CS711008Z Algorithm Design and Analysis

Lecture 6. Basic algorithm design technique: Dynamic programming

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

Outline

- The first example: MATRIXCHAINMULTIPLICATION
- Elements of dynamic programming technique;
- Various ways to describe subproblems: SEGMENTED LEAST SQUARES, KNAPSACK, RNA SECONDARY STRUCTURE, SEQUENCE ALIGNMENT, and SHORTEST PATH;
- Connection with greedy technique: INTERVAL SCHEDULING, SHORTEST PATH.

Dynamic programming and its connection with divide-and-conquer

- Dynamic programming typically applies to the **optimization problems with optimal-substructure property**; that is, the original problem can be divided into smaller subproblems, and the optimal solution to the original problem can be obtained by **combining the optimal solutions to subproblems**.
- To identify meaningful recursions, one of the key steps is to define **an appropriate general form of sub-problems**. For this aim, it is helpful to describe the solving process as **a multiple-stage decision process**.
- Unlike the general divide-and-conqueror framework, a dynamic programming algorithm usually **enumerate** all possible dividing strategies. In addition, **the repetition of computing the common subproblems** is avoided through "programing". [1]

[1] Here, "programming" means "tabular" rather than "coding". The best example of programming might be the calculation of Fibonacci numbers. Program: [Date: 1600-1700; Language: French; 'to write before']

On what problems can we apply divide and conqueror?

- Suppose a problem is related to the following data structure, perhaps we can try to divide it into sub-problems.
  - An array with n elements
  - A matrix
  - A set of n elements
  - A tree
  - A graph

MatrixChainMultiplication problem: recursion over **sequences**

INPUT:

A sequence of n matrices $A_1, A_2, ..., A_n$; matrix A_i has dimension $p_{i-1} \times p_i$;

OUTPUT:

Fully parenthesizing the product $A_1 A_2 ... A_n$ in a way to minimize the number of scalar multiplications.

Let's start from a simple example

$$A_1 = \begin{bmatrix} 1 & 2 \end{bmatrix} \quad A_2 = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad A_3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

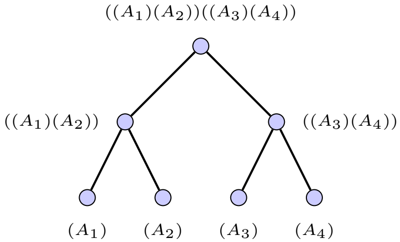
1×2 $\quad 2 \times 3$ $\quad 3 \times 4$

$$\begin{array}{rcc} \texttt{Solutions:} & ((A_1)(A_2))(A_3) & (A_1)((A_2)(A_3)) \\ \texttt{\#Multiplications:} & 1 \times 2 \times 3 & 2 \times 3 \times 4 \\ & +1 \times 3 \times 4 & +1 \times 2 \times 4 \\ & = 18 & = 32 \end{array}$$

- Here, the calculation of $A_1 A_2$ needs $1 \times 2 \times 3$ scalar multiplications.
- The objective is to determine a calculation sequence such that the number of multiplications is minimized.

The solution space size

- Intuitively, a calculation sequence can be described as a binary tree, where each node corresponds to a subproblem.



- The total number of possible calculation sequences: $\binom{2n}{n} - \binom{2n}{n-1}$ (Catalan number)
- Thus, it takes exponential time to enumerate all possible calculation sequences.
- Question: can we design an efficient algorithm?

A dynamic programming algorithm (by S. S. Godbole, 1973?)

Reduce into smaller sub-problems

1. It is not easy to solve the problem directly when n is large. Let's investigate whether it is possible to reduce into smaller sub-problems.
2. Solution: a full parentheses. Let's describe the solving process as a process of **multiple-stage decisions**, where each decision is to add parentheses at a position.
3. Suppose we have already worked out the optimal solution O, where the first **decision** adds two parentheses as $(A_1 ... A_k)(A_{k+1} ... A_n)$.
4. This decision decomposes the original problem into two **independent sub-problems**: to calculate $A_1 ... A_k$ and $A_{k+1} ... A_n$.
5. Summarizing these two cases, we define the general form of sub-problems as: to calculate $A_i ... A_j$ with the minimal number of scalar multiplications.

General form of sub-problems and optimal substructure property

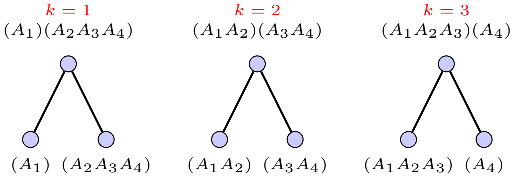
- The general form of sub-problems: to calculate $A_i ... A_j$ with the minimal number of scalar multiplications. Let's denote the optimal solution value to the sub-problem as $OPT(i, j)$, thus the original problem can be solved via calculating $OPT(1, n)$.
- The optimal solution to the original problem can be obtained through combining the optimal solutions to sub-problems, implying the following **optimal substructure property**: $OPT(1, n) = OPT(1, k) + OPT(k + 1, n) + p_1 p_{k+1} p_{n+1}$

Proof of the optimal substructure property

- "Cut-and-paste" proof:
  - Suppose for $A_i...A_k$, there is another parentheses $OPT'(i,k)$ better than $OPT(i,k)$. Then the combination of $OPT'(i,k)$ and $OPT(k+1,j)$ leads to a new solution with lower cost than $OPT(i,j)$: a contradiction.
  - Here, the independence between $A_i...A_k$ and $A_{k+1}...A_j$ guarantees that the substitution of $OPT(i,k)$ with $OPT'(i,k)$ does not affect solution to $A_{k+1}...A_j$.

A recursive solution

- So far so good! The only difficulty is that we have no idea of the first splitting position k in the optimal solution.
- How to overcome this difficulty? **Enumeration!** We **enumerate all possible options of the first decision**, i.e. for all k, $i \leq k < j$.



- Thus we have the following recursion:

$$OPT(i,j) = \begin{cases} 0 & i = j \\ \mathbf{min}_{\mathbf{i \leq k < j}} \{OPT(i,k) + OPT(k+1,j) + p_i p_{k+1} p_{j+1}\} & otherwise \end{cases}$$

Implementing the recursion: trial 1

Trial 1: Explore the recursion in the top-down manner

RECURSIVE_MATRIX_CHAIN(i, j)

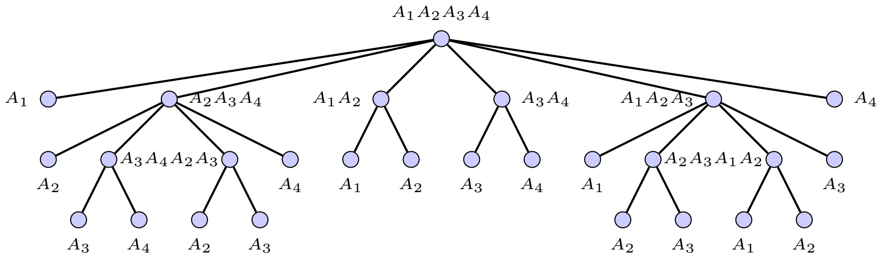
if $i == j$ **then**
    return 0;
end if
$OPT(i, j) = +\infty$;
for $k = i$ to $j - 1$ **do**
    $q =$ RECURSIVE_MATRIX_CHAIN(i, k)
        $+$ RECURSIVE_MATRIX_CHAIN$(k + 1, j)$
        $+p_i p_{k+1} p_{j+1}$;
    if $q < OPT(i, j)$ **then**
        $OPT(i, j) = q$;
    end if
end for
return $OPT(i, j)$;

- Note: The optimal solution to the original problem can be obtained through calling RECURSIVE_MATRIX_CHAIN$(1, n)$.

An example

$$A_1 = \begin{bmatrix} 1 & 2 \end{bmatrix} \quad A_2 = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad A_3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} \quad A_4 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

1×2 $\quad$ 2×3 $\quad$ 3×4 $\quad$ 4×5



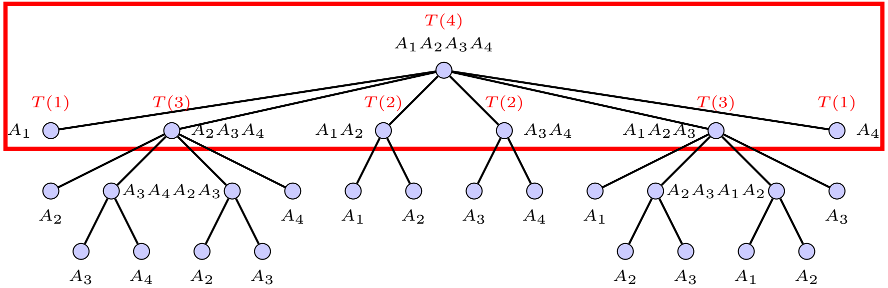
Note: each node of the recursion tree denotes a subproblem.

However, this is not a good implementation

Theorem

Algorithm RECURSIVE-MATRIX-CHAIN *costs exponential time.*

Let $T(n)$ denote the time used to calculate product of n matrices. Note that $T(n) \geq 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1)$ for $n > 1$.



Theorem

Algorithm RECURSIVE-MATRIX-CHAIN *costs exponential time.*

Proof.

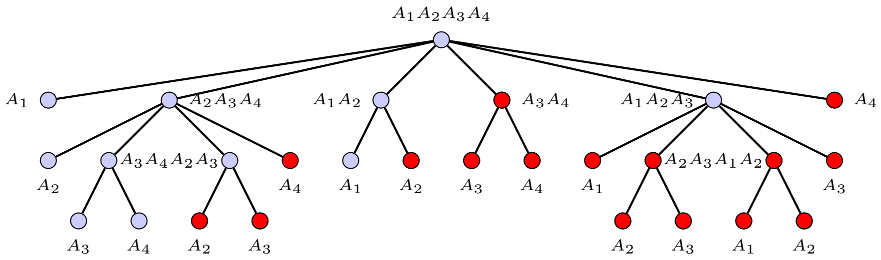
- We shall prove $T(n) \geq 2^{n-1}$ using the substitution technique.
  - Basis: $T(1) \geq 1 = 2^{1-1}$.
  - Induction:

$$
\begin{aligned}
T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) && (1)\\
&= n + 2\sum_{k=1}^{n-1} T(k) && (2)\\
&\geq n + 2\sum_{k=1}^{n-1} 2^{k-1} && (3)\\
&\geq n + 2(2^{n-1} - 1) && (4)\\
&\geq n + 2^n - 2 && (5)\\
&\geq 2^{n-1} && (6)
\end{aligned}
$$

□

Implementing the recursion: trial 2

Why the first trial failed?



