Matrix Multiplication

Data Structures and Algorithms

Andrei Bulatov

Matrix Multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute C = A·B.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

Brute force. $\Theta(n^3)$ arithmetic operations.

Fundamental question. Can we improve upon brute force?

Matrix Multiplication: Divide and Conquer

Divide: partition A and B into ½n-by-½n blocks.

Conquer: multiply 8 ½n-by-½n recursively.

Combine: add appropriate products using 4 matrix additions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$\mathrm{T}(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \qquad \Rightarrow \mathrm{T}(n) = \Theta(n^3)$$

Better Divide and Conquer

Key idea: multiply 2-by-2 block matrices with only 7 multiplications.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned}
P_1 &= A_{11} \times (B_{12} - B_{22}) \\
P_2 &= (A_{11} + A_{12}) \times B_{22} \\
P_3 &= (A_{21} + A_{22}) \times B_{11} \\
P_4 &= A_{22} \times (B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\
P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\
P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12})
\end{aligned}$$

$$\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}$$

Fast Matrix Multiplication

Fast matrix multiplication. (Strassen, 1969)

Divide: partition A and B into ½n-by-½n blocks.

Compute: 14 ½n-by-½n matrices via 10 matrix additions.

Conquer: multiply 7 ½n-by-½n matrices recursively.

Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.

Assume n is a power of 2.

T(n) = # arithmetic operations.

$$\mathrm{T}(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow \mathrm{T}(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Matrix Multiplication: Practice

Implementation issues.

- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm around n = 128.

Common misperception: "Strassen is only a theoretical curiosity."

- Advanced Computation Group at Apple Computer reports 8x speedup on G4 Velocity Engine when n ~ 2,500.
- Range of instances where it's useful is a subject of controversy.

Remark. Can "Strassenize" Ax=b, determinant, eigenvalues, and other matrix operations.

Matrix Multiplication: Theory

Q. Multiply two 2-by-2 matrices with only 7 scalar multiplications?

A. Yes! [Strassen, 1969] $\Theta(n^{\log_2 7}) = O(n^{2.81})$

Q. Multiply two 2-by-2 matrices with only 6 scalar multiplications?

A. Impossible. [Hopcroft and Kerr, 1971] $\Theta(n^{\log_2 6}) = O(n^{2.59})$

Q. Two 3-by-3 matrices with only 21 scalar multiplications?

A. Also impossible. $\Theta(n^{\log_3 21}) = O(n^{2.77})$

Q. Two 70-by-70 matrices with only 143,640 scalar multiplications?

A. Yes! [Pan, 1980] $\Theta(n^{\log_{70} 143640}) = O(n^{2.80})$

Decimal wars.

- December, 1979: $O(n^{2.521813})$.
- January, 1980: $O(n^{2.521801})$.

Matrix Multiplication: Theory (cntd)

Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987.]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat:

Theoretical improvements to Strassen are progressively less practical.

Dynamic Programming

Data Structures and Algorithms

Andrei Bulatov

Algorithmic Paradigms

Greed.

Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer.

Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming.

Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Algorithmic Paradigms

Bellman.

Pioneered the systematic study of dynamic programming in the 1950s.

Etymology.

Dynamic programming = planning over time.

Secretary of Defense was hostile to mathematical research.

Bellman sought an impressive name to avoid confrontation.

- "it's impossible to use dynamic in a pejorative sense"
- "something not even a Congressman could object to"

Algorithmic Paradigms

Bellman.

Pioneered the systematic study of dynamic programming in the 1950s.

Etymology.

Dynamic programming = planning over time.

Secretary of Defense was hostile to mathematical research.

Bellman sought an impressive name to avoid confrontation.

- "it's impossible to use dynamic in a pejorative sense"
- "something not even a Congressman could object to"

Weighted Interval Scheduling

Weighted interval scheduling problem.

Instance

A set of n jobs.

Job j starts at s_j, finishes at f_j, and has weight or value v_j .

Two jobs compatible if they don't overlap.

Objective

Find maximum weight subset of mutually compatible jobs.

Unweighted Interval Scheduling: Review

Recall:

