12 Dynamic Programming

21 January 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2025-01-21 01:41 H`

Learning Outcomes

Unit 12: ***Dynamic Programming***

1. Be able to apply the DP paradigm to solve new problems.

Outline

12.1 Elements of Dynamic Programming

Introduction

applicable to many problems

- *Dynamic Programming (DP)* is a powerful algorithm **design pattern**
  for exact solutions to **optimization** problems

- Some commonalities with Greedy Algorithms,
  but with an element of brute force added in

  DP = "careful brute force" (Erik Demaine)

- often yields polynomial time, but usually not linear time algorithms

- for many problems the *only* way we know to build efficient algorithms

- **Naming fun:** The term "dynamic programming", due to Richard Bellman from around 1953,
  does not refer to computer programming; rather to a program (= plan, schedule) changing with time.
  It seems to have been at least partly marketing babble devoid of technical meaning . . .

Plan of the Unit

1. Abstract steps of DP (briefly)
2. Details on a concrete example (*matrix chain multiplication*)
3. More examples!

6 Steps of Dynamic Programming

1. Define **subproblems** (and relate to original problem)
2. **Guess** (part of solution) $\rightsquigarrow$ local brute force
3. Set up **DP recurrence** (for quality of solution)
4. Recursive implementation with **Memoization**
5. Bottom-up **table filling** (topological sort of subproblem dependency graph)
6. **Backtracing** to reconstruct optimal solution

- Steps 1–3 require insight / creativity / intuition;
  Steps 4–6 are mostly automatic / same each time

$\rightsquigarrow$ Correctness proof usually at level of DP recurrence

running time too! worst case time = #subproblems · time to find single best guess

When does DP (not) help?

- *No Silver Bullet*
  DP is the most widely applicable design technique, but can't *always* be applied

1. Vitally important for DP to be correct:

   Bellman's Optimality Criterion

   For a *correctly guessed* fixed part of the solution,
   ***any* optimal solution to the corresponding subproblems**
   must yield an *optimal solution* to the overall problem (once combined).

   at most polynomial in n

2. Also, the total **number of different subproblems** should be "*small*"
   (DP potentially still works correctly otherwise, but won't be *efficient*.)

12.2 DP & Matrix Chain Multiplication

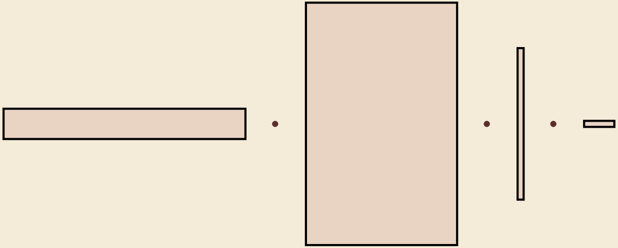
The Matrix-Chain Multiplication Problem

Consider the following exemplary problem

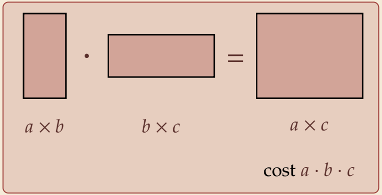
- We have a product $M_0 \cdot M_1 \cdot \dots \cdot M_{n-1}$ of n matrices to compute
- Since (matrix) multiplication is associative, it can be evaluated in different orders.
- For non-square matrices of different sizes, different order can change costs dramatically
  - Assume elementary matrix multiplication algorithm:
  - $\rightsquigarrow$ Multiplying $a \times b$-matrix with $b \times c$ matrix costs $a \cdot b \cdot c$ integer multiplications
- **Given:** Row and column counts $c[0..n)$ and $r[0..n)$ with $r[i+1] = c[i]$ for $i \in [0..n-1)$ (corresponding to matrices $M_0, \dots, M_{n-1}$ with $M_i \in \mathbb{R}^{r[i] \times c[i]}$)
- **Goal:** parenthesization of the product chain with minimal cost

really a binary tree with n leaves!

Matrix-Chain Multiplication – Example



M_0 10×80 · M_1 80×50 · M_2 50×2 · M_3 2×10



Parenthesization	Cost (integer multiplications)		
$M_0 \cdot \big(M_1 \cdot (M_2 \cdot M_3)\big)$	$1000 + 40\,000 + 8000$	$=$	$49\,000$
$M_0 \cdot \big((M_1 \cdot M_2) \cdot M_3\big)$	$8000 + 1600 + 8000$	$=$	$17\,600$
$(M_0 \cdot M_1) \cdot (M_2 \cdot M_3)$	$40\,000 + 1000 + 5000$	$=$	$46\,000$
$\big(M_0 \cdot (M_1 \cdot M_2)\big) \cdot M_3$	$8000 + 1600 + 200$	$=$	$9\,800$
$\big((M_0 \cdot M_1) \cdot M_2\big) \cdot M_3$	$40\,000 + 1000 + 200$	$=$	$41\,200$

first or last operation

Greedy fails both ways!

