

Containers

Stanislav Protasov

Agenda

- Lists, Sorted list
 - Unrolled linked list
 - Skip list
- Set, Sorted set
 - Search trees
- Persistent data structures
 - Techniques
 - V-List

Abstract data types and their implementations

List

List

Countable number of ordered non-unique values. Finite sequence.

	ArrayList	LinkedList
creating an empty list		
testing a list is empty		
prepend an entity		
append an entity		
determining the "head" of a list		
accessing the element at a given index		

List

Countable number of ordered non-unique values. Finite sequence.

	ArrayList	LinkedList
creating an empty list	O(1)	O(1)
testing a list is empty	O(1)	O(1)
prepend an entity / inserting at position	O(N)	O(1)
append an entity	O(N), O _A (1)	O(1)
determining the "head" of a list	O(1)	O(1)
accessing the element at a given index	O(1)	O(N)

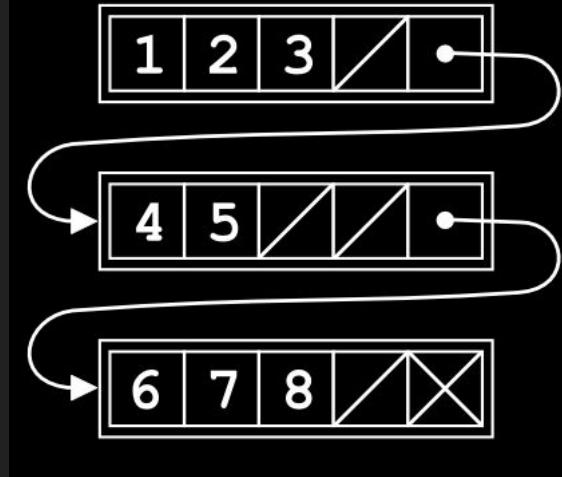
Unrolled *linked list* (1994)

Increases cache performance

Decreases memory overhead

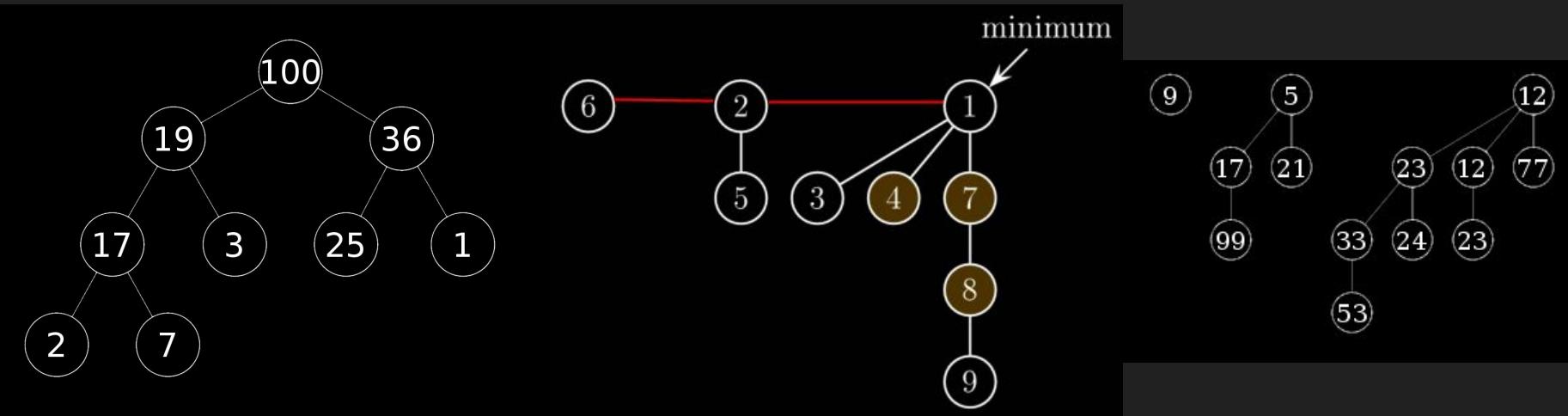
Ideas:

- Nodes capacity is constant k . Thus, **search is $O(n/k)$** .
- Each node stores it's size and \sim cache line of data.
- Nodes are preserved at worst case half-full.
 - On insert overflow: **split** a node into 2 of $(k/2)+1$ and $(k/2)$ in $+O(k)$.
 $O(n/k + k)$ for insert
 - On delete: either **steal** from the next or **merge** is possible in $+O(k)$.
 $O(n/k + k)$ for insert



Priority queue and Sorted list

Heaps (priority queues)



Sorted list: new requirements for the list

1. Fast “TOP N” operation (including peak() and pop())
2. Sublinear **search(x)**
 - a. Sublinear insert
 - b. Sublinear delete

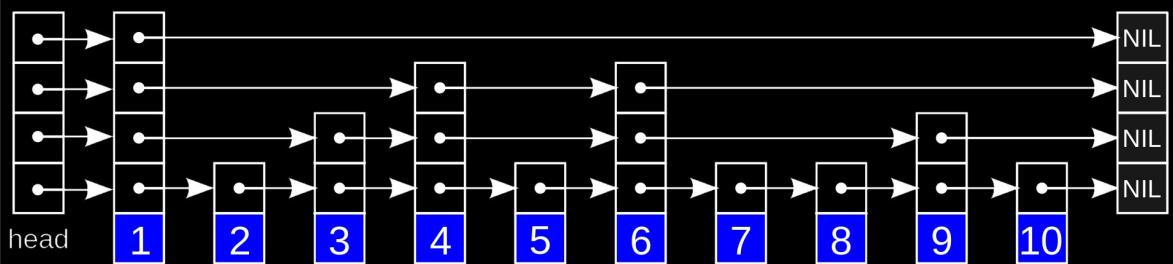
	ArrayList	LinkedList
Search		
Insert		
Delete		

Sorted list: new requirements for the list

1. Fast “**TOP N**” operation (including peak() and pop())
2. Sublinear **search(x)**
 - a. Sublinear insert
 - b. Sublinear delete

	Array List	Linked List
Search	$O(\log(N))$	$O(N)$
Insert	$O(N)$	$O(N)$
Delete	$O(N)$	$O(N)$

Skip List



- 1) Based on the linked list
 - a) Instead of `Node* next` it has `Node*[LEVELS] next;`
- 2) Introduces idea of search “shortcuts”
- 3) **Probabilistic insertion algorithm**
 - a) Insert into basic linked list
 - b) Increment a level. If maximum -
 - c) Toss a coin ($p = 0.5$ or any other). **If “win” - goto (b)**
- 4) **Expected** search time for a list with n elements: $T_E(n) = \frac{1}{p} \log_{1/p} n$
 - a) if `node.next[level]` is null or `node.next[max_level].value > x`
 - i) then: `level--` (or return None on level 0)
 - ii) else: `node = node.next[level]`

Set, multiset, map and sorted

Set+

- 1) Basic operations
 - a) Union, Intersection, Difference, IsSubset

	HashTable	Union-Find
Union (n, k)		
Intersection (n, k)		
Difference (n, k)		
IsSubsetOf (n, k)		

Set+

- 1) Basic operations
 - a) Union, Intersection, Difference, IsSubset
- 2) IsElementOf
- 3) Iterate/Enumerate*, **
- 4) Add(x), Remove(x)

	HashTable	Union-Find
Union (n, k)	$O(\min(n+k))$	$O(1)$
Intersection (n, k)	$O(\min(n, k))^*$	$O(\min(n,k))^{**}$
Difference (n, k)	$O(n+k)^*$	$O(n+k)^{**}$
IsSubsetOf (n, k)	$O(n)^*$	$O(n)^{**}$

	HashTable	Union-Find
IsElementOf		
Iterate		
Add		
Remove		

Set+

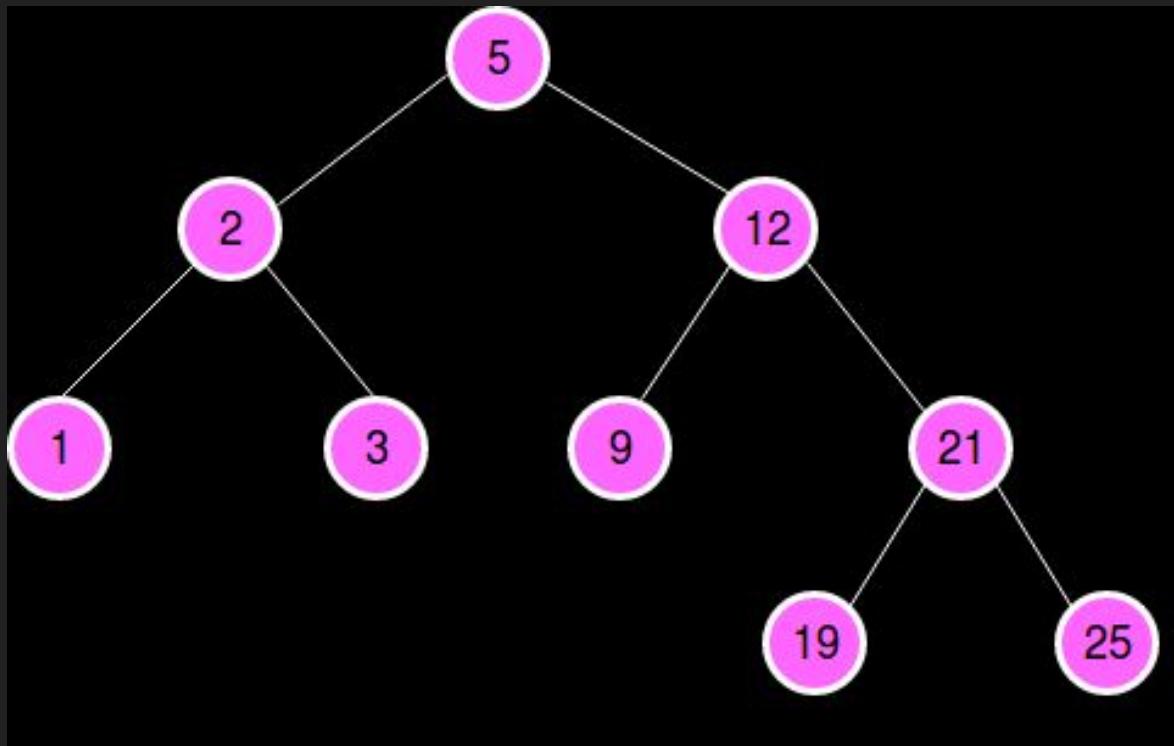
- 1) Basic operations
 - a) Union, Intersection, Difference, IsSubset
- 2) IsElementOf
- 3) Iterate/Enumerate*, **
- 4) Add(x), Remove(x)

	HashTable	Union-Find
Union (n, k)	$O(\min(n+k))$	$O(1)$
Intersection (n, k)	$O(\min(n, k))^*$	$O(\min(n,k))^{**}$
Difference (n, k)	$O(n+k)^*$	$O(n+k)^{**}$
IsSubsetOf (n, k)	$O(n)^*$	$O(n)^{**}$

	HashTable	Union-Find
IsElementOf	$O(1)$	$O(1)$
Iterate	$O(n)^*$	--
Add	$O_A(1)$	$O(1)$
Remove	$O(1)$	--

Sorted sets: [binary] search trees

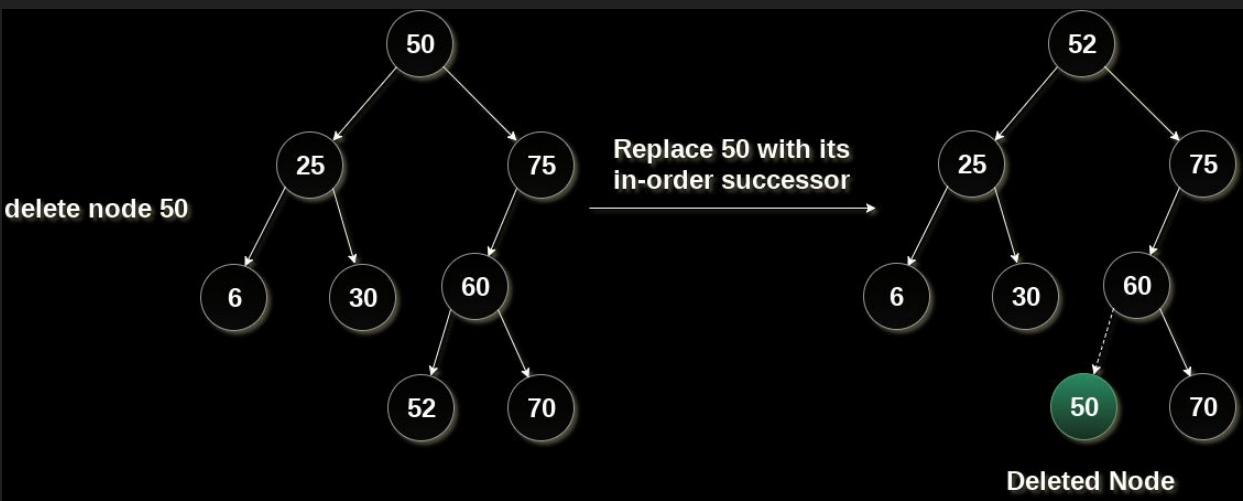
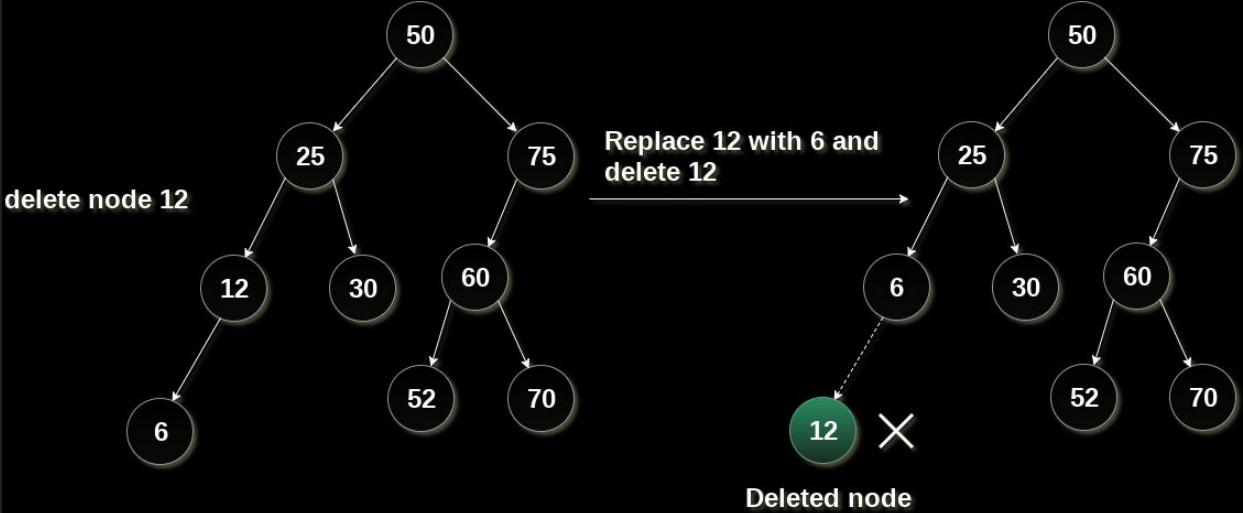
Idea



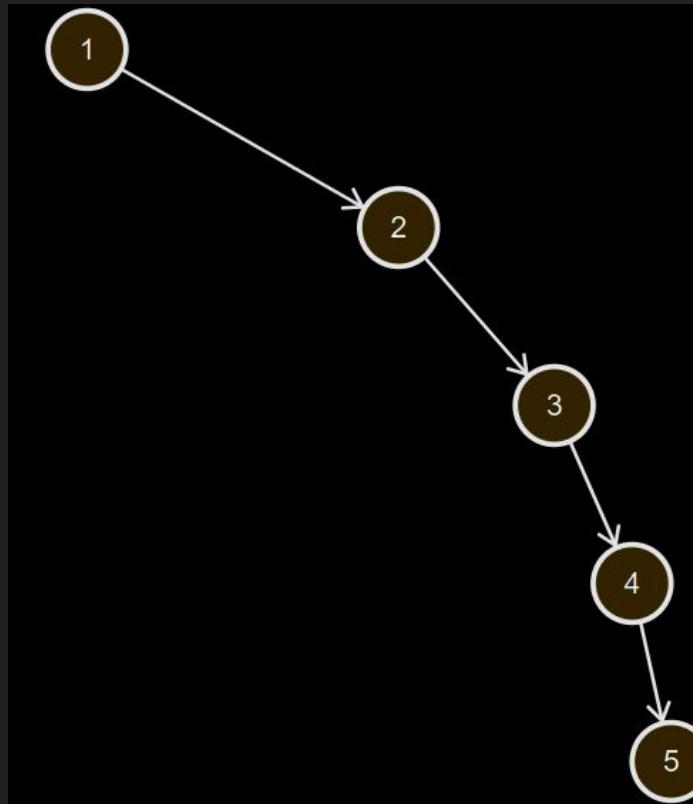
Non trivial delete

- 1) Find
- 2) If no ancestors - trivial
- 3) Single child - pull it up
- 4) 2 children

a) Find pre/in-order ancestor and replace with it



Skewed binary tree



Balanced (self-balancing) binary search trees

AVL-tree (Adelson-Velskii, Landis, 1962)

Restriction: subtrees height differ by not more than 1.

$$|h(\text{left}) - h(\text{right})| \leq 1$$

Thus, worst case is:

$$n_0 = 0,$$

$$n_1 = 1,$$

$$n_h = n_{h-2} + n_{h-1} + 1.$$

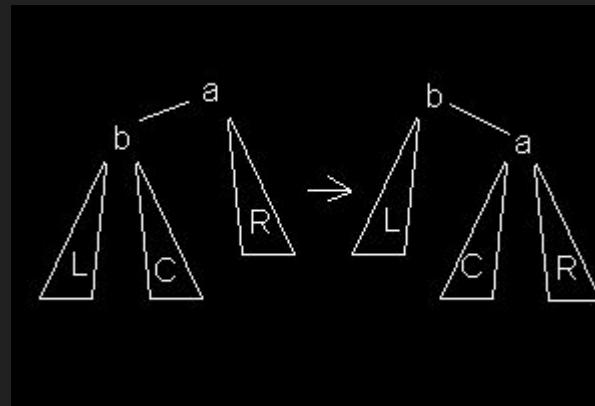
$$N_h = \Phi_{h+2} - 1$$

$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rfloor$$

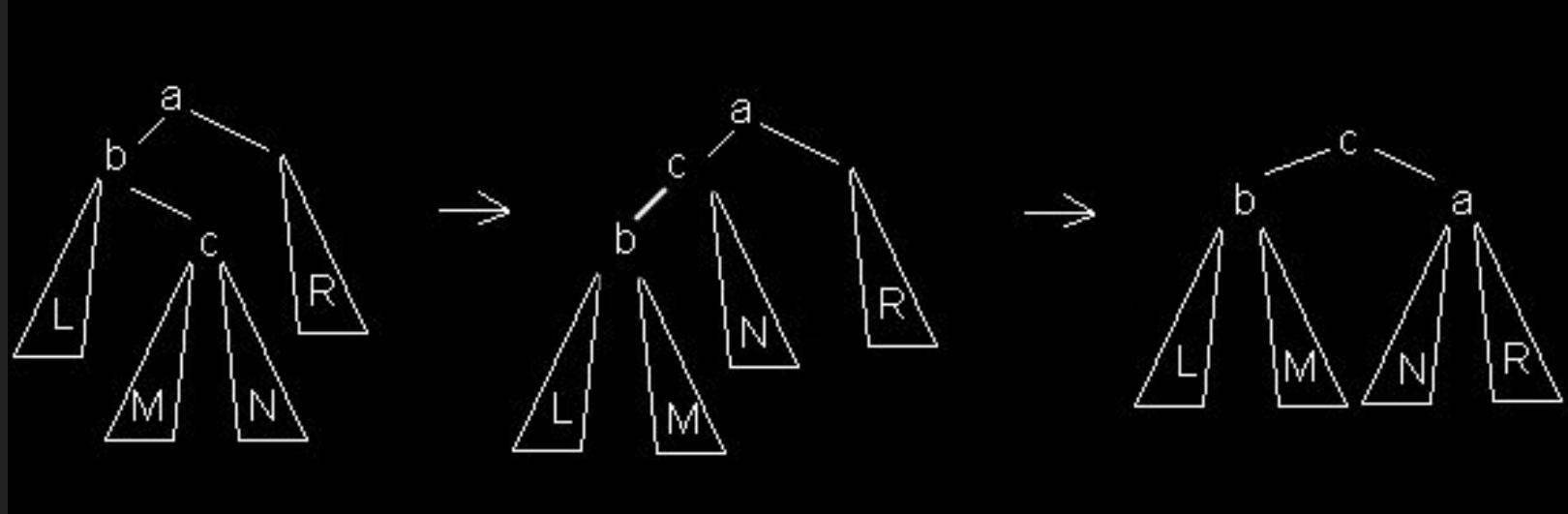
AVL Operations

Search - trivial BT search

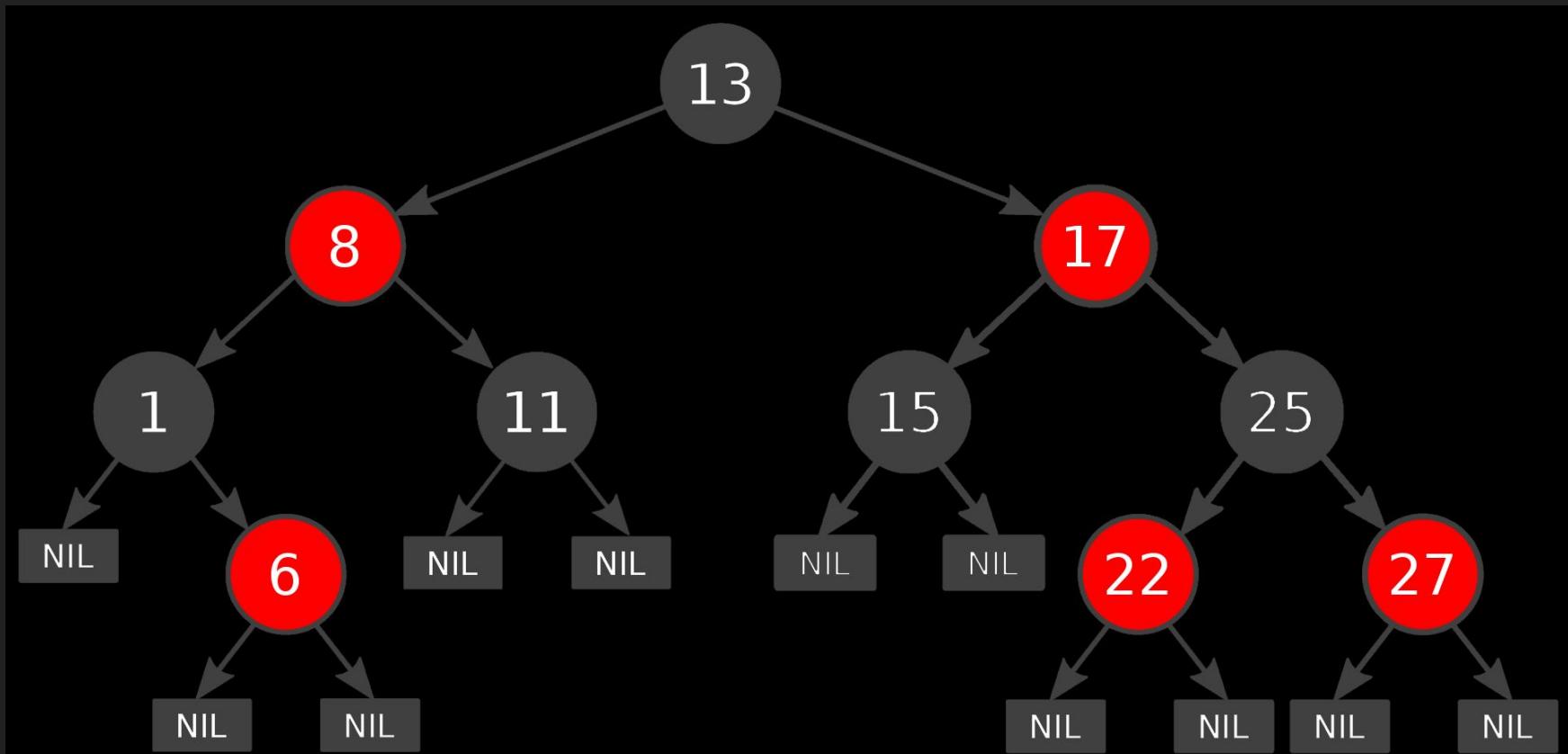
Removal, Insertion:



Left and right ***rotations and “big rotations”***



Red-Black Trees (Bayer, 1972)



