FIT2004: Supplementary notes for lecture 4

Dynamic Programming: Part II

DANIEL ANDERSON [1]

Now that we have been introduced to the powerful dynamic programming paradigm, we will take a look at a few classic problems that can be solved using it.

Summary: Dynamic Programming: Part II

In this lecture, we cover:

- The 0-1 knapsack problem (**The unlimited supply variant was covered in the lecture**)
- The matrix-chain multiplication problem
- The edit distance problem
- The space-saving trick

Recommended Resources: Dynamic Programming

- `http://users.monash.edu/~lloyd/tildeAlgDS/Dynamic/Edit/`
- `http://users.monash.edu/~lloyd/tildeAlgDS/Dynamic/`
- CLRS, Introduction to Algorithms, Chapter 15
- Weiss, Data Structures and Algorithm Analysis, Section 10.3
- Halim, Competitive Programming 3, Section 3.5

The 0-1 Knapsack Problem

NOTE: The unlimited supply knapsack problem was covered in the lecture. This is the 0-1 variant in which you can only take one copy of each item.

Suppose you find yourself in a room full of heavy but valuable items! You have with you a backpack that you can use to carry up to W kilograms of items. You have figured out for each item i how valuable it is p_i and how much it weighs w_i. Which items should you take in order to maximise the total value in your backpack?

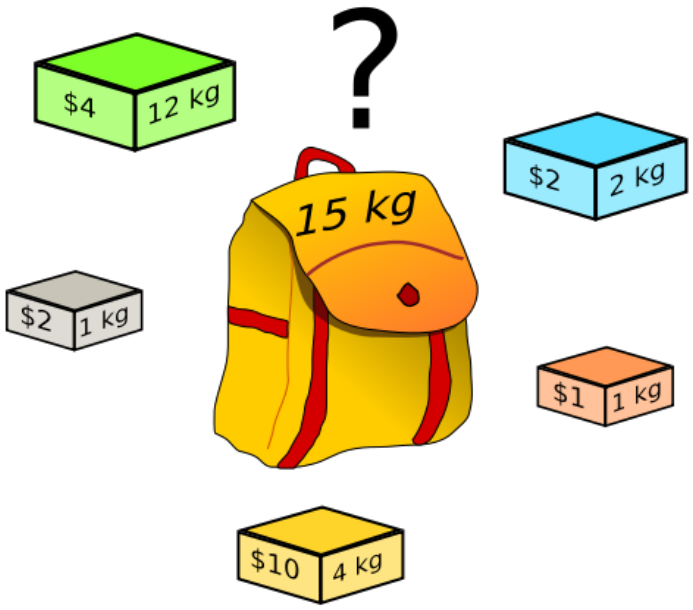
This problem is known in general as the 0-1 knapsack problem (0-1 because we can either take 0 of the item or 1 of it, there are other variants that permit us to take duplicate copies of items.) One method to solve the knapsack problem is to simply try all possibilities and see which one is the best feasible solution. If we have n items to choose from, then we would have to try all possible subsets of n items. The total number of subsets of a set of n items is 2^n, so trying all of them would lead to an exponential time complexity algorithm!

We can use dynamic programming to derive a much more efficient and elegant solution to the knapsack problem.

Identifying the optimal sub-problems

Consider an instance of the knapsack problem, for example, the one shown below.

[1] Faculty of Information Technology, Monash University: `daniel.anderson@monash.edu`. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.



An example instance of a knapsack problem. The optimal solution is to take all but the green box for a total value of \$15. Image taken from Wikimedia Commons[2].

The optimal solution for the five items shown given a backpack that carry a total weight of 15kg is to take the blue, silver, yellow and orange items for a total of \$15 value.

Suppose we are convinced that the optimal solution contains the blue item. Since it takes up precisely 2kg of the 15kg backpack, there is 13kg of space left for other items. We are then left with solving the sub-problem of finding the most value that we can accrue using 13kg of spacing using the remaining items. This is the optimal substructure of the knapsack problem. If we put an item weighing w_i into a backpack of size W kg, then we must fill the left over $W - w_i$ kgs optimally with the items that remain.