Matrix-Chain Multiplication – How about Brute Force?

If Greedy doesn't give optimal parenthesization, maybe just try all?

- parenthesizations for n matrices $=$ binary trees with n leaves
  $=$ binary trees with $n-1$ (internal) nodes
- How many such trees are there?
  - Let's write $m = n - 1$;
  - $C_0 = 1, C_1 = 1, C_2 = 2, C_3 = 5$
  - $C_m = \sum_{r=1}^{m} C_{r-1} \cdot C_{m-r} \qquad (m \geq 1)$

    generating functions / combinatorics / guess (OEIS!) & check ...
  - Can show $C_n = \dfrac{1}{n+1}\dbinom{2n}{n} \sim \dfrac{1}{\sqrt{\pi}} \cdot \dfrac{4^n}{n^{3/2}}$

  $\rightsquigarrow$ *exponentially many trees (almost 4^n)* $\qquad C_{20} = 6\,564\,120\,420, \; C_{30} = 3\,814\,986\,502\,092\,304$

$\rightsquigarrow$ A brute-force approach is utterly hopeless

$\rightsquigarrow$ *Dynamic programming to the rescue!*

Matrix-Chain Multiplication – Step 1: Subproblems

- Key ingredient for DP: Problem allows for recursive formulation
- Often requires to solve a more general problem
- Here: **Subproblems** = Ranges of matrices $[i..j)$ $\quad 0 \le i \le j \le n$
  i. e., optimal parenthesization
  for each range $M_i, M_{i+1}, \ldots, M_{j-1}$

$\rightsquigarrow$ **Original problem** = range $[0..n)$

- **Intuition:**
  - Any subtree in binary multiplication tree covers some range $[i..j)$
    (matrix multiplication is not commutative $\rightsquigarrow$ left-right order has to stay)
  - left and right factors of a multiplication don't "see/influence" each other

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

Matrix-Chain Multiplication – Step 2: Guess

- Usually, any subproblem can be split into smaller subproblems in different ways
- Which way to decompose gives best solution not known *a priori*

$\rightsquigarrow$ Assuming we can correctly *guess* this part; how to solve problem?

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

- Here: **Guess** last multiplication / root of binary tree

$\rightsquigarrow$ index $k \in [i+1\,..\,j)$ so that $[i..j)$ computed with **last** multiplication $(M_i \cdot \cdots \cdot M_{k-1}) \cdot (M_k \cdot \cdots \cdot M_{j-1})$

$\rightsquigarrow$ optimal parenthesization of $M_i, \ldots, M_{k-1}$ and $M_k, \ldots, M_{j-1}$ computed recursively (corresponds to subproblems $[i..k)$ and $[k..j)$)

Matrix-Chain Multiplication – Step 3: DP Recurrence

- With subproblems and guessed part fixed,
  we try to express total **value/cost of solution** *recursively*

⇝ *We ignore the actual solution and just compute its cost!*

- Often good to prove correctness at level of recurrence

> ***1.*** Subproblems
> ***2.*** Guess!
> ***3.*** DP Recurrence
> ***4.*** Memoization
> ***5.*** Table Filling
> ***6.*** Backtrace

- Here: **Recurrence** for $m[i,j]$ = total number of integer multiplications use in best parenthesization of $[i..j)$

⇝ Set up recurrence, including any base cases.

$$m[i,j] \;=\; \begin{cases} 0 & \text{if } j-i \le 1 \\ \underbrace{\min}_{\text{best } k \text{ chosen by } \textit{local brute force}} \left\{ \boxed{\underbrace{m[i,k]+m[k,j]}_{\text{recursive cost}} + \underbrace{r[i]\cdot r[k]\cdot c[j-1]}_{\text{cost of last multiplication}}} \;:\; k \in [i+1..j) \right\} & \text{otherwise} \end{cases}$$

Matrix-Chain Multiplication – Step 4: Memoization

- Write **recursive** function to compute recurrence
- But *memoize* all results!

⇝ First action of function: check if subproblem known
  - If so, return cached optimal cost
  - Otherwise, compute optimal cost and remember it!