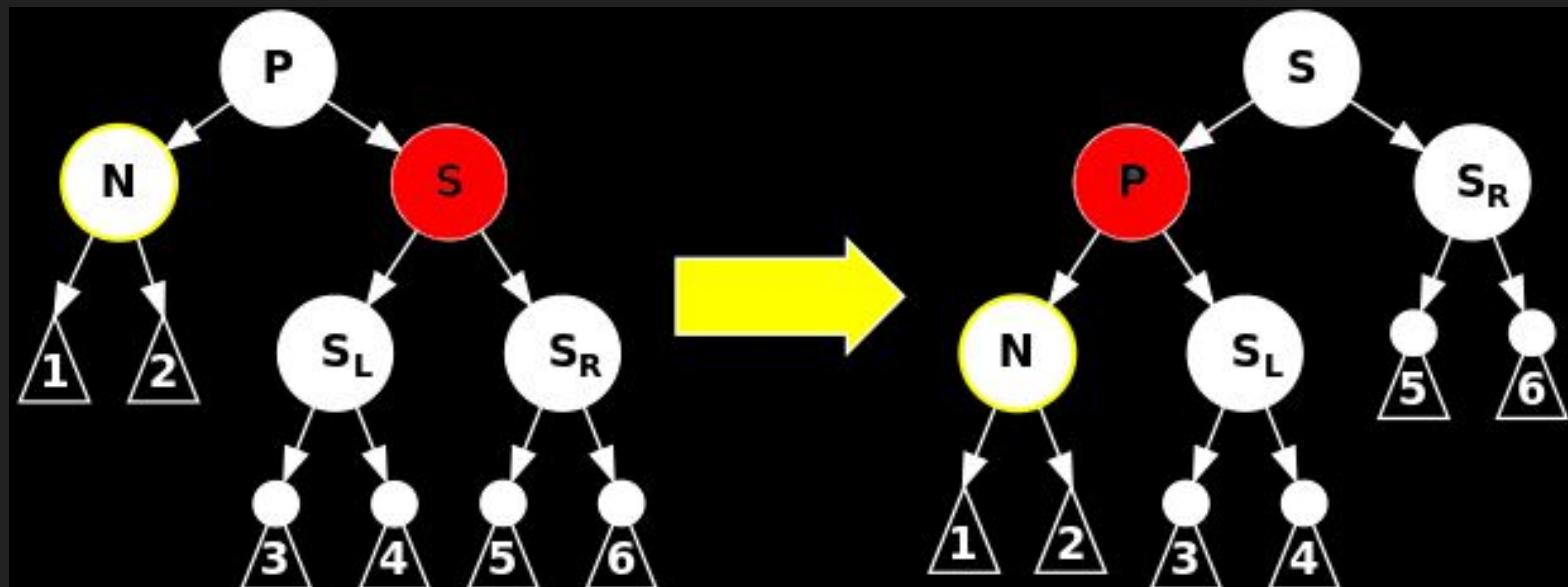
RB-tree restrictions

1. Each node is either **red** or **black**.
2. The **root is black**. This rule is sometimes omitted. Since the root can always be changed from red to black
3. All **leaves (NIL)** are black.
4. If a node is **red**, then both its **children** are black.
5. Every **path** from a given node to any of its descendant NIL nodes contains the **same number of black nodes**.

$$n \geq 2^{\frac{h(\text{root})}{2}} - 1 \leftrightarrow \log_2(n+1) \geq \frac{h(\text{root})}{2} \leftrightarrow h(\text{root}) \leq 2 \log_2(n+1).$$

RB-trees: techniques

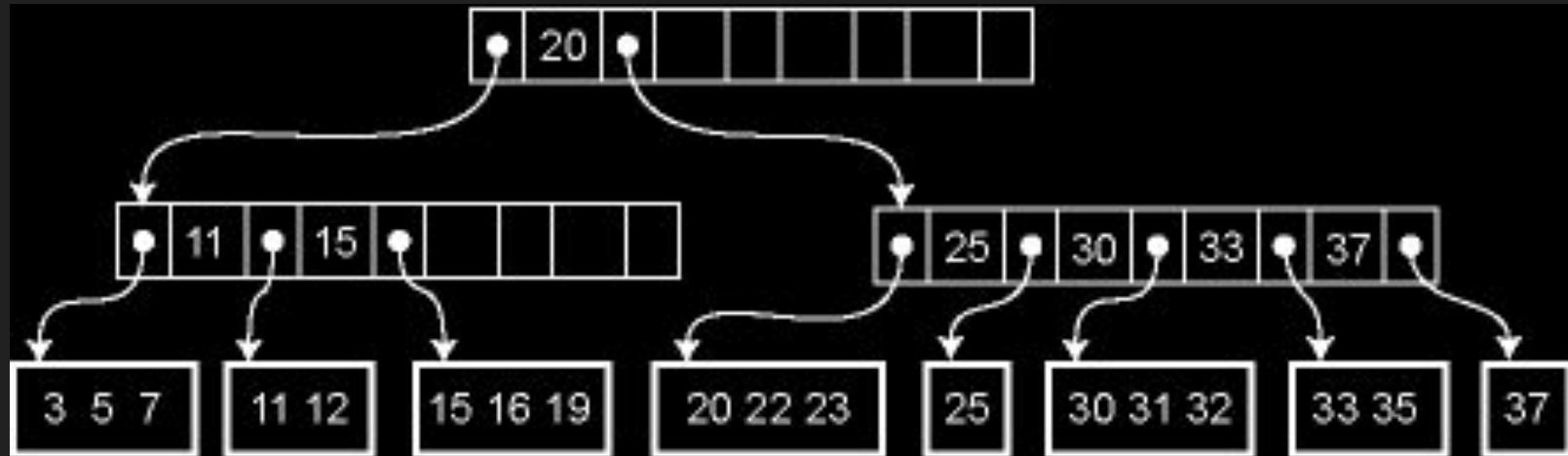
Restoring RB-properties (rotations, restructuring) requires $O(\log(N))$ or $O_A(1)$



Non-binary search tree: B-trees

Takes best from Unrolled linked lists and search trees:

1. Cache and HDD friendly
2. Minimizes restructuring (SPLIT, MERGE, BORROW)
3. Modifications are used in file systems and databases



Persistent lists

Techniques

Copy on Write

Fat node

Path copying

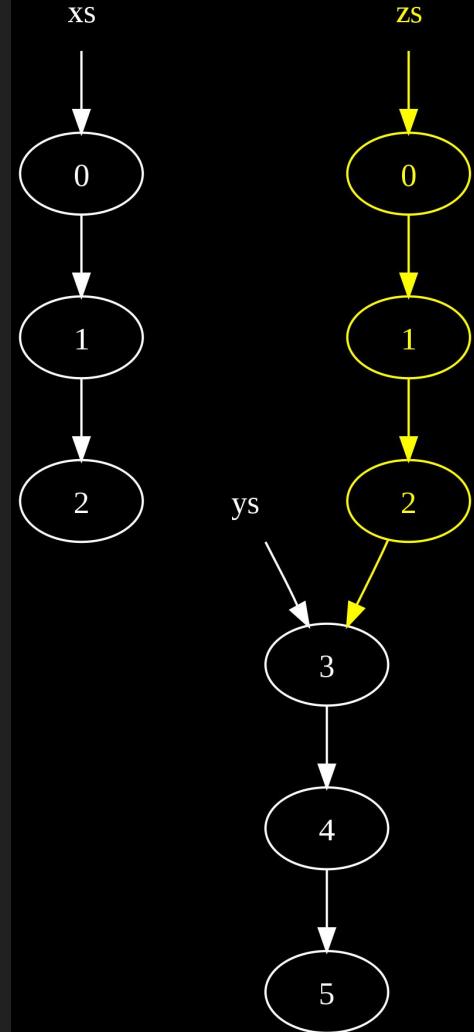
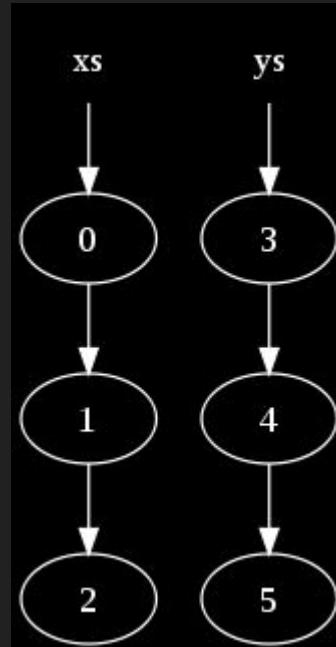
A combination fat node and path copying

Persistent singly-linked list

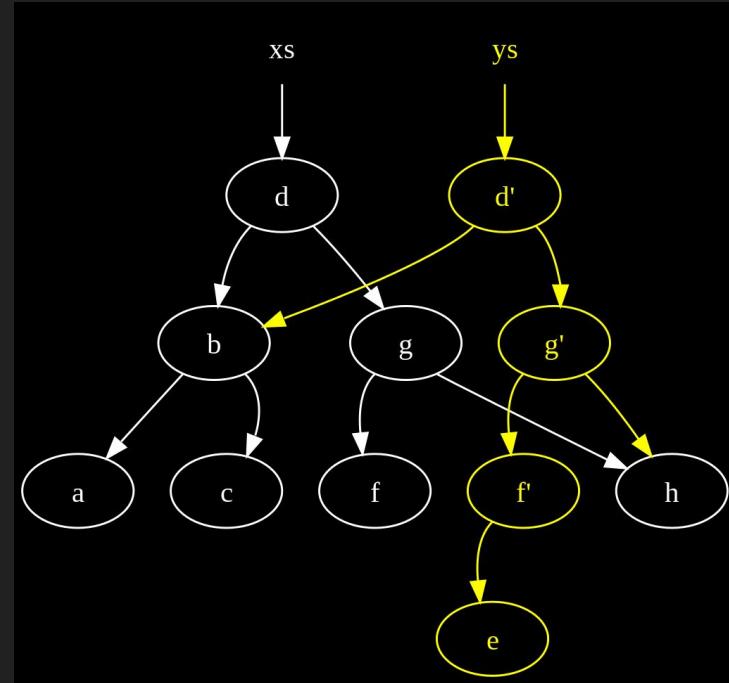
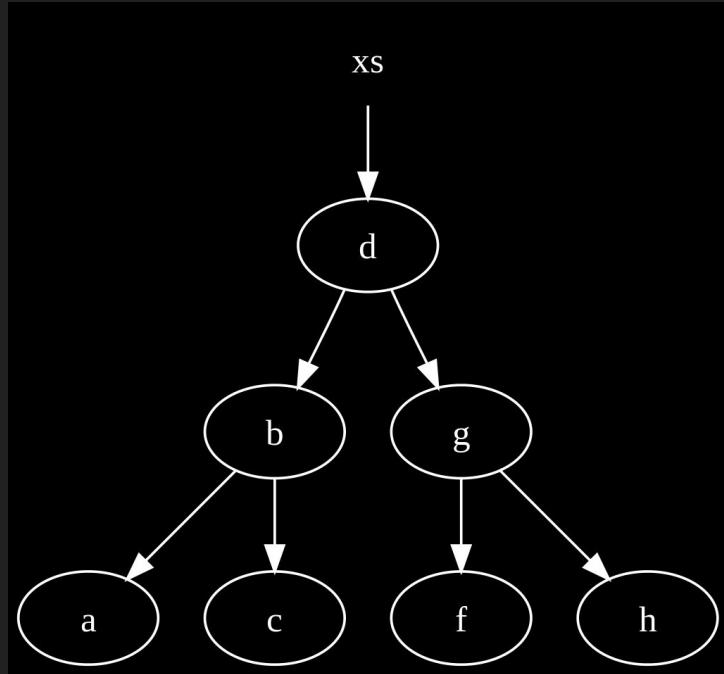
If we produce an operation, we need to preserve original structure unchanged.

If we cannot, we need to copy. Singly-linked Lists can be constructed and operated in persistent way with cons, car and cdr operations.

- $A[k] = O(k)$
- $A[1:k] = O(1)$
- Prepend = $O(1)$
- Append, insert before $k = O(k)$



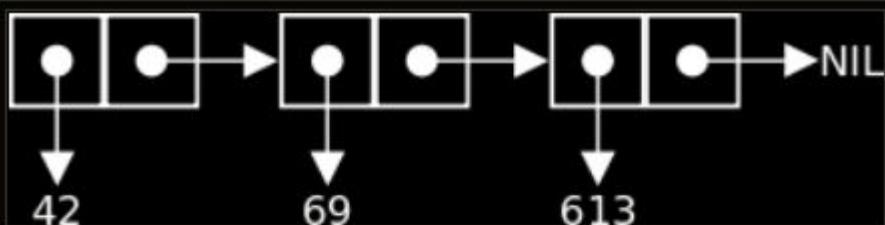
Persistent search trees



VList (Bagwell, 2002)

cons

CONS: constructs memory objects which hold two values



```
(cons 42 (cons 69 (cons 613 nil)))
```

and written with list:

```
(list 42 69 613)
```

```
(cons (cons 1 2) (cons 3 4))
```

```
*  
 / \  
* *  
/ \ / \  
1 2 3 4
```

car and cdr

car extracts first element of the pair, created by cons, cdr extracts the second

(cadr '(1 2 3)) = (car (cdr '(1 2 3))) = 2

(caar '((1 2) (3 4))) = (car (car '((1 2) (3 4)))) = 1

When cons cells are used to implement **singly linked lists** (rather than trees and other more complicated structures), the car operation returns the first element of the list

VList

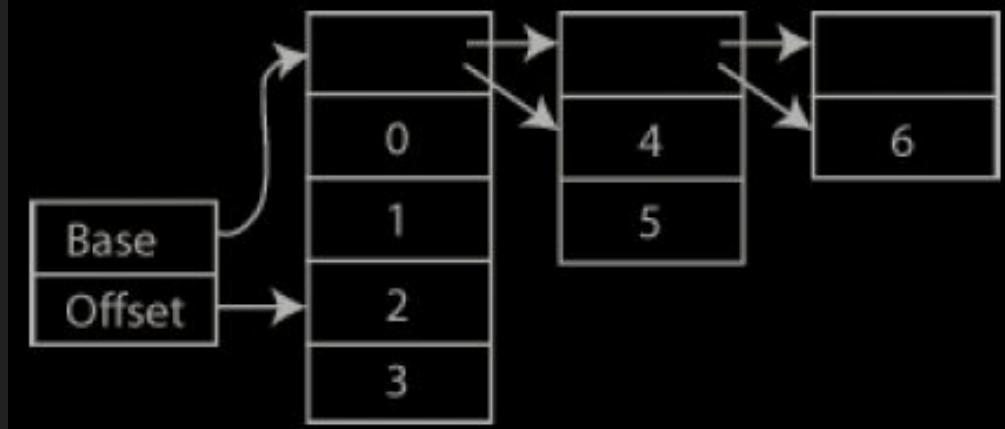
- $A[k]$ - $O_A(1)$ average, $O(\log n)$
 - **Prepend** - $O_A(1)$ average
 - $A[1:k]$ (**cdr**) - $O(1)$
 - **len(A)** - $O(\log n)$
-
- While immutability is a benefit, it is also a drawback, making it inefficient to **modify elements in the middle of the array**
 - $A[-1]$ **Access near the end** of the list can be as expensive as $O(\log n)$
 - **Wasted space** in the first block is proportional to n . This is similar to linked lists

VList

structure of a VList can be seen as a singly-linked list of arrays whose sizes decrease geometrically.

$A[k]$ timing comes from sum of geometric series

Any particular reference to a VList is actually a $\langle \text{base}, \text{offset} \rangle$ pair indicating the position of its first element



VList operations

$A[0]$ (car) = trivial lookup $O(1)$

$A[k]$ (car+cdr) = iterative process with $O(\log(N))$

remove (cdr) = increase the offset | increase the base, offset = 0 - $O(1)$

prepend (cons) = insert in front array or add new 2^r level $O_A(1)$

insert before k (cons in the middle) = see linked list example. We create new front array, place there an element, pointing to k 's base+offset - $O(\text{malloc}(n - k))$ for allocation

8.11 - midterm exam
Please, don't be late

Dynamic programming

General algorithmic techniques

- ▶ brute force (exhaustive search)
- ▶ recursion
- ▶ backtracking
- ▶ divide and conquer
- ▶ greedy algorithms (e.g. Dijkstra, ...)
- ▶ dynamic programming (e.g. Floyd-Warshall, shortest paths in DAGs, ...)
- ▶ branch and bound
- ▶ randomization

Dynamic Programming principles

- ▶ Characterize the structure of an optimal solution
- ▶ Express an optimal solution through optimal solutions of smaller problems (*dynamic programming relation*)
- ▶ Compute values of optimal solutions *bottom-up* (from smaller to larger)
- ▶ Construct an optimal solution from computed information

Example 1: Fibonacci numbers

- ▶ Compute n -th Fibonacci number

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

Example 1: Fibonacci numbers

- ▶ Compute n -th Fibonacci number

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

- ▶ $\mathcal{O}(n)$ -time solution: compute F_n iteratively from 0 to n
(dynamic programming)

Example 1: Fibonacci numbers

- ▶ Compute n -th Fibonacci number

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

- ▶ $\mathcal{O}(n)$ -time solution: compute F_n iteratively from 0 to n (**dynamic programming**)
- ▶ another $\mathcal{O}(n)$ -time solution: **memoization**
 - ▶ compute F_n recursively (*top-down*)
 - ▶ store computed values
 - ▶ before computing F_i check if it has been already computed
 - ▶ Note: **depth-first** exploration of the recursion (subproblem) tree

Example 2: Coin changing problem

- ▶ **Problem:** given a coin system (i.e. denominations of coins $\{C_1, C_2, \dots, C_k\}$) make change for N "rubles" with the minimal number of coins $opt(N)$
- ▶ **Greedy strategy:**
 - ▶ to change N , use the largest coin with value $C \leq N$
 - ▶ change $N-C$ recursively
 - ▶ Ex: 18 rubles = 10+5+2+1, 60 kopecks = 50+10
- ▶ **Does greedy strategy always lead to an optimal solution?**

Example 2: Coin changing problem

- ▶ **Problem:** given a coin system (i.e. denominations of coins $\{C_1, C_2, \dots, C_k\}$) make change for N "rubles" with the minimal number of coins $opt(N)$
- ▶ **Greedy strategy:**
 - ▶ to change N , use the largest coin with value $C \leq N$
 - ▶ change $N-C$ recursively
 - ▶ Ex: 18 rubles = 10+5+2+1, 60 kopecks = 50+10
- ▶ **Does greedy strategy always lead to an optimal solution?**
NO!
 - ▶ if we had 9 roubles in addition, then $18=9+9$
 - ▶ if we had $\{50, 20, 2, 1\}$ kopecks, then $60=20+20+20$ would be better than $60=50+2+2+2+2+2$ obtained by greedy

Average min number of coins

- ▶ In the USA, there are coins of 1¢ 5¢ 10¢ 25¢. Replacing 10¢ by 18¢ would improve the average min number of coins in the change from 4.7 to 3.89 (which is optimal for 4 denominations)
- ▶ If we had to (1,5,10,25) add another denomination, the best to add would be 32¢ (improves average to 3.46)

Mathematical Entertainments

Michael Kleber and Ravi Vakil, Editors

What This Country Needs Is an 18¢ Piece*

Jeffrey Shallit

Most businesses in the United States currently make change using four different types of coins: 1¢ (cent),¹ 5¢ (nickel), 10¢ (dime), and 25¢ (quarter). For people who make change on a daily basis, it is desirable to make change in as efficient a manner as possible. One criterion for efficiency is to use the smallest number of coins. For example, to make change for 30¢, one could, at least in principle, give a customer 30 1-cent coins, but most would probably prefer receiving a quarter and a nickel.

Formally, we can define the *optimal representation problem* as follows:

For the current system, where $(e_1, e_2, e_3, e_4) = (1, 5, 10, 25)$, a simple computation determines that $\text{cost}(100; 1, 5, 10, 25) = 4.7$. In other words, on average a change-maker must return 4.7 coins with every transaction.

Can we do better? Indeed we can. There are exactly two sets of four denominations that minimize $\text{cost}(100; e_1, e_2, e_3, e_4)$; namely, $(1, 5, 18, 25)$ and $(1, 5, 18, 29)$. Both have an average cost of 3.89. We would therefore gain about 17% efficiency in change-making by switching to either of these four-coin systems. The first system, $(1, 5, 18, 25)$,

Coin changing: solution 1

- ▶ Assume we always have coins of 1 ruble
- ▶ *Idea*: try all decompositions of N into two amounts, solve each amount, take minimum

$$opt(N) = \begin{cases} 1, & \text{if there exist coins of } N \\ \min_{i=1..N-1} \{opt(i) + opt(N - i)\}, & \text{otherwise} \end{cases}$$

- ▶ Can be seen as divide-and-conquer or brute-force
- ▶ Exponential time (tree of recursive calls)

Coin changing: solution 2

- ▶ *Idea*: try all possible coins and solve the difference; take minimum

$$opt(N) = \begin{cases} 0, & \text{if } N = 0 \\ \min_{C_i \leq N} \{1 + opt(N - C_i)\}, & \text{otherwise} \end{cases}$$

- ▶ still exponential time (but better than the previous solution)

Coin changing: solution 2

- ▶ *Idea*: try all possible coins and solve the difference; take minimum

$$opt(N) = \begin{cases} 0, & \text{if } N = 0 \\ \min_{C_i \leq N} \{1 + opt(N - C_i)\}, & \text{otherwise} \end{cases}$$

- ▶ still exponential time (but better than the previous solution)
- ▶ *memoization* \Rightarrow time $O(N \cdot K)$ where K is the number of distinct denominations

Coin changing: solution 3

- ▶ *Idea (dynamic programming)*: solve the problem for amounts $1, 2, \dots, N$. For each amount, use solutions for smaller amounts.

```
opt(0) ← 0;  
for  $j \leftarrow 1$  to  $N$  do  
     $1 + \min_{C_i \leq j} opt(j - C_i)$ 
```

- ▶ Replacing recursion by *iteration over subproblems*. Time $O(N \cdot K)$, where K is the number of distinct denominations

Coin changing: solution 3

- ▶ **Idea (dynamic programming):** solve the problem for amounts $1, 2, \dots, N$. For each amount, use solutions for smaller amounts.

```
opt(0) ← 0;  
for  $i \leftarrow 1$  to  $N$  do  
     $1 + \min_{C_i \leq i} opt(N - C_i)$ 
```

N is given “in unary”

- ▶ ~~Replacing recursion by iteration over subproblems.~~ Time $O(N \cdot K)$, where K is the number of distinct denominations

Coin changing: solution 3

- ▶ **Idea (dynamic programming):** solve the problem for amounts $1, 2, \dots, N$. For each amount, use solutions for smaller amounts.

```
opt(0) ← 0;  
for  $i \leftarrow 1$  to  $N$  do  
     $1 + \min_{C_i \leq i} opt(N - C_i)$ 
```

N is given “in unary”

- ▶ Replacing recursion by *iteration over subproblems*. Time $O(N \cdot K)$, where K is the number of distinct denominations
- ▶ How to recover an optimal decomposition?

Coin changing: number of decompositions

- ▶ $\{1,10,25,50\}$, $N=30$
- ▶ **Question:** how to count the number of all *distinct* decompositions?
- ▶ E.g. $\{1,2,3\}$, $N=5$
 $\{1,1,1,1,1\}, \{1,1,1,2\}, \{1,2,2\}, \{1,1,3\}, \{2,3\}$

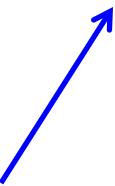
Coin changing: number of decompositions

- ▶ Assume denominations $\{C_1, C_2, \dots, C_k\}$ in increasing order and $C_1=1$
- ▶ $NUM(N, j)$: number of decompositions of N using coins $\{C_1, C_2, \dots, C_j\}$
- ▶ main relation: $NUM(N, j) = NUM(N, j - 1) + NUM(N - C_j, j)$

Coin changing: number of decompositions

- ▶ Assume denominations $\{C_1, C_2, \dots, C_k\}$ in increasing order and $C_1=1$
- ▶ $NUM(N, j)$: number of decompositions of N using coins $\{C_1, C_2, \dots, C_j\}$
- ▶ main relation: $NUM(N, j) = NUM(N, j - 1) + NUM(N - C_j, j)$

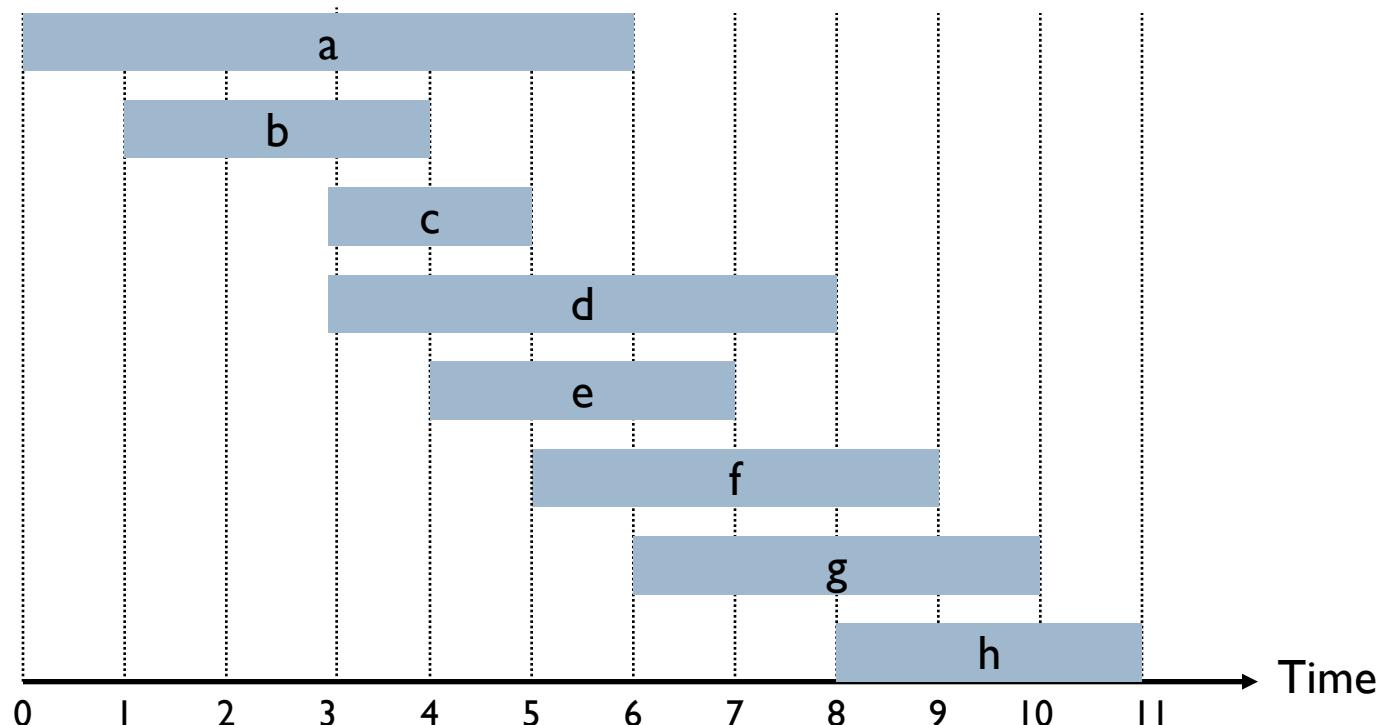
$$NUM(N, j) = \begin{cases} 1, & \text{if } j = 1 \text{ or } N = 0 \\ NUM(N, j - 1), & \text{elseif } N < C_j \\ NUM(N, j - 1) + NUM(N - C_j, j) & \text{otherwise} \end{cases}$$



conditions are checked in order

(Weighed) interval scheduling problem

- ▶ Weighted interval scheduling problem.
 - ▶ 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.
 - ▶ Goal: find maximum **weight** subset of mutually compatible jobs.