Deriving a recurrence

Consider an instance of the knapsack problem consisting of n items of weight w_i and value p_i for $1 \leq i \leq n$ and a backpack of weight W kg. Let us define $m_{i,j}$ for $0 \leq i \leq n$ and $0 \leq j \leq W$ to be the maximum possible value that we can fit in a backpack of weight j using choices from only the first i items.

The base cases are rather clear: we can take a maximum of \$0 value using the first zero items (no items at all) and can take a maximum of \$0 using a backpack that can hold 0kg. Thus

$$m_{0,0} = 0, \qquad m_{i,0} = 0, \qquad m_{0,j} = 0,$$

for $i \leq n$ and $j \leq W$. For the general case, if we are considering taking item i with j kg of space left in our backpack, then we have two choices:

1. We take item i and have $j - w_i$ kg of space left for the remaining items
2. We do not take item i and still have j kg of space for the remaining items

This leads to the general case recurrence for $m_{i,j}$ for $i, j \geq 1$,

$$m_{i,j} = \max \begin{cases} m_{i-1,j-w_i} + p_i & \text{if } w_i \leq j \\ m_{i-1,j} \end{cases} \quad \text{if } i,j > 0, \qquad m_{0,0} = 0, \qquad m_{i,0} = 0, \qquad m_{0,j} = 0,$$

for $i \leq n$, $j \leq W$. The value of the solution to the entire problem is then given by $m_{n,W}$.

[2]Taken from `https://en.wikipedia.org/wiki/File:Knapsack.svg` by Dake. Shared under a creative commons licence.

Implementation of the knapsack problem

Here is a bottom-up implementation of the dynamic programming algorithm corresponding to the recurrence derived above. Each row of the recurrence depends only on the previous row, so we will solve the sub-problems in row order. We will store the weights w_i in the array `weight[1..N]` and the values p_i in the array `value[1..N]`.

Algorithm: 0-1 Knapsack Problem

```
1: function KNAPSACK(weight[1..N], value[1..N], W)
2:     Set m[0..N][0..W] = [0,0,0,...]
3:     for i = 1 to N do
4:         for j = 1 to W do
5:             if weight[i] ≤ j then
6:                 m[i][j] = max(m[i-1][j], m[i-1][j-weight[i]] + value[i])
7:             else
8:                 m[i][j] = m[i-1][j]
9:             end if
10:        end for
11:    end for
12:    return m[N][W]
13: end function
```

Since there are $N \times W$ sub-problems and solving each one requires looking at no more than two previous sub-problems, the time and space complexities of this dynamic programming algorithm are $O(NW)$.

Matrix-Chain Multiplication

Recall the problem of multiplying two matrices A and B. If A is an $n \times m$ matrix and B is an $m \times o$ matrix, then their product will be an $n \times o$ matrix.

$$\underbrace{\begin{bmatrix} \\ A \\ \\ \end{bmatrix}}_{n \times m} \times \underbrace{\begin{bmatrix} \\ B \\ \\ \end{bmatrix}}_{m \times o} = \underbrace{\begin{bmatrix} \\ A \times B \\ \\ \end{bmatrix}}_{n \times o}$$

In general, the amount of work required to multiply an $n \times m$ matrix and an $m \times o$ matrix is $n \times m \times o$ scalar multiplications. Consider now the problem of multiplying a sequence of matrices of differing sizes. For example, suppose we want to compute the product of a 4×2, a 2×5 and a 5×3 matrix.

$$\underbrace{\begin{bmatrix} \\ A \\ \\ \end{bmatrix}}_{4 \times 2} \times \underbrace{\begin{bmatrix} \\ B \\ \end{bmatrix}}_{2 \times 5} \times \underbrace{\begin{bmatrix} \\ C \\ \\ \end{bmatrix}}_{5 \times 3}$$