- Reason: There are only $O(n^2)$ subproblems. However, some subproblems (in red) were solved repeatedly.
- Solution: **memorize the solutions to subproblems** using an array $OPT[1..n; 1..n]$ for further look-up.

The “memorizing” technique

MEMORIZE_MATRIX_CHAIN(i, j)

if $OPT[i, j] \neq NULL$ **then**
    return $OPT(i, j)$;
end if
if $i == j$ **then**
    $OPT[i, j] = 0$;
else
    for $k = i$ to $j - 1$ **do**
        $q =$ MEMORIZE_MATRIX_CHAIN(i, k)
            $+$MEMORIZE_MATRIX_CHAIN$(k + 1, j)$
            $+p_i p_{k+1} p_{j+1}$;
        if $q < OPT[i, j]$ **then**
            $OPT[i, j] = q$;
        end if
    end for
end if
return $OPT[i, j]$;

The "memorizing" technique cont'd

- The original problem can be solved by calling $\mathrm{MEMORIZE\_MATRIX\_CHAIN}(1, n)$ with all $OPT[i, j]$ initialized as NULL.
- Time-complexity: $O(n^3)$ (The calculation of each entry $OPT[i, j]$ makes $O(n)$ recursive calls in line 8.)
- Note: the calculation of Fibonacci number is a good example of the power of the "memorizing" technique.

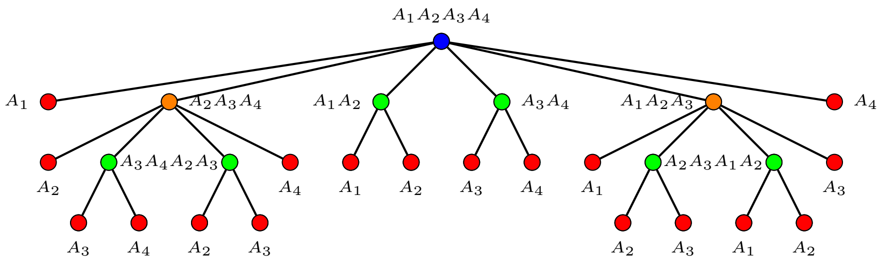
Implementing the recursion faster: trial 3

Trial 3: Faster implementation: unrolling the recursion in the bottom-up manner

MATRIX_CHAIN_MULTIPLICATION(P)

```
for i = 1 to n do
    OPT(i, i) = 0;
end for
for l = 2 to n do
    for i = 1 to n - l + 1 do
        j = i + l - 1;
        OPT(i, j) = +∞;
        for k = i to j - 1 do
            q = OPT(i, k) + OPT(k + 1, j) + p_i p_{k+1} p_{j+1};
            if q < OPT(i, j) then
                OPT(i, j) = q;
                S(i, j) = k;
            end if
        end for
    end for
end for
return OPT(1, n);
```

Recursion tree: an intuitive view of the bottom-up calculation



Solving sub-problems in a bottom-up manner, i.e.

1. Solving the sub-problems in red first;
2. Then solving the sub-problems in green;
3. Then solving the sub-problems in orange;
4. Finally we can solve the original problem in blue.

Step 1 of the bottom-up algorithm

OPT

1	2	3	4	
0	6			1
	0	24		2
		0	60	3
			0	4

SPLITTER

1	2	3	4	
	1			1
		2		2
			3	3
				4

Step 1:

$OPT[1,2] = p_0 \times p_1 \times p_2 = 1 \times 2 \times 3 = 6;$

$OPT[2,3] = p_1 \times p_2 \times p_3 = 2 \times 3 \times 4 = 24;$

$OPT[3,4] = p_2 \times p_3 \times p_4 = 3 \times 4 \times 5 = 60;$

Step 2 of the bottom-up algorithm

OPT

1	2	3	4	
0	6	18		1
	0	24	64	2
		0	60	3
			0	4

SPLITTER

1	2	3	4	
	1	2		1
		2	3	2
			3	3
				4

Step 2:

$$OPT[1,3] = \min \begin{cases} OPT[1,2] + OPT[3,3] + p_0 \times p_2 \times p_3 (= 18) \\ OPT[1,1] + OPT[2,3] + p_0 \times p_1 \times p_3 (= 32) \end{cases}$$

Thus, $SPLITTER[1,2] = 2$.

$$OPT[2,4] = \min \begin{cases} OPT[2,2] + OPT[3,4] + p_1 \times p_2 \times p_4 (= 90) \\ OPT[2,3] + OPT[4,4] + p_1 \times p_3 \times p_4 (= 64) \end{cases}$$

Thus, $SPLITTER[2,4] = 3$.

OPT

1	2	3	4	
0	6	18	38	1
	0	24	64	2
		0	60	3
			0	4

SPLITTER

1	2	3	4	
	1	2	3	1
		2	3	2
			3	3
				4

`Step 3:`

$$OPT[1,4] = \min \begin{cases} OPT[1,1] + OPT[2,4] + p_0 \times p_1 \times p_4 (= 74) \\ OPT[1,2] + OPT[3,4] + p_0 \times p_2 \times p_4 (= 81) \\ OPT[1,3] + OPT[4,4] + p_0 \times p_3 \times p_4 (= 38) \end{cases}$$

Thus, $SPLITTER[1,4] = 3$.

Question: We have calculated the optimal **value**, but how to get the optimal **calculation sequence**?

Final step: constructing an optimal solution through "backtracking" the optimal options

- Idea: **backtracking!** Starting from $OPT[1, n]$, we trace back the source of $OPT[1, n]$, i.e. which option we take at each decision stage.
- Specifically, an auxiliary array $S[1..n, 1..n]$ is used.
  - Each entry $S[i, j]$ records the optimal decision, i.e. the value of k such that the optimal parentheses of $A_i...A_j$ occurs between $A_k A_{k+1}$.
  - Thus, the optimal solution to the original problem $A_{1..n}$ is $A_{1..S[1,n]} A_{S[1,n]+1..n}$.
- Note: The optimal option cannot be determined before solving all subproblems.

Backtracking: step 1

OPT

	1	2	3	4
1	0	6	18	38
2		0	24	64
3			0	60
4				0

SPLITTER

	1	2	3	4
1		1	2	3
2			2	3
3				3
4				

Step 1: $(\; A_1 A_2 A_3 \;)\;(\; A_4 \;)$

OPT

1	2	3	4	
0	6	18	38	1
	0	24	64	2
		0	60	3
			0	4

SPLITTER

1	2	3	4	
	1	2	3	1
		2	3	2
			3	3
				4

`Step 1:` $(\ A_1A_2A_3\ )\ (\ A_4\ )$

`Step 2:` $(\ (\ A_1A_2\ )\ (\ A_3\ )\ )\ (\ A_4\ )$

Backtracking: step 3

OPT

1	2	3	4	
0	6	18	38	1
	0	24	64	2
		0	60	3
			0	4

SPLITTER

1	2	3	4	
	1	2	3	1
		2	3	2
			3	3
				4

Step 1: $(\ A_1 A_2 A_3\ )\ (\ A_4\ )$

Step 2: $(\ (\ A_1 A_2\ )\ (\ A_3\ )\ )\ (\ A_4\ )$

Step 3: $(\ (\ (\ A_1\ )\ (\ A_2\ )\ (\ A_3\ )\ )\ (\ A_4\ )$

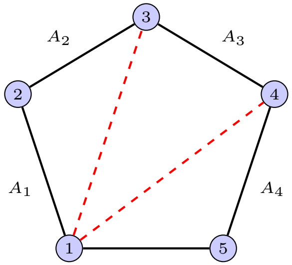
Summary: elements of dynamics programming

1. It is usually not easy to solve a large problem directly. Let's consider whether the problem can be decomposed into smaller sub-problems. How to define sub-problems? [2]
   - Let's describe the solving process as a process of multiple-stage **decisions** first.
   - Suppose that we have already worked out the optimal solution. Let's consider the **first/final decision (in some order)** in the optimal solution. The **first/final decision** might have several options.
   - We enumerate all possible options for the decision, and observe the generated sub-problems. The general form of sub-problems can be defined via summarising all possible forms of sub-problems.
2. Show the **optimal substructure property**, i.e. the optimal solution to the problem contains within it optimal solutions to subproblems.
3. Programming: if recursive algorithm solves the same subproblem over and over, "tabular" can be used to avoid the repetition of solving same sub-problems.

[2] Sometimes problem should be extended to identify meaningful recursion.

Question: is $O(n^3)$ the lower bound?

An $O(n \log n)$ algorithm by Hu and Shing 1981



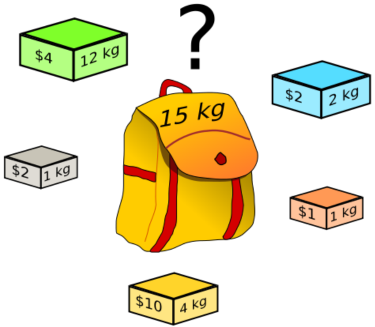
- One-to-one correspondence between parenthesis and partioning a convex polygon into non-intersecting triangles.
  - Each node has a weight w_i, and a triangle corresponds to a product of the weight of its nodes.
  - The decomposition (red, dashed lines) has a weight sum of 38. In fact, it corresponds to the parenthesis (((A_1) (A_2) (A_3)) (A_4)).
- The optimal decomposition can be found in $O(n \log n)$ time.

(See Hu and Shing 1981 for details)

0/1 Knapsack problem: recursion over sets

A Knapsack instance

Given a set of items, each item has a weight and a value, to select a subset of items such that the total weight is less than a given limit and the total value is as large as possible.



What's the best solution?