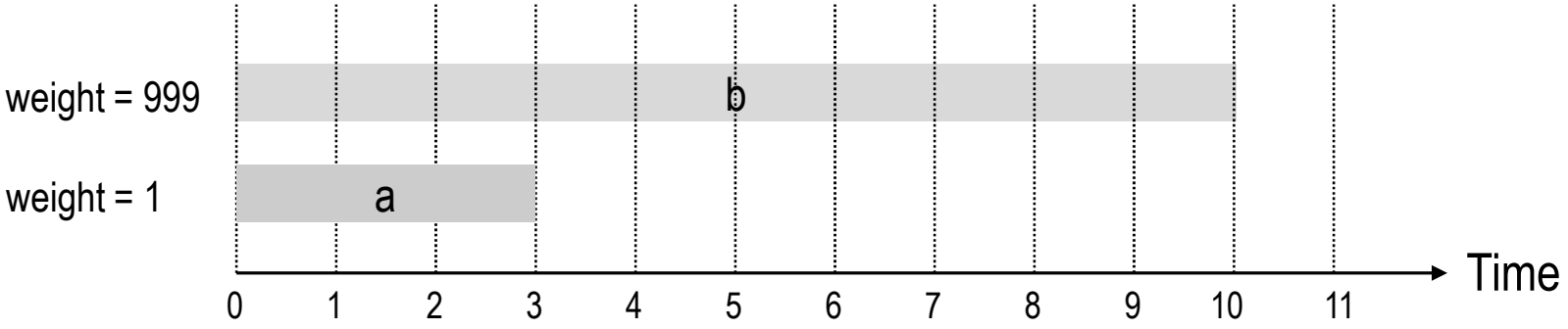
Greedy algorithm works if all weights are 1.

Consider jobs in ascending order of finish time.

Add job to subset if it is compatible with previously chosen jobs.

Observation.

Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



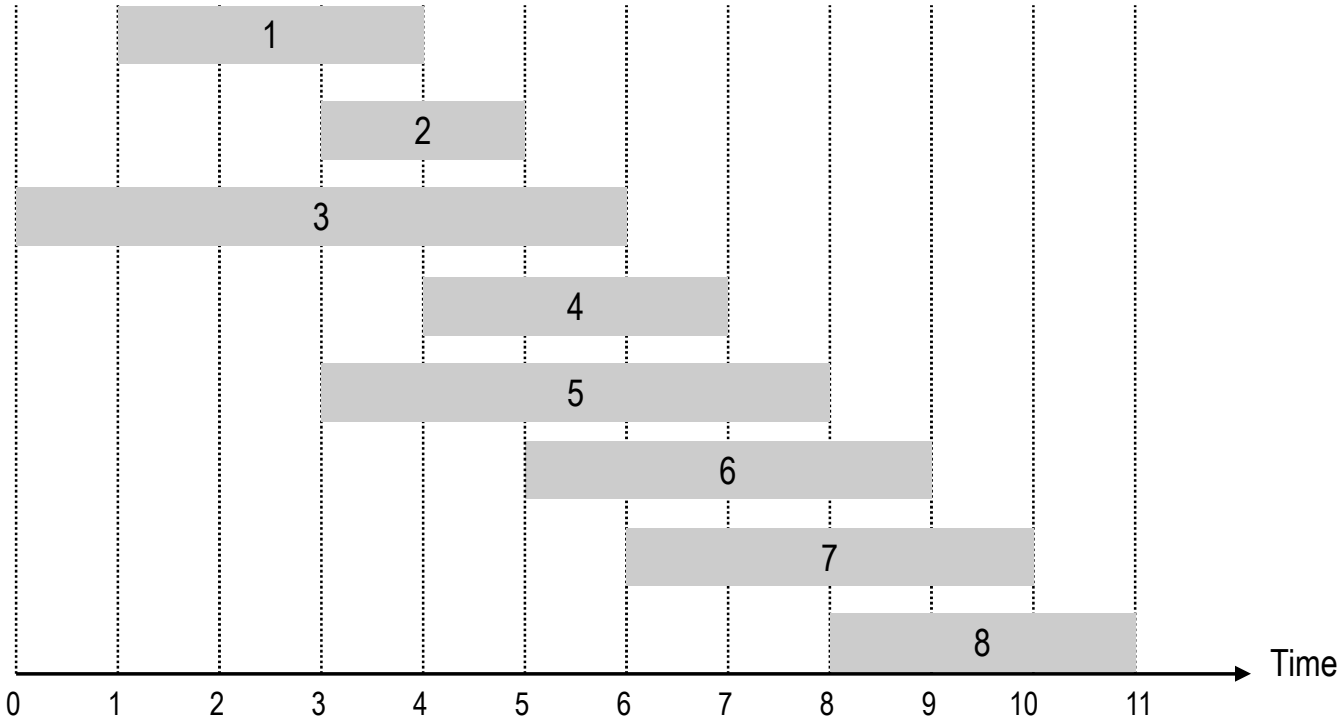
Weighted Interval Scheduling

Notation: Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Let p(j) be the largest index $i < j$ such that job i is compatible with j.

Example.

p(8) = 5, p(7) = 3, p(2) = 0.



Dynamic Programming: Binary Choice

Let OPT(j) denote the value of an optimal solution to the problem consisting of job requests 1, 2, ..., j.

