Dynamic Programming

Ruo Yu Tao, Sitong Chen

January 25, 2018

This is the augmented transcript of a lecture given by Luc Devroye on the 25th of January 2017 for a Data Structures and Algorithms class (COMP 252, McGill University). The subject was Dynamic Programming.

In dynamic programming, we build lookup tables that provide all sub-solutions necessary to find a solution of a problem. In our previous introductory lecture on dynamic programming, we computed the binomial coefficients, the partition of n into k sets and the travelling salesman tour. In today's lecture, additional examples are given.

Longest Common Subsequence Problem

Given the following two ordered sequences,

$$x_1, x_2, \ldots, x_n,$$
$$y_1, y_2, \ldots, y_m,$$

where each value is in the set of an alphabet (e.g., $\{0, 1\}$, $\{A, G, T\}$), find the longest common subsequence (figure 1).

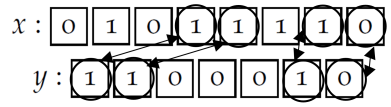


Figure 1: Example of a two sequences with longest common subsequence 1110.

We say that two subsequences $i_1, \ldots, i_\ell$ and $j_1, \ldots, j_\ell$ identify a common subsequence if

$$x_{i_1} = y_{j_1}, \ldots, x_{i_\ell} = y_{j_\ell}.$$

The largest such ℓ is the length of the longest common subsequence.

Solution

Definition 1. We define the matrix element $L[i, j]$ as the length of the longest common subsequence of $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$. This means that the last element in the matrix would be:

$$L[n, m] = \begin{cases} 1 + L[n-1, m-1], \text{ if } x_n = x_m, \\ \max(L[n-1, m], L[n, m-1]), \text{ if } x_n \neq x_m. \end{cases}$$

We can plot a table for all values in L (figure 2). In order to find the values in each cell, we can run the following algorithm to compute L.

$computeLCS(n, m)$

1 **for** all $i = 1$ **to** n
2 $\quad L[i, 0] = 0$
3 **for** all $j = 1$ **to** m
4 $\quad L[0, j] = 0$ // initialize all 0 index values
5 **for** all $i = 1$ **to** n
6 $\quad$ **for** all $j = 1$ **to** m
7 $$L[i,j] = \begin{cases} 1 + L[i-1, j-1] & \textbf{if} \quad x_i = x_j \\ \max(L[i-1,j], L[i,j-1]) & \textbf{if} \quad x_i \neq x_j \end{cases}$$

This matrix should always return the length of the Longest Common Subsequence.

We can now define an algorithm that takes in the matrix defined above and returns the longest common subsequence:

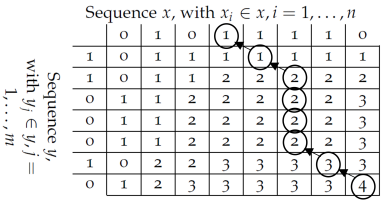


Figure 2: Example of two sequences with a longest common subsequence of length 4.

LCS

1 let $i = n$, $j = m$, $r =$ empty list for results.
// We start at the cell of the last row of the last column.
2 **while** $i \geq 0$ && $j \geq 0$ // We repeat this until we reach out of bounds of our matrix.
3 $\quad$ **if** $x_i = y_j$ // If the two elements $x_i = y_i$, we go North West (NW) one cell.
4 $\quad\quad$ append x_i to r.
5 $\quad\quad$ $i = i - 1, j = j - 1$.
6 **else** // Else, if $x_i \neq y_j$, then we choose the maximum between the number in
$\quad$ // the West cell and North cell.
7 $\quad$ **if** $L[i-1, j] > L[i, j-1]$
8 $\quad\quad$ $i = i - 1$.
9 $\quad$ **else** $j = j - 1$. // If the two values are equal, just go north.
10 **return** r

This algorithm is represented by the circles and arrows in Figure 2.
The construction of the matrix L takes time $\Theta(nm)$.

