12 Dynamic Programming

21 January 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2025-01-27 15:55`

Learning Outcomes

Unit 12: ***Dynamic Programming***

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

Outline

12 Dynamic Programming

12.1 Elements of Dynamic Programming

Introduction

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

  applicable to many 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!

The 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).

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

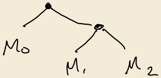
The Matrix-Chain Multiplication Problem

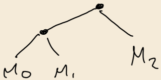
Consider the following exemplary problem

- We have a product $M_0 \cdot M_1 \cdot \cdots \cdot M_{n-1}$ 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 $\overset{r}{c}[0..n)$ and $\overset{c}{r}[0..n)$ with $r[i+1] = c[i]$ for $i \in [0..n-1)$
  (corresponding to matrices $M_0, \ldots, M_{n-1}$ with $M_i \in \mathbb{R}^{r[i] \times c[i]}$)
- **Goal:** parenthesization of the product chain with minimal cost

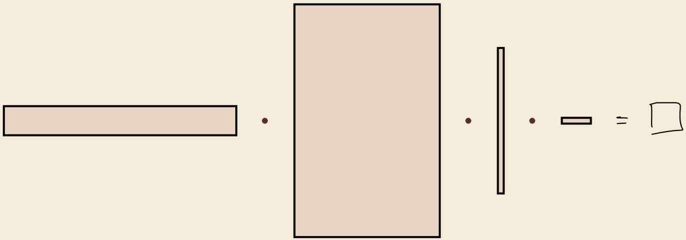
really a binary tree with n leaves!

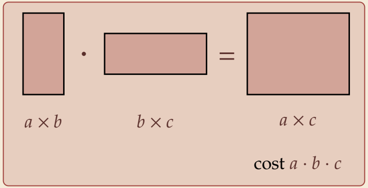
$(M_0 \cdot (M_1 \cdot M_2))$ $((M_0 \cdot M_1) \cdot M_2)$



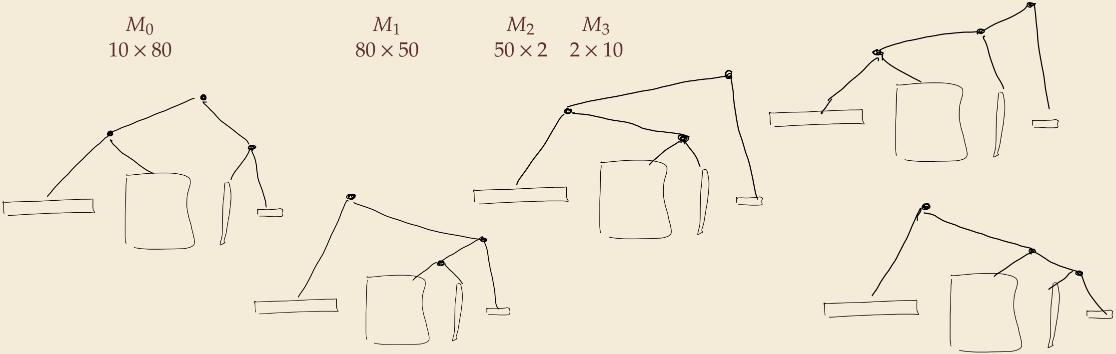


Matrix-Chain Multiplication – Example

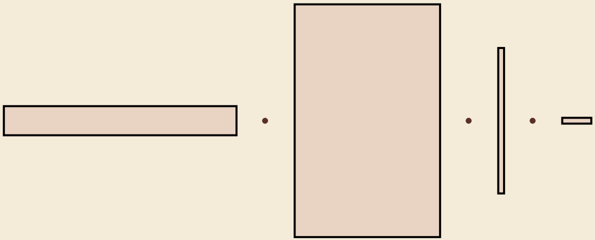


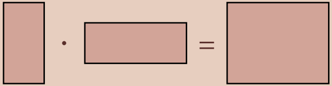


M_0	M_1	M_2	M_3
10×80	80×50	50×2	2×10



Matrix-Chain Multiplication – Example





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

first or last operation

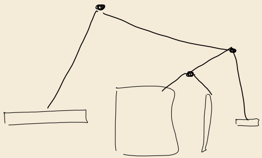
Greedy fails both ways!

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 (*evalution trees*)
  $=$ binary trees with $n-1$ (internal) nodes

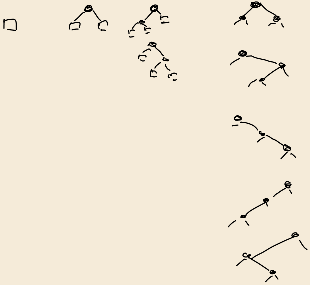
- How many such trees are there?