0/1 KNAPSACK problem

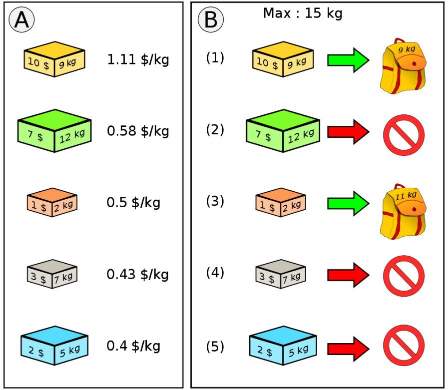
Formalized Definition:

- **Input:**
  A set of items. Item i has weight w_i and value v_i, and a total weight limit W;
- **Output:**
  A sub-set of items to maximize the total value with a total weight below W.

Note:

1. Here, "0/1" means that we should select an item (1) or abandon it (0), and we cannot select parts of an item.
2. In contrast, FRACTIONAL KNAPSACK problem allow one to select a fractional, say 0.5, of an item.

0/1 Knapsack problem: an intuitive algorithm



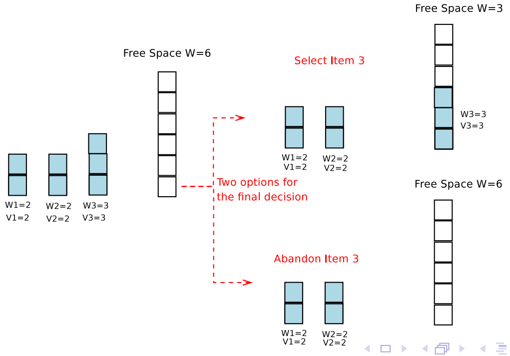
- Intuitive method: selecting “expensive” items first.
- But this is not the optimal solution.

Key observation

- It is not easy to solve the problem with n items. Let's whether it is possible to reduce into smaller sub-problems.
- Solution: a subset of items. Describe the solving process as as a process of multiple-stage **decisions**. At the i-th decision stage, we decide whether item i should be selected.
- Suppose we have already worked out the optimal solution.
- Consider the **first** decision, i.e. whether the optimal solution contains item n or not. The decision has two options:
  1. SELECT: then it suffices to select items as "expensive" as possible from $\{1, 2, ..., n-1\}$ with weight limit $W - w_n$.
  2. ABANDON: Otherwise, we should select items as "expensive" as possible from $\{1, 2, ..., n-1\}$ with weight limit W.
- In both cases, the original problem is reduced into smaller sub-problems.

Key observation cont'd

- Summarizing these two cases, the general form of sub-problems is: to select items as "expensive" as possible from $\{1, 2, ..., i\}$ with weight limit w. Denote the optimal solution value as $OPT(i, w)$.
- Optimal sub-structure property:
  $OPT(n, W) = \max\{OPT(n-1, W), OPT(n-1, W - w_n) + v_n\}$
  (Enumerating two possible decisions for item n.)



Algorithm

KNAPSACK(n, W)

```
for w = 1 to W do
    OPT[0, w] = 0;
end for
for i = 1 to n do
    for w = 1 to W do
        OPT[i, w] = max{OPT[i-1, w], v_i + OPT[i-1, w-w_i]};
    end for
end for
```

Example I

Initially all OPT[0,w] = 0

	W = 0	1	2	3	4	5	6
i=3							
i=2							
i=1							
i=0	0	0	0	0	0	0	0

Example II

OPT[1,2] = max{
OPT[0,2] (=0),
OPT[0,0] + V1 (=0+2) }
= 2

	W = 0	1	2	3	4	5	6
i=3							
i=2							
i=1	0	0	2	2	2	2	2
i=0	0	0	0	0	0	0	0

Example III

OPT[2,4] = max{
OPT[1,4] (=2),
OPT[1,2] + V2 (=2+2) }
= 4

	W = 0	1	2	3	4	5	6
i=3							
i=2	0	0	2	2	4	4	4
i=1	0	0	2	2	2	2	2
i=0	0	0	0	0	0	0	0

Example IV

OPT[3,3] = max{
OPT[2,3] (=2),
OPT[2,0] + V3 (=0+3) }
= 3

	W = 0	1	2	3	4	5	6
i=3	0	0	2	3	4	5	5
i=2	0	0	2	2	4	4	4
i=1	0	0	2	2	2	2	2
i=0	0	0	0	0	0	0	0

Backtracking: step 1

Backtracking

OPT[3,6] = max{
    OPT[2,6] (=4),
    OPT[2,3] + V3 (=2+3) }
        = 5
Decision: Select item 3

	W = 0	1	2	3	4	5	6
i=3	0	0	2	3	4	5	5
i=2	0	0	2	2	4	4	4
i=1	0	0	2	2	2	2	2
i=0	0	0	0	0	0	0	0

Backtracking: step 2

Backtracking

OPT[3,6] = max{
OPT[2,6] (=4),
OPT[2,3] + V3 (=2+3) }
= 5
Decision: Select item 3

OPT[2,3] = max{
OPT[1,3] (=2),
OPT[1,1] + V2 (=0+2) }
= 2
Decision: Select item 2

	W = 0	1	2	3	4	5	6
i=3	0	0	2	3	4	5	5
i=2	0	0	2	2	4	4	4
i=1	0	0	2	2	2	2	2
i=0	0	0	0	0	0	0	0

Time complexity analysis

- Time complexity: $O(nW)$. (Hint: for each entry in the matrix, only a comparison is needed; we have $O(nW)$ entries in the matrix.)
- Notes:
  1. This algorithm is inefficient when W is large, say $W = 1M$.
  2. Remember that a polynomial time algorithm costs time polynomial in the **input length**. However, this algorithm costs time $mW = m2^{\log W} = m2^{\text{input length}}$. Exponential!
  3. Pseudo-polynomial time algorithm: polynominal in the value of W rather than the **length** of W ($\log W$).
  4. We will revisit this algorithm in approximation algorithm design.

Note: the general form of subproblems

- Here the items were considered in an order. Why?
- Let's consider two general forms of sub-problems:
  1. Selecting items as "expensive" as possible from a subset s with weight limit w: the number of sub-problems is **exponential**.
  2. Selecting items as "expensive" as possible from $\{1, 2, ..., i\}$ with weight limit w: the number of sub-problems is **O(n)**.
- In fact, the first one is a recursion over sets, while the second one is a recursion over sequences.

Extension: The first public-key encryption system

- *Cryptosystems based on the knapsack problem were among the first public key systems to be invented, and for a while were considered to be among the most promising. However, essentially all of the knapsack cryptosystems that have been proposed so far have been broken. These notes outline the basic constructions of these cryptosystems and attacks that have been developed on them.*

See *The Rise and Fall of Knapsack Cryptosystems* for details.

VERTEX COVER: recursion over **trees**

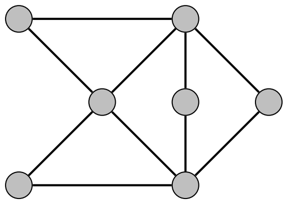
Vertex Cover Problem

- Practical problem:
  Given n sites connected with paths, how many guards (or cameras) should be deployed on sites to surveille **all** *the paths?*

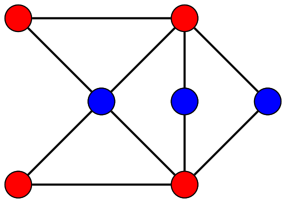
Formalized Definition:

Input: Given a graph $G =< V, E >$
Output: the minimum of nodes $S \subseteq V$, such that each edge has at least one of its endpoints in S

- An example: how many nodes are needed to cover all edges in the following graph?

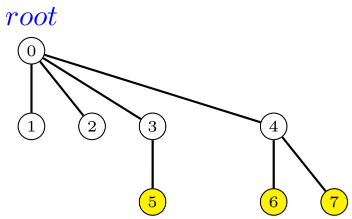


An example



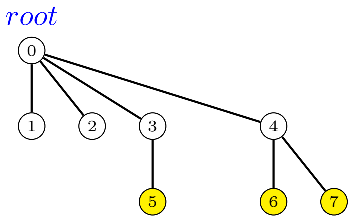
- The nodes in red form a vertex cover.
- VERTEX COVER is a hard problem for general graph.
- However, it is easy to find the minimum vertex cover for trees.

An example



- It is not easy to solve the problem with n nodes. Let's whether it is possible to reduce into smaller sub-problems.
- Solution: selection a subset of nodes. Describe the solving process as as a process of multiple-stage **decisions**. At each decision stage, we decide whether a node should be selected.
- Suppose we have already worked out the optimal solution.
- Consider the **first** decision, i.e. whether the optimal solution contains the root node or not. The decision has two options:
  1. SELECT: it suffices to consider the sub-trees;
  2. ABANDON: we should select all the children nodes, and then consider all grand-children.

An example



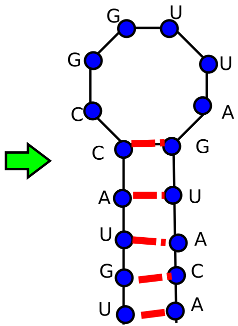
- In both cases, the original problem is reduced into smaller sub-problems.
- General form of sub-problems: find minimum vertex cover on a tree rooted at node v. Let's denote the optimal solution as $OPT(v)$.
- Thus we have the following recursion:
  $OPT(root) =$
  $$\begin{cases} 1 + \sum_c OPT(c) & c \text{ is a children of root} \\ k + \sum_g OPT(c) & c \text{ is a grand-children of root}, k = \#children \end{cases}$$
- Time complexity: $O(n)$

RNA SECONDARY STRUCTURE PREDICTION: recursion over **trees**

RNA Secondary Structure

- RNA is a sequence of nucleic acids. It will automatically form structures in water through the formation of bonds $A - U$ and $C - G$.
- The native structure is the conformation with the lowest energy. Here, we simply use the number of base pairs as the energy function.