(Weighed) interval scheduling problem

- ▶ What about a greedy solution?
- ▶ Many possible greedy strategies:
 1. Choose the earliest starting next job
 2. Choose the shortest next job
 3. Choose the earliest finishing next job

(Weighed) interval scheduling problem

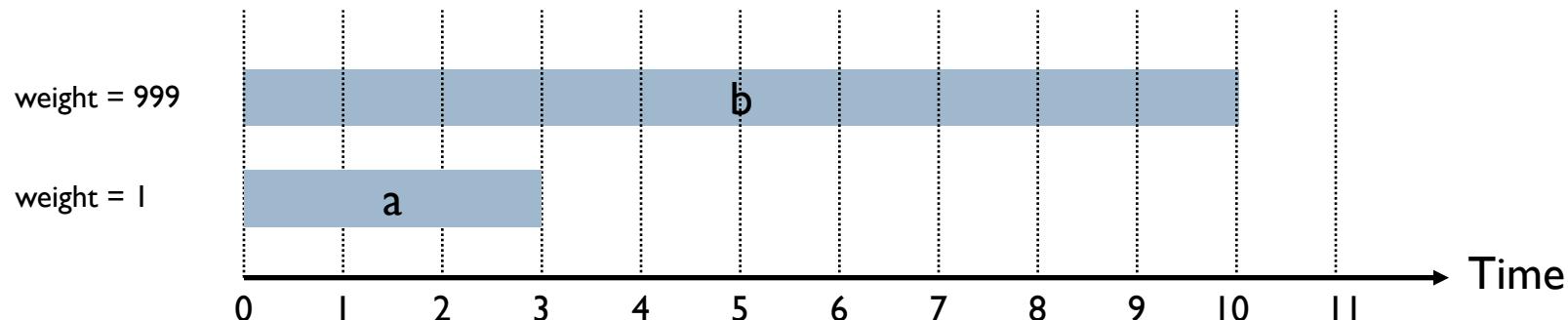
- ▶ What about a greedy solution?
- ▶ Many possible greedy strategies:
 1. Choose the earliest starting next job
 2. Choose the shortest next job
 3. Choose the earliest finishing next job
- ▶ Strategies 1 and 2 do not produce an optimal solution.
Strategy 3 does but *only if all intervals have equal weight*. Easy implementation:
 - ▶ Consider jobs in ascending order of finish time.
 - ▶ Go through the jobs and add a job to the current subset if it is compatible with previously chosen jobs.

Exercise

- ▶ Prove that Strategy 3 computes an optimal solution if all weights are equal

(Weighed) interval scheduling problem

- ▶ *Observation:* Strategy 3 can fail spectacularly if arbitrary weights are allowed.

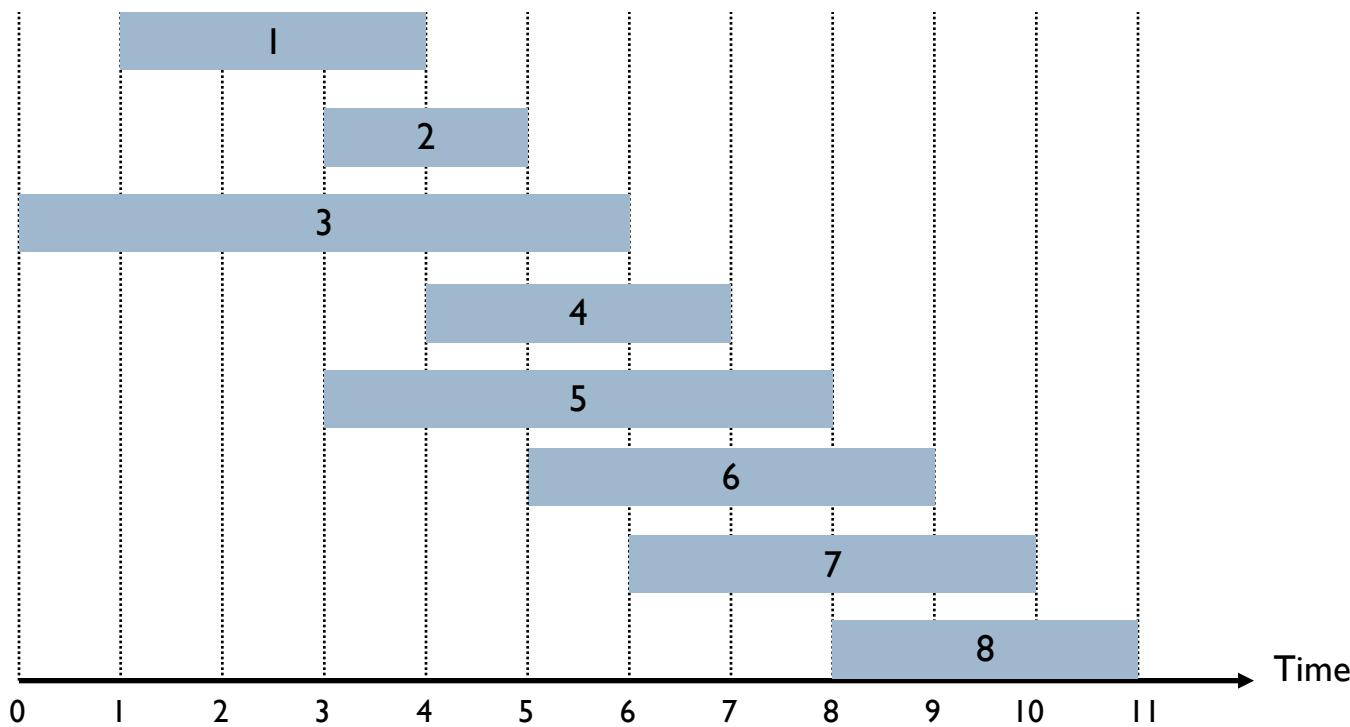


(Weighed) interval scheduling problem

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

Def: $p(j)$ = largest index $i < j$ such that job i is compatible with j .

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



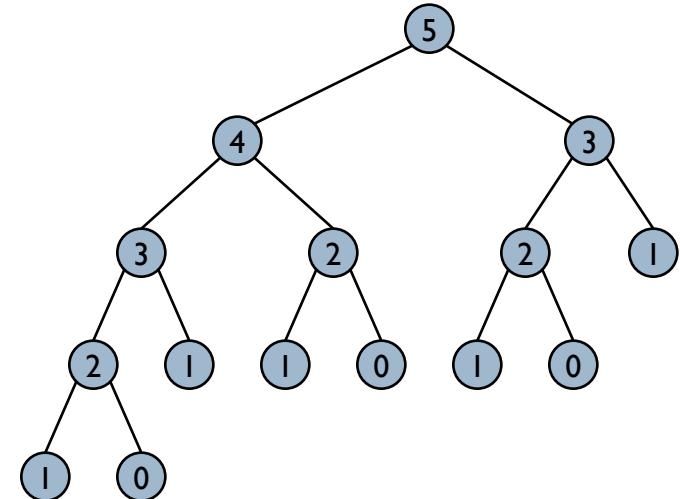
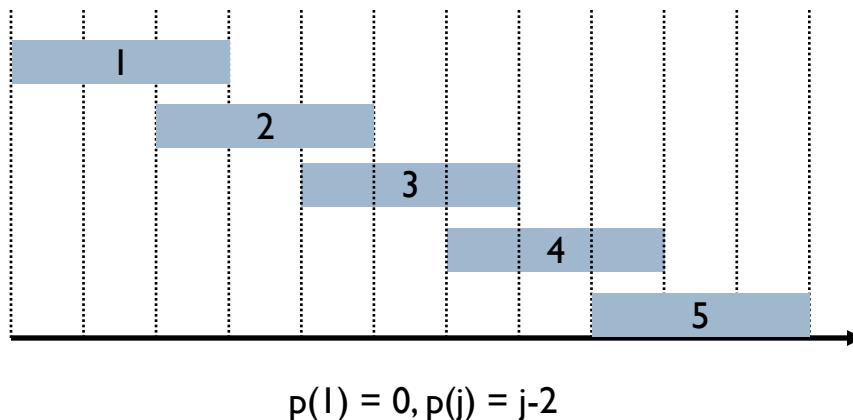
(Weighed) interval scheduling problem

- ▶ *Notation:* $OPT(j)$ = value of optimal solution to the problem consisting of job requests $1, 2, \dots, j$.
- ▶ Case 1: OPT selects job j .
 - ▶ can't use incompatible jobs $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$
 - ▶ must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, p(j)$
- ▶ Case 2: OPT does not select job j .
 - ▶ must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, j-1$

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

(Weighed) interval scheduling problem

- ▶ *Observation.* Computing $\text{OPT}(n)$ by recursive algorithm leads to exponential time because of redundant subproblems
- ▶ *Exercise:* Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



(Weighed) interval scheduling problem

- ▶ *Memoization*: Store results of each sub-problem in an array; lookup as needed.

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

Sort jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p(1), p(2), \dots, p(n)$

```
for j = 1 to n
    M[j] = empty
M[0] = 0
```

```
M-Compute-Opt(j) {
    if (M[j] is empty)
        M[j] = max( $w_j + M\text{-Compute-Opt}(p(j))$ ,  $M\text{-Compute-Opt}(j-1)$ )
    return M[j]
}
```

(Weighed) interval scheduling problem

- ▶ Bottom-up dynamic programming. Unwind recursion.

```
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)
```

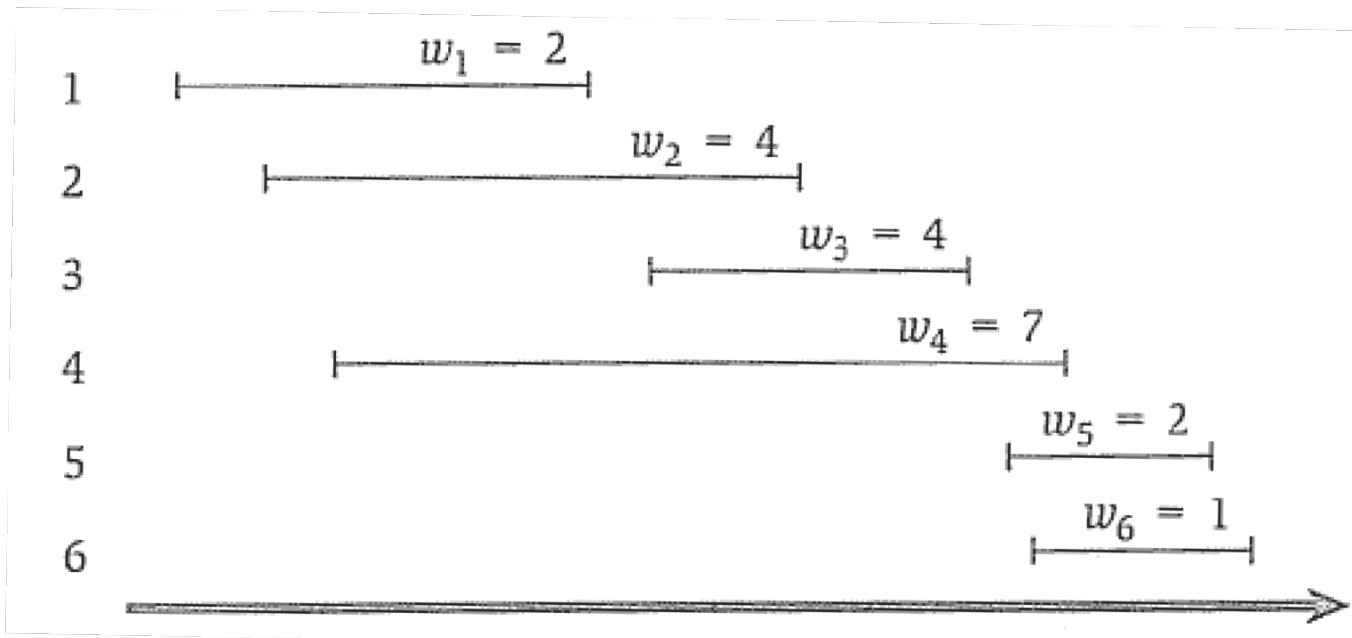
```
Iterative-Compute-Opt {
    M[0] = 0
    for j = 1 to n
        M[j] = max(vj + M[p(j)], M[j-1])
}
```

(Weighed) interval scheduling problem

- ▶ Time complexity:
 - ▶ Sort by finish time: $O(n \log n)$
 - ▶ Computing $p(\cdot)$: $O(n)$ after sorting by start time

Exercise

- Solve the interval scheduling problem for the following case:



Dynamic Programming scenario

- ▶ Analyse the structure of an optimal solution
 - ▶ usually to an optimization problem
- ▶ Express an optimal solution to an instance via optimal solutions to "smaller" instances
 - ▶ usually leads to a recursive relation whose direct application usually leads to exponential time
- ▶ Iterate through the instances from "smallest" to "largest" ("bottom-up") until obtaining the solution to the desired instance
- ▶ Construct the solution from computed information

Some history

- ▶ Richard Bellman (1920-1984) 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"
- from [Bellman, R., *Eye of the Hurricane, An Autobiography*]



Exercise

- Given a sequence of numbers $A[1..n]$, compute the *longest increasing subsequence*, i.e. the maximum k and a subsequence $1 \leq i_1 < i_2 < \dots < i_k \leq n$ such that

$$A[i_1] < A[i_2] < \dots < A[i_k]$$

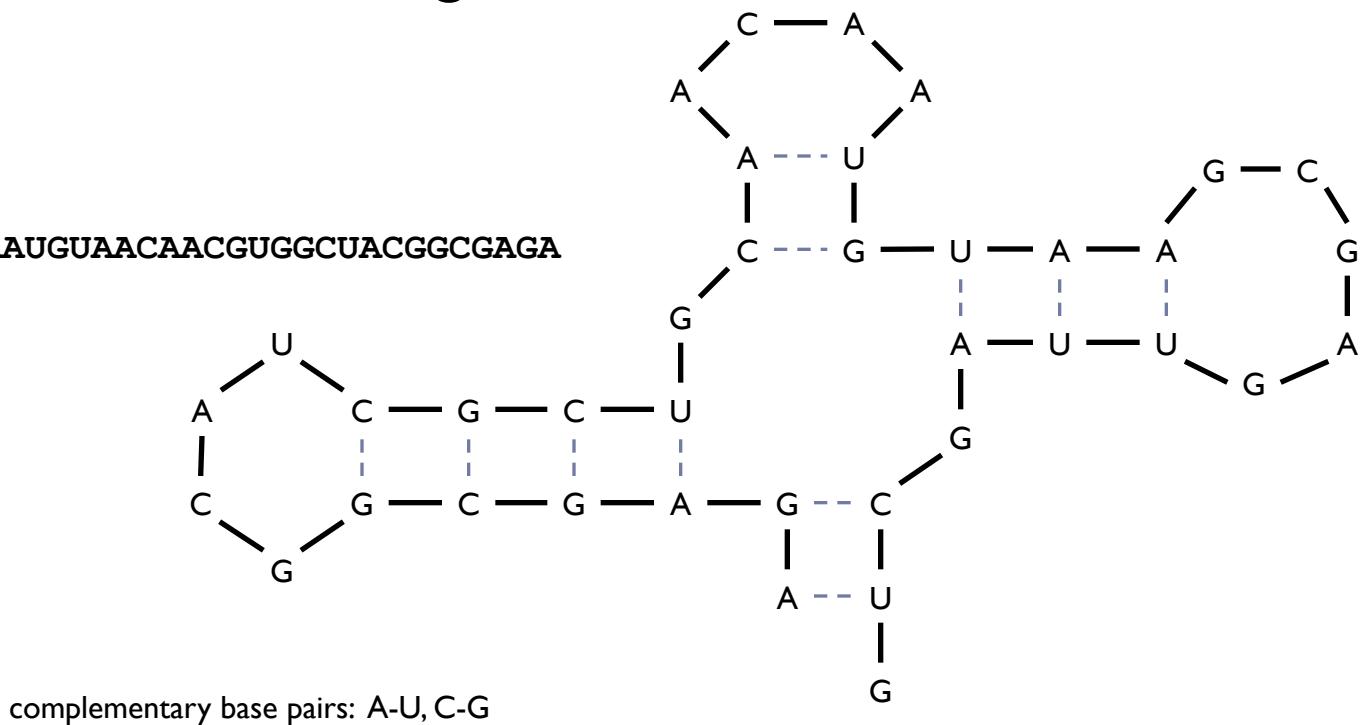
RNA Secondary Structure

Dynamic Programming over intervals

RNA Secondary Structure

- ▶ RNA. String $B = b_1b_2\dots b_n$ over alphabet { A, C, G, U }.
- ▶ Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

Ex: GUCGAUUGAGCGAAUGUAACAAACGUGGUACGGCGAGA

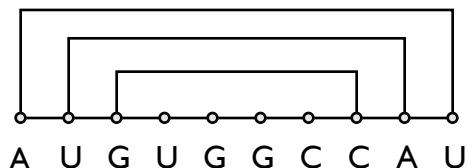
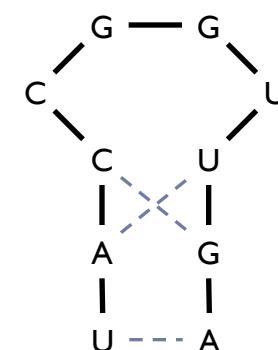
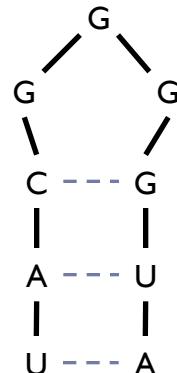
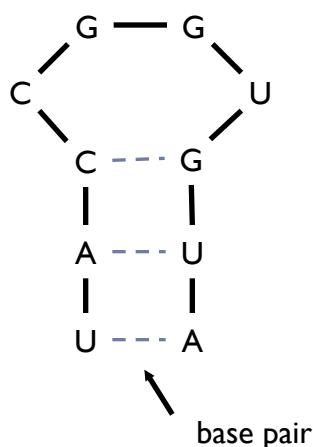


RNA Secondary Structure

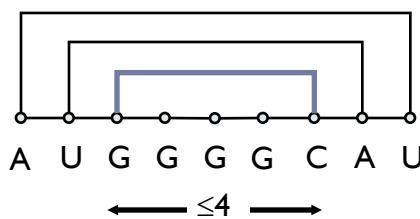
- ▶ **Secondary structure:** A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:
 - ▶ [Watson-Crick.] S is a matching and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C.
 - ▶ [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
 - ▶ [Non-crossing.] If (b_i, b_j) and (b_k, b_l) are two pairs in S , then we cannot have $i < k < j < l$.
- ▶ **Free energy:** Usual hypothesis is that an RNA molecule will form the secondary structure with the minimum total free energy.
 - ↓
approximate by max number of base pairs
- ▶ **Goal:** Given an RNA molecule $B = b_1 b_2 \dots b_n$, find a secondary structure S that maximizes the number of base pairs.

RNA Secondary Structure: Examples

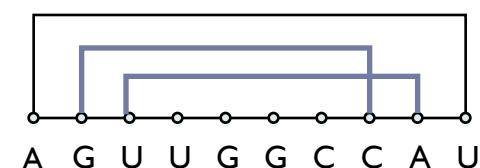
► Examples:



ok



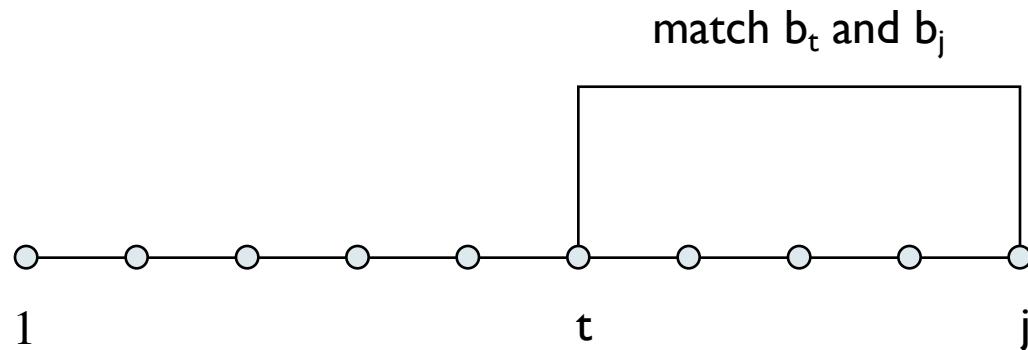
sharp turn



crossing

RNA Secondary Structure: Subproblems

- ▶ *First attempt:* $\text{OPT}(j) = \text{maximum number of base pairs in a secondary structure of the prefix } b_1 b_2 \dots b_j.$



- ▶ *Difficulty:* Results in two sub-problems.

- ▶ Finding secondary structure in: $b_1 b_2 \dots b_{t-1}$. ← $\text{OPT}(t-1)$
- ▶ Finding secondary structure in: $b_{t+1} b_{t+2} \dots b_{j-1}$. ← need more sub-problems

Dynamic Programming Over Intervals

- ▶ Notation: $\text{OPT}(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \dots b_j$.
- ▶ Case 1. If $i \geq j - 4$
 - ▶ $\text{OPT}(i, j) = 0$ by no-sharp turns condition.
- ▶ Case 2. Base b_j is not involved in a pair.
 - ▶ $\text{OPT}(i, j) = \text{OPT}(i, j-1)$
- ▶ Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$
 - ▶ non-crossing constraint decouples resulting sub-problems
 - ▶ $\text{OPT}(i, j) = 1 + \max_t \{ \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1) \}$

↑
take max over t such that $i \leq t < j-4$ and
 b_t and b_j are Watson-Crick complements

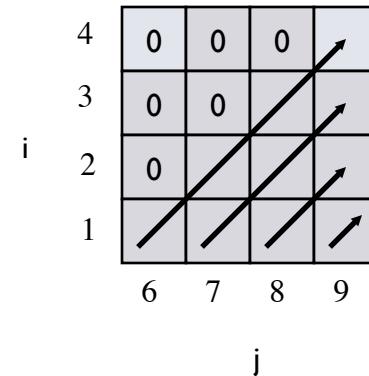
DP relation: summary

$$OPT(i, j) = \begin{cases} 0, & \text{if } i \geq j - 4 \\ \max\{1 + \max_t \{OPT(i, t - 1) + OPT(t + 1, j)\}, OPT(i, j - 1)\}, & \text{otherwise} \end{cases}$$

Bottom Up Dynamic Programming Over Intervals

- ▶ Q. What order to solve the sub-problems?
- ▶ A. Do shortest intervals first.

```
RNA(b1, ..., bn) {  
    for k = 5, 6, ..., n-1  
        for i = 1, 2, ..., n-k  
            j = i + k  
            Compute OPT(i, j)  
                ↗ using recurrence  
    return OPT  
}
```



- ▶ Running time: $\Theta(n^3)$.

Sequence comparison

more Dynamic Programming

I. Hamming distance

- ▶ defined for sequences of the same length
- ▶ $\text{Ham}(S,T) = \text{number of mismatches}$

GAGGCTCACCTGACCTTCCAGGCGCGAGA**TGCC**TCT
||| | | | | | | | | | | | | | | | | | | | | | | | | | | |
TAGCCTCACAAGACCG**CTTGGGT**CGATT**TGCC**GAC

- ▶ can be computed in $O(n)$

II. LCS: Longest Common Subsequence

- ▶ longest *subsequence* common to the two strings
- ▶ $\text{LCS}(\text{ACACGA}, \text{CAAGTAGAG})=4$

II. LCS: Longest Common Subsequence

- ▶ longest *subsequence* common to the two strings
- ▶ $\text{LCS}(\text{ACACGA}, \text{CAAGTAGAG})=4$

ACACGA

CAAGTAGAG

II. LCS: Longest Common Subsequence

- ▶ longest *subsequence* common to the two strings
- ▶ $\text{LCS}(\text{ACACGA}, \text{CAAGTAGAG})=4$

A**CACGA**

CAAGTAGAG

II. LCS: Longest Common Subsequence

- ▶ longest *subsequence* common to the two strings
- ▶ $\text{LCS}(\text{ACACGA}, \text{CAAGTAGAG})=4$

A C A C G A

C A A G T A G A G

- $d(S, T) = |S| + |T| - 2 \cdot \text{LCS}(S, T)$
- $d(\text{ACACGA}, \text{CAAGTAGAG})=7$
- $d(S, T)$: minimum number of letter insertions/deletions needed to transform one sequence to the other

II. LCS: Longest Common Subsequence

- ▶ longest *subsequence* common to the two strings
- ▶ $\text{LCS}(\text{ACACGA}, \text{CAAGTAGAG})=4$

A**C****A**—**C****G**—**A**—
| | | |
—**C****A****A**—**G****T****A****G****A**
The sequence above shows the longest common subsequence between "ACACGA" and "CAAGTAGAG". The subsequence is highlighted in bold and blue. Vertical lines above the subsequence indicate its position in both strings.