There are several ways that we could compute this product owing to the fact that matrix multiplication is associative. This means that we can bracket the product in any way we want and we will still obtain the same answer. We can compute this product in either the order $(A \times B) \times C$ or $A \times (B \times C)$. Surprisingly, the order that choose to do the multiplications actually has a substantial effect on the amount of computational time required. If we choose the first order $(A \times B) \times C$, then we perform $(4 \times 2 \times 5) + (4 \times 5 \times 3) = 100$ scalar multiplications to obtain the answer. If we use the second order $A \times (B \times C)$, then we only perform $(2 \times 5 \times 3) + (4 \times 2 \times 3) = 54$ scalar multiplications. That's almost half the amount of work! The general matrix-chain multiplication problem is the problem of deciding for a sequence of n matrices, the best way to multiply them to minimise the number of scalar multiplications required.

For a sequence of four matrices, there are five different ways to parenthesise them as follows: $A \times ((B \times C) \times D)$, $A \times (B \times (C \times D))$, $(A \times B) \times (C \times D)$, $(((A \times B) \times C) \times D)$ and $(A \times (B \times C)) \times D$. In general, the number of ways to parenthesise a sequence of n matrices is exponential in n (it is related to the famous Catalan number sequence), so trying all of them would be way too slow. Dynamic programming can be used to find a fast solution to the matrix-chain multiplication problem.

Identifying the optimal sub-problems

Consider a sequence of n matrices $A_1, A_2, A_3, A_4, ...A_n$ to be multiplied (assume that they are of valid dimensions to be multiplied.) Think about the final multiplication that happens in the sequence. This multiplication must be the product of two matrices

$$\underbrace{(A_1 \times A_2 ... \times A_k)}_{\substack{\text{Result of multiplying} \\ \text{matrices 1 to } k}} \times \underbrace{(A_{k+1} \times A_{k+2} \times ... \times A_n)}_{\substack{\text{Result of multiplying} \\ \text{matrices } k+1 \text{ to } n}},$$

where $(A_1 \times A_2 ... \times A_k)$ and $(A_{k+1} \times A_{k+2} \times ... \times A_n)$ were multiplied earlier. If this is the optimal order in which to multiply the matrices, then the two sub-problems of multiplying $(A_1 \times A_2 ... \times A_k)$ and $(A_{k+1} \times A_{k+2} \times ... \times A_n)$ must also be multiplied in optimal order. If they were not, we could substitute them for a better order and obtain a better solution. This is the optimal substructure for the matrix-chain multiplication problem.

The dynamic programming solution for the matrix-chain multiplication therefore involves trying every possible "split point" k at which to perform the final multiplication. The problem is then recursively solved by asking for the optimal order in which to multiply matrices 1 to k and the remaining matrices $k+1$ to n.

Deriving a recurrence

Let us denote the number of columns in the matrices $A_1, A_2, ...A_n$ by the sequence $c_1, c_2, ..., c_n$. Recalling that two adjacent matrices can only be multiplied if the number of columns of the first is the same as the number of rows of the second, we know that if the sequence of matrices (A_i) is valid, that the number of rows in $A_2, ...A_n$ must be $c_1, ...c_{n-1}$. For completeness, we will further denote the number of rows of A_1 as c_0.

Let us denote by $m_{i,j}$ the optimal number of scalar multiplications required to multiply the sequence of matrices $A_i, A_{i+1}, ...A_j$. The base case for our recurrence occurs when the chain consists of a single matrix on its own, which requires no scalar multiplications, hence

$$m_{i,i} = 0,$$

for all $1 \leq i \leq n$.