A C A U G A U U G G C C A U G U

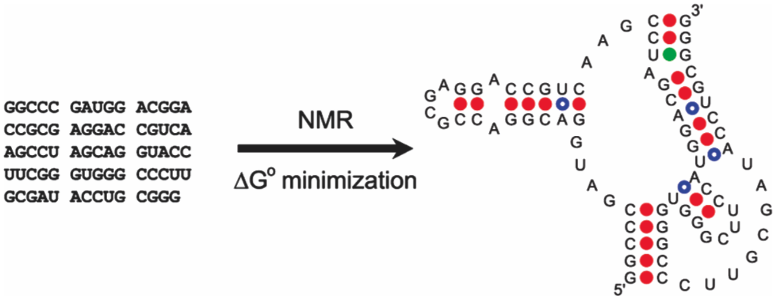


INPUT:
A sequence in alphabet $\Sigma = \{A, U, C, G\}$;
OUTPUT:
A pairing scheme with the maximum pairing number

Requirements of base pairs:

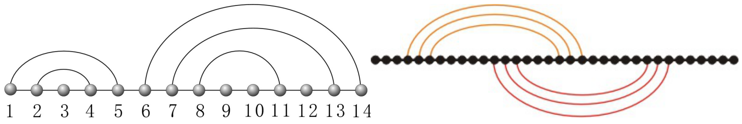
1. Watson-Crick pair: A pairs with U, and C pairs with G;
2. There is no base occurring in more than 1 base pairs;
3. No cross-over (nesting): there is no crossover under the assumption of free pseudo-knots.
4. And two bases i, j $(|i - j| \leq 4)$ cannot form a base pair.

An example

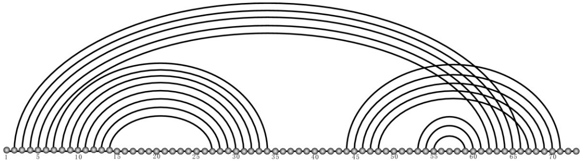


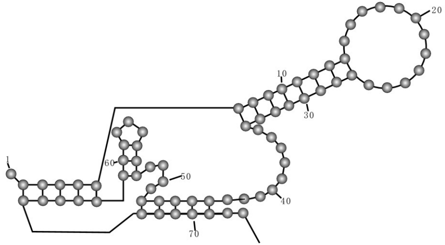
GGCCC GAUGG ACGGA
CCGCG AGGAC CGUCA
AGCCU AGCAG GUACC
UUCGG GUGGG CCCUU
GCGAU ACCUG CGGG
NMR
ΔG° minimization
5'
3'

Nesting and Pseudo-knot



Left: nesting of base pairs (no cross-over); Right: pseudo-knots (cross-over);





Feymann graph: an intuitive representation form of RNA secondary structure, i.e. two bases are connected by an edge if they form a Watson-Crick pair.

Key observation I

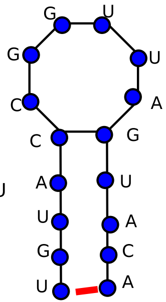
- Solution: a set of nested base pairs. Describe the solving process as as a process of multiple-stage **decisions**. At the i-th decision stage, we determine whether base i forms pair or not.
- Suppose we have already worked out the optimal solution.
- Consider the $first$ decision made for base n. There are two options:
  1. Base n pairs with a base i: we should calculate optimal pairs for regions $i+1...n-1$ and $1..i-1$.
     Note: these two sub-problems are independent due to the "nested" property.
  2. Base n doesn't form a pair: we should calculate optimal pairs for regions $1...n-1$.
- Thus we can design the general form of sub-problems as: to calculate the optimal pairs for region $i...j$. (Denote the optimal solution value as: $OPT(i,j)$.)

Key observation II

Option 1:

j and t form apair, t <=j-1

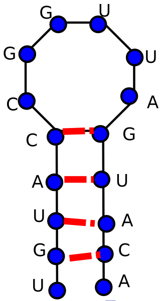
A C A U G A U U G G C C A U G U
i t j



Option 2:

j forms no pair

A C A U G A U U G G C C A U G U
i t j



- Optimal substructures property:

Algorithm

RNA2D(n)

Initialize all $OPT[i, j]$ with 0;
for $i = 1$ to n **do**
    for $j = i + 5$ to n **do**
        $OPT[i, j] = \max\{OPT[i, j-1], \max_t\{1 + OPT[i, t-1] + OPT[t+1, j-1]\}\}$;
        /* t and j can form Watson-Crick base pair. */
    end for
end for

INPUT:

ACCGGUAGU

i=3	0	0	0	
i=2	0	0		
i=1	0			
i=0				
	j = 6	7	8	9

INPUT:

A C C G G U A G U

	j = 6	7	8	9
i=3	0	0	0	1
i=2	0	0	1	
i=1	0	0		
i=0	1			

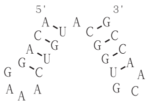
INPUT:

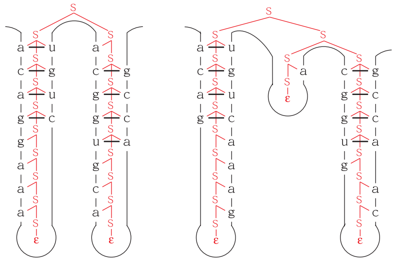
A C C G G U A G U

i=3	0	0	0	1
i=2	0	0	1	1
i=1	0	0	1	
i=0	1	1		
	j = 6	7	8	9

i=3	0	0	0	1
i=2	0	0	1	1
i=1	0	0	1	1
i=0	1	1	1	2
	j = 6	7	8	9

Time complexity: $O(n^3)$.



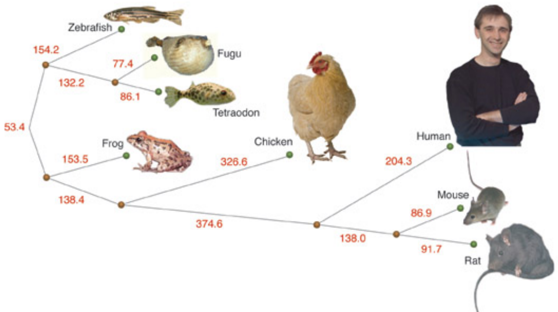


(see extra slides)

Sequence Alignment problem: recursion over **sequence pairs**

Practical problem: genome similarity

- To identify homology genes of two species, say $Human$ and $Mouse$. E.g., Human and Mouse NHPPEs (in KRAS genes) show a high sequence homology (Ref: Cogoi, S., et al. NAR, 2006).
    - `GGGCGGTGTGGGAA-GAGGGAAG-AGGGGGAG`
    - `||| || | ||||| |||||| | |||| |`
    - `GGGAGG-GAGGGAAGGAGGGAGGGAGGGAG--`
- Having calculating the similarity of genomes of various species, a reasonable phylogeny tree can be estimated (See https://www.llnl.gov/str/June05/Ovcharenko.html)



Practical problem: spell tool to correct typos

- When you type in ``OCURRANCE'', spell tools might guess what you really want to type through the following alignment, i.e. ``OCURRANCE'' is very similar to ``OCCURRENCE'' except for INS/DEL/MUTATION operations.
  - O-CURRANCE
  - OCCURRENCE
- But the following instance is a bit difficult:
  - abbbaa-bbbbaab
  - ababaaabbbba-b

INPUT:
Two sequences S and T, $|S| = m$, and $|T| = n$;
OUTPUT:
To identify an alignment of S and T that maximizes a scoring function.

Note: for the sake of simplicity, the following indexing schema is used: $S = S_1 S_2 ... S_m$.

What is an alignment?

- An example of alignment:
  - `O-CURRANCE`
  - `| |||| |||`
  - `OCCURRENCE`
- Basic idea:
  1. Alignment is usually used to describe the generating process of an erroneous word from the correct word.
  2. Make the two sequences to have the same length through adding space '-', i.e. changing S to S' through adding spaces at some positions, and changing T to T' through adding spaces at some positions, too. The only requirement is: $|S'| = |T'|$. There are three cases:
     1. $T'[i] =' -'$: $S'[i]$ is simply an `INSERTION`.
     2. $S'[i] =' -'$: $S'[i]$ is simply a `DELETION` of $T'[i]$.
     3. Otherwise, $S'[i]$ is a copy of $T'[i]$ (with possible `MUTATION`).
  3. Thus, an alignment clearly illustrates how to change T into S with a series of `INS/DEL/MUTATION` operations.

How to measure an alignment in the sense of sequence similarity?

The similarity is defined as the sum of score of aligned letter pairs, i.e.

$$d(S,T) = \sum_{i=1}^{|S'|} \delta(S'[i], T'[i])$$

The simplest $\delta(a,b)$ is:

1. Match: $+1$, e.g. $\delta('C', 'C') = 1$.
2. Mismatch: -1, e.g. $\delta('E', 'A') = -1$.
3. Ins/Del: -3, e.g. $\delta('C', '-') = -3$.

[3]

[3] Ideally, the score function is designed such that $d(S,T)$ is proportional to $\log \Pr[S$ is generated from $T]$. See extra slides for the statistical model for sequence alignment, and better similarity definition, say BLOSUM62, PAM250 substitution matrix, etc.

Alignment is useful

- Observation 1: Using alignment, we can determine the most likely source of "OCURRANCE".
  1. $T =$"OCCUPATION":

```
S': OC_URRA___NCE
    || |  |   |
T': OCCU_PATION__
```

  $$d(S', T') = 1+1-3+1-3-3-1+1-3-3-3+1-3-3 = -28.$$
  2. $T =$ "OCCURRENCE":

```
S': O_CURRANCE
    | ||||  |||
T': OCCURRENCE
```

  $$d(S', T') = 1-3+1+1+1+1-1+1+1+1 = 4.$$

- Conjecture: it is more likely that "ocurrance" comes from "occurrence" relative to "occupation".

Alignment is useful cont'd

- Observation 2: In addition, we can also determine the most likely operations changing "occurrence" into "ocurrance".
  1. Alignment 1:

```
S': O_CURRANCE
    | |||| |||
T': OCCURRENCE
```

$$d(S', T') = 1 - 3 + 1 + 1 + 1 + 1 - 1 + 1 + 1 + 1 = 4.$$

  2. Alignment 2:

```
S': O_CURR_ANCE
    | ||||  |||
T': OCCURRE-NCE
```

$$d(S', T') = 1 - 3 + 1 + 1 + 1 - 3 - 3 + 1 + 1 + 1 = -1.$$

- Conjecture: the first alignment might describes the real generating process of "ocurrance" from "occurrence".