Knapsack Problem

We are given a knapsack which can hold a total integer capacity of k. We have items $x_1, x_2, \ldots, x_n$, all positive integers, that we want to add to our knapsack. We want to find a subset $S \subseteq \{1, \ldots, n\}$ such that

$$\sum_{i \in S} x_i = k.$$

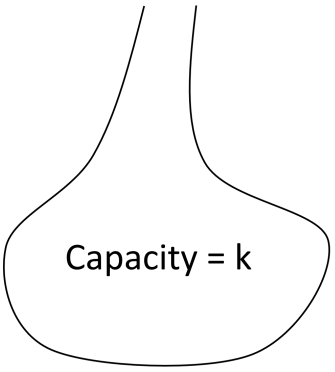


Figure 3: Our knapsack.

Solution

The knapsack problem exhibits optimal-substructure because an optimal solution to the problem contains within it optimal solutions to subproblems.[1]

Definition 2. If we have our n-th positive integer with k capacity remaining, we can build a matrix P that tells us whether or not there exists a solution with all the elements up to n and with a knapsack of capacity k. Formally, we define:

$$P[n,k] = \begin{cases} 1 & \text{if } \exists \text{ solution with a subset of } x_1, x_2, \ldots, x_n \text{ given capacity } k, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 3. We also define a matrix S where we grab the nth element with j capacity remaining. So S is a matrix that tells us if we should select a certain element or not:

$$S[n,k] = \begin{cases} 1 & \text{if we select item } n \text{ in the solution,} \\ 0 & \text{otherwise.} \end{cases}$$

We can write a dynamic program to calculate all values in the matrix P:

computeChoices

```
1  for all i = 1 to n
2      P[i,0] = 1, S[i,0] = 0 // initialize all 0 capacity knapsacks with
       // any element to 1, and don't select them.
3  for all j = 1 to k
4      P[0,j] = 0, S[0,j] = 0 // initialize all naught elements with
       // any capacity to 0, and don't select them.
5  P[0,0] = 1, S[0,0] = 1
6  for all i = 1 to n
7      for all j = 1 to k
8          if x_i > j // If element x_i is greater than remaining capacity
9              P[i,j] = P[i-1,j] // This cell will be set depending on
               // the existence of a solution in a subset of {1,...i-1}
10             S[i,j] = 0 // We do not select this element
11         elseif x_i == j
12             P[i,j] = 1 // There definitely exists a solution with x_i in it.
13             S[i,j] = 1 // We pick this to add to the solution.
14         elseif x_i < j
15             P[i,j] = max(P[i-1,j], P[i-1,j-x_i])
16             if P[i,j] == 0
17                 S[i,j] = 0
18             elseif P[i-1,j-x_i] == 1
19                 S[i,j] = 1
20             else S[i,j] = 0
```

The time taken by this algorithm is $O(kn)$.

Optimal Binary Search Tree

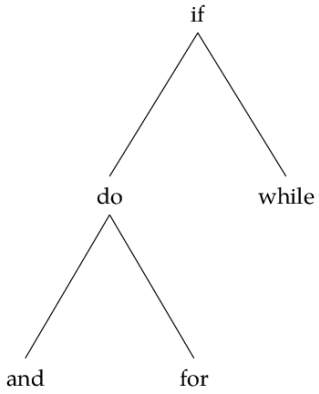
Background

Suppose that we are designing a compiler for a language, in which there are n syntactic keywords with corresponding semantics. For each occurrence of a keyword, we would want to perform a lookup operation by building a static binary search tree with n syntactic words as keys and their semantics as data stored in corresponding nodes. For the efficiency of the compiler, we would like to design a static binary search tree that minimizes total search time. [1]

We know that for a balanced tree, we can ensure an $O(\log n)$ search time per occurrence; however those syntactic words can appear with different frequencies. For example, if a frequently used word such as "if" is placed at the leave of this tree, it will greatly increase the total search time and hence the compiling time, vice versa. Therefore, given that we know the frequency of each key word appearing, we would like to organize a binary search tree in a way that minimizes the overall number of nodes visited. Such a tree is known as an optimal binary search tree. Moreover, it may be intuitive to consider a tree with smallest depth and key words of highest frequency at the root as an optimal binary search tree. However neither condition is necessary.[1]

[1]

Key words	Frequency(w)
and	w_1
do	w_2
for	w_3
if	w_4
while	w_5



Formally, let the sorted keys be $k_1 \dots k_n$. Their frequencies are denoted by $w_1 \dots w_n$. The algorithm is to place key k_i in a binary search tree at depth d_i such that

$\sum_{i=1}^{n} d_i w_i$

is minimal. Here $d_i = 1$ if k_i is the root.