For the general case, to find the optimal number of multiplications for the sequence $A_i, ...A_j$ where $i < j$, we must try every possible "split point" and recurse. If we perform the split after matrix k, where $i \leq k < j$, then the total number of required multiplications is $m_{i,k} + m_{k+1,j}$ plus the number of multiplications required to multiply $(A_i \times ... \times A_k)$ and $(A_{k+1} \times ... \times A_j)$. The matrix $(A_i \times ... \times A_k)$ has exactly c_{i-1} rows and c_k columns, and the matrix $(A_i \times ... \times A_k)$ and $(A_{k+1} \times ... \times A_j)$ has exactly c_k rows and c_j columns. Hence the number of scalar multiplications required to multiply them is $c_{i-1} \times c_k \times c_j$. Therefore the total number of required multiplications to multiply matrices i to j if we split after matrix k is given by

$$m_{i,j} = m_{i,k} + m_{k+1,j} + c_{i-1} \times c_k \times c_j$$

Combining everything together and trying all possible split points, we end up with the recurrence

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq k < j} (m_{i,k} + m_{k+1,j} + c_{i-1} c_k c_j) & \text{if } i < j. \end{cases}$$

for $i \leq j \leq n$. Note that we do not care about defining $m_{i,j}$ for $j > i$ since it does not make sense. With this recurrence, the answer to the entire problem is $m_{1,n}$, the optimal number of scalar multiplications required to multiply the entire sequence $A_1 \times ... \times A_n$.

Implementation of Matrix-Chain Multiplication

We will demonstrate a bottom-up implementation of the dynamic programming solution to matrix-chain multiplication. First, we must consider the order in which the sub-problems depend on each-other. For each sub-problem $m_{i,j}$, we need to have already computed the values of $m_{i,k}$ and $m_{k+1,j}$ for all $i \leq k < j$. The essential observation is that the sub-problems depend on sub-problems consisting of a shorter sequence of matrices. An effective order in which to compute the solutions is therefore in order of length, ie. we compute the answer for all contiguous subsequences of 2 matrices, then all contiguous subsequences of 3 matrices and so on.

Algorithm: Optimal Matrix-Chain Multiplication

```
1:  function MATRIX_ORDER(c[0..N])
2:      Set m[1..N][1..N] = [0,0,0,...]
3:      for i = 1 to N do
4:          m[i][i] = 0
5:      end for
6:      for length = 2 to N do
7:          for i = 1 to N - length + 1 do
8:              Set j = i + length - 1
9:              Set best = ∞
10:             for k = i to j - 1 do
11:                 best = min(best, m[i][k] + m[k+1][j] + c[i-1]*c[k]*c[j])
12:             end for
13:             m[i][j] = best
14:         end for
15:     end for
16:     return m[1][N]
17: end function
```

Since there are $N^2/2$ sub-problems and the computation of each sub-problem i, j requires looping over $j-i$ previous sub-problems, the time complexity of the dynamic programming solution for matrix-chain multiplication is

$$\begin{aligned}
\sum_{i=1}^{N}\sum_{j=i+1}^{N}(j-i) &= \sum_{i=1}^{N}\left(\sum_{j=i+1}^{N} j - \sum_{j=i+1}^{N} i\right) \\
&= \sum_{i=1}^{N}\sum_{j=i+1}^{N} j - \sum_{i=1}^{N}(N-i)i \\
&= \sum_{i=1}^{N}\left(\frac{N(N+1)}{2} - \frac{i(i+1)}{2}\right) - N\sum_{i=1}^{N} i + \sum_{i=1}^{N} i^2 \\
&= \frac{N^2(N+1)}{2} - \frac{1}{2}\left(\sum_{i=1}^{N} i^2 + \sum_{i=1}^{N} i\right) - \frac{N^2(N+1)}{2} + \sum_{i=1}^{N} i^2 \\
&= \frac{N(N+1)}{4} + \frac{1}{2}\sum_{i=1}^{N} i^2 \\
&= \frac{N(N+1)}{4} + \frac{N(N+1)(2N+1)}{12}
\end{aligned}$$

which is $O(N^3)$ and the space complexity is $O(N^2)$.

String Alignments – The Edit Distance Problem

The edit distance problem is a way to formally classify how similar or dissimilar two strings of text are. For example, the words "computer" and "commuter" are similar, as we can change one into the other by just modifying one letter, 'p' to 'm.' The words "sport" and "sort" are similar, as one can be changed into the other by deleting one letter from the first (or equivalently, inserting a new letter into the second).