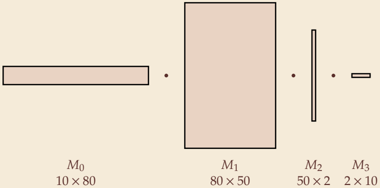
1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

$$m[i,j] \;=\; \begin{cases} 0 & \text{if } j-i \le 1 \\ \min\Big\{ m[i,k] + m[k,j] + r[i]\cdot r[k]\cdot c[j-1] \;:\; k \in [i+1..j) \Big\} & \text{otherwise} \end{cases}$$

```
procedure totalMults(r[i..j), c[i..j))
    if j − i ≤ 1
        return 0
    else
        best := +∞
        for k := i + 1, ..., j − 1
            m_l := cachedTotalMults(r[i..k), c[i..k))
            m_r := cachedTotalMults(r[k..j), c[k..j))
            m := m_l + m_r + r[i] · r[k] · c[j − 1]
            best := min{best, m}
        end for
        return best
```

```
procedure cachedTotalMults(r[i..j), c[i..j))
    // M[0..n)[0..n) initialized to NULL at start
    if M[i][j] == NULL
        M[i][j] := totalMults(r[i..j), c[i..j))
    return M[i, j]
```

Matrix-Chain Multiplication – Example Memoization



$n = 4$

$r[0..n) = [10, 80, 50, 2]$

$c[0..n) = [80, 50, 2, 10]$

$M[i][j]$

$i \backslash j$	0	1	2	3	4
0	0	0	40000	9600	9800
1	—	0	0	8000	9600
2	—	—	0	0	1000
3	—	—	—	0	0
4	—	—	—	—	0

Matrix-Chain Multiplication – Runtime Analyses

```
procedure totalMults(r[i..j), c[i..j))
    if j − i ≤ 1
        return 0
    else
        best := +∞
        for k := i + 1, . . . , j − 1
            m_l := cachedTotalMults(r[i..k), c[i..k))
            m_r := cachedTotalMults(r[k..j), c[k..j))
            m := m_l + m_r + r[i] · r[k] · c[j − 1]
            best := min{best, m}
        end for
        return best
```

```
procedure cachedTotalMults(r[i..j), c[i..j))
    // M[0..n)[0..n) initialized to NULL at start
    if M[i][j] == NULL
        M[i][j] := totalMults(r[i..j), c[i..j))
    return M[i, j]
```

- With memoization, compute each subproblem at most once
- nonrecursive cost (totalMults): $O(j - i) = O(n)$
- Number of subproblems $[i..j)$ for $0 \le i \le j \le n$

$$\sum_{0 \le i \le j \le n} 1 \;=\; \sum_{i=0}^{n} \sum_{j=i}^{n} 1 \;=\; \Theta(n^2)$$

$\rightsquigarrow$ total running time $\Theta(n^3)$

Matrix-Chain Multiplication – Step 5: Table Filling

- Recurrence induces a DAG on subproblems (who calls whom)
  - Memoized recurrence traverses this DAG
  - We can slightly improve performance by systematically computing subproblems following a fixed topological order
- **Topological order** here: by **increasing length** $\ell = j - i$, then i

```
procedure totalMultsBottomUp(r[0..n), c[0..n))
    M[0..n)[0..n) // M[i][j] stores m[i, j]
    for ℓ = 0, 1, ..., n // iterate over subproblems ...
        for i = 0, 1 ..., n // ... in topological order
            j := i + ℓ
            if ℓ ≤ 1
                M[i][j] := 0
            else
                M[i][j] := +∞
                for k := i + 1, ..., j − 1
                    m := M[i][k] + M[k][j] + r[i] · r[k] · c[j − 1]
                    M[i][j] := min{M[i][j], m}
    return M[0..n)[0..n)
```

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

- Same Θ-class as memoized recursive function
- In practice usually substantially faster
  - lower overhead
  - predictable memory accesses

Matrix-Chain Multiplication – Step 6: Backtracing

- So far, only determine the **cost** of an optimal solution
  - But we also want the solution itself!
- By *retracing* our steps, we can determine one
- Here: output a parenthesized term

```
procedure matrixChainMult(r[0..n), c[0..n))
    M[0..n)[0..n) := totalMultsBottomUp(r[0..n), c[0..n))
    return traceback([0..n))

procedure traceback([i..j))
    if j − i == 1
        return M_i
    else
        for k := i + 1, . . . , j − 1
            m := M[i][k] + M[k][j] + r[i] · r[k] · c[j − 1]
            if M[i][j] == m
                return (traceback([i..k))) · (traceback([k..j)))
        end for
    end if
```