- $d(S, T) = |S| + |T| - 2 \cdot \text{LCS}(S, T)$
- $d(\text{ACACGA}, \text{CAAGTAGAG})=7$
- $d(S, T)$: minimum number of letter insertions/deletions needed to transform one sequence to the other

III. Levenshtein (edit) distance

- ▶ minimum number of edit operations (letter substitutions, insertions, deletions) required to transform S into T
- ▶ $\text{edit}(\text{ACACGA}, \text{CAAGTAGAG})=6$

A C A C G - A ---
| | | |
- C A A G T A G A G

Bioinformatics: “CIGAR strings”

part of SAM format

RefPos:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Reference:	C	C	A	T	A	C	T	G	A	A	C	T	G	A	C	T	A	A	C
Read :					A	C	T	A	G	A	A		T	G	G	C	T		

POS: 5

CIGAR: 3M1I3M1D5M

IV. Sequence alignment (weighted edit distance)

- Given two sequences RDISLVKNAGI and RNILVSDAKNVGI

R	D	I	S	L	V	-	-	-	K	N	A	G	I
R	N	I	-	L	V	S	D	A	K	N	V	G	I

- 3 types of columns corresponding to 3 elementary evolutionary events
 - matches
 - substitution (mismatch)
 - insertion, deletion (*indel*)
- Assign a score (positive or negative) to each “event”.
- Alignment score* = sum of scores over all columns.
- Optimal alignment* = one that maximizes the score

Sequence alignment: scoring

- ▶ scoring function:

Mismatch :

DN : 1
AV, LD : 0

Match :

A, I, L, S, V : 4

Indel :

G, N : 6
R, K : 5 -5

R D I S L V - - - K N A G I
| | | | | | | | | |
R N I - L V S D A K N V G I

R D I - - S L V K N A - - - G I
| | | | | | | | | | | |
R N I L V S - - - D A K N V G I

Score=19

R D I - - S L V K N A G I
| | | | | | | | | |
R N I L V S D A K N V G I

Score=-11

Score=25

Sequence alignment: scoring

- ▶ BLOSUM62 matrix for protein sequences

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W	
C	9																			C	
S	-1	4																		S	
T	-1	1	5																	T	
P	-3	-1	-1	7																P	
A	0	1	0	-1	4															A	
G	-3	0	-2	-2	0	6														G	
N	-3	1	0	-2	-2	0	6													N	
D	-3	0	-1	-1	-2	-1	1	6												D	
E	-4	0	-1	-1	-1	-2	0	2	5											E	
Q	-3	0	-1	-1	-1	-2	0	0	2	5										Q	
H	-3	-1	-2	-2	-2	-2	1	-1	0	0	8									H	
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5								R	
K	-3	0	-1	-1	-1	-2	0	-1	1	1	-1	2	5							K	
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5						M	
I	-1	-2	-1	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4					I	
L	-1	-2	-1	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4				L	
V	-1	-2	0	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4			V	
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6		F	
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7	<th>Y</th>	Y
W	-2	-3	-2	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11	W
	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W	

Example of an alignment

Sequences producing significant alignments:

Score E
(bits) value

- [gnl|Pfam|pfam00155](#) aminotran_1, Aminotransferases class-I 338 4e-94
- [gnl|Pfam|pfam00155](#), aminotran_1, Aminotransferases class-I

query to multiple alignment, display sequences

Length = 428
Score = 338 bits (857), Expect = 4e-94

Query: 23	KSTWFSEVQMGPPDAILGVTEAFKKDTNPKKIN----LGAGAYRDDNTQPPVLPSVREAE	78
Sbjct: 1	LSRNATFNSHGQDSSYFLGWQEYEKNPYHEVHNTNGIQMGLAENQLCFDLLESWLAKNP	60
Query: 79	KRVSRS-----LDKEYATIIGIPEFYNKIAELALGKGSKRLLAAKHNVTAQSISCTGA	131
Sbjct: 61	EAAAFKKNGESIFAAELALFQDYHGLPAFKAMVDFMAEIRGNKVTFDPNHLVLTAGATSA	120
Query: 132	LRIGAAF LAKFWQGNREIYIPSPSWGNHV-AIFEHA GLPVNRYRYDKDTCALDFGGLIE	190
Sbjct: 121	NETFIFCLADPGE---AVLIPTPYPGFDRDLKWRT GVE IVPIHCTSSNGFQITETALEE	177
Query: 191	DLKKIPE---KSIVL LHACAH NPTGVDPTLEQWR EISALV KRNLYPFIDMAYQGFATGD	247
Sbjct: 178	AYQEAEKRNLRLVKGV LVTNPS NPLGTTMTRNELYLLSFVEDKG IHLISDEI YSGTAFSS	237
Query: 248	IDRDAQAVRTFEAD-----GHDFCLAQSF AKNMGLYGERAGA F T VLCSDEEE AARV	298
Sbjct: 238	P--SFISVMEVLKDRCNDENSEWQRVHV VY SLSKD L GLPCFRVGAIYSNDMVVA AATK	295
Query: 299	M-----SQVKILIRGLYSNP---PVH GARI AAEILNNEDLRAQWLKDVKLMADRIIDV	348
Sbjct: 296	MSSFGLVS SQTQHLLSAML SDKKLTKNYIAENHKRLKQRQKKLVSGLQKSG- ISCLNGNA	354
Query: 349	RTKLKD NLIKLGSSQNWDHIVNQIGMF CFTGLKPEQVQK-LIKDH S VYLTNDGR VSMAGV	407
Sbjct: 355	GLFCWVDMRHLL---SNTFEAEMELWKKIVYEVHLNISP GSSCH TEPGWFR VCFANL	410
Query: 408	TSKNVEYLAESIHKVTK 424	
Sbjct: 411	PERTLD LA MQR L KAFVG 427	

Longest Common Subsequence

- ▶ consider score match:1, indel: 0, mismatch: -1

```
-AGGCTCACCTGACT-CCAGGC-CGA--TGCC---  
||| | ||||| | | | | | | | | | | | | | | | |  
TAG-CTCAC--GAC-GC--GG-TCGATTGCCCCGAC
```

- optimal alignment \sim *longest common subsequence (LCS)*
- $\text{Score}(S,T) = \text{LCS}(S,T)$

Levenshtein distance

- ▶ consider score match:0, indel: -1, mismatch: -1

The diagram illustrates a sequence alignment between two DNA strands. The top strand is: **-AGGCTCACCTGACTCACA GGCCGA -- TGCC ---**. The bottom strand is: **TAG-CTCAC--GACGC--GGTCGA TT TGCC GAC**. Vertical blue lines connect matching nucleotides between the two strands. A horizontal blue line connects the two strands at the position of the insertion of 'CT' in the top strand. Horizontal red lines above and below the strands indicate the positions of matches and mismatches respectively.

- optimal alignment \sim Levenshtein (edit) distance
- $\text{edit}(S,T) = -\text{Score}(S,T)$

Computing Score(S,T)

- ▶ assume d is indel penalty (usually $d < 0$), $s(x,y)$ score of aligning x and y (match or mismatch), $S[1..n]$ and $T[1..m]$ are input strings
- ▶ Idea: compute $\text{Score}(i,j)$: optimum score between $S[1..i]$ and $T[1..j]$

Computing Score(S,T)

- ▶ assume d is indel penalty (usually $d < 0$), $s(x,y)$ score of aligning x and y (match or mismatch), $S[1..n]$ and $T[1..m]$ are input strings
- ▶ Idea: compute $\text{Score}(i,j)$: optimum score between $S[1..i]$ and $T[1..j]$

$$\text{Score}(i,j) = \max \begin{cases} \text{Score}(i-1, j-1) + s(S[i], T[j]) \\ \text{Score}(i-1, j) + d \\ \text{Score}(i, j-1) + d \end{cases}$$

- initialization: $\text{Score}(0,0)=0$, $\text{Score}(0,j)=jd$, $\text{Score}(i,0)=id$
- resulting score: $\text{Score}(n,m)$

Computing Score(S,T)

- ▶ assume d is indel penalty (usually $d < 0$), $s(x,y)$ score of aligning x and y (match or mismatch), $S[1..n]$ and $T[1..m]$ are input strings
- ▶ Idea: compute $\text{Score}(i,j)$: optimum score between $S[1..i]$ and $T[1..j]$

$$\text{Score}(i,j) = \max \begin{cases} \text{Score}(i-1,j-1) + s(S[i], T[j]) \\ \text{Score}(i-1,j) + d \\ \text{Score}(i,j-1) + d \end{cases}$$

- initialization: $\text{Score}(0,0)=0$, $\text{Score}(0,j)=jd$, $\text{Score}(i,0)=id$
- resulting score: $\text{Score}(n,m)$
- *Implementation: Dynamic Programming!*

Example

$s(x,x)=2$, $s(x,y)=-1$ for $x \neq y$, $d=-2$

A C G G C T A T

Example

$s(x,x)=2$, $s(x,y)=-1$ for $x \neq y$, $d=-2$

A C G G C T A T

Example

$s(x,x)=2$, $s(x,y)=-1$ for $x \neq y$, $d=-2$

A C G G C T A T

Example

$$s(x,x)=2, s(x,y)=-1 \text{ for } x \neq y, d=-2$$

A C G G C T A T

Example

$s(x,x)=2$, $s(x,y)=-1$ for $x \neq y$, $d=-2$

A C G G C T A T

Example

$$s(x,x)=2, s(x,y)=-1 \text{ for } x \neq y, d=-2$$

A C G G C T A T

Example

$$s(x,x)=2, s(x,y)=-1 \text{ for } x \neq y, d=-2$$

A C G G C T A T

Example

$$s(x,x)=2, s(x,y)=-1 \text{ for } x \neq y, d=-2$$

A C G G C T A T

Example

$s(x,x)=2$, $s(x,y)=-1$ for $x \neq y$, $d=-2$

A C G G C T A T

	0	1	2	3	4	5	6	7	8
0	0	-2	-4	-6	-8	-10	-12	-14	-16
A	-2	2	0	-2	-4	-6	-8	-10	-12
C	-4	0	4	2	0	-2	-4	-6	-8
T	-6	-2	2	3	1	-1	0	-2	-4
G	-8	-4	0	4	5	3	1	-1	-3
T	-10	-6	-2	2	3	4	5	3	1
A	-12	-8	-4	0	1	2	3	7	5
T	-14	-10	-6	-2	-1	0	4	5	9

Example

$s(x,x)=2$, $s(x,y)=-1$ for $x \neq y$, $d=-2$

	A	C	G	G	C	T	A	T	
	0	1	2	3	4	5	6	7	8
0	0	-2	-4	-6	-8	-10	-12	-14	-16
A	-2	2	0	-2	-4	-6	-8	-10	-12
C	-4	0	4	2	0	-2	-4	-6	-8
T	-6	-2	2	3	1	-1	0	-2	-4
G	-8	-4	0	4	5	3	1	-1	-3
T	-10	-6	-2	2	3	4	5	3	1
A	-12	-8	-4	0	1	2	3	7	5
T	-14	-10	-6	-2	-1	0	4	5	9

Score(S,T)

How to recover the alignment?

$$s(x,x)=2, s(x,y)=-1 \text{ for } x \neq y, d=-2$$

A C G G C T A T

	0	1	2	3	4	5	6	7	8
0	0	-2	-4	-6	-8	-10	-12	-14	-16
A	-2	2	0	-2	-4	-6	-8	-10	-12
C	-4	0	4	2	0	-2	-4	-6	-8
T	-6	-2	2	3	1	-1	0	-2	-4
G	-8	-4	0	4	5	3	1	-1	-3
T	-10	-6	-2	2	3	4	5	3	1
A	-12	-8	-4	0	1	2	3	7	5
T	-14	-10	-6	-2	-1	0	4	5	9

Score(S,T)

How to recover the alignment?

$$s(x,x)=2, s(x,y)=-1 \text{ for } x \neq y, d=-2$$

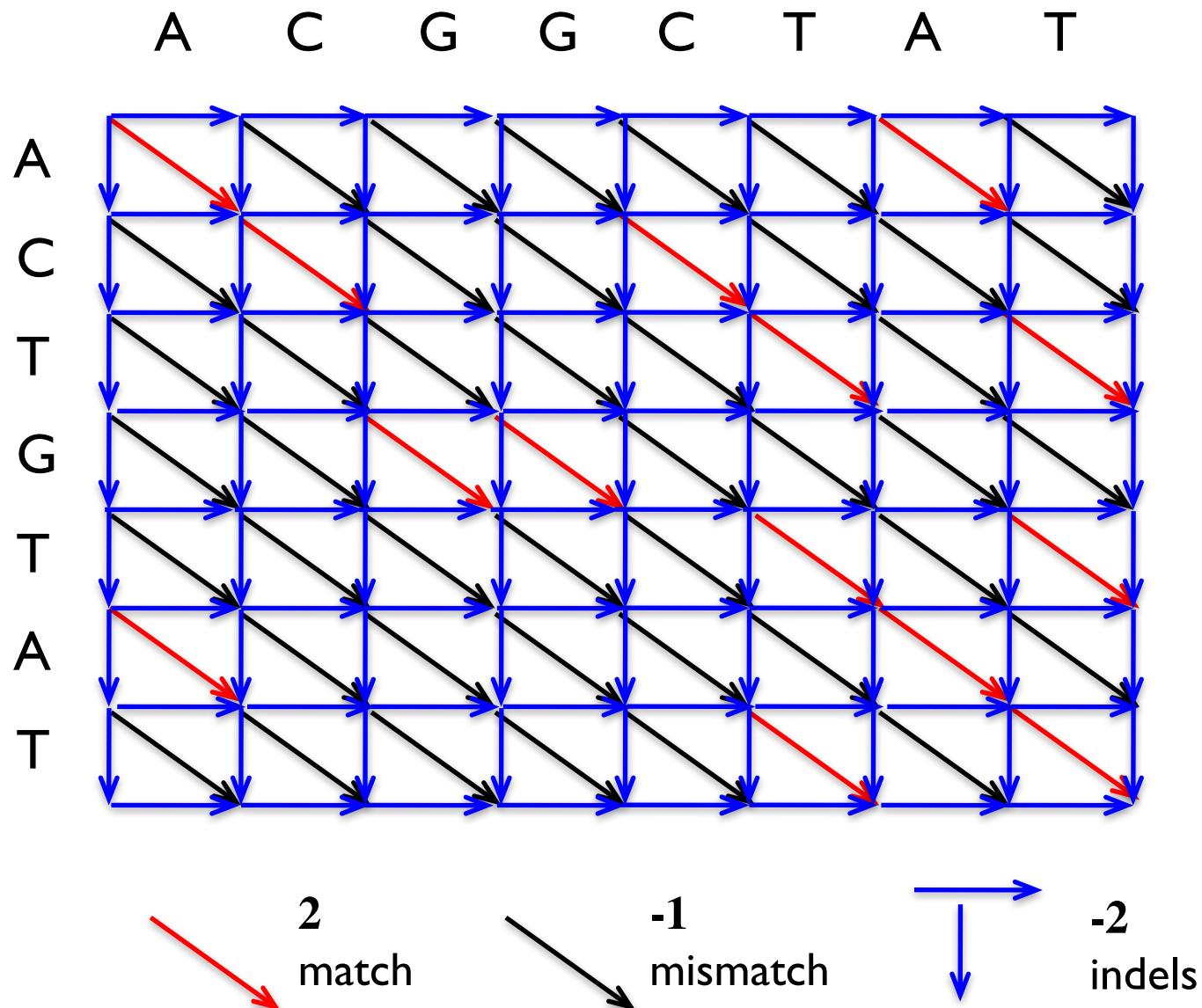
ACGGCTAT
ACTG-TAT

A C G G C T A T

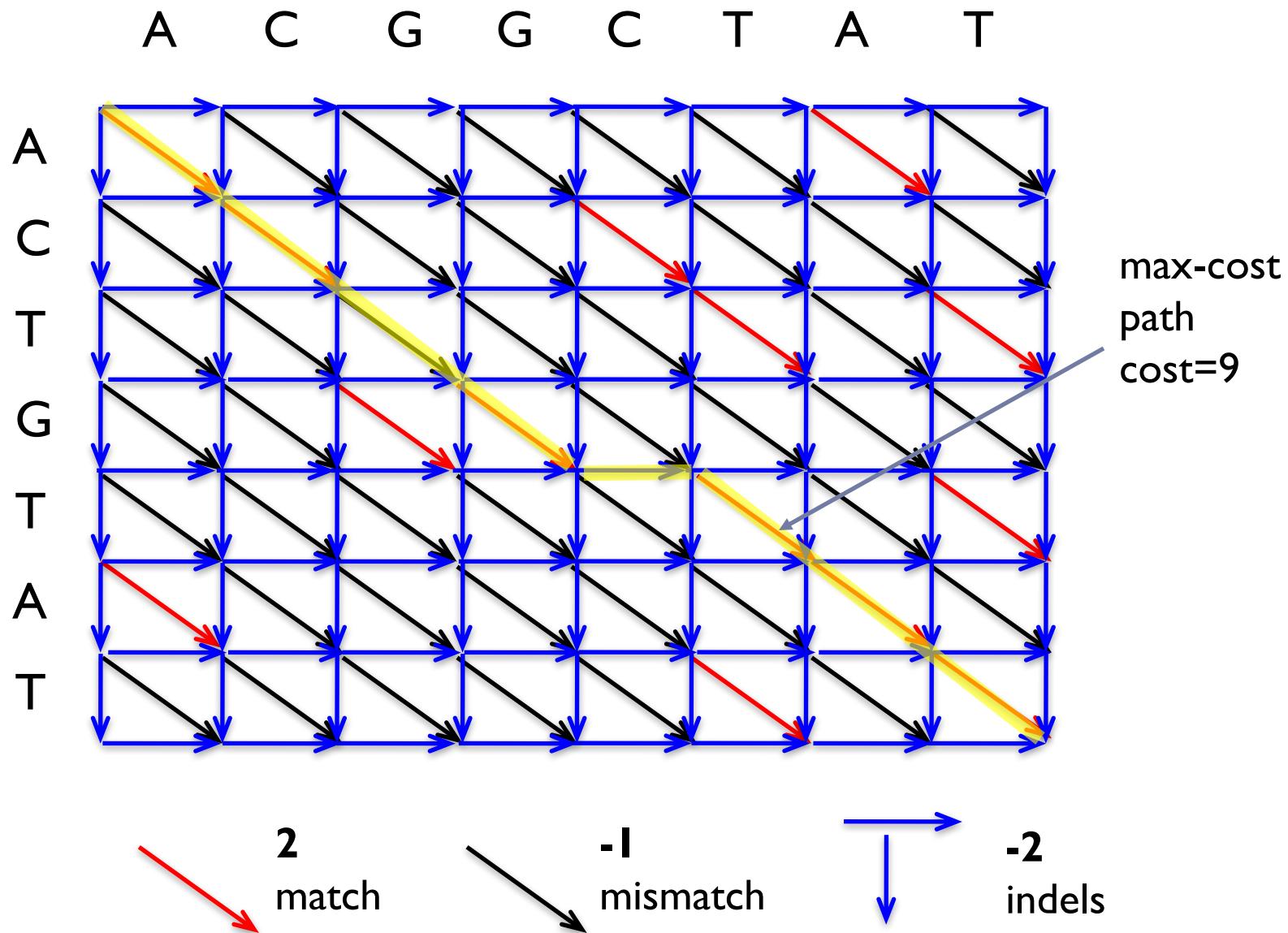
	0	1	2	3	4	5	6	7	8
0	0	-2	-4	-6	-8	-10	-12	-14	-16
A	-2	2	0	-2	-4	-6	-8	-10	-12
C	-4	0	4	2	0	-2	-4	-6	-8
T	-6	-2	2	3	1	-1	0	-2	-4
G	-8	-4	0	4	5	3	1	-1	-3
T	-10	-6	-2	2	3	4	5	3	1
A	-12	-8	-4	0	1	2	3	7	5
T	-14	-10	-6	-2	-1	0	4	5	9

Score(S,T)

Alignment: graph formulation



Alignment: graph formulation



Shortest path view (flashback)

- ▶ Previous DP solution implies that *on this graph*, the single-source shortest/longest paths problem can be solved in time $O(nm)$
- ▶ ... i.e. in time $O(|V|+|E|)$ as the graph has $O(nm)$ nodes and $O(nm)$ edges
- ▶ Not a surprise! We knew this already, cf lecture on shortest paths in DAGs
- ▶ However, no need here for topological sort because of the regular graph structure

Exercise

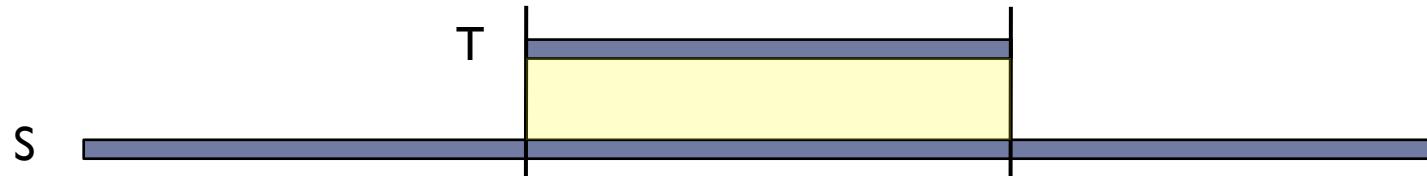
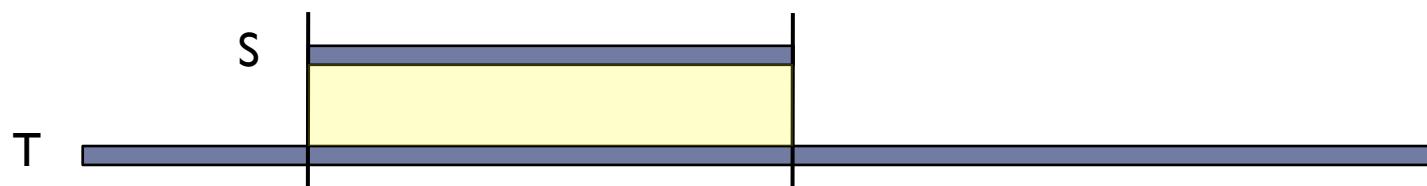
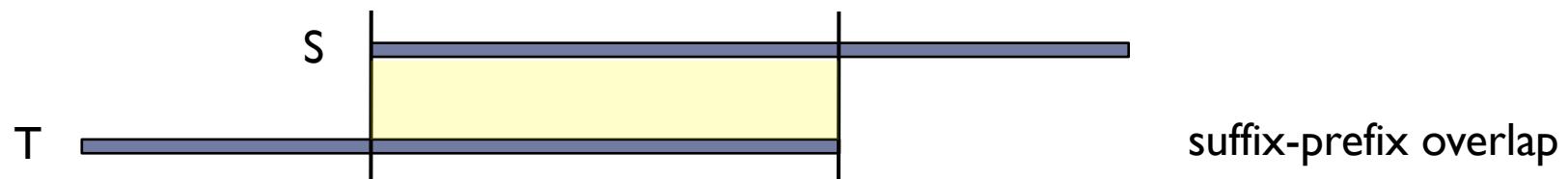
- ▶ Give all optimal alignments between ACCGTTG and CGAATGAA if the match score is 2, the mismatch penalty is -1 and the gap penalty (indel score) is -2

Comments

- ▶ algorithm known as Needleman-Wunsch algorithm (1970)
- ▶ note that optimal alignment is generally not unique
- ▶ the problem considered is called *global alignment*
- ▶ both time and space complexity is $O(n^2)$
- ▶ space complexity is $O(n)$ if only the optimum score has to be computed (e.g. line-by-line, keep two lines at a time)
- ▶ time can be reduced to $O(n^2/\log^2 n)$ (assuming RAM model) [Masek, Paterson 80] using “four-russians technique” (another solution in [Crochemore, Landau, Ziv-Ukelson 03])
- ▶ proved to be unlikely solvable in time $O(n^{2-\varepsilon})$ [Abboud, Williams, Weimann 14] (by reduction from 3SUM to some versions of alignment problem); similar result for LCS [Abboud, Backurs, Williams 15]

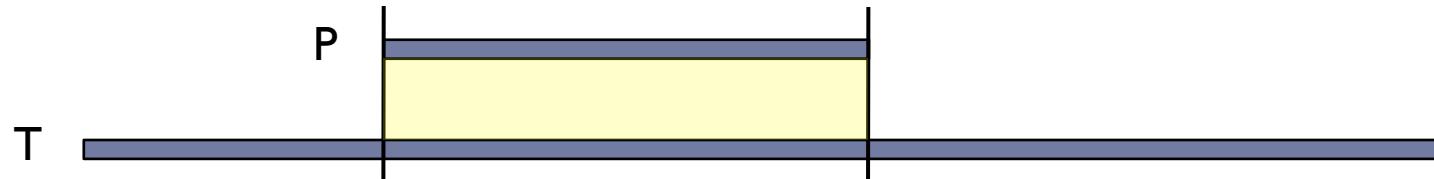
End-space free alignment

- ▶ Compute the best alignment of S and T such that spaces at string borders contribute 0



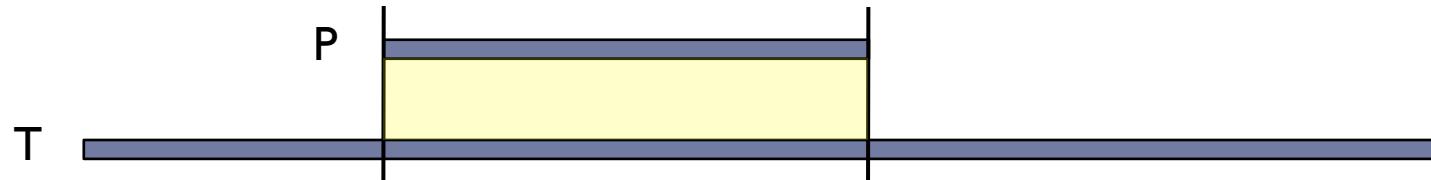
Approximate string matching

- ▶ Compute all alignments such that $\text{Score}(P, T[i..j]) > \delta$



Approximate string matching

- ▶ Compute all alignments such that $\text{Score}(S, T[i..j]) > k$



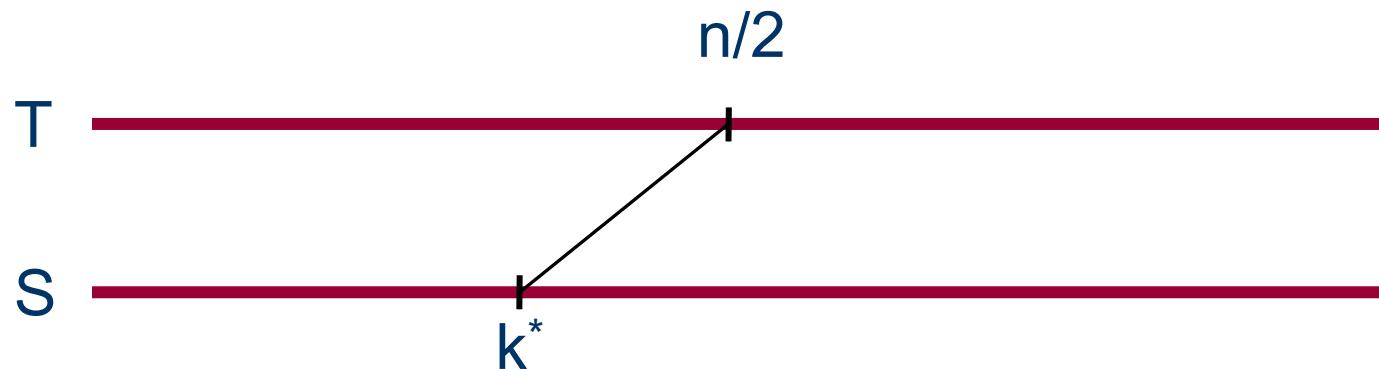
- Particular cases
 - edit distance ($< k$): $O(kn)$ [[Landau&Vishkin 85](#), [Galil&Park 89](#), ...]
 - Hamming distance: $O(n \cdot \log(m))$ [[Fischer&Paterson 73](#)], $O(nk)$ [[Galil&Giancarlo 86](#)], $O(n\sqrt{k} \cdot \log(k))$ [[Amir&Lewenstein&Porat 04](#)], ...

