Chapter 4:
Advanced Design and Analysis Techniques

2023 Oct 31

Nemo

Contents

1 Dynamic Programming

Dynamic programming applies when the subprobelms overlap. We typically apply dynamic programming to *optimization problems*.

When developing a dynamic programming algorithm, we follow a sequence of four steps:
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

1.1 Rod Cutting

The *rod cutting problem* is the following: Given a rod of length n inches and a table of prices p_i according to the length i, determine the maximum revenue r_n obtainable by cutting the rod and selling them.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

1.2 Matrix Chain Multiplication

The problem of matrix-chain multiplication is that given a sequence of n matrices to be multiplied, we wish to conpute the product $A_1A_2A_3...A_n$ at the minimum cost.

1.2.1 Applying Dynamic Programming

We follow the following four steps in order to apply dynamic programming:
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

1.2.1.1 Step 1: The structure of the optimal solution

First, for convenience we adopt the notation of $A_{i...j}$ for $A_iA_{i+1}...A_j$.

For a nontrival problem of MCM, to parenthesize the chain of matrices, we can think it backwards: Since we can the operation of multiplication is defined between two matrices, the last step of MCM must be a multiplication of two matrices. Then we can break down the problem into two subproblems of smaller scales, i.e. we can split the chain an a point between A_k and A_{K+1}. Then the same way, we can break down the problem recursively.

Since we can do break down the problem recursively, it's natural to think if the optimal solutions to the problems combined together is the solution to the original problem. i.e. We need to try to check out if the problem contians optimal structure, and the answer is yes. Proof: Suppose that the the chain is optimally parenthesized, then the two subchain divided between k and $k+1$ is also optimally parenthesized. Then, if the chain $A_{i...k}$ or $A_{k+1...j}$ is not optimally parenthesized, i.e. there exists a better way to parenthesize either of the chains. Since the way we parenthesize the matirx chain dose not affect the shape of the outcome matrix, then the cost of multiplying the result of the two product of the chain stays the same. Then there exists a better way to parenthesize the chain, contridict to the assumption that it is the optimal way to parenthesize the chian. So the problem contians optimal structure.

Proved above that we can construct the optimal solution, then we can solve the problem by solving the subproblems. Then we can move to the next step.

1.2.1.2 Step 2: A recursive solution

Now we define the cost of the terms of the optioml solption the subproblems. We let $m[i, j]$ be the cost of the cost of $A_{i...j}$, then for the full problem should be $m[i, j]$. According to the optimal structure, we should define the cost $m[i, j]$ as follows:

- When $i = j$, there is no multiplication, so $m[i, j] = 0$.
- When $i < j$, $m[i, j] = \min_{i \le k < j}\big(m[i, k] + m[k + 1, j] + p_{i-1}p_k p_j\big)$, where p is the colums of matrices.

1.2.1.3 Step 3: Computing the Opitmal Costs

We can foresee that since we have relatively few distince subprobelms than all the possibilities, we would enconter a subprobelms multiple times if we adopt a recursive method, i.e. subproblems overlap, together with the optimal structure, are two hallmarks of applying dynamic programming.

Instead of using an recursive way, we use a bottom-up tabular way to calculate the cost.

```
MATRIX-CHAIN-ORDER(p):
1  let n=p.length
2  let cost[i...n, 1...n] sep[1...n, 1...n] be new tables=0
3  for i from 1 to n:
4      for j from i+1 to n:
5          for k from i to j-1:
6              cost[i,j]=max(cost[i,j],cost[i,k]+cost[k+1,j]+p[i]p[k]p[j])
7              if cost[i,j] update: sep[i,j]=k
8  return cost,sep
```

1.3 Elements of Dynamic Programming

1.3.1 Opitmal Structure

1.3.2 Overlap Subproblems

1.3.3 Reconstruct the Optimal Solution

1.3.4 Memoization

1.4 Longest Common Subsequence

1.4.1 Step 1: Characterizing a Longest Common Substructure

***Theorem* 1: Optimal Substructure of an LCS**

Let $X = \langle x_1, x_2, ..., x_m\rangle$ and $Y = \langle y_1, y_2, ..., y_n\rangle$ be sequances, and let $Z = \langle z_1, z_2, ..., z_k\rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1}
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y
3. If $x_m \neq y_n$, then $z_k \neq y_m$ implies that Z is an LCS of Y_{m-1} and X