Key observation I

- It is not easy to consider long sequences directly. Let's consider whether it is possible to reduce into smaller subproblem.
- Solution: alignment of two sequences. Describe the solving process as as a process of multiple-stage **decisions**. At each decision stage, we decide how to align two letters, i.e., how to generate $S[i]$ from $T[j]$.

Pair	Insertion	Deletion
S: $\boxed{\texttt{OCURRANC}}$ E	S: $\boxed{\texttt{OCURRANC}}$ E	S: $\boxed{\texttt{OCURRANCE}}$ -
T: $\boxed{\texttt{OCCURRENC}}$ E	T: $\boxed{\texttt{OCCURRENCE}}$ -	T: $\boxed{\texttt{OCCURRENC}}$ E

Key observation II

- Suppose we have already worked out the optimal solution. Consider the **first** decision made for $S[m]$. There are three cases:
  1. $S[m]$ pairs with $T[n]$, i.e. $S[m]$ comes from $T[n]$. Then it suffices to align $S[1..m-1]$ and $T[1..n-1]$;
  2. $S[m]$ pairs with a space '-', i.e. $S[m]$ is an `INSERTION`. Then we need to align $S[1..m-1]$ and $T[1..n]$;
  3. $T[n]$ pairs with a space '-', i.e. $S[m]$ is a `DELETION` of a letter in T. Then we need to align $S[1..m]$ and $T[1..n-1]$.
- In the three cases, the original problem can be reduced into smaller sub-problems.

Pair	Insertion	Deletion
S: OCURRANC E	S: OCURRANC E	S: OCURRANCE -
T: OCCURRENC E	T: OCCURRENCE -	T: OCCURRENC E

Key observation III

- Thus, we can design the general form of sub-problems as: alignment a **prefix** of S (denoted as $S[1..i]$) and **prefix** of T (denoted as $T[1..j]$). Denote the optimal solution value as $OPT(i, j)$.
- Optimal substructure property:

$$OPT(i, j) = \max \begin{cases} \delta(S_i, T_j) + OPT(i-1, j-1) \\ \delta(`\_', T_j) + OPT(i, j-1) \\ \delta(S_i, `\_') + OPT(i-1, j) \end{cases}$$

Needleman-Wunsch algorithm 1970

NEEDLEMAN_WUNSCH(S, T)

for $i = 0$ to m; **do**
    $OPT[i, 0] = -3 * i$;
end for
for $j = 0$ to n; **do**
    $OPT[0, j] = -3 * j$;
end for
for $i = 1$ to m **do**
    for $j = 1$ to n **do**
        $OPT[i, j] = \max\{OPT[i-1, j-1] + \delta(S_i, T_j), OPT[i-1, j] - 3, OPT[i, j-1] - 3\}$;
    end for
end for
return $OPT[m, n]$;

Note: the first row is introduced to describe the alignment of prefixes $T[1..i]$ with an empty sequence ϵ, so does the first column.

The first row/column of the alignment score matrix

S:	''	O	C	U	R	R	A	N	C	E
T: ''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3									
C	−6									
C	−9									
U	−12									
R	−15									
R	−18									
E	−21									
N	−24									
C	−27									
E	−30									

Score: `d("OCU", "") = -9`
Alignment:
```
S'= OCU
T'= ---
```

Score: `d("", "OC") = -6`
Alignment:
```
S'= --
T'= OC
```

Why should we introduce the first row/column?

S:	''	O	C	U	R	R	A	N	C	E
T: ''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Score: d("OC", "O") = $\max \begin{cases} d(\text{"OC"}, \text{""}) & -3 \quad (=-9) \\ d(\text{"O"}, \text{""}) & -1 \quad (=-4) \\ d(\text{"O"}, \text{"O"}) & -3 \quad (=-2) \end{cases}$

Alignment: S'= OC
T'= O-

T \ S	''	O	C	U	R	R	A	N	C	E
''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Score: d("OCUR", "OC") = $\max \begin{cases} d(\text{"OCUR"}, \text{"O"}) & -3 \quad (=-11) \\ d(\text{"OCU"}, \text{"O"}) & -1 \quad (=-6) \\ d(\text{"OCU"}, \text{"OC"}) & -3 \quad (=-4) \end{cases}$

Alignment:

```
S'= OCUR
T'= OC--
```

The final entry

	S: ''	O	C	U	R	R	A	N	C	E
T: ''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Score: `d("OCURRANCE", "OCCURRENCE") =` $\max \begin{cases} \text{d(“OCURRANCE“, “OCCURRENC“)} & -3 \quad (=\text{-}3) \\ \text{d(“OCURRANC“, “OCCURRENC“)} & +1 \quad (=4) \\ \text{d(“OCURRANC“, “OCCURRENCE“)} & -3 \quad (=\text{-}3) \end{cases}$

Alignment:
```
S'= O-CURRANCE
T'= OCCURRENCE
```

Question: how to find the alignment with the highest score?

Find the optimal alignment via backtracking

T \ S:	''	O	C	U	R	R	A	N	C	E
T: ''	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
O	-3	1	-2	-5	-8	-11	-14	-17	-20	-23
C	-6	-2	2	-1	-4	-7	-10	-13	-16	-19
C	-9	-5	-1	1	-2	-5	-8	-11	-12	-15
U	-12	-8	-4	0	0	-3	-6	-9	-12	13
R	-15	-11	-7	-3	1	1	-2	-5	-8	-11
R	-18	-14	-10	-6	-2	2	$-$	-3	-6	-9
E	-21	-17	-13	-9	-5	-1	1	-1	-4	-5
N	-24	-20	-16	-12	-8	-4	-2	2	-1	-4
C	-27	-23	-19	-15	-11	-7	-5	-1	3	0
E	-30	-26	-22	-18	-14	-10	-8	-4	0	4

Optimal Alignment:

```
S'= O-CURRANCE
T'= OCCURRENCE
```

Optimal alignment versus sub-optimal alignments

- It should be noted that in practice, **sub-optimal alignments (as an ensemble)** are more robust than the optimal alignment due to inaccuracy in the scoring model.
- Please refer to Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids for details.

Space efficient algorithm: reducing the space requirement from $O(mn)$ to $O(m+n)$ (D. S. Hirschberg, 1975)

- Key observation 1: it is easy to calculate **the final score** $OPT(S,T)$ **only!**, i.e. **the alignment information are not recorded.**

T \ S:	''	O	C	U	R	R	A	N	C	E
''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Technique 1: two arrays are enough if only score is needed

- Why? Only column $j-1$ is needed to calculate column i. Thus, we use two arrays $score[1..m]$ and $newscore[1..m]$ instead of the matrix $OPT[1..m, 1..n]$.

T: \ S:	''	O	C	U	R	R	A	N	C	E
''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

- Why? Only column $j-1$ is needed to calculate column i. Thus, we use two arrays $score[1..m]$ and $newscore[1..m]$ instead of the matrix $OPT[1..m, 1..n]$.

S:	''	O	C	U	R	R	A	N	C	E
T: ''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Technique 1: two arrays are enough if only score is needed

- Why? Only column $j-1$ is needed to calculate column i. Thus, we use two arrays $score[1..m]$ and $newscore[1..m]$ instead of the matrix $OPT[1..m, 1..n]$.

T \ S:	''	O	C	U	R	R	A	N	C	E
''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Algorithm

PREFIX_SPACE_EFFICIENT_ALIGNMENT(S, T, SCORE)

for $i = 0$ to m **do**
    $score[i] = -3 * i;$
end for
for $i = 1$ to m **do**
    $newscore[0] = 0;$
    for $j = 1$ to n **do**
        $newscore[j] = \max\{score[j-1] + \delta(S_i, T_j), score[j] - 3, newscore[j-1] - 3\};$
    end for
    for $j = 1$ to n **do**
        $score[j] = newscore[j];$
    end for
end for
return $score[n]$;

Technique 2: aligning suffixes instead of prefixes

- Key observation: Similarly, we can align **suffixes** of S and T instead of **prefixes** and obtain the same score and alignment.

4	0	−4	−10	−12	−16	−18	−22	−26	−30	O
5	3	−1	−7	−9	−13	−15	−19	−23	−27	C
3	6	2	−4	−6	−10	−12	−16	−20	−24	C
−1	2	5	−1	−3	−7	−9	−13	−17	−21	U
−5	−2	1	4	0	−4	−6	−10	−14	−18	R
−9	−6	−3	0	3	−1	−3	−7	−11	−15	R
−13	−10	−7	−4	−1	2	0	−4	−8	−12	E
−15	−12	−9	−6	−3	0	3	−1	−5	−9	N
−19	−16	−13	−10	−7	−4	−1	2	−2	−6	C
−23	−20	−17	−14	−11	−8	−5	−2	1	−3	E
−27	−24	−21	−18	−15	−12	−9	−6	−3	0	''
O	C	U	R	R	A	N	C	E	''	S \ T

Final difficulty: identify optimal alignment besides score

1. However, **only the recent two columns** of the matrix were kept, the optimal alignment cannot be restored via **backtracking**.
2. A clever idea: Suppose we have already obtained the optimal alignment. Consider the position **where $S_{[\frac{m}{2}]}$ is aligned to** (denoted as q). We have

$$OPT(S,T) = OPT(S[1..\frac{m}{2}], T[1..q]) + OPT(S[\frac{m}{2}+1..m], T[q+1..n])$$

3. Notes:
    - Things will be easy as soon as q was determined. The equality holds due to the definition of $d(S,T)$.
    - $\frac{m}{2}$ is chosen for the sake of time-complexity analysis.