Case 1: OPT selects job j.

cannot use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }

must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

Case 2: OPT does not select job j.

must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Brute Force

Input:
  n, $s_1,\ldots,s_n$, $f_1,\ldots,f_n$, $v_1,\ldots,v_n$

  sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$
  compute p(1), p(2), …, p(n)
  return Compute-Opt(n)

```
Compute-Opt(j)
  if (j = 0)
   return 0
  else
   return max(v_j+Compute-Opt(p(j)),Compute-Opt(j-1))
```

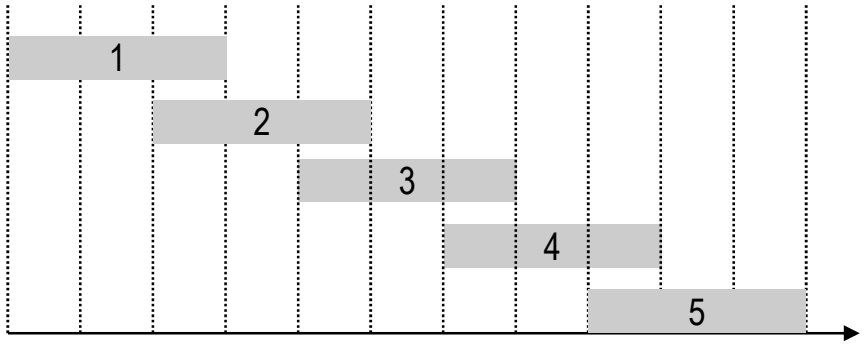
Weighted Interval Scheduling: Brute Force

Observation.

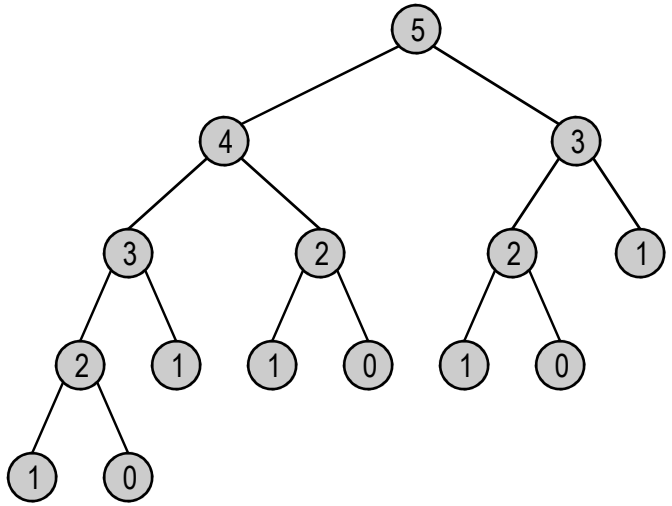
Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

Example

Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



$p(1) = 0,\ p(j) = j - 2$



Weighted Interval Scheduling: Memoization

Memoization:

Store results of each sub-problem in a cache; lookup as needed.

```
Input:
   n, s1,…,sn , f1,…,fn , v1,…,vn

sort jobs by finish times so that f1≤f2≤…≤ fn
compute p(1), p(2), …, p(n)
set OPT[0]:=0
for j=1 to n do
   set OPT[j]:=max(vj+OPT[p(j)],OPT[j-1])
endfor
return OPT[n]
```

Weighted Interval Scheduling: Running Time

Theorem

Memoized version of algorithm takes O(n log n) time.

Proof

Sort by finish time: O(n log n).

Computing p(·) : O(n) after sorting by finish time

Each iteration of the for loop: O(1)

Overall time is O(n log n)

QED

Remark.

O(n) if jobs are pre-sorted by finish times

Automated Memoization

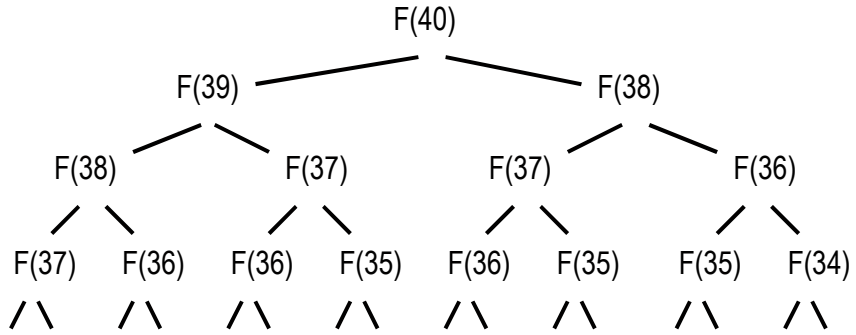
Automated memoization.
Many functional programming languages (e.g., Lisp) have built-in support for memoization.
Q. Why not in imperative languages (e.g., Java)?

```
(defun F (n)
  (if
    (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2)))))
```

Lisp (efficient)

```
static int F(int n) {
   if (n <= 1) return n;
   else return F(n-1) + F(n-2);
}
```

Java (exponential)



Finding a Solution

Dynamic programming algorithm computes optimal value.
What if we want the solution itself?

Do some post-processing

```
Find-Solution(j)
  if j = 0 then
      output nothing
   else if  v_j+M[p(j)]>M[j-1] then do
      print j
      Find-Solution(p(j))
   endif
   else
      Find-Solution(j-1)
  endif
```