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 (*evalution trees*)
  $=$ binary trees with $\underbrace{n-1}_{m}$ (internal) nodes
- How many such trees are there?
  - Let's write $m = n - 1$;
  - $C_0 = 1$, $C_1 = 1$, $C_2 = 2$, $C_3 = 5$



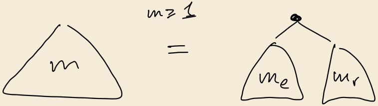
Matrix-Chain Multiplication – How about Brute Force?

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

- parenthesizations for n matrices = binary trees with n leaves (*evalution trees*)
  = binary trees with $n-1$ (internal) nodes

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

rank of root



$m_l + m_r = m - 1$

$0 \leq m_l, m_r \leq m - 1$

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 (*evalution trees*)
  $=$ 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
  Need to decide:
  1. What are the **subproblems** to consider?
  2. How can the **original problem** be expressed as subproblem(s)?

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

Matrix-Chain Multiplication – Step 1: Subproblems

- Key ingredient for DP: Problem allows for recursive formulation
  Need to decide:
  1. What are the **subproblems** to consider?
  2. How can the **original problem** be expressed as subproblem(s)?
- Often requires to solve a more general version of the problem

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

Matrix-Chain Multiplication – Step 1: Subproblems

- Key ingredient for DP: Problem allows for recursive formulation
  Need to decide:
  1. What are the **subproblems** to consider?
  2. How can the **original problem** be expressed as subproblem(s)?
- Often requires to solve a more general version of the problem

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

Here:

1. **Subproblems** = Ranges of matrices $[i..j)$ $\quad 0 \le i \le j \le n$
   i. e., optimal parenthesization for each range $M_i, M_{i+1}, \ldots, M_{j-1}$

Matrix-Chain Multiplication – Step 1: Subproblems

- Key ingredient for DP: Problem allows for recursive formulation
  Need to decide:
  1. What are the **subproblems** to consider?
  2. How can the **original problem** be expressed as subproblem(s)?
- Often requires to solve a more general version of the problem

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

Here:

1. **Subproblems** = Ranges of matrices $[i..j)$ $\quad 0 \le i \le j \le n$
   i. e., optimal parenthesization for each range $M_i, M_{i+1}, \ldots, M_{j-1}$
2. **Original problem** = range $[0..n)$

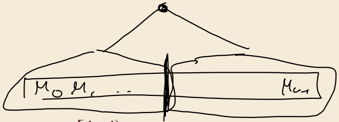
Matrix-Chain Multiplication – Step 1: Subproblems

- Key ingredient for DP: Problem allows for recursive formulation
  Need to decide:
  1. What are the **subproblems** to consider?
  2. How can the **original problem** be expressed as subproblem(s)?
- Often requires to solve a more general version of the problem

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

Here:

1. **Subproblems** = Ranges of matrices $[i..j)$ $\quad 0 \le i \le j \le n$
   i. e., optimal parenthesization for each range $M_i, M_{i+1}, \ldots, M_{j-1}$
2. **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

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

Matrix-Chain Multiplication – Step 2: Guess

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

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

⇝ Assuming we can correctly *guess* this part; how to solve problem?

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

⇝ index $k \in [i+1 \mathinner{..} j)$ so that $[i..j)$ computed with **last** multiplication $\underline{(M_i \cdot \dots \cdot M_{k-1})} \cdot \underline{(M_k \cdot \dots \cdot M_{j-1})}$

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

try all k!

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

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

subproblem [i..j)

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

⇝ Set up recurrence, including any base cases.

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

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

Matrix-Chain Multiplication – Correctness

Claim: Let $m(i,j)$ for $0 \le i \le j \le n$ be defined by the recurrence

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

Then $m(i,j)$ = #integer multiplications in best parenthesization of $M_i \cdots M_{j-1}$.

Proof:

Matrix-Chain Multiplication – Correctness

Claim: Let $m(i,j)$ for $0 \le i \le j \le n$ be defined by the recurrence

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

Then $m(i,j)$ = #integer multiplications in best parenthesization of $M_i \cdots M_{j-1}$.

Proof:

- **IB:** When $j - i \le 1$ we have an empty product ($j = i$) or a single matrix ($j = i + 1$)
  In both cases, no multiplications are needed and $m(i,j) = 0$. ✓

Matrix-Chain Multiplication – Correctness

Claim: Let $m(i,j)$ for $0 \leq i \leq j \leq n$ be defined by the recurrence

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

Then $m(i,j) \;=\;$ #integer multiplications in best parenthesization of $M_i \cdots M_{j-1}$.