Computing alignment in linear space

- ▶ Hirschberg (1975) proposed a nice trick in order to compute the optimal alignment *in linear space* (at the price of doubling the time)
- ▶ Linear space is trivially sufficient if we only need to compute the *optimum score*:
 - ▶ fill entries of DP matrix line-by-line
 - ▶ keeping only the previous line is sufficient
- ▶ But how to do this if we need an *optimal alignment*?
 - ▶ if we don't keep the whole matrix, traceback becomes impossible

Hirschberg: main idea

- If we cut one sequence into two equal part, to which cut in the second sequence will it correspond?



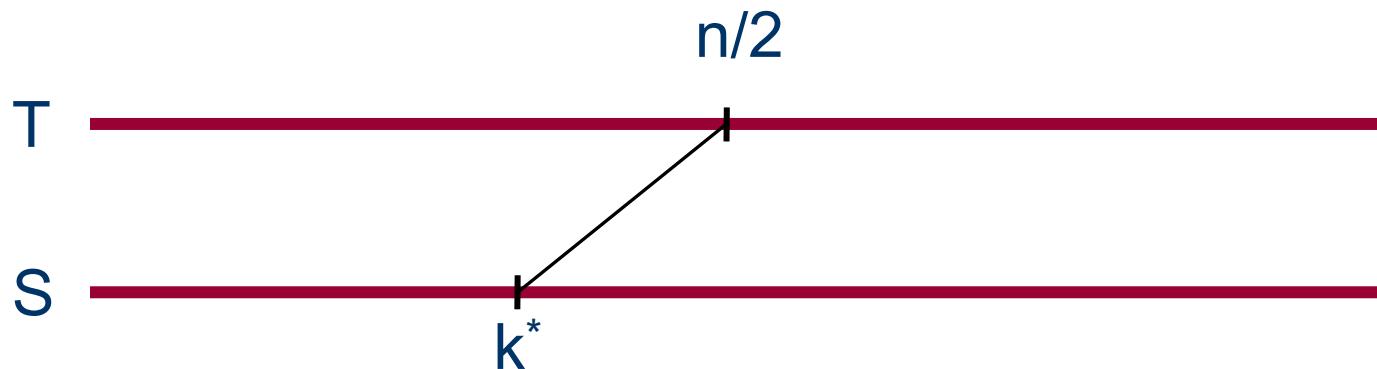
Example: $s(x,x)=1$, $s(x,y)=-1$ for $x \neq y$, $d=-2$

T = GAACTGCG

S = CAAGAC

Hirschberg: main idea

- If we cut one sequence into two equal part, to which cut in the second sequence will it correspond?



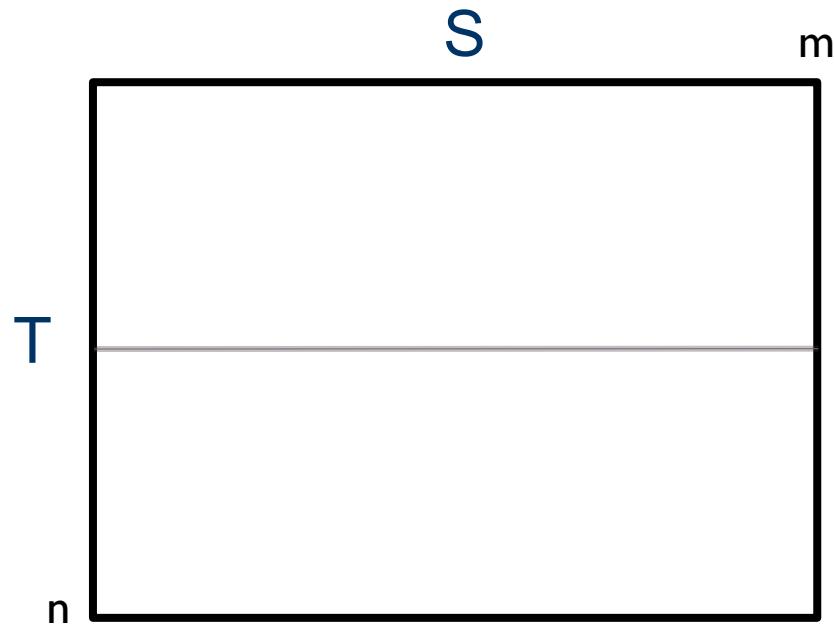
Example: $s(x,x)=4$, $s(x,y)=-1$ for $x \neq y$, $d=-2$

$T = GAAC|TGCG$

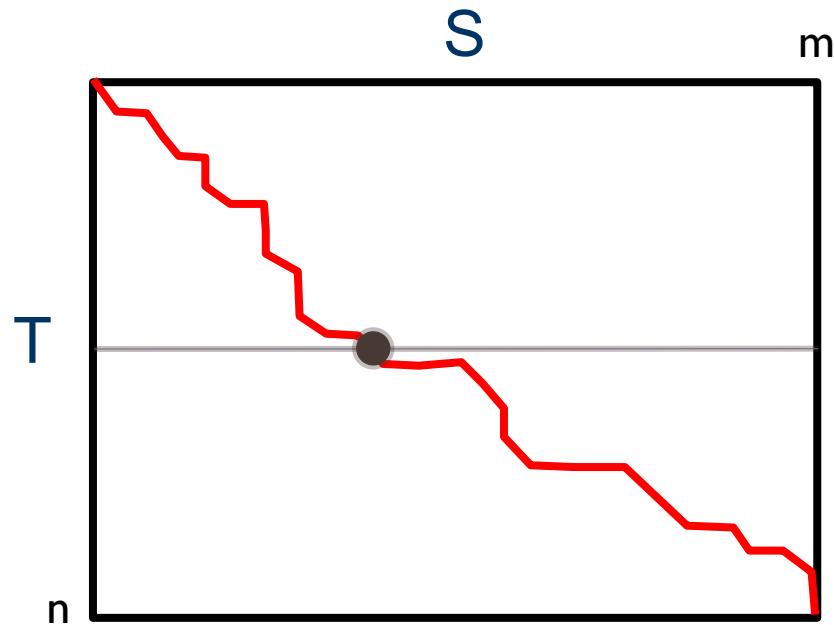
$S = CAAG|AC$

GAAC|TGCG
|| |
CAAGA-C-

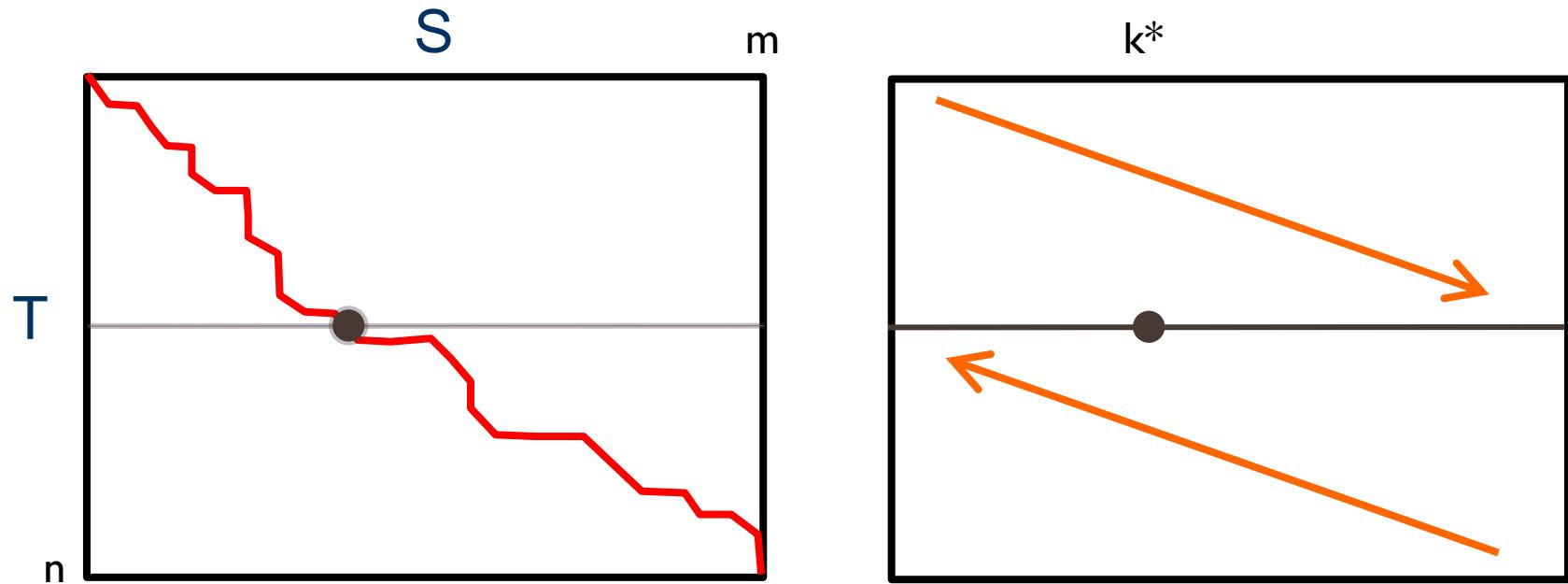
Hirschberg explained on graphs



Hirschberg explained on graphs

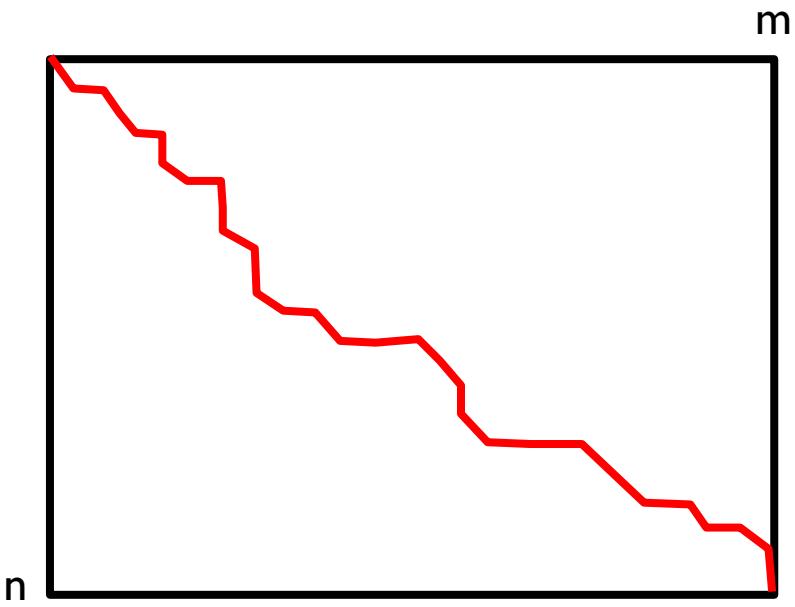


Hirschberg explained on graphs



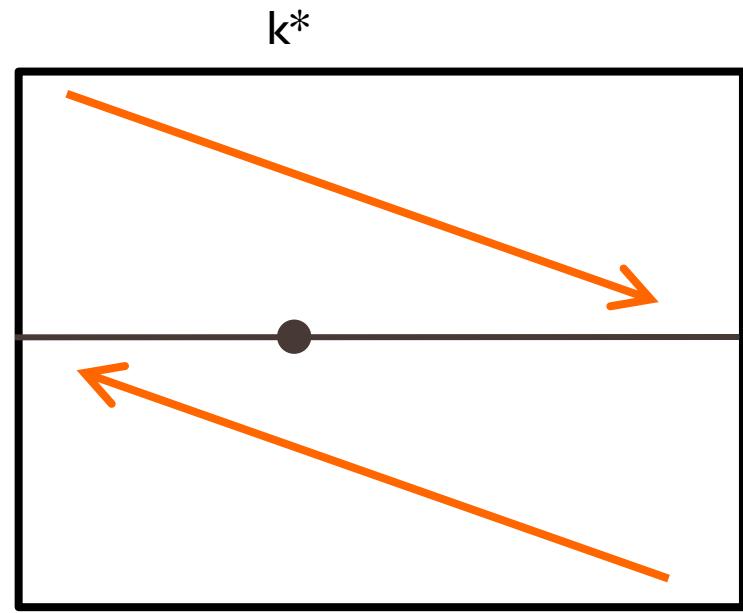
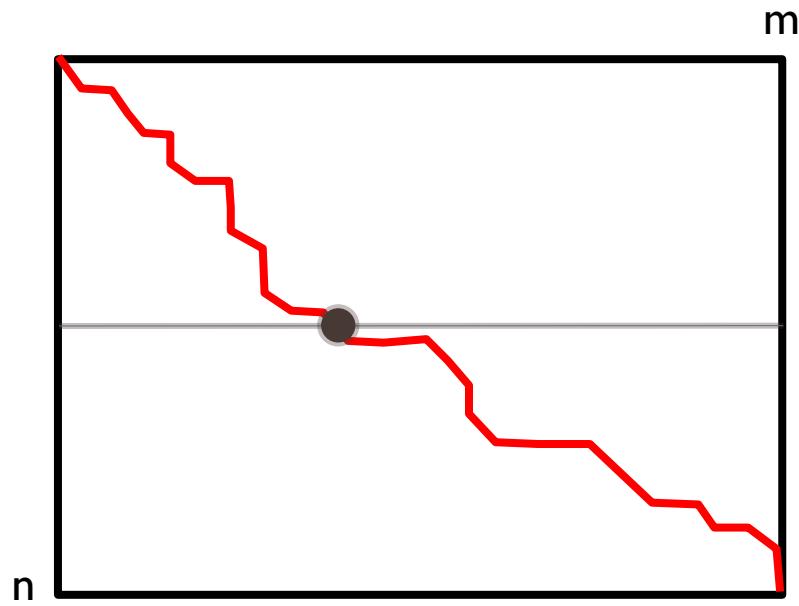
computing k^ (cf. bidirectional search):*

- for all $0 \leq k \leq m$, compute (forward) the max score $\text{Score}(n/2, k)$ of a path from $(0,0)$ to $(n/2, k)$
- for all $0 \leq k \leq m$, compute (backward) the max score $\text{Score}^R(n/2, k)$ of a path from $(n/2, k)$ to (n, m)
- choose $k^* = \arg\max_k (\text{Score}(n/2, k) + \text{Score}^R(n/2, k))$

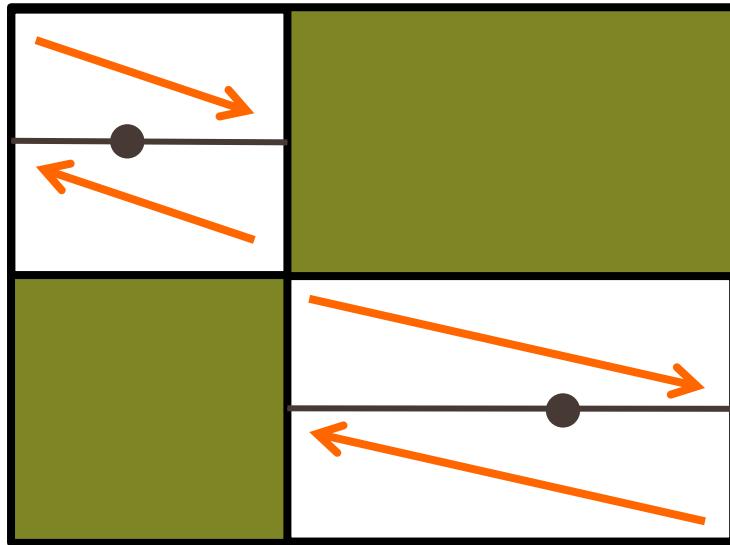
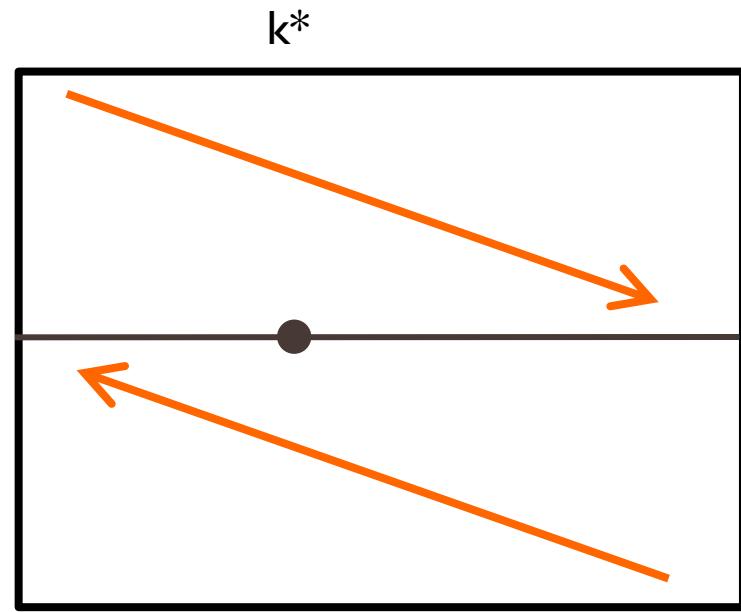
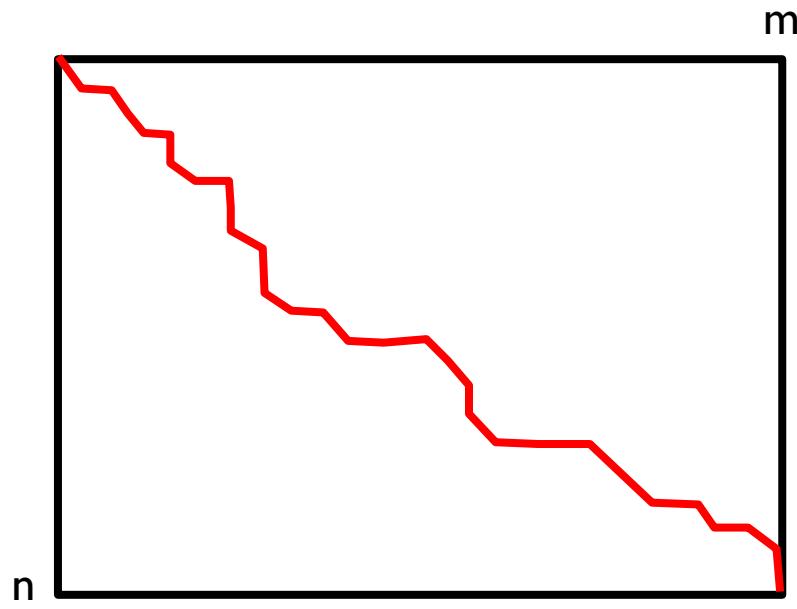


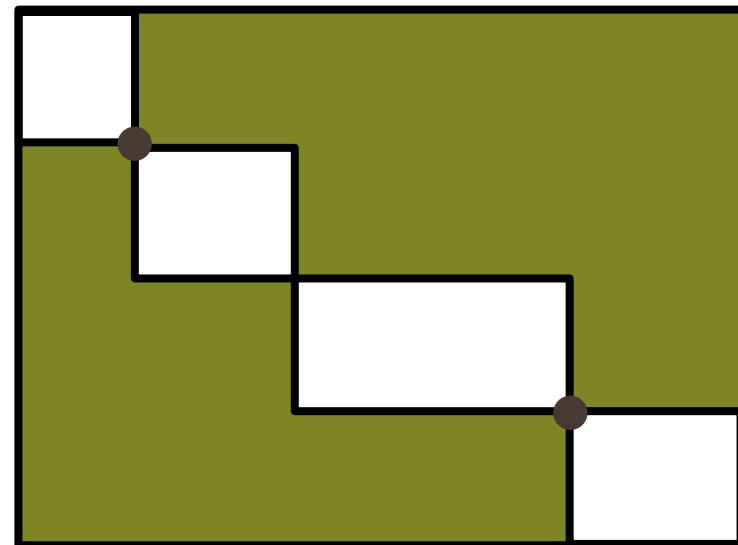
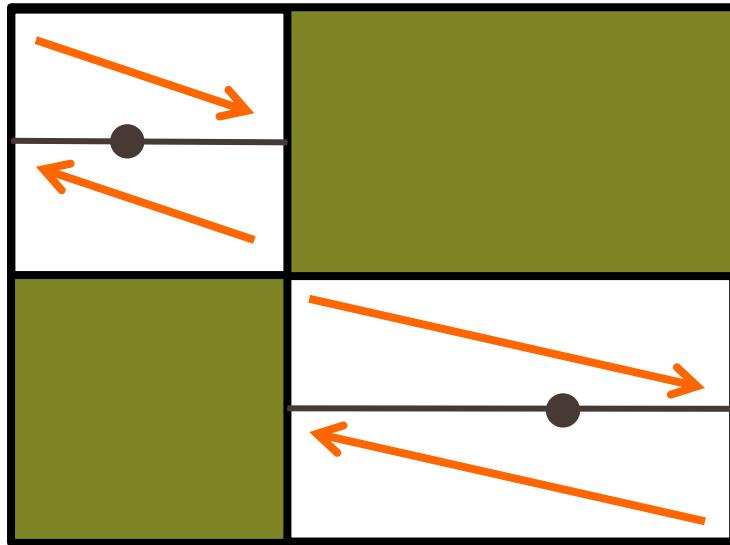
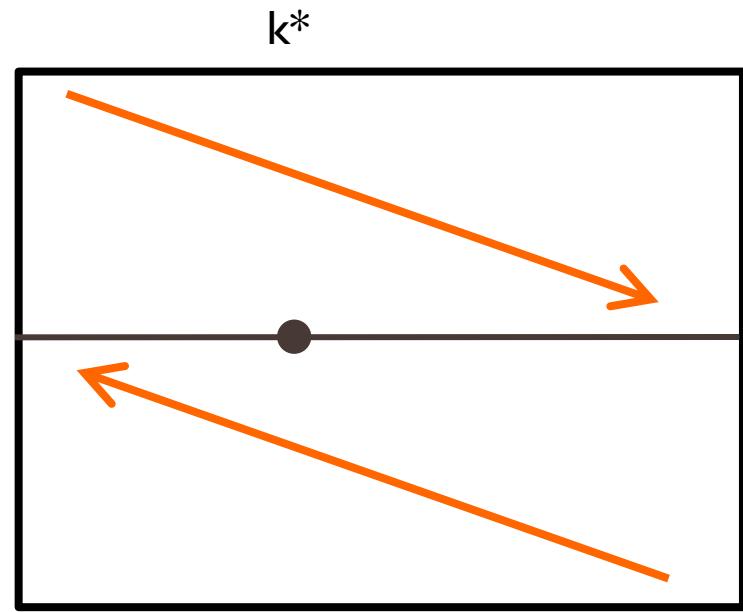
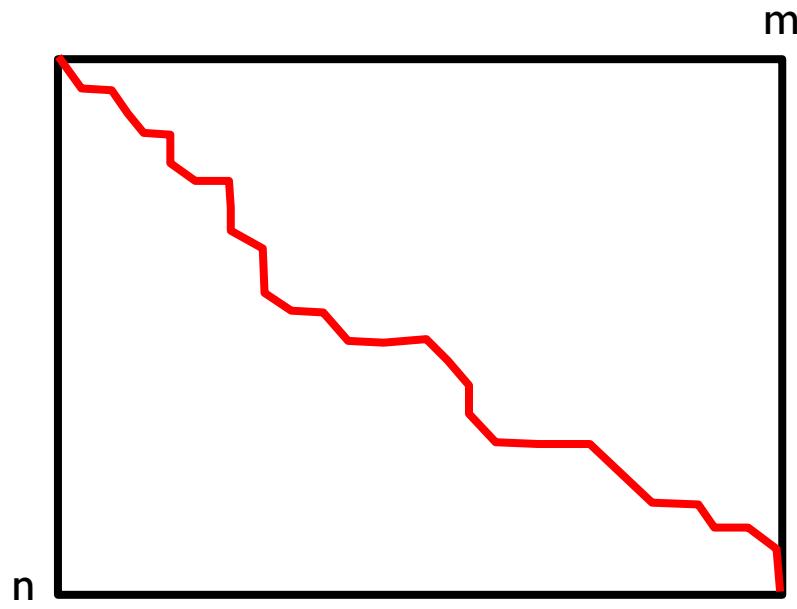
m

n



compute
 $k^* = \text{argmax}_k (\text{Score}[n/2, k] + \text{Score}^R[n/2, m-k])$



