Algorithms and Data Structures for Mathematicians

Lecture 6: Dynamic Programming

Peter Kostolányi
`kostolanyi at fmph and so on`
Room M-258

2 November 2017

Dynamic Programming

Dynamic programming refers to:

- An approach to solving optimisation problems
- Programming in a mathematical sense, not computer programming
- Can be thought of as a paradigm for efficient algorithm design
- What follows is an "algorithm design viewpoint"
- A typical "optimisation viewpoint" is slightly different

Dynamic Programming vs. Divide and Conquer

Divide and conquer:

- Applies to problems with "recursive structure"
- That is: when a solution to a problem can be expressed in terms of solutions to its subproblems
- The solution is computed recursively in a top-down manner
- This may be inefficient in case of large overlaps between the subproblems...
- ...and particularly inefficient if some subproblems can have subproblems in common

Dynamic Programming vs. Divide and Conquer

Dynamic programming:

- Applies mostly to optimisation problems with "recursive structure"
- The optimum (value) for a problem has to be expressed recursively in terms of optima for its subproblems
- The optimum is computed in a bottom-up manner, while storing optima for subproblems, e.g., in some form of a table
- This avoids dealing with the same subproblems multiple times
- One can usually modify this approach to find a solution with optimal value as well

Matrix Chain Multiplication

- Suppose we are given matrices $A_1, \ldots, A_n$ such that A_i is of size $r_{i-1} \times r_i$ for $i = 1, \ldots, n$ and some $r_0, \ldots, r_n$ in $\mathbb{N}$
- We want to compute their product $A_1 A_2 \ldots A_n$
- This can be done by calling the "naive" matrix multiplication algorithm for pairs of matrices several times
- To multiply a pair of matrices of sizes $p \times q$ and $q \times r$, precisely $p \cdot q \cdot r$ scalar multiplications are needed
- The order, in which these pairs are chosen can influence the overall performance
- Which order minimises the total number of scalar multiplications?

Matrix Chain Multiplication

- The order of multiplying pairs of matrices is determined by making the matrix chain fully parenthesised

For example, consider matrices A_1, A_2, A_3 such that:

- A_1 is of size 2×500
- A_2 is of size 500×5
- A_3 is of size 5×1000

The first possible multiplication order is given by $A_1(A_2A_3)$:

- First compute A_2A_3 ($500 \cdot 5 \cdot 1000 = 2500000$ multiplications)
- The resulting matrix A_2A_3 is of size 500×1000
- Next compute $A_1(A_2A_3)$ ($2 \cdot 500 \cdot 1000 = 1000000$ multiplications)
- $2500000 + 1000000 = 3500000$ multiplications in total

The second order is given by $(A_1A_2)A_3$:

- First compute A_1A_2 ($2 \cdot 500 \cdot 5 = 5000$ multiplications)
- The resulting matrix A_1A_2 is of size 2×5
- Next compute $(A_1A_2)A_3$ ($2 \cdot 5 \cdot 1000 = 10000$ multiplications)
- $5000 + 10000 = 15000$ multiplications in total

Matrix Chain Multiplication

Given matrices $A_1, \ldots, A_n$ and their sizes, the optimum value is:

- The minimum number of scalar multiplications needed when multiplying the matrix chain according to some full parenthesisation

The optimal solution is:

- A full parenthesisation, for which the number of scalar multiplications needed is optimal

Checking all possible parenthesisations is inefficient

Matrix Chain Multiplication via Dynamic Programming

For each $1 \leq i \leq j \leq n$, consider the following subproblem:

- Compute the product $A_i \dots A_j$
- If $i = j$, we need no scalar multiplications
- If $i < j$, then for each parenthesisation there is k such that:
  - $i \leq k < j$
  - The last pair of matrices multiplied is $A_i \dots A_k$ and $A_{k+1} \dots A_j$
- Given such k, the following has to be done to multiply the chain:
  - Compute $A_i \dots A_k$
  - Compute $A_{k+1} \dots A_j$
  - Compute $(A_i \dots A_k)(A_{k+1} \dots A_j)$ in $r_{i-1} \cdot r_k \cdot r_j$ multiplications
- We are interested in k, for which the above three steps involve the least number of multiplications

Matrix Chain Multiplication via Dynamic Programming

- For $1 \leq i \leq j \leq n$, let $m_{i,j}$ be the least number of scalar multiplications needed for the matrix chain $A_i \dots A_j$
- The optimum value for the whole problem thus is $m_{1,n}$

The values $m_{i,j}$ satisfy the following recurrence:

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq k < j} (m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j) & \text{if } i < j \end{cases}$$

- We shall now compute the values $m_{i,j}$ using a bottom-up approach

Matrix Chain Multiplication via Dynamic Programming

OPTVALUE(n, r):

Input : Integer $n \geq 1$; Array $r = \langle r[0], \ldots, r[n] \rangle$ of positive integers such that A_i is of size $r[i-1] \times r[i]$ for $i = 1, \ldots, n$