Proof:

- **IB:** When $j - i \leq 1$ we have an empty product ($j = i$) or a single matrix ($j = i + 1$)
  In both cases, no multiplications are needed and $m(i,j) = 0$.
- **IS:** Given $j - i \geq 2$ matrices and an optimal evalution tree T for them.
  - T's root must be a last product of left and right subterms $(M_i \cdots M_{k-1}) \cdot (M_k \cdots M_{j-1})$ for some $i < k < j$, with cost $r[i]r[k]c[j-1]$.

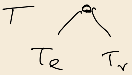
Matrix-Chain Multiplication – Correctness

Claim: Let $m(i,j)$ for $0 \le i \le j \le n$ be defined by the recurrence

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

Then $m(i,j) \;=\;$ #integer multiplications in best parenthesization of $M_i \cdots M_{j-1}$.

Proof:



- **IB:** When $j - i \le 1$ we have an empty product ($j = i$) or a single matrix ($j = i + 1$)
  In both cases, no multiplications are needed and $m(i,j) = 0$.
- **IS:** Given $j - i \le 2$ matrices and an optimal evalution tree T for them.
  - T's root must be a last product of left and right subterms $(M_i \cdots M_{k-1}) \cdot (M_k \cdots M_{j-1})$ for some $i < k < j$, with cost $r[i]r[k]c[j-1]$.
  - Moreover, left and right subtree T_ℓ and T_r of the root must be optimal evaluation trees for subproblems $[i..k)$ and $[k..j)$; (otherwise can improve T)

Matrix-Chain Multiplication – Correctness

Claim: Let $m(i,j)$ for $0 \le i \le j \le n$ be defined by the recurrence

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

Then $m(i,j) \;=\;$ #integer multiplications in best parenthesization of $M_i \cdots M_{j-1}$.

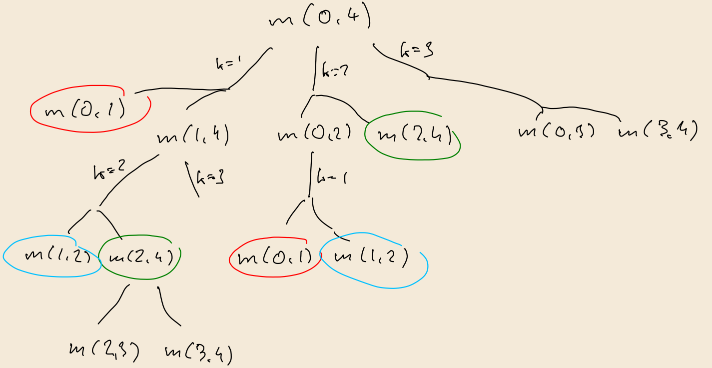
Proof: Induction over $j-i$

- **IB:** When $j-i \le 1$ we have an empty product ($j=i$) or a single matrix ($j=i+1$)
  In both cases, no multiplications are needed and $m(i,j)=0$.
- **IS:** Given $j-i \le 2$ matrices and an optimal evalution tree T for them.
  - T's root must be a last product of left and right subterms $(M_i \cdots M_{k-1}) \cdot (M_k \cdots M_{j-1})$ for some $i<k<j$, with cost $r[i]r[k]c[j-1]$.
  - Moreover, left and right subtree T_ℓ and T_r of the root must be optimal evaluation trees for subproblems $[i..k)$ and $[k..j)$; (otherwise can improve T)

  $\rightsquigarrow$ By IH, the cost of T_ℓ and T_r are given by $m(i,k)$ and $m(k,j)$

  $\rightsquigarrow$ $m(i,j) =$ cost of T $\square$

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



Matrix-Chain Multiplication – Step 4: Memoization

- Write **recursive** function to compute recurrence
- But *memoize* all results! (symbol table: subproblem $\mapsto$ optimal cost)

$\rightsquigarrow$ 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

Matrix-Chain Multiplication – Step 4: Memoization

- Write **recursive** function to compute recurrence
- But *memoize* all results! (symbol table: subproblem $\mapsto$ optimal cost)

$\rightsquigarrow$ 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

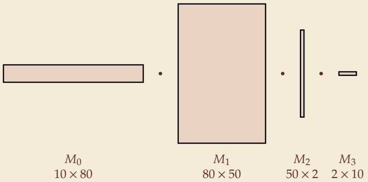
implements recurrence

```
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
```