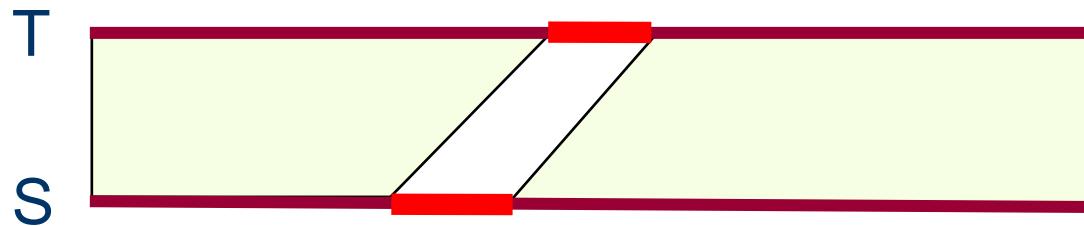


Resulting time complexity

- ▶ if the Score computation on a $p \times q$ matrix takes time $c \cdot pq$, then computing the first “cut” takes $2 \cdot c \cdot (n/2) \cdot m = c \cdot nm$
- ▶ the first halving results in time $c \cdot (n/2) \cdot k^* + c \cdot (n/2) \cdot (m - k^*) = 1/2 \cdot c \cdot nm$
- ▶ all recursive calls take time
 $c \cdot nm + 1/2 \cdot c \cdot nm + 1/4 \cdot c \cdot nm + \dots \leq 2c \cdot nm$

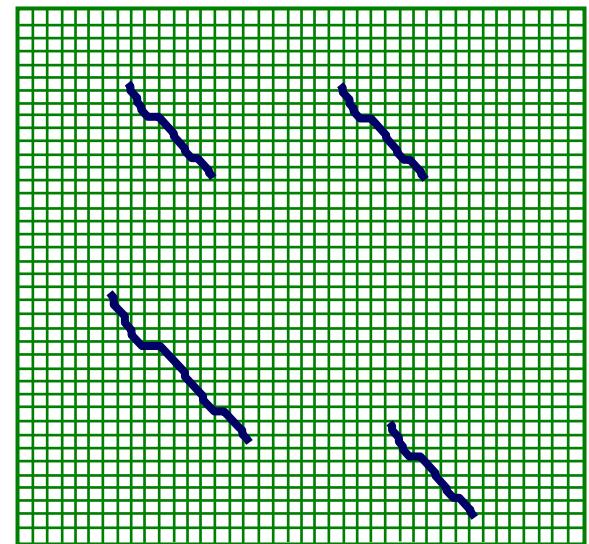
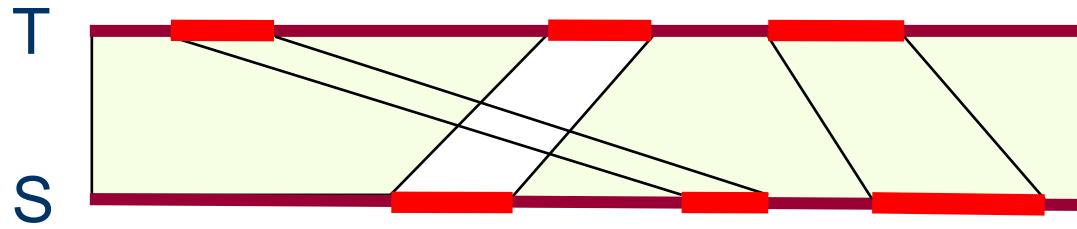
Local alignment

- ▶ Biologists are mostly interested in *local alignments* that may ignore arbitrary prefixes and suffixes of input sequences



Local alignment

- ▶ Biologists are mostly interested in *local alignments* that may ignore arbitrary prefixes and suffixes of input sequences



- *Problem:* Compute **all significant** local alignments, i.e. all alignments of score above a threshold

Smith-Waterman algorithm (1981)

- ▶ Assume matches are scored positively and mismatches/indels are scored negatively
- ▶ $\text{Score}[i,j]$: maximum score over all substrings of S that end at position i and all substrings of T that end at position j
- initialization: $\text{Score}[0,j]=\text{Score}[i,0]=0$

$$\text{Score}[i,j] = \max \left\{ \begin{array}{l} 0 \\ \text{Score}[i-1,j-1] + s(S[i], T[j]) \\ \text{Score}[i-1,j] + d \\ \text{Score}[i,j-1] + d \end{array} \right\}$$

Smith-Waterman: example

EAWACQGKL vs ERDAWCQPGKWKY

$s(x,x)=1, s(x,y)=-3$ for $x \neq y, d=-1$

T	j	0	1	2	3	4	5	6	7	8	9	10	11	12
i		$y[j]$	E	R	D	A	W	C	Q	P	G	K	W	Y
0	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
1	E	0	1	0	0	0	0	0	0	0	0	0	0	0
2	A	0	0	0	0	1	0	0	0	0	0	0	0	0
3	W	0	0	0	0	0	2	1	0	0	0	0	1	0
4	A	0	0	0	0	1	1	0	0	0	0	0	0	0
5	C	0	0	0	0	0	0	2	1	0	0	0	0	0
6	Q	0	0	0	0	0	0	1	3	2	1	0	0	0
7	G	0	0	0	0	0	0	0	2	1	3	2	1	0
8	K	0	0	0	0	0	0	0	1	0	2	4	3	2
9	L	0	0	0	0	0	0	0	0	0	1	3	2	1

resulting local alignment:

$$\begin{pmatrix} A & W & A & C & Q & - & G & K \\ A & W & - & C & Q & P & G & K \end{pmatrix}$$

Comments

- ▶ Score matrix is important
- ▶ There exists a statistical model (Karlin&Altschul 90) that allows to relate the score of a local alignment and the probability for this alignment to appear in random sequences (p-value)
- ▶ The average value of score matrix should be negative

More complex gap penalty systems

- ▶ Affine gap penalty: $h+q \cdot i$

h: gap opening penalty

q: gap extension penalty

$O(mn)$ algorithm [Gotoh 82]

- ▶ Convex gap penalty

$O(mn \cdot \log n)$

- ▶ Arbitrary gap penalty

$O(mn^2 + nm^2)$

(Hidden) Markov models

Discrete-time random processes

- ▶ $\{X_i\}_{i=1..\infty}$ where X_i are random variables
- ▶ *Our case*: X_i take a finite number of possible values (finite-space r.p.); equivalently, values of X_i are letters from a finite alphabet



362543525311...

Some examples

- ▶ Market trends: Bull market – Stagnant market – Bear market
- ▶ Credit risk grades (e.g. Standard&Poor, Moody's&Fitch): AAA – AA – A – BBB – BB – B – CCC – D
- ▶ Weather changes: Sunny – Rainy – Cloudy
- ▶ Working on EADS course in a day: Not working – Just getting an update – Spending time reading/thinking/coding
- ▶ ...

Why consider probabilistic models of sequences?

- ▶ Classification
 - ▶ Machine learning
 - ▶ Data mining
- } which category (*family, ...*) the sequence belongs to?
-
- ▶ Estimating likelihood of an observation
 - ▶ Average-case analysis of algorithms
 - ▶ ...

Knowledge assumptions

- ▶ Basics of probability theory

- ▶ Conditional probability $P(A|B) = \frac{P(A \& B)}{P(B)}$
- ▶ Bayes formula $P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$

Probabilistic model of sequences

- ▶ *Simplest model*: all letters are i.i.d. **Bernoulli** distributed random variables X_i
- ▶ Q : finite set of states (*letters*)
- ▶ $P(X_i = q) = p_q, \sum_{q \in Q} p_q = 1$

e.g. $P(A)=P(C)=P(G)=P(T)=0.25$

or $P(A)=P(T)=0.2$ and $P(C)=P(G)=0.3$

Markov chains (models)

Markov chain of order k : probability distribution of X_i depends on values of $X_{i-1}, X_{i-2}, \dots, X_{i-k}$



A.A.Марков (1856-1922)

Specified by probabilities

$P(X_i = q_0 | X_{i-1} = q_1, \dots, X_{i-k} = q_k)$ for $q_0, q_1, \dots, q_k \in Q$
and initial probabilities $P(X_0 = q)$ for $q \in Q$

ПЕРВЫЙ ЧИСТЕРІ СОРСНОУ АКАДЕМІІІ СЛУЖБА. — 1913.
(Bulletin de l'Académie Impériale des Sciences de St.-Pétersbourg).

Примѣръ статистическаго изслѣдованія надъ
текстомъ „Евгения Онѣгина“ иллюстрирующій
связь испытаній въ цѣль.

А. А. Марковъ.

(Докладено въ засѣданіи Физико-Математического Отдѣленія 23 января 1913 г.).

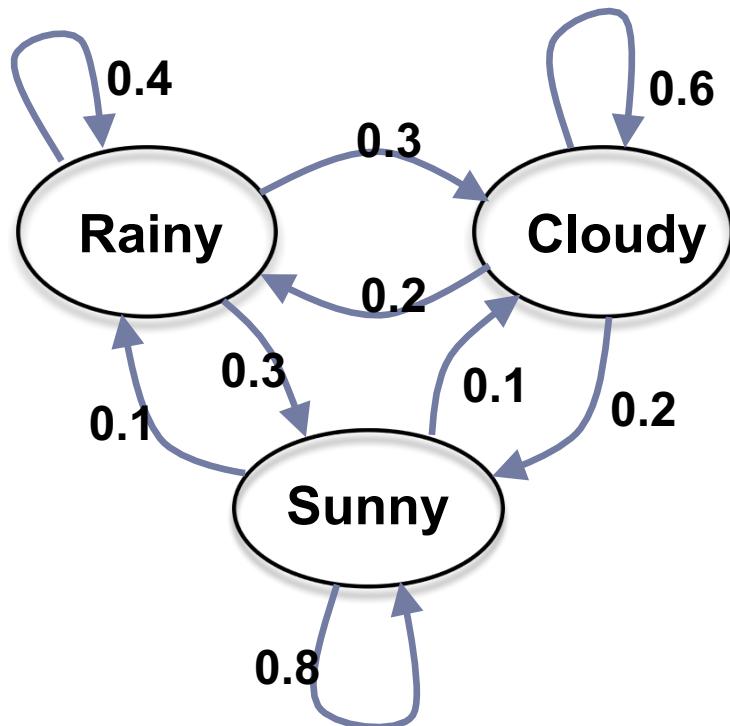
Наше изслѣдованіе относится къ послѣдовательности 20 000 русскихъ буквъ, не считая ъ и ь, въ романѣ А. С. Пушкина «Евгений Онѣгинъ», которая заключаетъ всю первую главу и шестьнадцать строфъ второй.

Эта послѣдовательность доставляетъ намъ 20 000 связанныхъ испытаний, каждое изъ которыхъ даетъ гласную или согласную букву.

Markov chains: example

- ▶ *Example:* assume three states {Rainy, Cloudy, Sunny}, and a Markov chain of order 1 (first order):

$$\begin{pmatrix} 0.4 & 0.3 & 0.3 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$



Markov chains (cont)

- If the weather today is **Sunny**, than the proba of following **S-S-R-R-S-C-S** is

$$P(SSSRRSCS|\text{Model}) =$$

$$P(S) \cdot P(S|S)^2 \cdot P(R|S) \cdot P(R|R) \cdot P(S|R) \cdot P(C|S) \cdot P(S|C) = \\ 1 \cdot 0.8^2 \cdot 0.1 \cdot 0.4 \cdot 0.3 \cdot 0.1 \cdot 0.2 \approx 1.536 \cdot 10^{-4}$$

- *Example*: given that today is **Cloudy**, what is the proba that it will be **Rainy** the day after tomorrow?

Markov chains (cont)

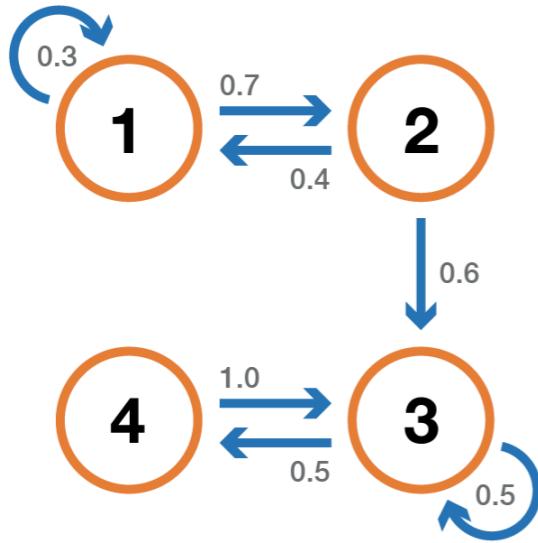
- ▶ Given that the model is in a known state, what is the probability it stays in that state for exactly d days?
- ▶ The answer is $p_i(d) = a_{ii}^{d-1}(1 - a_{ii})$
- ▶ Thus the expected number of consecutive days in the same state is $d_i = \frac{1}{1 - a_{ii}}$
- ▶ So the expected number of consecutive **Sunny** days, according to the model is 5.

Markov chains (cont)

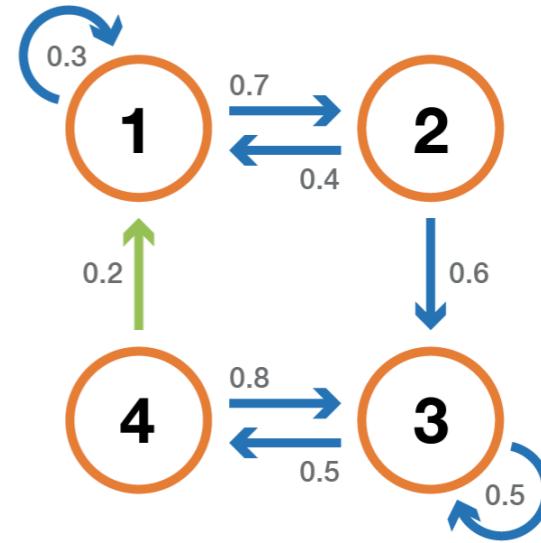
- ▶ Consider a first-order Markov chain and let $Q = \{1, \dots, |Q|\}$
- ▶ Let $\mathcal{P} = (p_{kl})$ be the matrix of transition probabilities, i.e. $p_{kl} = P(X_i = l | X_{i-1} = k)$
- ▶ Then $P(X_i = l | X_{i-2} = k) = \sum_{j=1}^{|Q|} p_{kj} \cdot p_{jl}$, i.e. “two-step probabilities” are specified by matrix \mathcal{P}^2
- ▶ Similarly, “ m -step probabilities” are specified by matrix \mathcal{P}^m
- ▶ $I \cdot \mathcal{P}^m$: probability distribution of states after m steps, starting from initial distribution I (row vector)

Properties of Markov chains

- ▶ **Irreducible** : each state reachable from any other (graph strongly connected)



not irreducible



irreducible

Properties of Markov chains (cont)

- ▶ **Irreducible** : each state reachable from any other (graph strongly connected)
- ▶ **Theorem**: in the case of finite number of states¹, each irreducible Markov chain has a stationary distribution $\beta = (\beta_1, \dots, \beta_{|Q|})$, where β_i is the limit fraction of visits to state i (equivalently, β_i is the probability that at a *random* moment $t \in [1..T]$, $T \rightarrow \infty$, the chain is at state i)
- ▶ This stationary distribution is the solution of $\beta = \beta \cdot \mathcal{P}$

¹for an infinite number of states, things are more complicated

Example

- ▶ For the (Rainy, Cloudy, Sunny) example

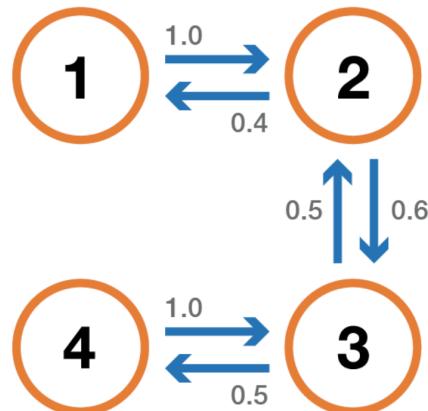
$$(\beta_1, \beta_2, \beta_3) = (\beta_1, \beta_2, \beta_3) \cdot \begin{pmatrix} 0.4 & 0.3 & 0.3 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$

- ▶ Resolving under condition $\beta_1 + \beta_2 + \beta_3 = 1$ gives

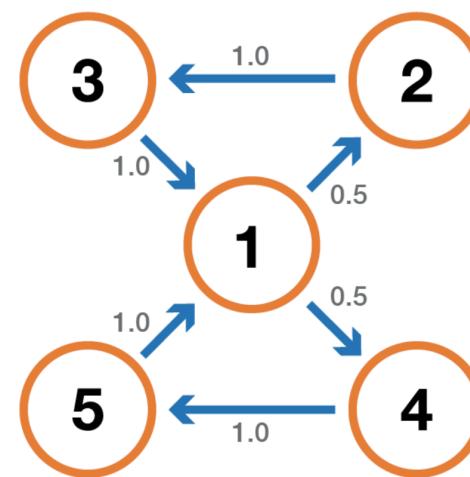
$$(\beta_1, \beta_2, \beta_3) = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$$

Properties of Markov chains (cont)

- ▶ A state is **periodic** with period k if, starting from this state, the chain must take a multiple of k steps to return back. If $k = 1$, then the state is **aperiodic**. The chain is **aperiodic** if all its states are aperiodic.



periodic $k = 2$



periodic $k = 3$

Properties of Markov chains (cont)

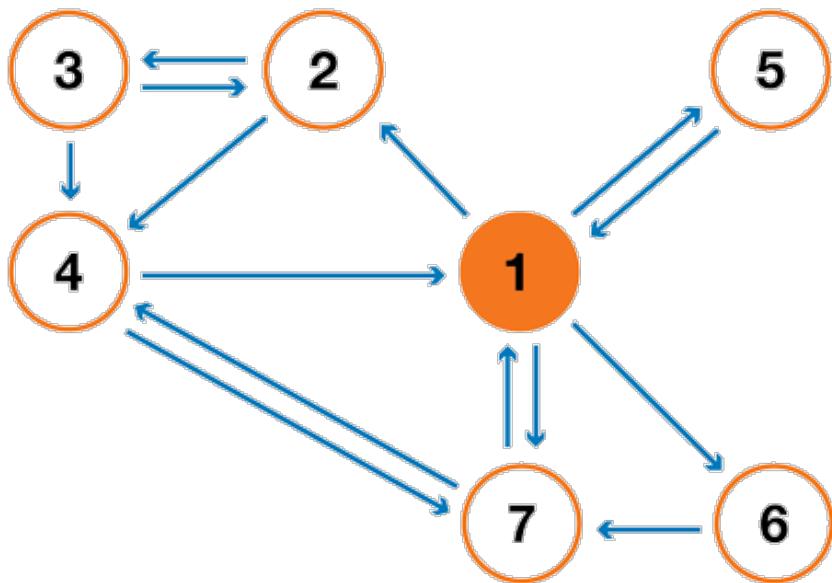
- ▶ If a chain is irreducible and aperiodic, it is called **ergodic**
- ▶ For an ergodic chain, the stationary distribution applies, in the limit, to *any* state (independent on the initial distribution). Formally

$$\beta_i = \lim_{t \rightarrow \infty} P(X_t = i)$$

Example: PageRank

- ▶ Consider
 - ▶ a set of webpages linked by hyperlinks
 - ▶ a person randomly navigating through these webpages
- ▶ This defines a Markov chain
- ▶ If it is irreducible and aperiodic, then it has a stationary distribution of “current pages”
- ▶ **Assumption (behind PageRank):** this distribution reflects the importance of pages (more probable pages are more important)

Example



$$\mathcal{P} = \begin{pmatrix} . & 0.25 & . & . & 0.25 & 0.25 & 0.25 \\ . & . & 0.5 & 0.5 & . & . & . \\ . & 0.5 & . & 0.5 & . & . & . \\ 0.5 & . & . & . & . & . & 0.5 \\ 1.0 & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ 0.5 & . & . & 0.5 & . & . & . \end{pmatrix}$$

Trajectory : 1

solving $\pi = \pi \cdot \mathcal{P}$ gives

$$\pi = (0.29, 0.10, 0.05, 0.19, 0.07, 0.07, 0.24)$$

Hidden Markov models

Hidden Markov Model (HMM)

- ▶ at each moment the model is at one of the “hidden states” (finite number)
- ▶ Each state has its own probability distribution of moving to another state (*transition probabilities*). Altogether, they define a *Markov chain* on the states
- ▶ each hidden state holds a (Bernoulli) distribution for “emitting” observations (*emission probabilities*)
- ▶ While in a certain state, the machine randomly decides:
 - ▶ what is the next state
 - ▶ what symbol is emitted
- ▶ *Example*: you don't know the weather (S,C,R) but you observe if the person you see carries an umbrella, and $P(\text{umbrella} | S)=0.05$, $P(\text{umbrella} | C)=0.2$, $P(\text{umbrella} | R)=0.9$

CpG-Islands

- ▶ Given 4 nucleotides: probability of occurrence is $\sim 1/4$. Thus, probability of occurrence of a dinucleotide is $\sim 1/16$.
- ▶ However, the frequencies of dinucleotides in DNA sequences vary widely.
- ▶ In particular, *CG* is typically underrepresented (frequency of *CG* is typically $< 1/16$)
- ▶ E.g. the human genome has 42% of {C,G}, i.e. the expected frequency of *CG* is $0.21 \cdot 0.21 \approx 4\%$ whereas the real frequency is $\approx 1\%$

Why CpG-Islands?

- ▶ **CG** is the least frequent dinucleotide because C in CG is easily methylated and has the tendency to mutate into T afterwards (cf epigenetics)
- ▶ However, the methylation is suppressed around genes in a genome. So, CG appears at relatively high frequency within these CpG islands
- ▶ So, finding the CpG islands in a genome is an important problem

Left: CpG sites at 1/10 nucleotides, constituting a CpG island. The sample is of a gene-promoter, the highlighted ATG constitutes the start codon.

Right: CpG sites present at every 1/100 nucleotides, constituting a more normal example of the genome, or a region of the genome that is commonly methylated.

CpG Islands and the “Fair Bet Casino”

- ▶ The *CG* islands problem can be modeled after a problem named “*The Fair Bet Casino*”
- ▶ The game is to flip coins, which results in only two possible outcomes: **H**ead or **T**ail
- ▶ The **F**air coin will give **H**eads and **T**ails with same probability 0.5 : $P(H|F) = P(T|F) = 0.5$
- ▶ The **B**iased coin will give **H**eads with probability 0.75 : $P(H|B) = 0.75, P(T|B) = 0.25$
- ▶ The dealer changes between **F**air and **B**iased coins with probability 0.1

The Fair Bet Casino Problem

- ▶ ***Input:*** sequence $x = x_1 x_2 x_3 \dots x_n$ of coin tosses, $x_i \in \{\mathbf{H}, \mathbf{T}\}$ made by two possible coins
- ▶ ***Output:*** sequence $\pi = \pi_1 \pi_2 \pi_3 \dots \pi_n$, with each π_i being either **F** or **B** indicating that x_i is the result of tossing the **Fair** or **Biased** coin respectively

Decoding problem

- ▶ Any observed outcome of coin tosses could have been generated by any sequence of states
- ▶ Goal: compute the *most likely* sequence π producing x , i.e. π maximizing $P(\pi|x)$
- ▶ This problem is called the *decoding problem*

Warm-up: what if the coin stays the same?