This notion of the number of "edits" required to transform one string into another form the basis of the edit distance metric. A single edit operations consists of one of the following three operations:

1. Insert a new symbol anywhere in the string
2. Delete one of the symbols from the string
3. Replace one of the symbols in the string with any other symbol

Definition: Edit Distance

The edit distance between two strings is the minimum number of edit operations required to convert one of the strings into the other.

We can also define an optimal alignment of two strings which shows where the optimal number of insertions, deletions and substitutions occur. An optimal alignment for "computer" and "commuter" would be

c	o	m	**p**	u	t	e	r
c	o	m	**m**	u	t	e	r
			*				

which contains one edit operation (substitute 'p' for 'm.') An optimal alignment for the words "kitten" and "sitting" is given by

k	i	t	t	**e**	n	-
s	i	t	t	**i**	n	**g**
*				*		*

which contains two substitutions and one insertion, hence the edit distance between them in 3.

Identifying the optimal sub-problems

Consider the problem of finding an optimal alignment for the strings "ACAATCC" and "AGCATCG." Considering just the final column of the alignment, we have three choices:

C		C		-
G	or	-	or	G
*		*		*

The first choice corresponds to substituting the last 'C' for 'G.' The second choice corresponds to deleting the last 'C,' and the third choice corresponds to inserting the character 'G' at the end of the string. Once we have made a choice of one of these three options, we are then left to find an optimal alignment of all of the remaining characters.

In other words, the optimal alignment of "ACAATCC" and "AGCATCG" is one of:

$$\texttt{OPTIMAL\_ALIGNMENT}(\text{“ACAATC”}, \text{“AGCATC”}) + \begin{bmatrix} \texttt{C} \\ \texttt{G} \\ * \end{bmatrix}$$

$$\texttt{OPTIMAL\_ALIGNMENT}(\text{“ACAATC”}, \text{“AGCATCG”}) + \begin{bmatrix} \texttt{C} \\ \texttt{-} \\ * \end{bmatrix}$$

$$\texttt{OPTIMAL\_ALIGNMENT}(\text{“ACAATCC”}, \text{“AGCATC”}) + \begin{bmatrix} \texttt{-} \\ \texttt{G} \\ * \end{bmatrix}$$

The alignment of the remaining prefixes must be optimal because if it were not, we could substitute it for a more optimal one and achieve a better overall alignment.

Deriving a recurrence

Using the optimal sub-problems observed above, we can derive a recurrence for the edit distance between two strings $S_1[1..n]$ and $S_2[1..m]$ like so. Let $D_{i,j}$ for $0 \leq i \leq n$, $0 \leq j \leq m$ denote the edit distance between the prefixes $S_1[1..i]$ and $S_2[1..j]$. The edit distance between the empty string and a string of length i is i (we simply perform i insertions) so the base cases for our recurrence are:

$$D_{0,0} = 0, \qquad D_{i,0} = i, \qquad D_{0,j} = j,$$

for $i \leq n$ and $j \leq m$. For the general case, consider the computation of the sub-problem $D_{i,j}$, where $i, j > 0$. We have three choices:

1. Align $S_1[i]$ with $S_2[j]$, which has a cost of zero if $S_1[i] = S_2[j]$, or one otherwise.
2. Delete the character $S_1[i]$, which has a cost of one
3. Insert the character $S_2[j]$, which has a cost of one

If we choose to align $S_1[i]$ with $S_2[j]$, then the sub-problem that remains to be solved is to align the remaining prefixes ie. $D_{i-1,j-1}$. If we delete the character $S_1[i]$, then we need to align the remaining prefix of S_1 with S_2, ie. we are interested in the sub-problem $D_{i-1,j}$. Finally, if we chose to insert the character $S_2[j]$, then we are left to align the remaining prefix of S_2 with S_1, ie. we want the sub-problem $D_{i,j-1}$.