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

```
procedure 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



$n = 4$

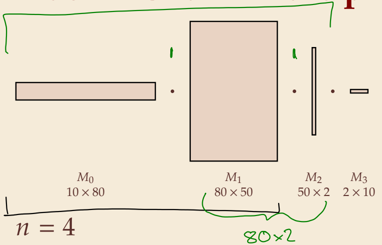
$r[0..n) = [10, 80, 50, 2]$

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

$m[i][j]$

$i \backslash j$	0	1	2	3	4
0	0	0			
1	—	0	0		
2	—	—	0	0	
3	—	—	—	0	0
4	—	—	—	—	0

Matrix-Chain Multiplication – Example Memoization



$n = 4$

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

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

$m(0,2) = 10 \cdot 80 \cdot 50 = 40000$

$m(0,3) = \min\{\overset{1600}{10 \cdot 80 \cdot 2} + \underset{8000}{m(1,3)},\ \overset{40000}{m(0,2)} + 10 \cdot 50 \cdot 2\}$

traceback(0,4) $k=3$

(traceback(0,3)) · (traceback(3,4))

$k=1$ $\quad$ $= M_3$

(traceback(0,1)) (traceback(1,3))

$= M_0$

(1,2) (2,3)

$= M_1$ $\quad$ M_2

$m[i][j]$

$i \backslash j$	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

$= ((M_0) \cdot ((M_1) \cdot (M_2))) \cdot (M_3)$

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
```

$\rightsquigarrow$ total running time $\overset{O}{\cancel{\Theta}}(n^3)$

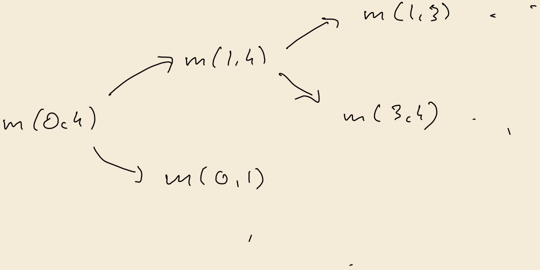
```
procedure cachedTotalMults(r[i..j), c[i..j)):
    // m[0..n)[0..n) initialized to NULL at start
    if m[i][j] == NULL
        m[i][j] := totalMults(r[i..j), c[i..j))
    return m[i, j]
```

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

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

Matrix-Chain Multiplication – Step 5: Table Filling

- Recurrence induces a DAG on subproblems (who calls whom)
  - Memoized recurrence traverses this DAG (DFS!)
  - We can slightly improve performance by systematically computing subproblems following a fixed topological order



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

Matrix-Chain Multiplication – Step 5: Table Filling

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

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

$$\underbrace{m(0,4)}_{\ell=4} > \underbrace{m(1,4) > m(0,3)}_{\ell=3} > \underbrace{m(2,4) > m(1,3) > m(0,2)}_{\ell=2} > \dots$$

Matrix-Chain Multiplication – Step 5: Table Filling

- Recurrence induces a DAG on subproblems (who calls whom)
  - Memoized recurrence traverses this DAG (DFS!)
  - We can slightly improve performance by systematically computing subproblems following a fixed topological order

- **Topological order** here: by **increasing length** $\ell = j - i$, then by i

```
procedure totalMultsBottomUp(r[0..n), c[0..n)):
    m[0..n)[0..n) := 0 // initialize to 0          m[i][j] = m(i,j)
    for ℓ = 2, 3, ..., n // iterate over subproblems ...
        for i = 0, 1 ..., n − ℓ // ... in topological order
            j := i + ℓ
            m[i][j] := +∞
            for k := i + 1, ..., j − 1
                q := m[i][k] + m[k][j] + r[i] · r[k] · c[j − 1]
                m[i][j] := min{m[i][j], q}
    return m[0..n)[0..n)
```

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

Matrix-Chain Multiplication – Step 5: Table Filling

- Recurrence induces a DAG on subproblems (who calls whom)
  - Memoized recurrence traverses this DAG (DFS!)
  - We can slightly improve performance by systematically computing subproblems following a fixed topological order

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

- **Topological order** here: by **increasing length** $\ell = j - i$, then by i

```
procedure totalMultsBottomUp(r[0..n), c[0..n)):
    m[0..n)[0..n) := 0 // initialize to 0
    for ℓ = 2, 3, ..., n // iterate over subproblems ...
        for i = 0, 1 ..., n − ℓ // ... in topological order
            j := i + ℓ
            m[i][j] := +∞
            for k := i + 1, ..., j − 1
                q := m[i][k] + m[k][j] + r[i] · r[k] · c[j − 1]
                m[i][j] := min{m[i][j], q}
    return m[0..n)[0..n)
```

- 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/construct one!
- Here: output a parenthesized term recursively