$\frac{m}{2}$

S: $\boxed{\texttt{OCUR}}\boxed{\texttt{RANCE}}$

T: $\boxed{\texttt{OCCUR}}\boxed{\texttt{RENCE}}$

$1 \leq q \leq n$

Hirschberg's algorithm for alignment

LINEAR_SPACE_ALIGNMENT(S, T)

Allocate two arrays f and b; each array has a size of m .
PREFIX_SPACE_EFFICIENT_ALIGNMENT$(S[1..\frac{m}{2}], T, f)$;
SUFFIX_SPACE_EFFICIENT_ALIGNMENT$(S[\frac{m}{2}+1..m], T, b)$;
Let $q^* = argmax_q(f[q] + b[q])$;
Free arrays f and b;
Record $< \frac{m}{2}, q^* >$ in array A;
LINEAR_SPACE_ALIGNMENT$(S[1..\frac{m}{2}], T[1..q^*])$;
LINEAR_SPACE_ALIGNMENT$(S[\frac{m}{2}+1..m], T[q^*+1..n])$;
return A;

- Key observation: at each iteration step, only $2n$ space is needed.
- How to determine q? **Identifying the largest entry in $f[q] + b[q]$.**

Step 1: Determine the optimal aligned position of $S_{[\frac{m}{2}]}$

T \ S:	""	O	C	U	R
""	0	-3	-6	-9	-12
O	-3	1	-2	-5	-8
C	-6	-2	2	-1	-4
C	-9	-5	-1	1	-2
U	-12	-8	-4	0	0
R	-15	-11	-7	-3	1
R	-18	-14	-10	-6	-2
E	-21	-17	-13	-9	-5
N	-24	-20	-16	-12	-8
C	-27	-23	-19	-15	-11
E	-30	-26	-22	-18	-14

f

$f+b$
-24
-17
-10
-5
0
4
-3
-8
-15
-22
-29

$f+b$

S:	R	A	N	C	E	""	T
	-12	-16	-18	-22	-26	-30	O
	-9	-13	-15	-19	-23	-27	C
	-6	-10	-12	-16	-20	-24	C
	-3	-7	-9	-13	-17	-21	U
	0	-4	-6	-10	-14	-18	R
	3	-1	-3	-7	-11	-15	R
	-1	2	0	-4	-8	-12	E
	-3	0	3	-1	-5	-9	N
	-7	-4	-1	2	-2	-6	C
	-11	-8	-5	-2	1	-3	E
	-15	-12	-9	-6	-3	0	""

b

The value of the largest item: Recall that 4 is actually the optimal score of S and T.

Step 2: Recursively solve sub-problems

f

T \ S:	""	O	C	U	R
""	0	-3	-6	-9	-12
O	-3	1	-2	-5	-8
C	-6	-2	2	-1	-4
C	-9	-5	-1	1	-2
U	-12	-8	-4	0	0
R	-15	-11	-7	-3	1
R	-18	-14	-10	-6	-2
E	-21	-17	-13	-9	-5
N	-24	-20	-16	-12	-8
C	-27	-23	-19	-15	-11
E	-30	-26	-22	-18	-14

f+b

f+b
-24
-17
-10
-5
0
4
-3
-8
-15
-22
-29

b

R	A	N	C	E	""	T \ S
-12	-16	-18	-22	-26	-30	O
-9	-13	-15	-19	-23	-27	C
-6	-10	-12	-16	-20	-24	C
-3	-7	-9	-13	-17	-21	U
0	-4	-6	-10	-14	-18	R
3	-1	-3	-7	-11	-15	R
-1	2	0	-4	-8	-12	E
-3	0	3	-1	-5	-9	N
-7	-4	-1	2	-2	-6	C
-11	-8	-5	-2	1	-3	E
-15	-12	-9	-6	-3	0	""

The position of the largest item: Generate two independent sub-problems.

Space complexity analysis

The total space requirement: $O(m + n)$.

- $\textsc{Prefix\_Space\_Efficient\_Alignment}(S[1..\frac{m}{2}), T, f)$ needs only $O(n)$ space;
- $\textsc{Suffix\_Space\_Efficient\_Alignment}(S[\frac{m}{2} + 1..m), T, b)$ needs only $O(n)$ space;
- Line 4 (Record $< \frac{n}{2}, q^* >$ in array A) needs only $O(m)$ space;

Time complexity analysis

Theorem

Algorithm LINEAR_SPACE_ALIGNMENT(S, T) *still takes* $O(mn)$ *time.*

Proof.

- The algorithm implies the following recursion:
  $T(m,n) = cmn + T(q, \frac{n}{2}) + T(m-q, \frac{n}{2})$;
- Difficulty: we have no idea of q before algorithm ends; thus, the master theorem cannot apply directly. **Guess and substitution!!!**
- Guess: $T(m', n') \leq km'n'$ follows for any $m' < m$ and $n' < n$.
- Substitution:

$$\begin{aligned}
T(m,n) &= cmn + T(q, \frac{n}{2}) + T(m-q, \frac{n}{2}) && (7)\\
&\leq cmn + kq\frac{n}{2} + k(m-q)\frac{n}{2} && (8)\\
&= cmn + kq\frac{n}{2} + km\frac{n}{2} - kq\frac{n}{2} && (9)\\
&\leq (c + \frac{k}{2})mn && (10)\\
&= kmn \qquad (set\ k = 2c) && (11)
\end{aligned}$$

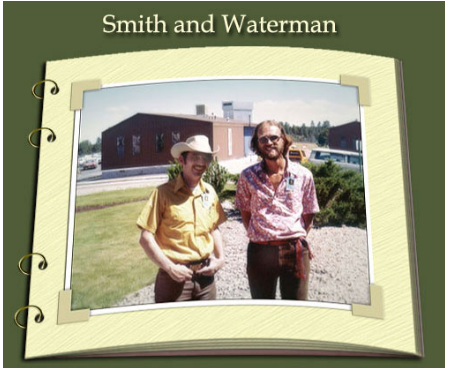
Extended Reading 1: From global alignment to local alignment

From Global alignment to Local alignment: Smith-Waterman algorithm

- Global alignment: to identify similarity between **two whole sequences;**
- Local alignment: It is often that we wish to find **similar SEGMENTS (sub-sequences)**.

Smith-Waterman algorithm [1981]

- Needleman-Wunsch **global alignment** algorithm was developed by biologists in 1970s, about twenty years later than Bellman-Ford algorithm was developed.
- Then Smith-Waterman **local alignment** algorithm was proposed.



Please refer to Smith and Waterman1981 for details.

Extended Reading 2: How to derive a reasonable scoring schema?

PAM250: one of the most popular substitution matrices in Bioinformatics

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	2	-2	0	0	-2	0	0	1	-1	-1	-2	-1	-1	-3	1	1	1	-6	-3	0	0	0	0	-8
R	-2	6	0	-1	-4	1	-1	-3	2	-2	-3	3	0	-4	0	0	-1	2	-4	-2	-1	0	-1	-8
N	0	0	2	2	-4	1	1	0	2	-2	-3	1	-2	-3	0	1	0	-4	-2	-2	2	1	0	-8
D	0	-1	2	4	-5	2	3	1	1	-2	-4	0	-3	-6	-1	0	0	-7	-4	-2	3	3	-1	-8
C	-2	-4	-4	-5	12	-5	-5	-3	-3	-2	-6	-5	-5	-4	-3	0	-2	-8	0	-2	-4	-5	-3	-8
Q	0	1	1	2	-5	4	2	-1	3	-2	-2	1	-1	-5	0	-1	-1	-5	-4	-2	1	3	-1	-8
E	0	-1	1	3	-5	2	4	0	1	-2	-3	0	-2	-5	-1	0	0	-7	-4	-2	3	3	-1	-8
G	1	-3	0	1	-3	-1	0	5	-2	-3	-4	-2	-3	-5	0	1	0	-7	-5	-1	0	0	-1	-8
H	-1	2	2	1	-3	3	1	-2	6	-2	-2	0	-2	-2	0	-1	-1	-3	0	-2	1	2	-1	-8
I	-1	-2	-2	-2	-2	-2	-2	-3	-2	5	2	-2	2	1	-2	-1	0	-5	-1	4	-2	-2	-1	-8
L	-2	-3	-3	-4	-6	-2	-3	-4	-2	2	6	-3	4	2	-3	-3	-2	-2	-1	2	-3	-3	-1	-8
K	-1	3	1	0	-5	1	0	-2	0	-2	-3	5	0	-5	-1	0	0	-3	-4	-2	1	0	-1	-8
M	-1	0	-2	-3	-5	-1	-2	-3	-2	2	4	0	6	0	-2	-2	-1	-4	-2	2	-2	-2	-1	-8
F	-3	-4	-3	-6	-4	-5	-5	-5	-2	1	2	-5	0	9	-5	-3	-3	0	7	-1	-4	-5	-2	-8
P	1	0	0	-1	-3	0	-1	0	0	-2	-3	-1	-2	-5	6	1	0	-6	-5	-1	-1	0	-1	-8
S	1	0	1	0	0	-1	0	1	-1	-1	-3	0	-2	-3	1	2	1	-2	-3	-1	0	0	0	-8
T	1	-1	0	0	-2	-1	0	0	-1	0	-2	0	-1	-3	0	1	3	-5	-3	0	0	-1	0	-8
W	-6	2	-4	-7	-8	-5	-7	-7	-3	-5	-2	-3	-4	0	-6	-2	-5	17	0	-6	-5	-6	-4	-8
Y	-3	-4	-2	-4	0	-4	-4	-5	0	-1	-1	-4	-2	7	-5	-3	-3	0	10	-2	-3	-4	-2	-8
V	0	-2	-2	-2	-2	-2	-2	-1	-2	4	2	-2	2	-1	-1	-1	0	-6	-2	4	-2	-2	-1	-8
B	0	-1	2	3	-4	1	3	0	1	-2	-3	1	-2	-4	-1	0	0	-5	-3	-2	3	2	-1	-8
Z	0	0	1	3	-5	3	3	0	2	-2	-3	0	-2	-5	0	0	-1	-6	-4	-2	2	3	-1	-8
X	0	-1	0	-1	-3	-1	-1	-1	-1	-1	-1	-1	-1	-2	-1	0	0	-4	-2	-1	-1	-1	-1	-8
*	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	1

Please refer to “PAM matrix for Blast algorithm” (by C. Alexander, 2002) for the details to calculate PAM matrix.

Extended Reading 3: How to measure the significance of an alignment?

Measure the significance of a segment pair

- When two random sequences of length m and n are compared, the probability of finding a pair of segments with a score greater than or equal to S is $1 - e^{-y}$, where $y = Kmne^{-\lambda S}$.