- ▶ Assume that the dealer never changes the coin
 - ▶ $P(x|\mathbf{F})$: probability of the outcome x provided that the dealer uses the **F** coin all along
 - ▶ $P(x|\mathbf{B})$: same if the dealer uses the **B** coin
- ▶ $P(x|\mathbf{F}) = P(x_1 \dots x_n | \mathbf{F}) = \prod_{i=1,n} P(x_i | \mathbf{F}) = (1/2)^n$
- ▶ $P(x|\mathbf{B}) = P(x_1 \dots x_n | \mathbf{B}) = (3/4)^k (1/4)^{n-k} = 3^k / 4^n$ where k is the number of **Heads** in x

What if the coin stays the same? (cont)

- ▶ $P(x|\mathbf{F})=P(x|\mathbf{B}) \Rightarrow (1/2)^n = 3^k/4^n \Rightarrow k = n / \log_2 3$ ($k \sim 0.63n$)
- ▶ We can compute the *log-odds ratio* to measure the discrimination of **F** vs **B**:

$$\log_2(P(x|\mathbf{F})/ P(x|\mathbf{B})) = n - k \log_2 3$$

HMM Parameters

$A = \langle \Sigma, Q, A, E, I \rangle$

Σ : set of emission characters (observations)

Ex.: $\Sigma = \{H, T\}$ for coin tossing

Q : set of hidden states, each emitting symbols from Σ

$Q = \{F, B\}$ for coin tossing

HMM Parameters (cont)

$A = (a_{kl})$: a $|Q| \times |Q|$ matrix of probability of changing from state k to state l

$$a_{FF} = 0.9 \quad a_{FB} = 0.1$$

$$a_{BF} = 0.1 \quad a_{BB} = 0.9$$

$E = (e_k(b))$: a $|Q| \times |\Sigma|$ matrix of probability of emitting symbol b while being in state k

$$e_F(T) = 0.5 \quad e_F(H) = 0.5$$

$$e_B(T) = 0.25 \quad e_B(H) = 0.75$$

$I = (p_o(k))$: initial state probabilities for every state $k \in Q$

Summary: HMM for Fair Bet Casino

- ▶ The *Fair Bet Casino* in *HMM* terms:

$$\Sigma = \{\mathbf{T}, \mathbf{H}\}$$

$\mathbf{Q} = \{\mathbf{F}, \mathbf{B}\}$: **F** for Fair and **B** for Biased coin

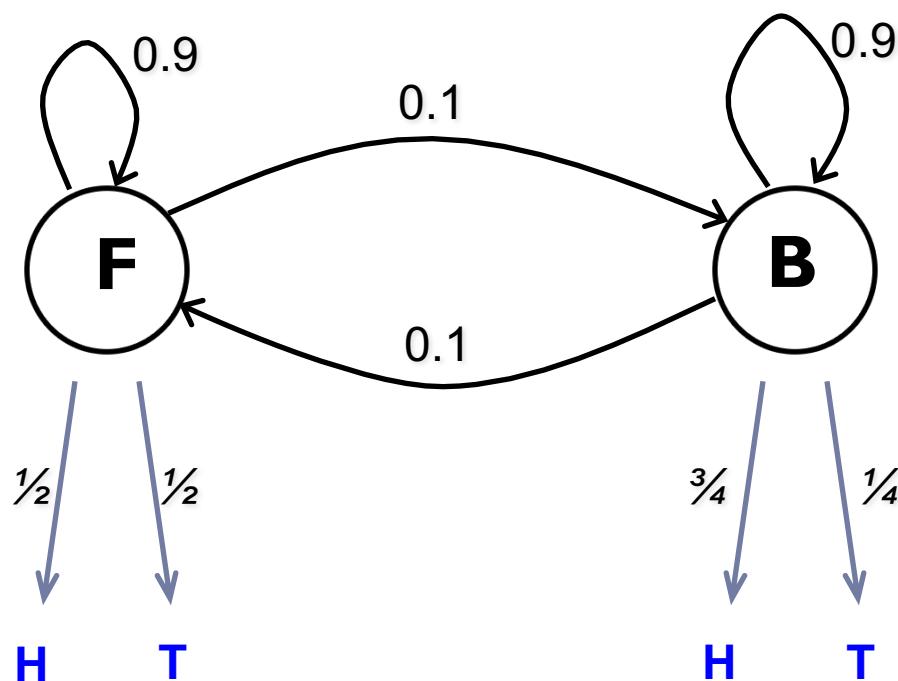
Transition Probabilities A

	Fair	Biased
Fair	$a_{FF} = 0.9$	$a_{FB} = 0.1$
Biased	$a_{BF} = 0.1$	$a_{BB} = 0.9$

Emission Probabilities E

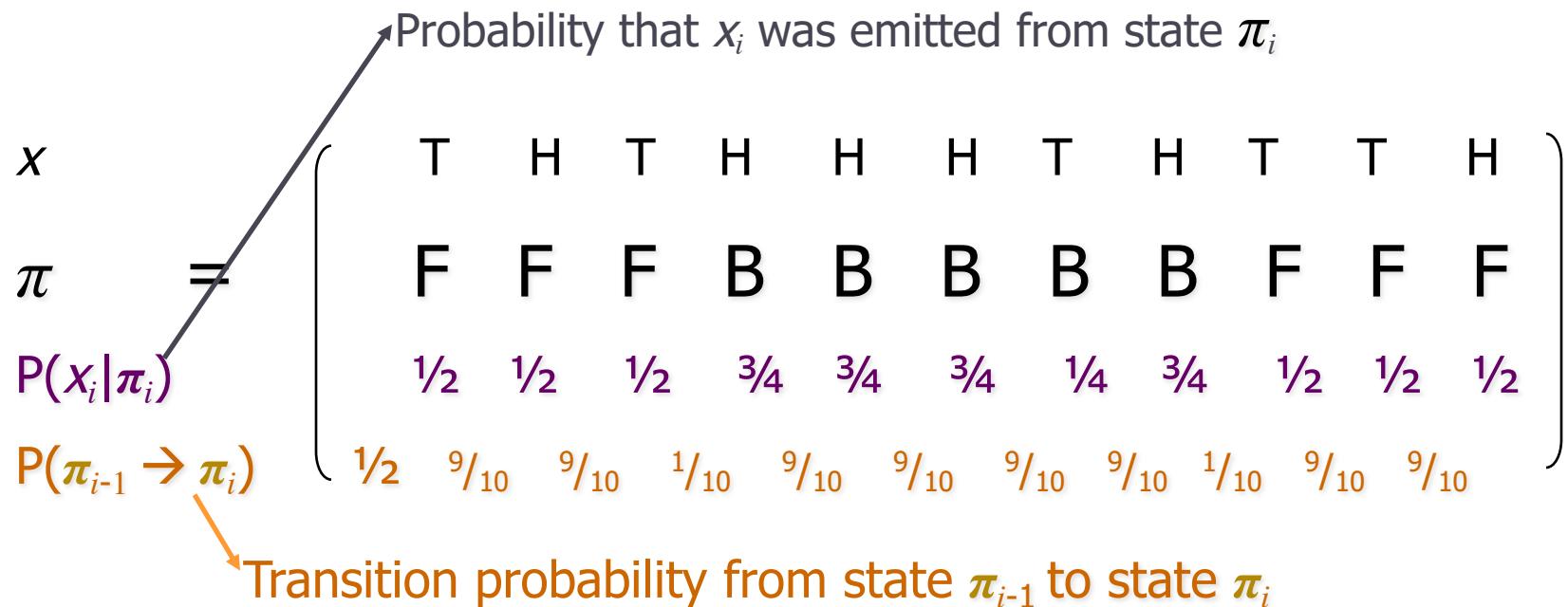
	Tails(0)	Heads(1)
Fair	$e_F(T) = \frac{1}{2}$	$e_F(H) = \frac{1}{2}$
Biased	$e_B(T) = \frac{1}{4}$	$e_B(H) = \frac{3}{4}$

HMM for Fair Bet Casino (cont)



Hidden Paths

- ▶ A **path** $\pi = \pi_1 \dots \pi_n$ in the HMM is defined as a sequence of states.
- ▶ Consider path $\pi = \text{FFFFBBBBBFFFF}$ and sequence $x = \text{THTHHHHTHTTH}$



$P(x|\pi)$ Calculation

- ▶ $P(x|\pi)$: Probability that sequence x was generated by the path π :

$$P(x|\pi) = \prod_{i=1}^n P(x_i | \pi_i) \cdot P(\pi_{i-1} \rightarrow \pi_i)$$

assuming that $P(\pi_0 \rightarrow \pi_1)$ is the probability $P(\pi_1)$ of π_1 to be the starting state

Decoding Problem

- ▶ **Goal:** Find an "optimal" (most likely) hidden path of states given observations.
- ▶ **Input:** Sequence of observations $x = x_1 \dots x_n$ generated by an HMM $M(\Sigma, Q, A, E)$
- ▶ **Output:** A path that maximizes $P(x|\pi)$ over all possible paths $\pi = \pi_1 \dots \pi_n$
- ▶ observe that $\max_{\pi} P(\pi|x) = \max_{\pi} P(\pi \text{ and } x) / P(x)$, i.e. we have to compute $\pi^* = \operatorname{argmax}_{\pi} P(\pi \text{ and } x)$

Viterbi algorithm (1967)



- ▶ Consider prefix $x_1 \dots x_i$
- ▶ For each hidden state $\pi_i = l$, let $s_{l,i}$ be the maximum probability (over $i-1$ previous visited states) to observe $x_1 \dots x_i$ and arrive to state l
- ▶ Why computing $s_{l,i}$? Because then we can extract the sequence of states π^* that realizes $\max\{s_{l,n} \mid l \in Q\}$

Viterbi algorithm (1967)

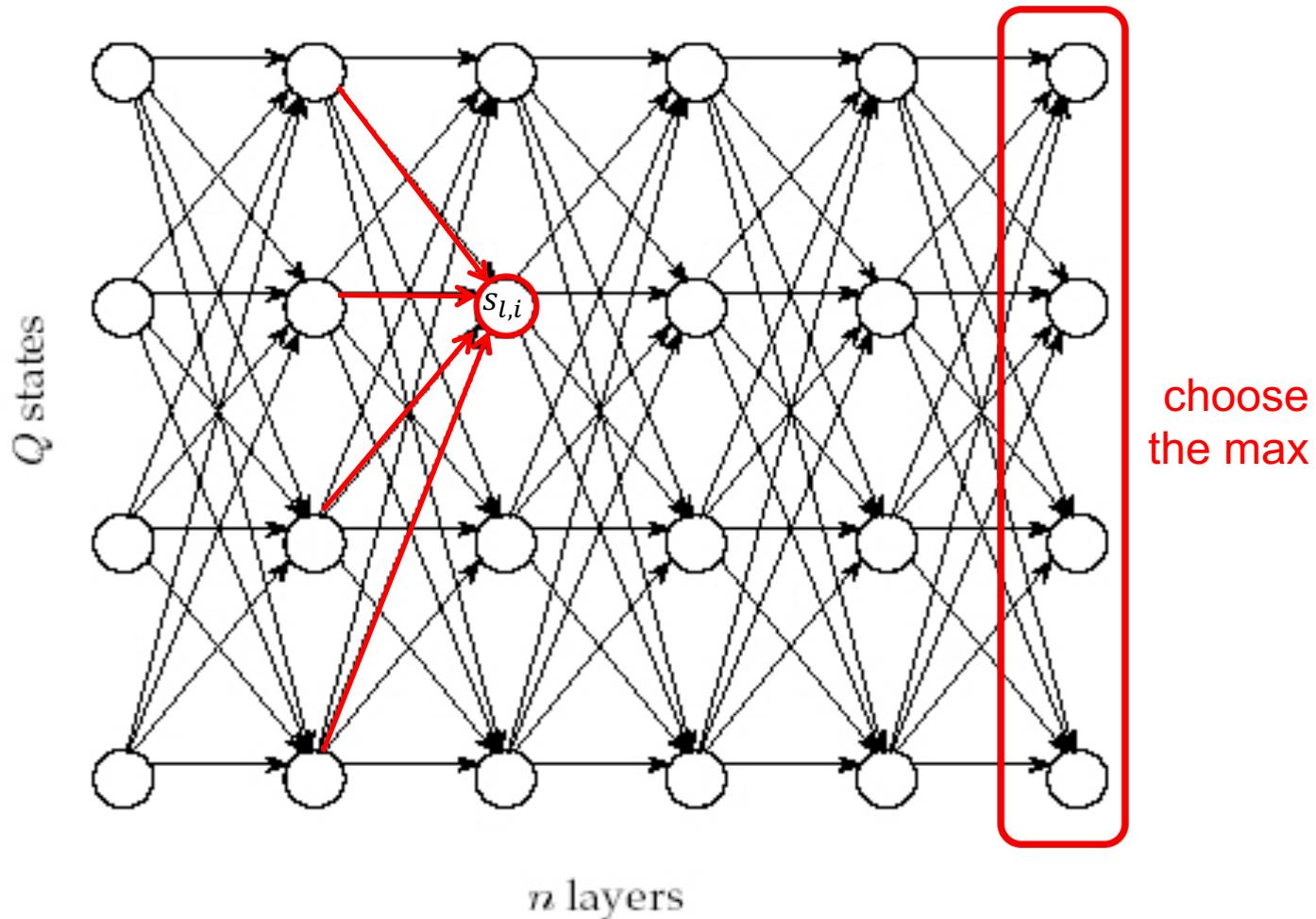


- ▶ Consider prefix $x_1 \dots x_i$
- ▶ For each hidden state $\pi_i = l$, let $s_{l,i}$ be the maximum probability (over $i-1$ previous visited states) to observe $x_1 \dots x_i$ and arrive to state l
- ▶ Why computing $s_{l,i}$? Because then we can extract the sequence of states π^* that realizes $\max\{s_{l,n} \mid l \in Q\}$
- ▶ How to compute $s_{l,i}$? By dynamic programming!

$$s_{l,i} = \max_{k \in Q} \{ s_{k,i-1} \cdot a_{k,l} \cdot e_l(x_i) \} = \\ e_l(x_i) \cdot \max_{k \in Q} \{ s_{k,i-1} \cdot a_{k,l} \}$$

DP implementation

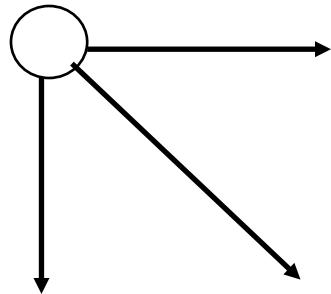
Consider the graph



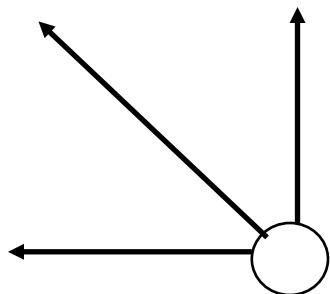
DP implementation

- ▶ Every choice of $\pi = \pi_1 \dots \pi_n$ corresponds to a path in the graph.
- ▶ *Initialization*: $s_{l,0}$ = probability for the model to start from l
- ▶ *DP recurrence*: $s_{l,i} = \max_{k \in Q} \{s_{k,i-1} \cdot a_{k,l} \cdot e_l(x_i)\}$
- ▶ Resulting path π is retrieved by "backtracing" starting from node $\operatorname{argmax}\{s_{l,n} \mid l \in Q\}$
- ▶ The graph has $\leq |Q|^2 n$ edges
- ▶ time complexity $O(|Q|^2 n)$

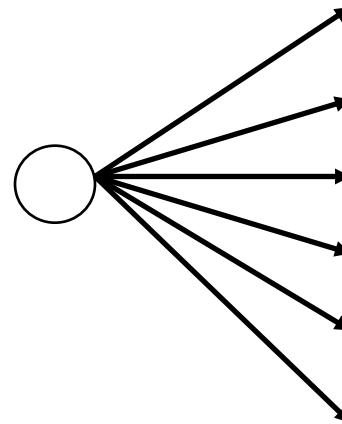
Decoding Problem vs. Alignment Problem



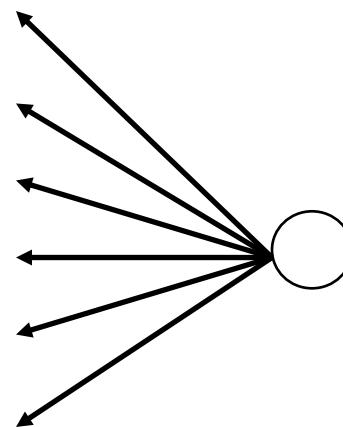
Valid directions in the *alignment problem*.



Computation in the *alignment problem*.



Valid directions in the *decoding problem*.



Computation in the *decoding problem*.

Decoding Problem as Finding a heaviest Path in a DAG

- ▶ The *Decoding Problem* can be reduced to finding a “heaviest” path in the *directed acyclic graph (DAG)*
- ▶ *Main difference:* the weight of the path is defined as the *product* of its edges’ weights, not the *sum*

```

Rolls      315116246446644245311321631164152133625144543631656626566666
Die       FFFFFFFF FFFFFFFF FFFFFF FFFFFF FFFF FFFF FFFF FFFF FFFF LLLL LLLL LLLL
Viterbi   FFFFFFFF FFFF LLLL LLLL LLLL

Rolls      65116645313265124563666463163666316232645523626666625151631
Die       LLLL LLLL FFFF FFFF
Viterbi   LLLL LLLL FFFF FFFF

Rolls      222555441666566563564324364131513465146353411126414626253356
Die       FFFF FFFF FLL LLLL LLLL LLLL LLLL FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FLL
Viterbi   FFFF FLL

Rolls      366163666466232534413661661163252562462255265252266435353336
Die       LLLL LLLL LLLL FFFF FFFF
Viterbi   LLLL LLLL LLLL FFFF FFFF

Rolls      233121625364414432335163243633665562466662632666612355245242
Die       FFFF FFFF
Viterbi   FFFF FFFF

```

300 rolls of a Fair (F) or Loaded (L) die and the prediction of Fair/Loaded by the Viterbi algorithm

(from Durbin et al. Biological sequence analysis, 1998)



Computer arithmetic problems

- ▶ The value of the product can become extremely small, which leads to overflowing
- ▶ To avoid overflowing, always use log value instead

$$s_{k,l} = \log e_l(x_i) + \max_{k \in Q} \{s_{k,i-1} + \log a_{k,l}\}$$

Example

- ▶ Two hidden states: *raining, not-raining*
- ▶ Proba to stay in the same state is 0.7, to change 0.3
- ▶ Probabilities modelling the person's behaviour:

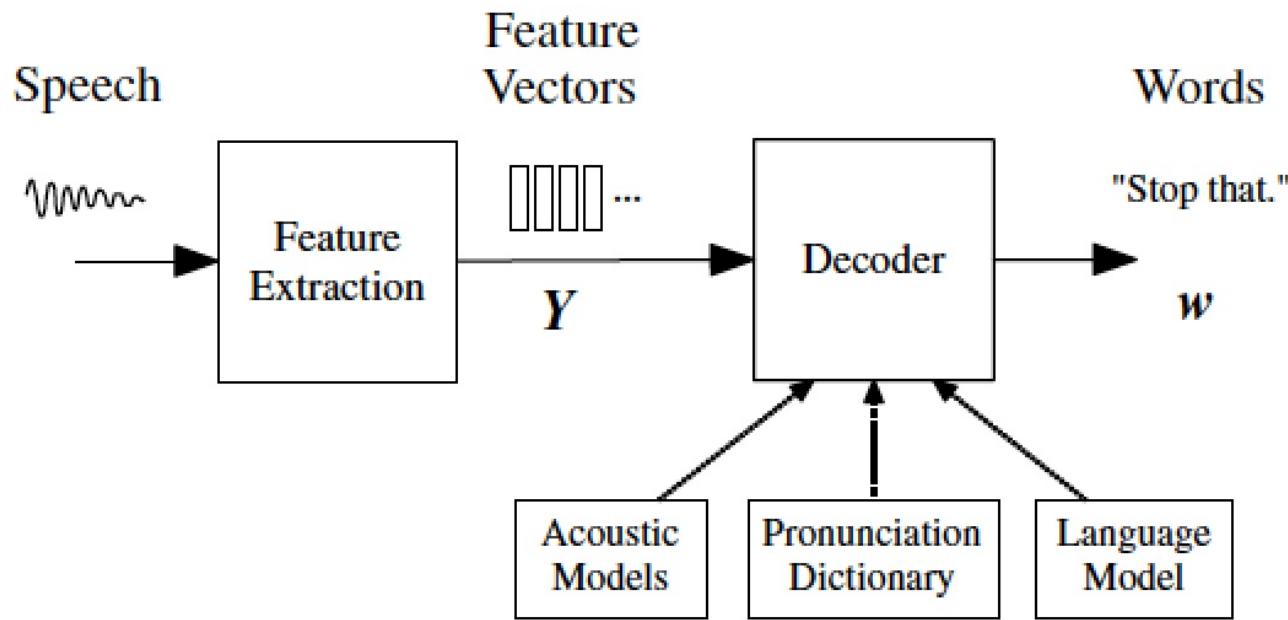
	umbrella	no umbrella
raining	0.9	0.1
not raining	0.2	0.8

- ▶ The initial probability of raining is 0.5
- ▶ *Question*: what is the most likely sequence of hidden states for (umbrella, umbrella, no umbrella)?

Many applications

- ▶ speech recognition
- ▶ handwriting recognition
- ▶ computational finance
- ▶ ...
- ▶ bioinformatics
 - ▶ gene prediction
 - ▶ protein classification
 - ▶ protein secondary structure and protein folding
 - ▶ DNA motif discovery (binding sites)
 - ▶

HMM in speech recognition



from [Gales&Young, Foundations and Trends in
Signal Processing, 2007]

Main problems for HMMs

- ▶ Given a sequence of observations x and an HMM M , compute the most likely sequence of hidden states
(Decoding problem)
- ▶ Given a sequence of observations x and an HMM M , compute the probability of x
- ▶ Given a sequence of observations x and an HMM M , find the probabilities of hidden states at time i . In other words, find $P(\pi_i = k | x)$ for any state k

- ▶ **Training the HMM:** adjust the parameters of HMM M to maximize $P(x|M)$

Computing the probability of x (exercise)

- ▶ How to compute the probability of observing $x = x_1 \dots x_n$?
- ▶ *Solution:* use Dynamic Programming. Similar to the Viterbi algorithm with max replaced by sum

Navigable Small-World Networks

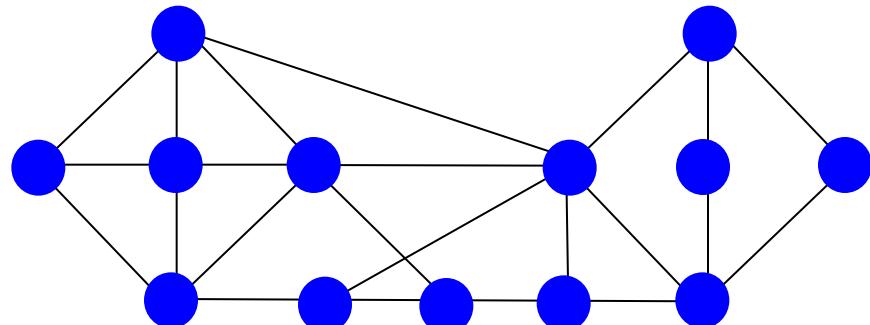
Based on slides by Šarūnas Girdzijauskas

Outline

- Small-Worlds vs. Random Graphs
- Navigation in Small-Worlds (Kleinberg's model)
- Small-World based Structured Overlays
- Non-uniform Structured Overlays

How is our society connected?

- Structure of our social networks
 - Most people's friends are located in their vicinity, be it colleagues, neighbours, or team mates in the local soccer club.
- Social networks were expected to be “grid-like”
 - Within a specific social dimension (e.g., profession, hobby, geographical distribution)



Implies the diameter of the social network is roughly $O(\sqrt{N})$.

Milgram's (Small-World) experiment

- Finding short chains of acquaintances linking pairs of people in USA who didn't know each other;
 - Source person in Nebraska and Kansas;
 - Target person in Massachusetts.
 - The letter could be only be given to persons one knows on a first name basis (*acquaintances*).



Milgram's (Small-World) experiment

- Average length of the chains that were completed lied between 5 and 6 steps;
- Coined as “Six degrees of separation” principle.
- This was far less than assumed under the 'grid-like' assumption !
 - Similar results have been found in many other social networks
- **BIG QUESTIONS:**
 - Why are there short chains of acquaintances linking together arbitrary pairs of strangers?
 - That is, why is the diameter of the graph low?



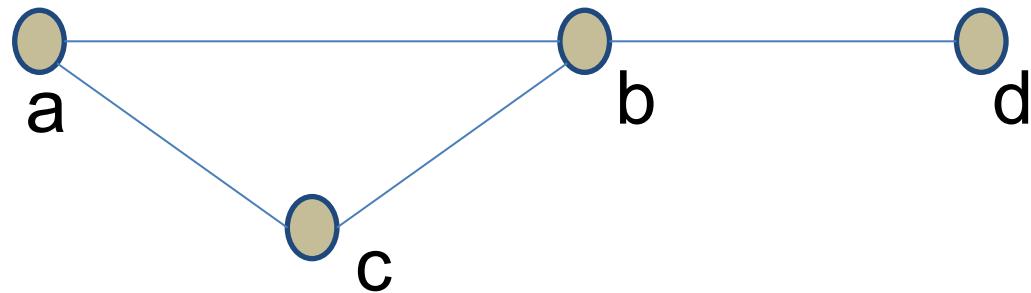
Random Graphs

- **Previously believed to be a Random graph**
 - A **random graph** is a graph that is generated by some random process.
 - When pairs or vertices are joined uniformly at random -> then any two vertices are connected by a short chain with high probability.
- **However..**
 - If A and B have a common friend C it is likely that they themselves will be friends! (clustering)
 - **Random networks tend not to be clustered**

Graphs

- A **graph** G formally consists of a set of vertices V and a set of edges E between them. That is, $G=(V,E)$.
- An **edge** e_{ab} connects vertex a with vertex b .
 - Edges can be directed or undirected.
- The neighbourhood N of a vertex a is defined as the set of its immediately connected vertices.
- The degree of a vertex is defined as the number of vertexes in its neighbourhood.
- Distance between two vertexes is the number of edges in the shortest path connecting the vertexes.
- The diameter of a graph is maximum distance for any vertex.

Example Unidirected Graph



$$V = (a, b, c, d)$$

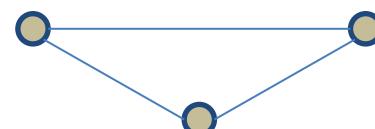
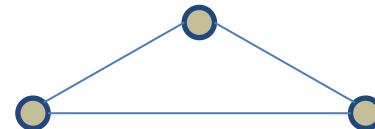
$$E = (e_{ab}, e_{ac}, e_{cb}, e_{bd})$$

$$N(a) = (b, c)$$

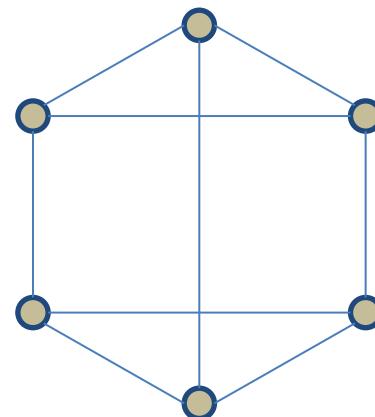
$$\deg(a) = 2$$

Regular Graph

- A regular graph is a Graph where each vertex has the same number of neighbors
 - In other words, nodes have the same degree
- Regular graphs can also be random
 - Random regular graph



2-regular graph



3-regular graph

Informally Clustering in Graphs

- High clustering => a given vertex's neighbours have lots of connections to each other
- Low clustering => a given vertex's neighbours have few connections to each other

Clustering coefficient

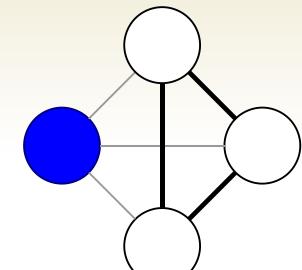
- The **local clustering coefficient** $C(v)$ of vertex v is a measure of how close v 's neighbours are to being a clique (a fully connected graph):

$$C(v) = \frac{e(v)}{\deg(v)(\deg(v) - 1)/2}$$

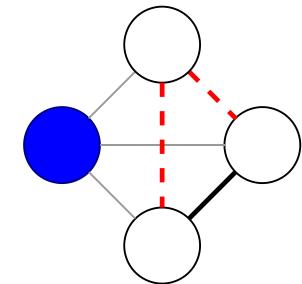
where $e(v)$ denotes the number of edges between the vertices in the v 's neighbourhood.