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

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/construct one!
- Here: output a parenthesized term recursively

```
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
            q := m[i][k] + m[k][j] + r[i] · r[k] · c[j − 1]
            if m[i][j] == q
                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

- follow recurrence a second time

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/construct one!
- Here: output a parenthesized term recursively

```
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
            q := m[i][k] + m[k][j] + r[i] · r[k] · c[j − 1]
            if m[i][j] == q
                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

- follow recurrence a second time
- always have for running time: backtracing $= O(\text{computing } M)$

$\leadsto$ computing optimal cost and computing optimal solution have same complexity

- speedup possible by remembering correct guess k for each subproblem

Summary: The 6 Steps of Dynamic Programming

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

1. Define **subprojects** and how **original problem** is solved
2. What part of solution to **guess**?
3. Set up **DP recurrence** for quality/cost of solution
   - ⇝ Prove **correctness** here: induction over subproblems following recurrence
   - ⇝ Analyze running **time complexity** here: #subproblems · non-recursive time

— (Basically) cookie-cutter approach from here on —

4. Recursive implementation with **Memoization**: mutually recursive functions with cache

or

5. Bottom-up **table filling**: define topological order of subproblem dependency graph
6. **Backtracing** to reconstruct optimal solution: Recursively retrace cost recurrence

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

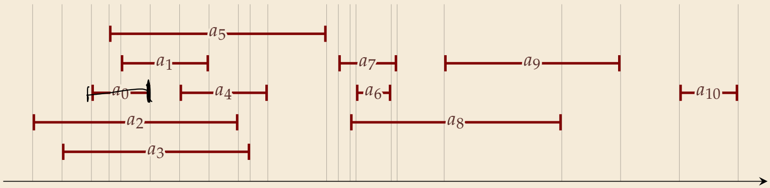
Dynamic Greedy

- Every Greedy Algorithm can also be seen as a DP algorithm **without guessing**
- ⇝ 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
- ⇝ 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 $i, j \in I \wedge i \ne j \implies [s_i, f_i) \cap [s_j, f_j) = \emptyset$
    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)

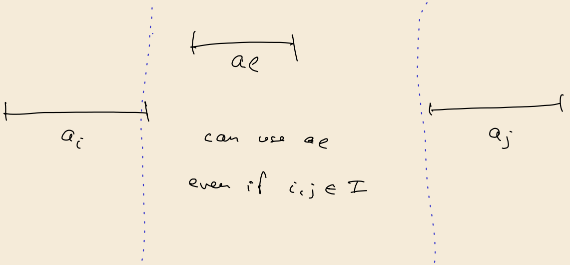


31

DP Algorithm for Activity Selection

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

   Original problem: $A_{-1,n}$ with dummy tasks $\cancel{s}f_{-1} = -\infty$, $\cancel{f}s_n = +\infty$



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

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)
   Original problem: $A_{-1,n}$ with dummy tasks $s_{-1} = -\infty$, $f_n = +\infty$

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

DP Algorithm for Activity Selection

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

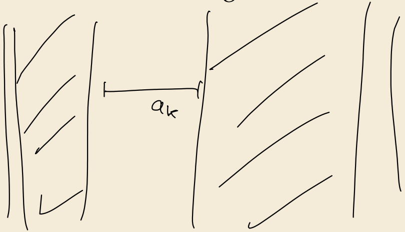
1. Subproblems: $A_{i,j} = \{a_\ell \in A : s_\ell \ge f_i \wedge f_\ell \le s_j\}$
(after a_i finishes and before a_j begins)
Original problem: $A_{-1,n}$ with dummy tasks $s_{-1} = -\infty$, $f_n = +\infty$

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

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

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

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



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)
   Original problem: $A_{-1,n}$ with dummy tasks $s_{-1} = -\infty$, $f_n = +\infty$

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

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

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

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

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

  No guess needed!

12.4 The Bellman-Ford Algorithm

Recall Shortest Paths

- **Single Source Shortest Path Problem (SSSPP)**
  - **Given:** directed, edge-weighted, simple graph $G = (V, E, c)$
    with edge costs $c : E \to \mathbb{R}$, a start vertex $s \in V$
  - **Goal:** a data structure that reports for every $v \in V$:
    $\delta_G(s, v)$: the shortest-path distance from s to v
    $\texttt{spath}(v)$: a shortest path from s to v (if it exists)
- $\delta_G(s, v) = \boxed{\inf\big(\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk in } G\}\big)}$
  - Write δ instead of δ_G when graph clear from context