Please refer to Altschul1990 for details.

Extended Reading 4: An FPGA implementation of Smith-Waterman algorithm

The potential parallelity of SmithWaterman algorithm

	-	S1	S2	S3	S4	S5	...
-	0	0	0	0	0	0	
T1	0	1	2	3	4	5	6
T2	0	2	3	4	5	6	7
T3	0	3	4	5	6	7	8
T4	0	4	5	6	7	8	9
T5	0	5	6	7	8	9	10
...		6	7	8	9	10	

For example, in the first cycle, only one element marked as (1) could be calculated. In the second cycle, two elements marked as (2) could be calculated. In the third cycle, three elements marked as (3) could be calculated, etc., and this feature implies that the algorithm has a very good potential parallelity.

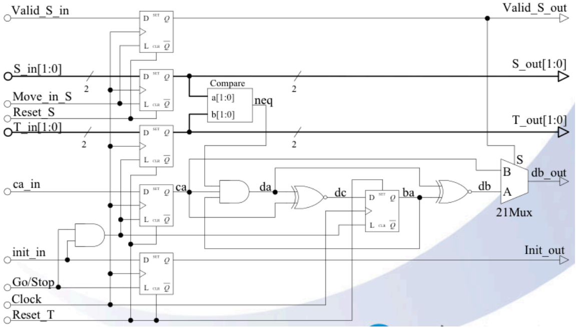
Mapping Smithg-Waterman algorithm on PE

PE Array	PE1	PE2	PE3	PE4	PE5	...

	-	S1	S2	S3	S4	S5	...
-	0	0	0	0	0	0	
T1	0	1	2	3	4	5	6
T2	0	2	3	4	5	6	7
T3	0	3	4	5	6	7	8
T4	0	4	5	6	7	8	9
T5	0	5	6	7	8	9	10
...		6	7	8	9	10	

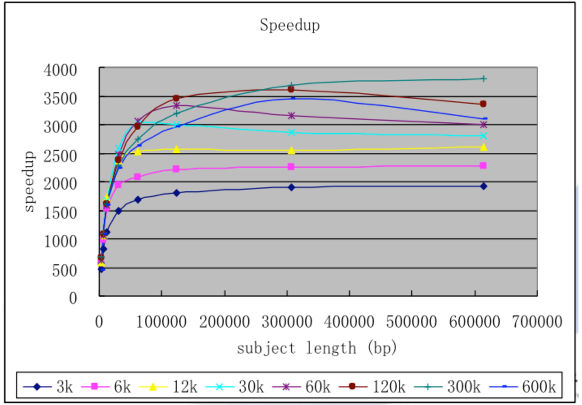
See *Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform* for details.

PE design of a card for Dawning 4000L



Smith-Waterman card for Dawning 4000L

Performance of Dawning 4000L

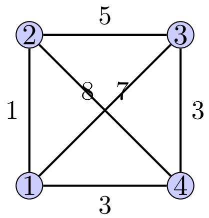


4

[4] Some pictures were excerpted from *Introduction to algorithms*

BELLMAN-HELD-KARP for TSP problem: recursion over **graphs**

INPUT: a list of n cities (denoted as V), and the distances between each pair of cities d_{ij} $(1 \leq i, j \leq n)$;
OUTPUT: the shortest tour that visits each city exactly once and returns to the origin city

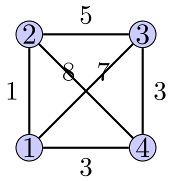


#Tours: 6

- Tour 1: $1 \to 2 \to 3 \to 4 \to 1$ (12)
- Tour 2: $1 \to 2 \to 4 \to 3 \to 1$ (21)
- Tour 3: $1 \to 3 \to 2 \to 4 \to 1$ (23)
- ....

Definition

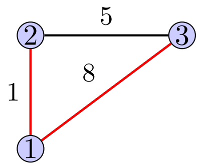
$D(S, e)$ = the minimum distance, starting from city 1, visiting all cities in S, and finishing at city $e \in S$.



- It suffices to calculate $D(S, e)$ for any $S \in \{1, 2, ..., n\}$ and city e since:
  - There are 3 cases of the city from which we return to 1.
  - Thus, the shortest tour can be calculated as:
    $\min\{D(\{1, 2, 3, 4\}, 2) + d_{2,1},$
    $D(\{1, 2, 3, 4\}, 3) + d_{3,1},$
    $D(\{1, 2, 3, 4\}, 4) + d_{4,1}\}$

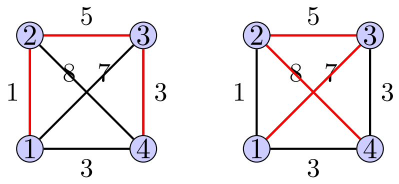
Let's start from the smallest problem

- It is trivial to calculate $D(S, e)$ when S consists of only 1 cities.



- $D(\{2\}, 2) = d_{12}$;
- $D(\{3\}, 3) = d_{13}$;
- But how to solve a large problem, say $D(\{2, 3, 4\}, 4)$?

Divide a larger problem into smaller problems



- $D(\{2,3,4\},4) = \min\{D(\{2,3\},3) + d_{34}, D(\{2,3\},2) + d_{24}\}$;
- Optimal substructure property:
  $D(S,e) =$
  $$\begin{cases} d_{1e} & \text{if } S = \{e\} \\ \min_{m \in S - \{e\}} (D(S - \{e\}, m) + d_{me}) & \text{otherwise} \end{cases}$$

Bellman-Held-Karp algorithm [1962]

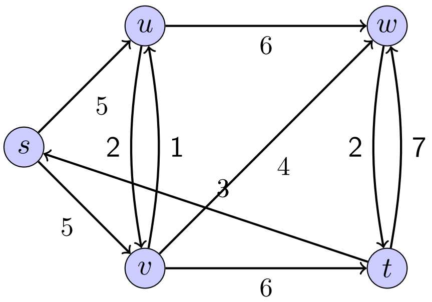
function $D(S, e)$

if $S = \{e\}$ **then**
    return d_{1e};
end if
$d = \infty$;
for all city $m \in S$, and $m \neq e$ **do**
    if $D(S - \{e\}, m) + d_{me} < d$ **then**
        $d = D(S - \{e\}, m) + d_{me}$;
    end if
end for
return d;

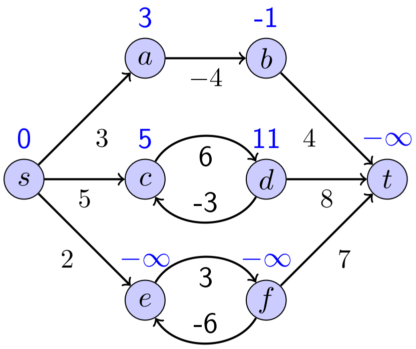
- Space complexity: $\sum_{k=2}^{n-1} k\binom{n-1}{k} + n - 1 = (n-1)2^{n-2}$
- Time complexity: $\sum_{k=2}^{n-1} k(k-1)\binom{n-1}{k} + n - 1 = O(2^n n^2)$.

SingleSourceShortestPath problem: recursion over **graphs**

INPUT:
A directed graph $G =< V, E >$. Each edge $e =< i, j >$ has a weight or distance $d(i, j)$. Two special nodes: source s, and destination t;
OUTPUT:
A shortest path from s to t; that is, the sum weight of the edges is minimized.



ShortestPath problem: cycles



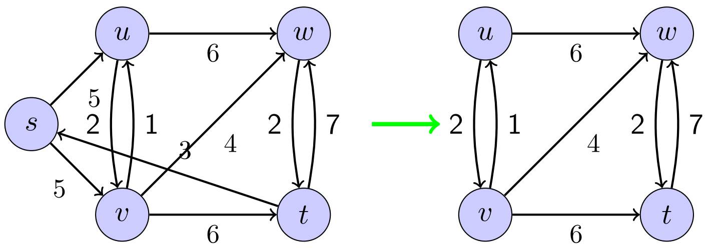
- Here $d(i, j)$ might be negative; however, there should be **no negative cycle**, i.e. the sum weight of edges in any cycle should be greater than 0.
- In fact, a negative cycle means an $-\infty$ shortest-path weight. Since e and f form a negative-weight cycle reachable from s, they have shortest-path weight of $-\infty$ from s.

Trial 1: describing the sub-problem as finding the shortest path in a graph

Trial 1: recursion over graphs

- Solution: a path. Describe the solving process as series of decisions. At each decision stage, we need to **determine an edge** to the subsequent node.
- Suppose we have already worked out the optimal solution O.
- Consider **the first decision** in O. The options are:
  - All edges starting from s: Suppose we use an edge $< s, v >$. Then it suffices to calculate the shortest path in graph $G' =< V', E' >$, where node s and related edges are removed.
- Thus, the original problem can be reduced into smaller sub-problems.
- General form of sub-problem: to find the shortest path from node v to t in graph G.

- General form of sub-problem: to find the shortest path from node v to t in graph G. Denote the optimal solution value as $OPT(G, v)$.
- Optimal substructure: $OPT(G, s) = \min_{v:<s,v>\in E}\{OPT(G', v) + d(s, v)\}$



- **Infeasible!** The number of sub-problems is **exponential**.

