 2008. [Online; accessed 13-October-2009]. Available from: `http://ace.cs.ohiou.edu/~razvan/courses/cs404/lecture17.pdf`.