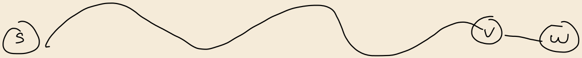
Recall Shortest Paths

- **Single Source Shortest Path Problem (SSSPP)**
  - **Given:** directed, edge-weighted, simple graph $G = (V, E, c)$
    with edge costs $c : E \to \mathbb{R}$, a start vertex $s \in V$
  - **Goal:** a data structure that reports for every $v \in V$:
    $\delta_G(s, v)$: the shortest-path distance from s to v
    $\mathtt{spath}(v)$: a shortest path from s to v (if it exists)
- $\delta_G(s, v) \;=\; \boxed{\inf\left(\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk in } G\}\right)}$
  - Write δ instead of δ_G when graph clear from context
- Here: Assume **negative-weight edges** are present (otherwise Dijkstra suffices)
  - but for now: assume there is **no negative cycle**
  - $\rightsquigarrow$ $\delta(s, v) > -\infty$ and can restrict to shortest **paths** (not walks)

"Pfad" "Weg"

Shortest Paths as DP – Last Edge Decomposition

- Idea: Every nontrivial shortest path has a **last edge**. *We don't know which; so guess!*



Shortest Paths as DP – Last Edge Decomposition

- Idea: Every nontrivial shortest path has a **last edge**. *We don't know which; so guess!*
  - ⇝ Subproblems: for $w \in V$, compute $\delta(s, w)$.
  - ⇝ Recurrence: $\delta(s, w) = \min\{\delta(s, v) + c(vw) : vw \in E\}$ $\delta(s,s) = 0$

Clicker Question

What is the problem with basing a DP algorithm on:
Subproblems: for $w \in V$, compute $\delta(s, w)$.
Recurrence: $\delta(s, w) = \min\{\delta(s, v) + c(vw) : vw \in E\}$

- A Bellman's Optimality Criterion is not satisfied.
- B Does not yield to an efficient algorithm: too many subproblems.
- C Does not yield to an efficient algorithm: non-recursive cost too high.
- D Subproblem dependency graph is cyclic.
- E Subproblem dependency graph is not connected.
- F Does not always compute correct distances.

→ sli.do/cs566

Clicker Question

What is the problem with basing a DP algorithm on:
Subproblems: for $w \in V$, compute $\delta(s, w)$.
Recurrence: $\delta(s, w) = \min\{\delta(s, v) + c(vw) : vw \in E\}$

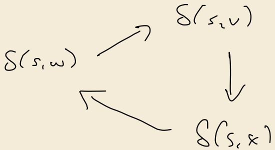
- A ~~Bellman's Optimality Criterion is not satisfied.~~
- B ~~Does not yield to an efficient algorithm: too many subproblems.~~
- C ~~Does not yield to an efficient algorithm: non-recursive cost too high.~~
- D Subproblem dependency graph is cyclic. ✓
- E ~~Subproblem dependency graph is not connected.~~
- F ~~Does not always compute correct distances.~~

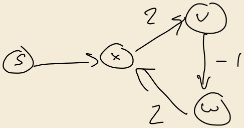
→ sli.do/cs566

Shortest Paths as DP – Last Edge Decomposition

- Idea: Every nontrivial shortest path has a **last edge**. *We don't know which; so guess!*
  - ⇝ Subproblems: for $w \in V$, compute $\delta(s, w)$.
  - ⇝ Recurrence: $\delta(s, w) = \min\{\delta(s, v) + c(vw) : vw \in E\}$

subproblem dependency graph is isomorphic to G^T! ⇝ doesn't work in general
  - ⇝ Yields usable (terminating!) algorithm *iff* G is a DAG.





Shortest Paths as DP – Last Edge Decomposition

- Idea: Every nontrivial shortest path has a **last edge**. *We don't know which; so guess!*
  - ⇝ Subproblems: for $w \in V$, compute $\delta(s, w)$.
  - ⇝ Recurrence: $\delta(s, w) = \min\{\delta(s, v) + c(vw) : vw \in E\}$

subproblem dependency graph is isomorphic to G^T! ⇝ doesn't work in general

- ⇝ Yields usable (terminating!) algorithm *iff* G is a DAG.

To break the cycles, let's turn them into a helix!

- Need to build "layers" in the subproblem dependency graph, so that edges can't go back up.

Shortest Paths as DP – Last Edge Decomposition

- Idea: Every nontrivial shortest path has a **last edge**. *We don't know which; so guess!*

  ⇝ Subproblems: for $w \in V$, compute $\delta(s,w)$.

  ⇝ Recurrence: $\delta(s,w) = \min\bigl\{\delta(s,v) + c(vw) : vw \in E\bigr\}$

  subproblem dependency graph is isomorphic to G^T! ⇝ doesn't work in general

  ⇝ Yields usable (terminating!) algorithm *iff* G is a DAG.