Output: The minimal number of scalar multiplications needed for the matrix chain $A_1 \ldots A_n$

```
Let m[1...n, 1...n] be a new two-dimensional array;
for i ← 1 to n do m[i, i] ← 0;
for len ← 2 to n do
    for i ← 1 to n − len + 1 do
        j ← i + len − 1;
        m[i, j] ← ∞;
        for k ← i to j − 1 do
            num ← m[i, k] + m[k + 1, j] + r[i − 1] · r[k] · r[j];
            if num < m[i, j] then m[i, j] ← num;
        end
    end
end
return m[1, n];
```

Matrix Chain Multiplication via Dynamic Programming

- So far we have computed the optimal value
- This is of little use, since we do not know the optimal solution
- In other words: we want to find out how to multiply the matrix chain optimally
- We shall extend $\mathrm{OPTVALUE}(n, r)$ to store an information about selected values of k for given i, j into an auxiliary array $\mathsf{aux}[i, j]$
- We shall be able to reconstruct an optimal solution from aux

Matrix Chain Multiplication via Dynamic Programming

OPTVALUEAUX(n, r):

```
Let m[1...n, 1...n] and aux[1...n-1, 2...n] be new arrays;
for i ← 1 to n do m[i,i] ← 0;
for len ← 2 to n do
    for i ← 1 to n − len + 1 do
        j ← i + len − 1;
        m[i,j] ← ∞;
        for k ← i to j − 1 do
            num ← m[i,k] + m[k+1,j] + r[i−1] · r[k] · r[j];
            if num < m[i,j] then
                m[i,j] ← num;
                aux[i,j] ← k;
            end
        end
    end
end
return aux;
```

Matrix Chain Multiplication via Dynamic Programming

- Let aux be the output of $\mathrm{OPTVALUEAUX}(n, r)$
- The call $\mathrm{RECONSTRUCT}(\mathsf{aux}, 1, n)$ will print the optimal parenthesisation

$\mathrm{RECONSTRUCT}(a, i, j)$:

if $i = j$ **then**
  Print "A_i";
end
else
  Print "(";
  $\mathrm{RECONSTRUCT}(a, i, a[i, j])$;
  $\mathrm{RECONSTRUCT}(a, a[i, j] + 1, j)$;
  Print ")";
end

Optimal Binary Search Trees

- Let $(S, \preceq)$ be a totally ordered set
- Suppose we are given elements $x_1 \prec x_2 \prec \ldots \prec x_n$ of S
- We want to build a binary search tree containing $x_1, \ldots, x_n$
- We have a fixed probability distribution of queries

The following probabilities are known and do not change in time:

- For $i = 1, \ldots, n$, let p_i be a probability of searching for x_i
- Let q_0 be a probability for searching for some x such that $x \prec x_1$
- For $i = 1, \ldots, n-1$, let q_i be a probability of searching for some x such that $x_i \prec x \prec x_{i+1}$
- Let q_n be a probability of searching for some x such that $x_n \prec x$

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Optimal Binary Search Trees

- We may add keys $u_0, \ldots, u_n$ representing unsuccessful searches
- This can be viewed as if $u_0 \prec x_1 \prec u_1 \prec \ldots \prec u_{n-1} \prec x_n \prec u_n$
- The probability of "searching for u_i" is q_i for $i = 0, \ldots, n$
- We shall require the leaves of a tree to be precisely $u_0, \ldots, u_n$
- We thus add one level of nodes to make each query successful

For each x in $\{u_0, x_1, u_1, \ldots, u_{n-1}, x_n, u_n\}$ and each search tree T:

- Let $d_T(x)$ be a depth of x in T
- Assume that searching for x has cost $d_T(x) + 1$

The expected cost of searching a tree then is:

$$E_T = 1 + \sum_{i=1}^{n} p_i \cdot d_T(x_i) + \sum_{i=0}^{n} q_i \cdot d_T(u_i)$$

How to construct T so that E_T is minimised?

Optimal Binary Search Trees via Dynamic Programming

- Consider a subproblem of constructing an optimal tree $T_{i,j}$ for keys $u_{i-1} \prec x_i \prec u_i \prec \ldots \prec u_{j-1} \prec x_j \prec u_j$
- If $j = i - 1$, then $T_{i,j}$ contains a single node u_{i-1}, which is its root
- If $j \geq i$, then the root of $T_{i,j}$ is x_k for some k with $i \leq k \leq j$
  - The left subtree of x_k has to be equal to $T_{i,k-1}$
  - The right subtree of x_k has to be equal to $T_{k+1,j}$
- We are interested in k such that $E_{T_{\boldsymbol{i},\boldsymbol{k-1}}} + E_{T_{\boldsymbol{k+1},\boldsymbol{j}}}$ are minimised

Optimal Binary Search Trees via Dynamic Programming

For all i, j such that $1 \leq i, j \leq n$ and $j \geq i - 1$:

- Let us write $E_{i,j}$ instead of $E_{T_{i,j}}$
- Let $p_{i,j}$ be $\sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$

The values of $E_{i,j}$ are then given by the following recurrence:

$$E_{i,j} = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ \min\limits_{i \leq k \leq j} (E_{i,k-1} + E_{k+1,j} + p_{i,j}) & \text{if } j \geq i \end{cases}$$

- The values $E_{i,j}$ can be computed using a bottom-up approach
- The table $E[i,j]$ is filled in gradually for increasing $j - i$, starting at $j - i = -1$
- By recording the choices for k, it is possible to build the optimal tree