Solution

Observe that any subtree of a binary search tree contains keys in a contiguous range $k_i \dots k_j$, for some $1 \leq i \leq j \leq n$. If an optimal binary search tree T has a subtree T' containing keys $k_i \dots k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i \dots k_j$.

Therefore, given a binary tree with keys $k_i \dots k_j$ and the root key k_r, where $i \leq r \leq j$, the left subtree contains keys $k_i \dots k_{r-1}$, while the right subtree contains keys $k_{r+1} \dots k_j$. If we check all possible candidate roots k_r, and identify the left and right subtree with minimum cost of searching, we are guaranteed to find an optimal binary search tree.

Let $C[i,j]$ denote the optimal cost for the tree containing k_i to k_j:

$C[i,j] = \sum_{k=i}^{j} w_k d_k$,

and let $W[i,j]$ denote the total frequency of all keywords between i and j:

$W[i,j] = \sum_{k=i}^{j} w_k$.

If k_r is the root of the optimal subtree for the tree containing k_i to k_j, we have:

$$C[i,j] = w_r + (C[i,r-1] + W[i,r-1]) + (C[r+1,j] + W[r+1,j]).$$

We also have:

$$W[i,j] = W[i,r-1] + W[r+1,j] + w_r.$$

We can rewrite $C[i,j]$ as:

$$C[i,j] = C[i,r-1] + C[r+1,j] + W[i,j].$$

Since we want to use the root that gives us the lowest cost of searching, we have the following recursive formula:

$$C[i,j] = \begin{cases} 0 & \text{, if } i < j, \\ w_i & \text{, if } i = j, \\ \min_{i \le r \le j} C[i,r-1] + C[r+1,j] + W[i,j] & \text{, if } i < j. \end{cases}$$

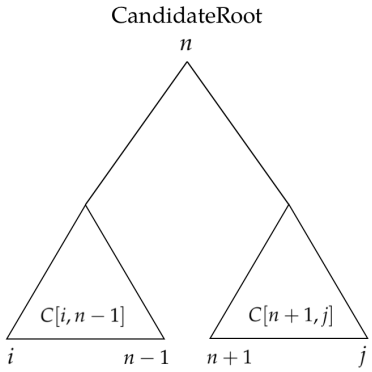
Moreover, based on the definition of $W[i,j]$, we have: [2]

$$W[i,j] = \begin{cases} w_i & \text{, if } i = j, \\ W[i,j-1] + w_i & \text{, if } i < j. \end{cases}$$

Finally, we also need to keep track of the roots of the optimal subtrees. Let $root[i,j]$ denote the index of the key that is the root of the optimal binary search tree for keys k_i up to k_j.

Our programs that compute W and C are called *computeFrequency*() and *computeCost*(), respectively. They have $O(n^2)$ and $O(n^3)$ worst-case complexity.

Remark. Knuth has shown that there are always roots of an optimal subtree such that $root[i,j-1] \le root[i,j] \le root[i+1,j]$ for all $1 \le i < j \le n$. Hence we can reduce the running time of computeCost() to $O(n^2)$ by replacing the innermost for loop `for r = i to j` with `for r = root[i,j-1] to root[i+1,j]`.[2]



[2]

The above is an tree view of our recursive formula for computing total cost of searching given a keyword is chosen as the root.

computeFrequency

```
1  for i = 1 to n
2      W[i,i] = w_i
3      for j = i + 1 to n
4          W[i,j] = W[i,j-1] + w_i
5  return W
```

computeCost

```
1  for i = 1 to n
2      C[i,i] = w_i
3  for sizeofSubtree = 2 to n
4      for i = 1 to n
5          j = i + sizeofSubtree - 1
6          if j ≤ n
7              C[i,j] = min_{i≤r≤j} C[i,r-1] + C[r+1,j] + W[i,j]
8              root[i,j] = one of the r's that minimizes C[i,j]
9  return C, root
```

References

[1] Thomas H. Cormen, Charles Eric. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Massachusetts Institute of Technology, 3rd edition, 2009.

[2] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1998.