To break the cycles, let's turn them into a helix!

- Need to build "layers" in the subproblem dependency graph, so that edges can't go back up.
- **Subproblems:** (w,ℓ) for $w \in V$, $\ell \in [0..n)$, compute $\delta_{\le \ell}(s,w)$
  where $\delta_{\le \ell}(s,v) = \min\bigl(\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk with } \le \ell \text{ edges}\}\bigr)$
- **Original problems:** $\ell = n - 1$ (without negative cycles, paths suffice)

Shortest Paths as DP – Last Edge Decomposition

- Idea: Every nontrivial shortest path has a **last edge**. *We don't know which; so guess!*

  $\rightsquigarrow$ Subproblems: for $w \in V$, compute $\delta(s, w)$.

  $\rightsquigarrow$ Recurrence: $\delta(s, w) = \min\{\delta(s, v) + c(vw) : vw \in E\}$

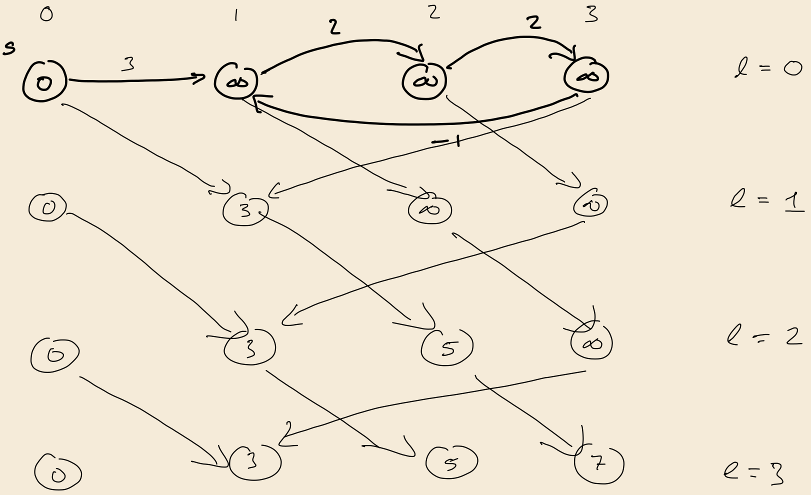
subproblem dependency graph is isomorphic to G^T! $\rightsquigarrow$ doesn't work in general

$\rightsquigarrow$ Yields usable (terminating!) algorithm *iff* G is a DAG.

To break the cycles, let's turn them into a helix!

- Need to build "layers" in the subproblem dependency graph, so that edges can't go back up.
- **Subproblems:** (w, ℓ) for $w \in V$, $\ell \in [0..n)$, compute $\delta_{\le \ell}(s, w)$
  where $\delta_{\le \ell}(s, v) = \min(\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk with } \boldsymbol{\le \ell} \text{ edges}\})$
- **Original problems:** $\ell = n - 1$ (without negative cycles, paths suffice)
- **Recurrence:** $$\delta_{\le \ell}(s, w) = \begin{cases} \infty & \text{if } \ell = 0 \text{ and } s \neq w \\ 0 & \text{if } \ell = 0 \text{ and } s = w \\ \min\{\delta_{\le \ell-1}(s, v) + c(vw) : vw \in E\} & \text{otherwise} \end{cases}$$

Shortest Paths as DP – Length Layers



Hold On – What about negative cycles?

- The recurrence for $\delta_{\leq \ell}$ seems to work fine with *negative* edges

  But G could contain a **negative-weight cycle** C ...

$$\delta_{\leq \ell}(s,w) = \begin{cases} \infty & \text{if } \ell = 0 \text{ and } s \neq w \\ 0 & \text{if } \ell = 0 \text{ and } s = w \\ \min\left\{\delta_{\leq \ell-1}(s,v) + c(vw) : vw \in E\right\} & \text{otherwise} \end{cases}$$

Isn't that a contradiction to the non-existence of shortest paths?

Hold On – What about negative cycles?

- The recurrence for $\delta_{\le\ell}$ seems to work fine with *negative* edges

  But G could contain a **negative-weight cycle** C ...

$$\delta_{\le\ell}(s,w) = \begin{cases} \infty & \text{if } \ell = 0 \text{ and } s \neq w \\ 0 & \text{if } \ell = 0 \text{ and } s = w \\ \min\left\{\delta_{\le\ell-1}(s,v) + c(vw) : vw \in E\right\} & \text{otherwise} \end{cases}$$

Isn't that a contradiction to the non-existence of shortest paths?