Trying all three possibilities and selecting the best one leads to the following general recurrence for $D_{i,j}$.

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} + 1_{S_1[i] \neq S_2[j]} \\ D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \end{cases} \qquad \text{if } i, j > 1, \qquad D_{0,0} = 0, \qquad D_{i,0} = i, \qquad D_{0,j} = j,$$

for $i \leq n$ and $j \leq m$ where $1_{S_1[i] \neq S_2[j]}$ is an indicator variable for $S_1[i] \neq S_2[j]$, in other words, it equals one if $S_1[i] \neq S_2[j]$, or zero otherwise. The optimal value of the entire problem is then given by $D_{n,m}$.

Implementation of edit distance

A bottom-up implementation of the dynamic programming solution for the edit distance problem might look like this. We note that the order in which we must compute the sub-problems corresponds to prefixes of increasing length since each sub-problem depends only on the sub-problems of one shorter length.

Algorithm: Edit Distance

```
1:  function EDIT_DISTANCE(S1[1..N], S2[1..M])
2:      Set D[0..N][0..M] = [0,0,0,...]
3:      for i=1 to N do
4:          D[i][0] = i
5:      end for
6:      for j = 1 to M do
7:          D[0][j] = j
8:      end for
9:      for i = 1 to N do
10:         for j = 1 to M do
11:             if S1[i] = S2[j] then
12:                 D[i][j] = D[i-1][j-1]
13:             else
14:                 D[i][j] = min(D[i-1][j-1], D[i-1][j], D[i][j-1]) + 1
15:             end if
16:         end for
17:     end for
18:     return D[N][M]
19: end function
```

There are a total of $N \times M$ sub-problems, and computing the solution to each one requires us to look back at just three previous ones, hence the time and space complexity of the edit distance algorithm is $O(NM)$.

Constructing the optimal alignment

In order to construct the alignment for the strings S_1 and S_2, we need to backtrack through the dynamic programming table from $D_{n,m}$ to $D_{0,0}$, examining the choices that were made along the way. The following function takes the filled dynamic programming table `D[0..N][0..M]` and produces the alignment `A1, A2`.

Algorithm: Alignment

```
1: function ALIGNMENT(S1[1..N], S2[1..M], D[0..N][0..M])
2:     Set A1 = "", A2 = ""
3:     Set i, j = N, M
4:     while i+j > 0 do
5:         if i = 0 then
6:             A1 += "-"
7:             A2 += S2[j]
8:             j -= 1
9:         else if j = 0 then
10:            A1 += S1[i]
11:            A2 += "-"
12:            i -= 1
13:        else
14:            Set best = min(D[i-1][j-1], D[i][j-1], D[i-1][j])
15:            if S1[i] = S2[j] or DP[i-1][j-1] = best then
16:                A1 += S1[i]
17:                A2 += S2[j]
18:                i -= 1, j -= 1
19:            else if DP[i-1][j] = best then
20:                A1 += S1[i]
21:                A2 += "-"
22:                i -= 1
23:            else
24:                A1 += "-"
25:                A2 += S2[j]
26:                j -= 1
27:            end if
28:        end if
29:    end while
30:    return reverse(A1), reverse(A2)
31: end function
```

Note that we build up the alignment in reverse and then reverse the strings at the end, since this saves us from expensively inserting characters at the beginning of the alignment strings.

The Space-Saving Trick

You may have noticed that the recurrences for the edit distance problem and the knapsack problem both had a very interesting property. Despite having a table of $N \times M$ sub-problems, each sub-problem only ever depended on sub-problems from the previous row in order to compute it's optimal solution. Because of this, we do not need to keep the entire table in memory all at once. It is enough to simply store two rows of the table at a time, and re-use the same memory for every second row of sub-problems that we compute. This reduces the space complexity of the edit distance and knapsack problems to $O(min(N, M))$ and $O(W)$ respectively. Of course, we have to remember that if we employ the space saving trick, we can no longer backtrack through the table to construct the optimal solution.