Trial 2: another problem form with a new variable

Trial 2: simplifying sub-problem form via limiting path length

- Solution: the shortest path from node s to t is a path with **at most n nodes** (Why? no negative cycle $\Rightarrow$ removing cycles in a path can shorten the path). Describe the solving process as a process of multiple-stage **decisions**; at each decision stage, we decide **the subsequent node** from current node.
- Suppose we have already worked out the optimal solution O.
- Consider the first decision in O. The feasible options are:
  - All adjacent nodes of s: Suppose we choose an edge $< s, v >$ to node v. Then the left-over is to find the shortest path from v to t via at most $n-2$ edges.
- Thus the general form of subproblem can be designed as: to find the shortest path from node v to t with **at most** k edges ($k \leq n-1$). Denote the optimal solution value as $OPT(v, t, k)$.
- Optimal substructure:

$$OPT[v, t, k] = \min \begin{cases} OPT[v, t, k-1], \\ \min_{<v,w>\in E}\{OPT[w, t, k-1] + d(v, w)\} \end{cases}$$

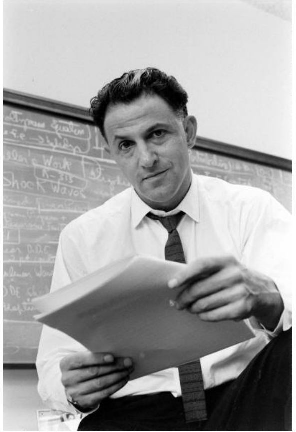
Bellman-Ford algorithm [1956, 1958]

BELLMAN_FORD(G, s, t)

for any node $v \in V$ **do**
  $OPT[v, t, 0] = \infty;$
end for
for $k = 0$ to $n - 1$ **do**
  $OPT[t, t, k] = 0;$
end for
for $k = 1$ to $n - 1$ **do**
  for all node v (in an arbitrary order) **do**
    $OPT[v, t, k] =$
    $$\min \begin{cases} OPT[v, t, k-1] \\ \min_{<v,w>\in E}\{OPT[w, t, k-1] + d(v, w)\} \end{cases}$$
  end for
end for
return $OPT[s, t, n-1];$

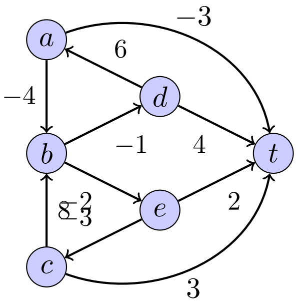
Note that the algorithm actually finds the shortest path from every possible source to t (or from s to every possible destination) by constructing a shortest path tree.

Richard Bellman

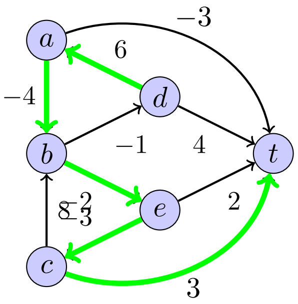


See "Richard Bellman on the birth of dynamic programming" (S. Dreyfus, 2002) and "On the routing problem" (R. Bellman, 1958) for details.

An example



Source node	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
t	0	0	0	0	0	0
a	-	-3	-3	-4	-6	-6
b	-	-	0	-2	-2	-2
c	-	3	3	3	3	3
d	-	4	3	3	2	0
e	-	2	0	0	0	0



Source node	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
t	0	0	0	0	0	0
a	-	-3	-3	-4	-6	-6
b	-	-	0	-2	-2	-2
c	-	3	3	3	3	3
d	-	4	3	3	2	0
e	-	2	0	0	0	0

Note: the shortest paths from all nodes to t form a *shortest path tree*.

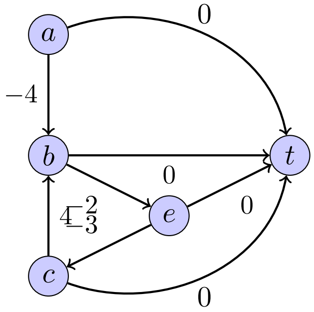
Time complexity

1. Cursory analysis: $O(n^3)$. (There are n^2 subproblems, and for each subproblem, we need at most $O(n)$ operations in line 7.
2. Better analysis: $O(mn)$. (Efficient for sparse graph, i.e. $m << n^2$.)
   - For each node v, line 7 need $O(d_v)$ operations, where d_v denotes the degree of node v;
   - Thus **the inner for loop** (lines 6-8) needs $\sum_v d_v = O(m)$ operations;
   - Thus **the outer for loop** (lines 5-9) needs $O(nm)$ operations.

Extension: detecting negative cycle

Theorem

If t is reachable from node v, and v is contained in a negative cycle, then we have: $\lim_{k\to\infty} OPT(v,t,k) = -\infty$.



Intuition: a traveling of the negative cycle leads to a shorter length. Say,

$length(b \to t) = 0$

$length(b \to e \to c \to b \to t) = -1$

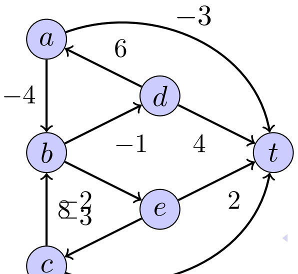
$length(b \to e \to c \to b \to e \to c \to b \to t) = -2$

$\ldots\ldots$

Corollary

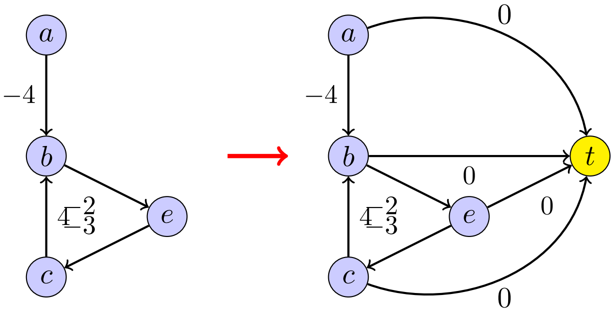
If there is no negative cycle in G, then for all node v, and $k \geq n$, $OPT(v, t, k) = OPT(v, t, n)$.

Source node	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9	k=10	k=11
t	0	0	0	0	0	0	0	0	0	0	0	0
a	-	-3	-3	-4	-6	-6	-6	-6	-6	-6	-6	-6
b	-	-	0	-2	-2	-2	-2	-2	-2	-2	-2	-2
c	-	3	3	3	3	3	3	3	3	3	3	3
d	-	4	3	3	2	0	0	0	0	0	0	0
e	-	2	0	0	0	0	0	0	0	0	0	0

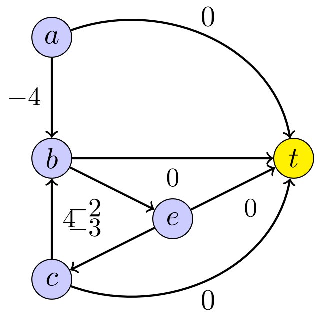


Detecting negative cycle via adding edges and a node t

- Expanding G to G' to guarantee that t is reachable from the negative cycle:
  1. Adding a new node t;
  2. For each node v, adding a new edge $< v, t >$ with $d(v, t) = 0$;
- Property: G has a negative cycle C, say, $b \rightarrow e \rightarrow c \rightarrow b) \Rightarrow t$ is reachable from a node in C. Thus, the above theorem applies.



An example of negative cycle



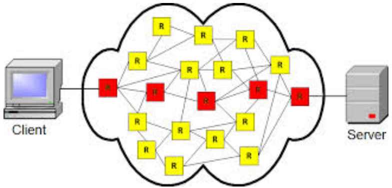
Source node	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	$\cdots$
t	0	0	0	0	0	0	0	0	0	$\cdots$
a	-	0	-4	-6	-9	-9	-11	-11	-12	$\cdots$
b	-	0	-2	-5	-5	-7	-7	-8	-8	$\cdots$
c	-	0	0	0	-2	-3	-3	-3	-4	$\cdots$
e	-	0	-3	-3	-5	-5	-6	-6	-6	$\cdots$

Application of Bellman-Ford algorithm: Internet router protocol

Internet router protocol

Problem statement:

- Each node denotes a route, and the weight denotes the **transmission delay** of the link from router i to j.
- The objective to design a protocol to determine the quickest route when router s wants to send a package to t.



Internet router protocol: Dijkstra's algo vs. Bellman-Ford algo

- Choice: Dijkstra algorithm.
- However, the algorithm needs **global knowledge**, i.e. the knowledge of the whole graph, which is (almost) impossible to obtain.
- In contrast, the Bellman-Ford algorithm **needs only local information**, i.e. the information of its **neighboorhood** rather than **the whole network.**

Application: Internet router protocol

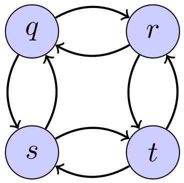
AsynchronousShortestPath(G, s, t)

Initially, set $OPT[t,t] = 0$, and $OPT[v,t] = \infty$;
Label node t as "active";
while exists an active node **do**
    arbitrarily select an active node w;
    remove w's active label;
    for all edges $< v, w >$ (in an arbitrary order) **do**
        $OPT[v,t] = \min \begin{cases} OPT[v,t] \\ OPT[w,t] + d(v,w) \end{cases}$
        if $OPT[v,t]$ was updated **then**
            label v as "active";
        end if
    end for
end while

A related problem: LONGESTPATH problem

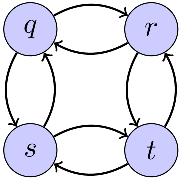
LongestPath problem

INPUT:
A directed graph $G =< V, E >$. Each edge (u, v) has a distance $d(u, v)$. Two nodes s and t;
OUTPUT:
The longest simple path from s to t;



Hardness: LongestPath problem is NP-hard. (Hint: it is obvious that LongestPath problem contains $Hamilton$ path as its special case.)

- Divide: Wrong! The subproblems are not independent.
- Consider dividing problem to find a path from q to t into two subproblems: to find a path from q to r, and to find a path from r to t.



- Suppose we have already solved the sub-problems. Let's try to combine the solutions to the two sub-problems:
  1. $P(q, r) = q \to s \to t \to r$
  2. $P(r, t) = r \to q \to s \to t$,

  We will obtain a path $q \to s \to t \to r \to q \to s \to t$, which is not simple.

Subtlety: LONGESTPATH problem II

- In other words, the use of s in the first subproblem prevents us from using s in the second subproblem. However, we cannot obtain the optimal solution to the second subproblem without using s.

- In contrast, the SHORTESTPATH does not have this difficulty.
- Why? The solutions to the subproblems **share no node**. Suppose the shortest paths $P(q, r)$ and $P(r, t)$ share a node $w (w \neq r)$. Then there will be a cycle $w \to \cdots \to r \to \cdots \to w$. Removing this cycle leads to a shorter path (no negative cycle). A contradiction.
- This means that the two subproblems are independent: the solution of one subproblem does not affect the solution to another subproblem.

A greedy algorithm exists when posing a stricter limit, i.e., all edges have a positive weight.

We will talk about this in next lectures.