- No. If we restrict the length, shortest walks always exist.
- But: If there is a negative cycle $C[0..k]$ with paths $s \rightsquigarrow C$ and $C \rightsquigarrow w$, then $\delta_{\le\ell}(s,w) > \delta_{\le\ell+k}(s,w) > \delta_{\le\ell+2k}(s,w) > \cdots$ (and $\delta(s,w) = -\infty$)

Hold On – What about negative cycles?

- The recurrence for $\delta_{\leq \ell}$ seems to work fine with *negative* edges

  But G could contain a **negative-weight cycle** C …

$$\delta_{\leq \ell}(s,w) = \begin{cases} \infty & \text{if } \ell = 0 \text{ and } s \neq w \\ 0 & \text{if } \ell = 0 \text{ and } s = w \\ \min\left\{\delta_{\leq \ell-1}(s,v) + c(vw) : vw \in E\right\} & \text{otherwise} \end{cases}$$

Isn't that a contradiction to the non-existence of shortest paths?

- No. If we restrict the length, shortest walks always exist.
- But: If there is a negative cycle $C[0..k]$ with paths $s \rightsquigarrow C$ and $C \rightsquigarrow w$,

  then $\delta_{\leq \ell}(s,w) > \delta_{\leq \ell+k}(s,w) > \delta_{\leq \ell+2k}(s,w) > \cdots$ (and $\delta(s,w) = -\infty$)

$\rightsquigarrow$ We can *detect* if any negative cycle is reachable from s by including more layers $\ell \geq n$ and check if some vertex still improves.

  - *How many further layers do we need / when is it safe to stop?*

Detecting negative cycles

We can detect reachable negative cycles by including just the *single* extra layer $\ell = n$!

Lemma: $\exists w : \delta_{\leq n}(s,w) < \delta_{\leq n-1}(s,w)$ *iff* negative cycle reachable from s

"$\Rightarrow$"
- If some vertex w improves further, i. e., $\delta_{\leq n}(s,w) < \delta_{\leq n-1}(s,w)$
  a walk $W[0..n]$ with $c(W) = \delta_{\leq n}(s,w)$ was the **shortest** way to reach w

$\rightsquigarrow$ W is a non-simple walk, i. e., it contains a cycle

- Let $P[0..k]$ be the path resulting from W by shortcutting all cycles $\rightsquigarrow$ $k \leq n-1$

$\rightsquigarrow$ $c(P) \geq \delta_{\leq n-1}(s,w) > \delta_{\leq n}(s,w) = c(W)$

$\rightsquigarrow$ $\exists$ negative cycle reachable from s

Detecting negative cycles

We can detect reachable negative cycles by including just the *single* extra layer $\ell = n$!

Lemma: $\exists w : \delta_{\le n}(s,w) < \delta_{\le n-1}(s,w)$ *iff* negative cycle reachable from s

"$\Rightarrow$"

- If some vertex w improves further, i. e., $\delta_{\le n}(s,w) < \delta_{\le n-1}(s,w)$
  a walk $W[0..n]$ with $c(W) = \delta_{\le n}(s,w)$ was the **shortest** way to reach w

$\rightsquigarrow$ W is a non-simple walk, i. e., it contains a cycle

- Let $P[0..k]$ be the path resulting from W by shortcutting all cycles $\rightsquigarrow$ $k \le n-1$

$\rightsquigarrow$ $c(P) \ \ge\ \delta_{\le n-1}(s,w) \ >\ \delta_{\le n}(s,w) \ =\ c(W)$

$\rightsquigarrow$ $\exists$ negative cycle reachable from s

"$\Leftarrow$"

- Conversely, let negative cycle $C[0..k]$ be reachable from s

$\rightsquigarrow$ $c(C) = \sum_{i=0}^{k-1} c(C[i]C[i+1]) < 0$

- Assume towards a contradiction that $\forall w : \delta_{\le n}(s,w) = \delta_{\le n-1}(s,w)$

$\rightsquigarrow$ $\forall vw \in E : \delta_{\le n-1}(s,w) \le \delta_{\le n-1}(s,v) + c(vw)$ (no update in layer $\ell = n$)

- summing this inequality over $C[0..k]$ yields (abbreviating $\delta(w) := \delta_{\le n-1}(s,w)$)

$$\sum_{i=1}^{k}\delta(C[i]) \ \le\ \sum_{i=1}^{k}\Big(\delta(C[i-1]) + c(C[i]C[i+1])\Big) \ =\ \sum_{i=0}^{k-1}\delta(C[i]) + \underbrace{\sum_{i=1}^{k} c(C[i]C[i+1])}_{=\,c(C)\,<\,0}$$

$\rightsquigarrow$ $0 \le c(C) < 0$ ↯

$\square$