1.4.2 Step 2: A Recursive Solution

Let define $c[i, j]$ to be the length of an LCS of the sequences of X_i and Y_j. So, we have:

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

1.4.3 Step 3: Computing the length of an LCS

```
LCS_LENGTH(X, Y):
 1  m = X.length
 2  n = Y.length
 3  let b[1...m,1...n] and c[1...m,1...n] be new tables
 4  for i from 1 to m:
 5      c[i,0]=0
 6  for j from 1 to n:
 7      c[0,j]=0
 8  for i from 1 to m:
 9      for j from 1 to n:
10          if X[i]==Y[j]:
11              c[i,j]=c[i-1,j-1]+1
12              b[i,j]=↖
13          elseif c[i-1,j] >= c[i,j-1]:
14              c[i,j]=c[i-1,j]
15              b[i,j]=↑
16          else:
17              c[i,j]=c[i,j-1]
18              b[i,j]=←
19  return c and b
```

1.4.4 Step 4: Reconstruct an LCS

When b[i,j]=↖ add c[i,j] in the LCS.

1.5 Optimal Binary Search Tree

Binary search tree is a rooted binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree.

1.5.1 Step 1: Opitmal Substructure

Easy to prove.

1.5.2 Step 2: A recursive Solution

Let us define $e[i, j]$ be the search cost of an optimal BST contianing the nodes of $k_i, ..., k_j$ (contiguous). When constructing an optimal BST containing keys $k_i, ..., k_j$ from optimal subtrees, we need to choose one key as the root, then there exists a combination cost, which is all nodes of subtrees have their depth adding 1. So the combination cost is:

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

And so the cost is:

$$\begin{aligned} e[i,j] &= p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)) \\ &= e[i,r-1] + e[r+1,j] + w(i,j) \end{aligned}$$

So,

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min_{i \le r \le j}\{e[i,r-1] + e[r-1,j] + w(i,j)\} & \text{if } i \le j \end{cases}$$

1.5.3 Step 3: Computing the Expected Search Cost of an Opitmal BTS

pseudocode

2 Greedy Algorithms

3 Amortized Analysis

3.1 Aggregate Analysis

The basic idea of aggregate analysis is that each operation costs $\frac{T(n)}{n}$, where $T(n)$ is the worst cost of a sequence of n operations.

3.1.1 Stack Operations

Consider a sequence of n operations containing PUSH, POP, MULTIPOP, what is the *amortized cost*?

For the worst case, all the operations are MULTIPOP, and the worst case cost of MULTIPOP is $O(n)$. So the worst cost of the sequence is $O(n^2)$ and the amortized cost is $\frac{O(n^2)}{n} = O(n)$.

Is it correct? The answer is no. That's a quite rough estimate, there are better upper bounds. When calculating the amortized cost, we cannot ignore the inner relationship between the individual operations. In this case the time POP can be called is no more than the time PUSH is called. So $T(n)$ is at most $O(n)$, and the amortized cost is $\frac{O(n)}{n} = O(1)$.

3.1.2 Incrementing a Binary Counter

Assume the cost of incrementing a binary counter is in proportion to the bits filpped in an operation.

So by observiation, we know that the lowest bit is flipped n times in a sequence of n operations and the second is filpped $\frac{n}{2}$ times and etc. Then we can obtain the total cost is $\sum_{i=0}^{k-1}\left(\frac{n}{2^i}\right) < \sum_{i=0}^{\infty}\left(\frac{n}{2^i}\right) = 2n = O(n)$, and so the amortized cost is $\frac{O(n)}{n} = O(1)$

3.2 The Accounting Method

The accounting method is like this:

1. We assign amortized cost for each type of operation in the sequence
2. We take the difference of the assigned amortized cost and the actual cost as a *credit* and attach it to the data object.
3. If we can pay all the actual cost of the opeartions using the credit of the data object, then the amortized cost hold, and we can obtain the upper bound of the total actual cost.

3.3 The Potential Method

The potential method is somehow similiar to the accounting method, but we do not assign the exceeding part to the individual data objects, instand we attach the *potential* to the data structure as a whole and accumulate them in the *potential function*: we define $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$, where $\hat{c_i}$ is the

assigned amortized cost, the c_i is the acutal cost of an operation, $\Phi(D_i)$ is the potential funciton, inside the D_i can be understood as the i^{th} stage of the data structure.

What we need to do is to first assign the amortized cost and ensure that after the sequence of operations the potential function is non-negative.

3.4 Dynamic Tables