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

- **backtracing** through M has at most the same complexity as **computing** M
- speedup possible by remembering correct guess k for each subproblem

12.3 Greedy as Special Case of DP

Dynamic Greedy

- Every Greedy Algorithm can also be seen as a DP algorithm **without guessing**

$\rightsquigarrow$ For new problems, it can help to first follow the DP roadmap and then check if we can select the "correct" guess without local brute force

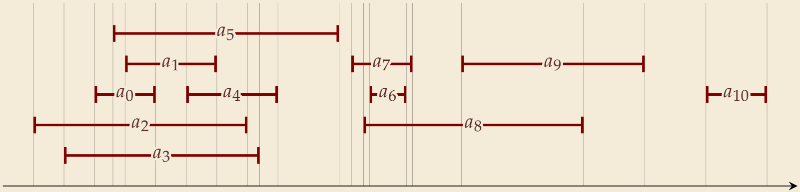
- If so, we then recurse on a single branch of subproblems

$\rightsquigarrow$ Greedy Algorithm doesn't need memoization or bottom-up table filling, but can do direct recursion instead

Recall Unit 11

The Activity selection problem

- **Activity Selection:** scheduling for *single* machine, jobs with *fixed* start and end times
  pick a *subset* of jobs without *conflicts*
  Formally:
  - **Given:** Activities $A = \{a_0, \ldots, a_{n-1}\}$, each with a start time s_i and finish time f_i $(0 \le s_i < f_i < \infty)$
  - **Goal:** Subset $I \subseteq [0..n)$ of tasks such that $i, j \in I \land i \neq j \implies [s_i, f_i) \cap [s_j, f_j) = \emptyset$
    and $|I|$ is maximal among all such subsets
  - We further assume that jobs are sorted by finish time, i. e., $f_0 \le f_1 \le \cdots \le f_{n-1}$
    (if not, easy to sort them in $O(n \log n)$ time)



31

DP Algorithm for Activity Selection

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

1. **Subproblems:** $A_{i,j} = \{a_\ell \in A : s_\ell \geq f_i \wedge f_\ell \leq s_j\}$
   (after a_i finishes and before a_j begins)

2. **Guess:** Task $k \in I^*$

3. **DP Recurrence:** Denote $c[i,j] = I^*(A_{i,j})$ = maximum #independent tasks in $A_{i,j}$

$$\rightsquigarrow \quad c[i,j] = \begin{cases} 0\,, & \text{if } A_{i,j} = \emptyset; \\ \max\{c[i,k] + c[k,j] + 1 : a_k \in A_{i,j}\} & \text{otherwise.} \end{cases}$$

4.–6. ***Omitted*** (can be done following the standard scheme)

4. Problem-specific insight from Unit 11 $\rightsquigarrow$ Can always use $k = \min\{k : a_k \in A_{ij}\}$
   (earliest finish time)

   No guess needed!

12.4 The Bellman-Ford Algorithm

Back to Shortest Paths!

- Consider again the single-source shortest path problem (SSSPP) on weighted digraphs
- We left open how to deal with negative-weight edges (in general graphs)!

Shortest Paths as DP

12.5 Making Change in pre 1971 UK

Pre-Decimal English Coins

Making Change by DP

Exact Knapsack Solution by DP

Pseudopolynomial Algorithms

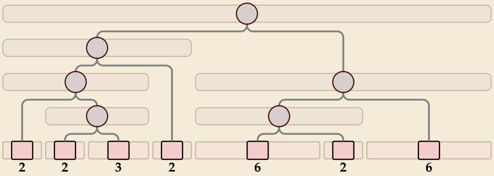
12.6 Optimal Merge Trees & Optimal BSTs

Recall Unit 4

Good merge orders

Let's take a step back and breathe.

- Conceptually, there are two tasks:
  1. Detect and use existing runs in the input $\rightsquigarrow$ $\ell_1, \ldots, \ell_r$ (easy)
  2. **Determine a favorable *order of merges* of runs** ("automatic" in top-down mergesort)



Merge cost = total area of
= total length of paths to all array entries

$$= \sum_{w \text{ leaf}} weight(w) \cdot depth(w)$$

$\rightsquigarrow$ *optimal* merge tree
= optimal *binary search tree*
for leaf weights $\ell_1, \ldots, \ell_r$
(optimal expected search cost)

well-understood problem
with known algorithms

29

Optimal Alphabetic Trees

Optimal Binary Search Trees

The Bisection Heuristic

12.7 Edit Distance

Edit Distance

Edit Distance Example

- x

Dynamic Programming – Summary