- Network average clustering coefficient** \tilde{C} is given by the fraction of:

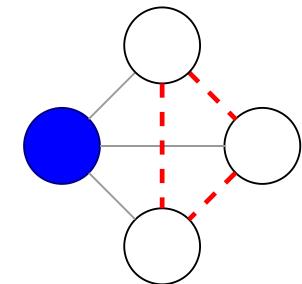
$$\tilde{C} = \frac{1}{N} \sum_{i=1}^N C(i)$$



$$c = 1$$



$$c = 1/3$$



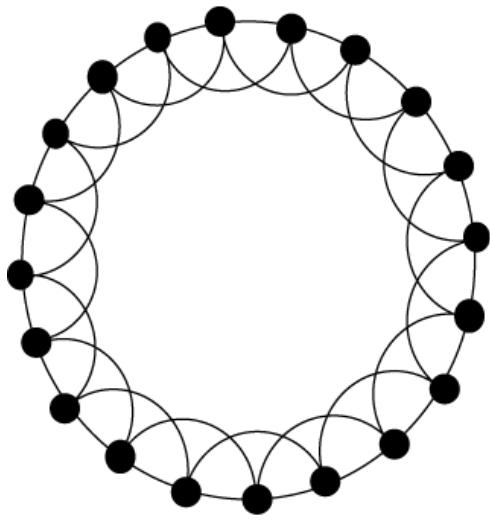
$$c = 0$$

Clustering

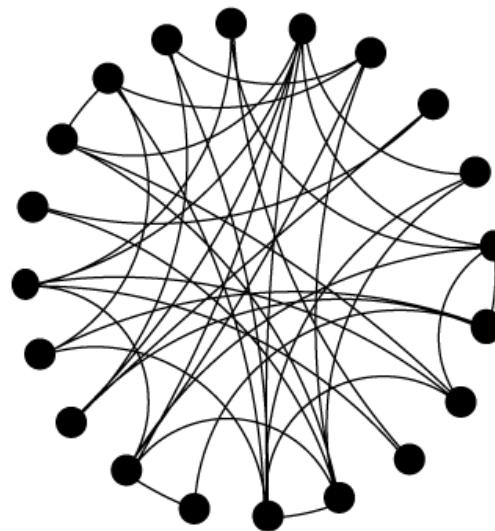
- Clustering measures the fraction of neighbours of a node that are connected among themselves
- Regular Graphs have a high clustering coefficient
 - but also a high diameter
- Random Graphs have a low clustering coefficient
 - but a low diameter
- Both models do match some properties expected from real networks - such as Milgram's!

Random vs. regular graphs

Regular



Random

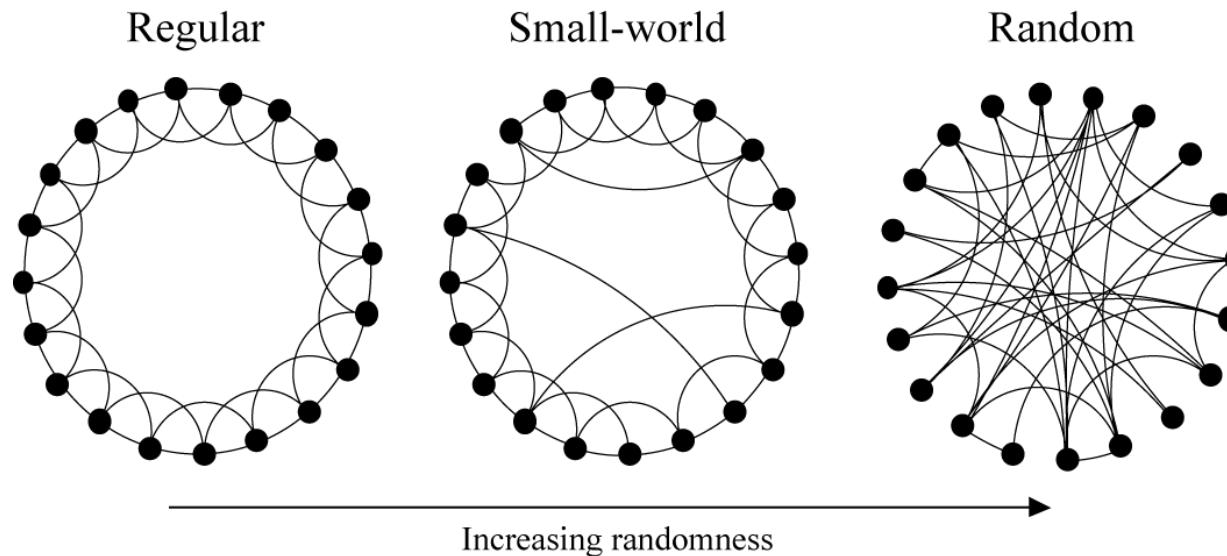


- Long paths
 - $L \sim N/(2k)$
- Highly clustered
 - $C \sim 3/4$

- Short path length
 - $L \sim \log_k N$
- Almost no clustering
 - $C \sim k/N$

Small-World networks

- Random rewiring procedure of regular graph (by Watts and Strogatz)
- With probability p rewire each link in a regular graph:
 - Exhibit properties of both: random and regular graphs:
 - High clustering coefficient;
 - Low diameter.

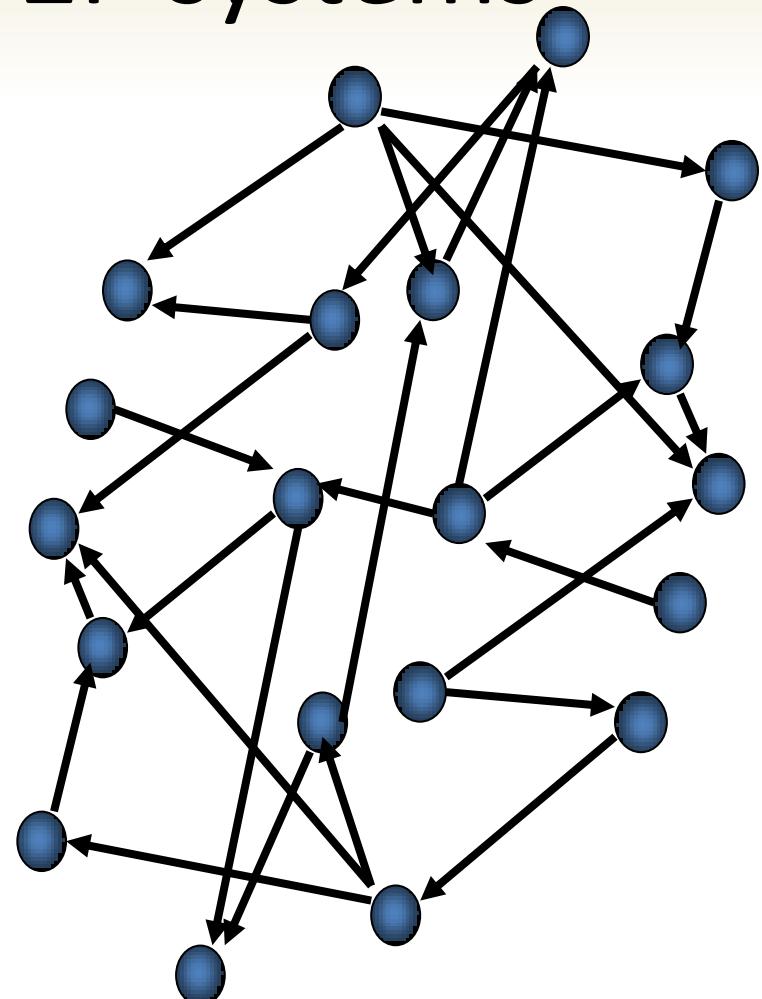


Small-World: remaining questions

- This is still not enough to explain Milgram's experiment:
 - If there exists a shortest path between any two nodes - where is the global knowledge that we can use to find this shortest path?
 - Why should arbitrary pairs of strangers be able to find short chains of acquaintances that link them together?
 - Why do decentralized “search algorithms” work?

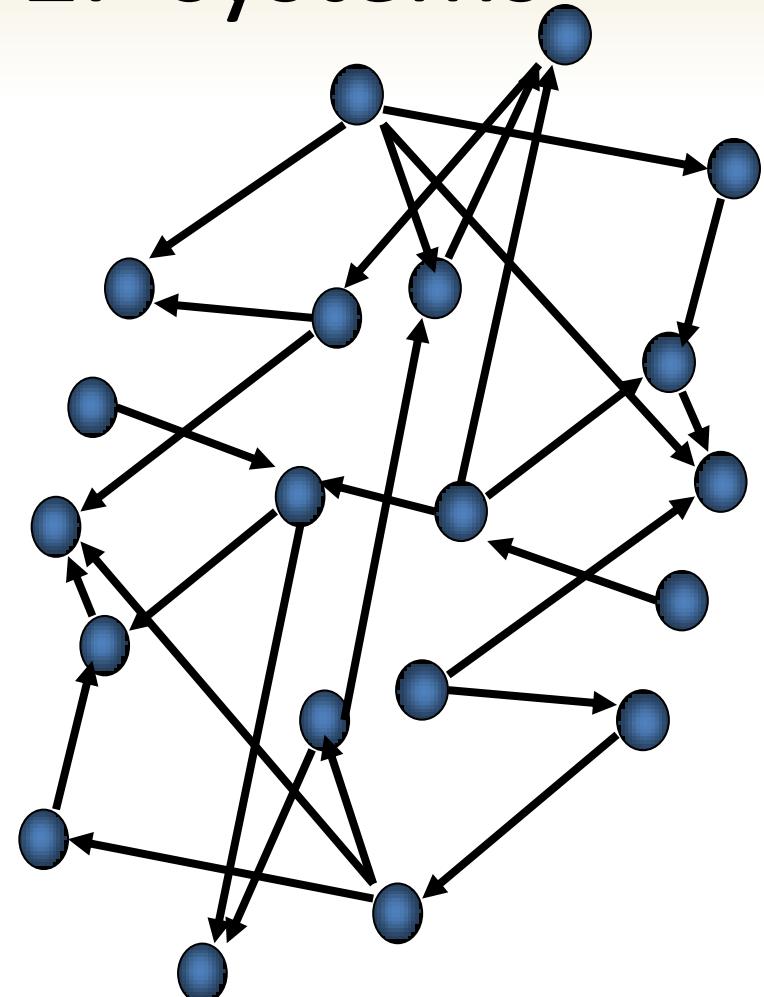
Implications for P2P systems

- Each P2P system can be interpreted as a directed graph where peers correspond to the nodes and their routing table entries as directed edges (links) to the other nodes



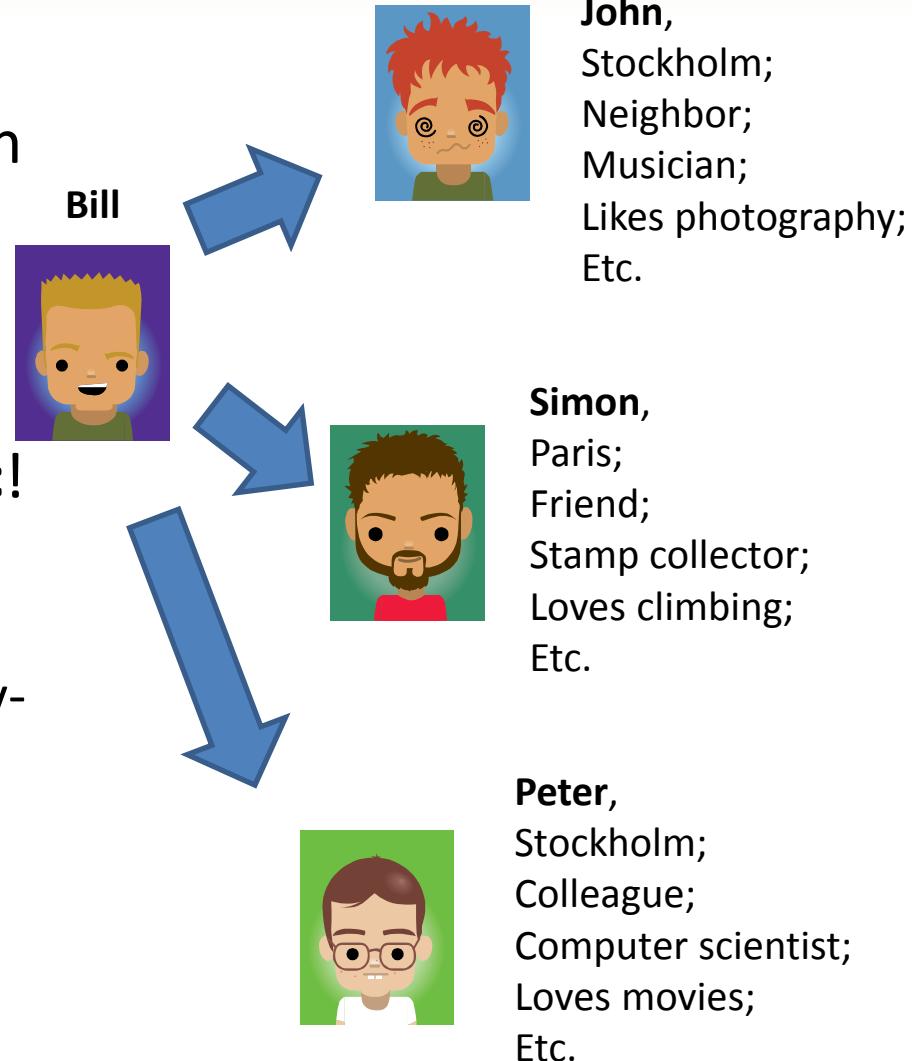
Implications for P2P systems

- Task for P2P:
 - Invent a decentralized search algorithm that would route message from any node A to any other node B with relatively few hops compared with the size of the graph
- Is it possible?
 - Milgram's experiment suggests YES!



Why did Milgram's experiment work?

- A social network is not a simple graph, but a graph with certain “labels”
 - “labels” representing various dimensions of our life
- We internalize a “**labeling space**” with a **distance metric**!
- We can greedily minimize the distance!
 - Decentralized search: a greedy-routing algorithm
 - We need to build the right graph where a decentralized search algorithm might perform the best



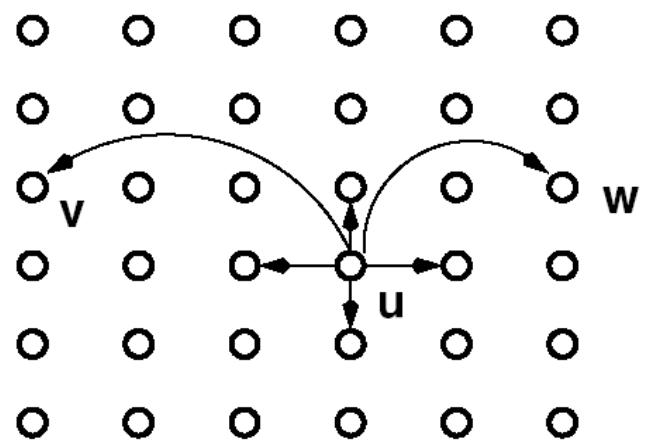
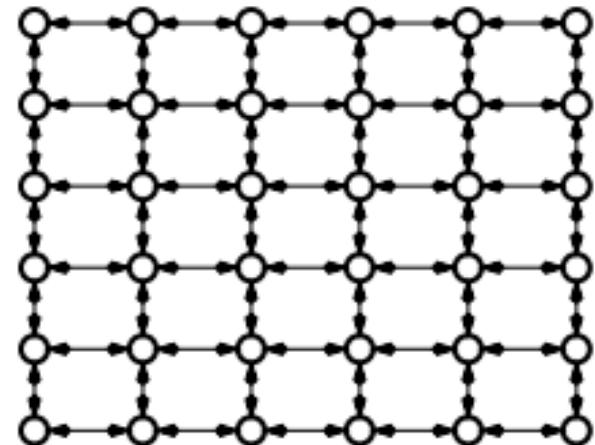
Kleinberg's model of Small-Worlds

- **Research of Jon Kleinberg:**
 - Claim that there is no decentralized algorithm capable performing effective search in the class of SW networks according to Watts and Strogatz model;
 - J. Kleinberg presented the infinite family of Small World networks that generalizes Watts and Strogatz model and shows that decentralized search algorithms can find short paths with high probability;
 - It was proven that there exists only one unique model within that family for which decentralized search algorithms are effective.

Navigable Small-World networks

- Kleinberg's Small-World's model
 - 2-dimensional lattice
 - Lattice (Manhattan) **distance**
 - Two type of edges:
 - Lattice edges (short range)
 - Long range
 - Probability for a node u to have a node v as a long range contact is proportional to

$$P(u \rightarrow v) \sim \frac{1}{d(u, v)^r}$$



Influence of “r”

- Each peer u has an edge to the peer v with probability $\frac{1}{d(u,v)^r}$ where $d(u,v)$ is the manhattan distance between u and v .
- **Tuning “r”**
 - When $r < \text{dim}$ (dimension of the euclidean space) we tend to choose more far away neighbours (search algorithm quickly approaches the target area, but slows down till it finally reaches the target).
 - When $r > \text{dim}$ we tend to choose closer neighbours (search algorithm reaches the target area slowly if it is far away, but finds the target quickly in its neighbourhood).
 - When $r=0$ – long range contacts are chosen uniformly. Random graph theory proves there exists short paths between every pair of vertices, **but there is no search algorithm capable of finding these paths.**
 - When $r=\text{dim}$, the algorithm exhibits optimal performance.

Performance with $r=\dim$

- When $q = 1$ (there is one long range link)
 - The expected search cost is bounded **by $O(\log^2 N)$**
- When $q = k$ (there are a constant number of long range links)
 - The expected search cost is bounded by
 $O(\log^2 N)/k$
- When $q = \log N$
 - The expected search cost is bounded **by $O(\log N)$**

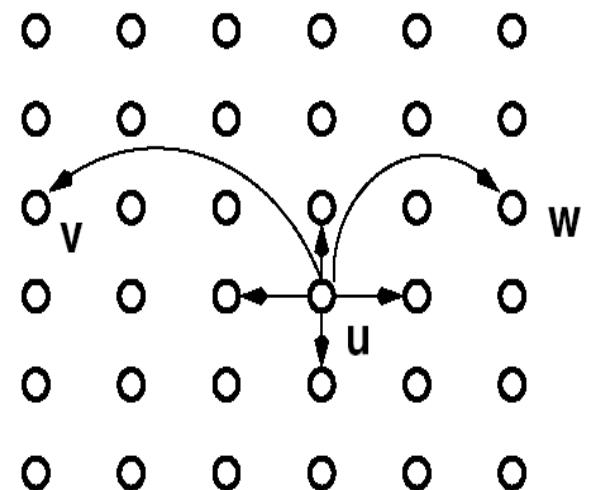
How does it work in practice?

$$P(u \rightarrow v) \sim \frac{1}{d(u, v)^r}$$

- Normalization constant has to be calculated:

$$\sum_{\forall i \neq u} \frac{1}{d(u, i)^r}, i \in N$$

$$P(u \rightarrow v) = \frac{1}{d(u, v)^r} \cdot \frac{1}{\sum_{\forall i \neq u} \frac{1}{d(u, i)^r}}, i \in N$$



Example

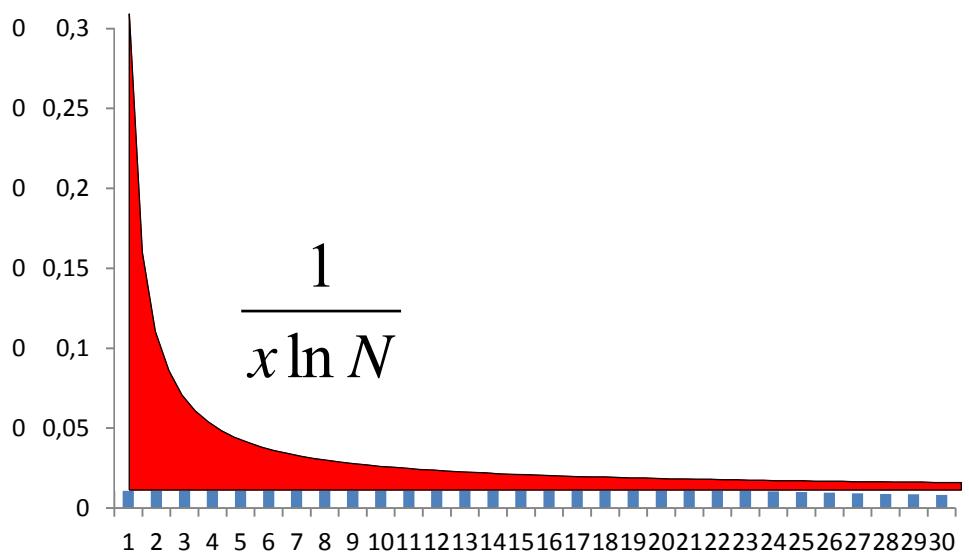
- Choose among 3 friends (1-dimension)
 - A (1 mile away)
 - B (2 miles away)
 - C (3 miles away)
- Normalization constant

$$P(\text{chooseA}) = \frac{\frac{1}{11}}{\frac{6}{11}} = \frac{6}{11}$$

$$P(\text{chooseB}) = \frac{\frac{1}{11}}{\frac{6}{11}} = \frac{3}{11}$$

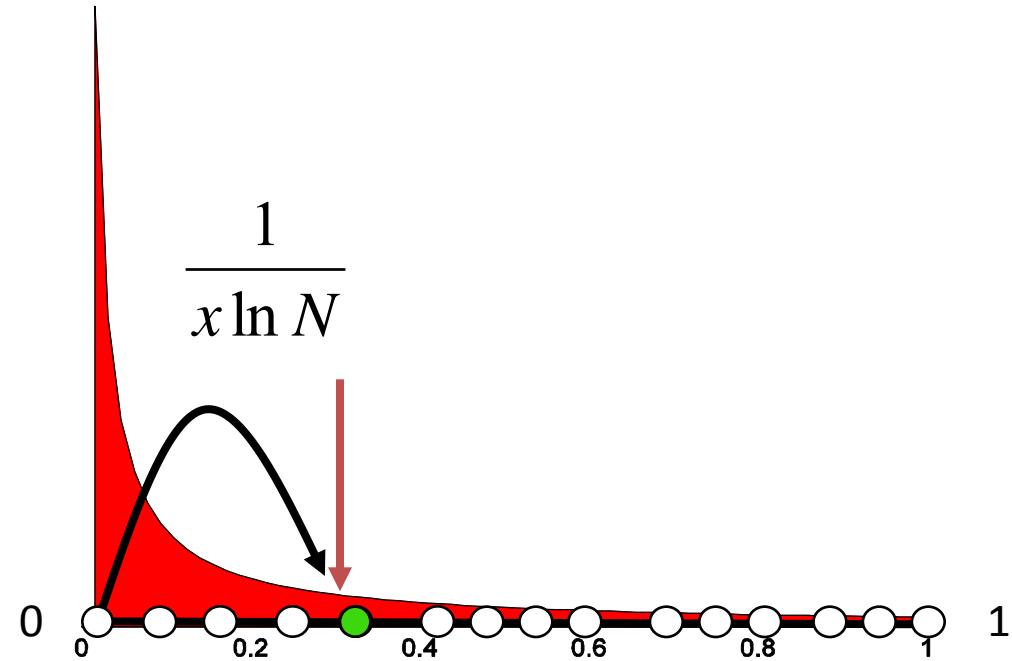
$$P(\text{chooseC}) = \frac{\frac{1}{11}}{\frac{6}{11}} = \frac{2}{11}$$

$$\sum_{\forall i \neq u} \frac{1}{d(u, i)} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} = \frac{11}{6}$$

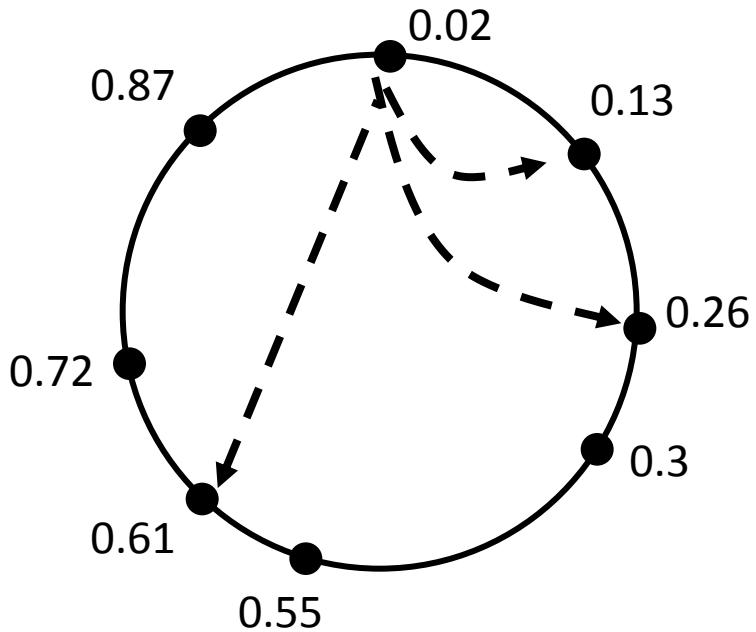


1-dimensional continuous case

- Peers uniformly distributed on a unit interval (or a ring structure)
- Long range links chosen with the probability $P \sim 1/d$
- Search cost
 $O(\log^2 N/k)$ with k long-range links
 $O(\log N)$ with $O(\log N)$ long-range links
- Systems:
 - Symphony (Manku et al, USITS 2003)
 - Accordion (Li et al, NSDI 2005)



Small-World based P2P Overlay



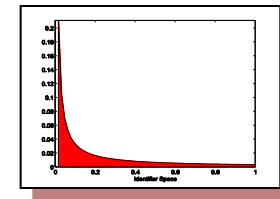
Systems:

Symphony (Manku et al, USITS 2003)

Accordion (Li et al, NSDI 2005)

- Peers mapped onto positions on the ring
 - Uniform hash function (e.g., SHA-1) for peerId
 - Establish successor and predecessor ring links
- Small-World **connectivity establishment**

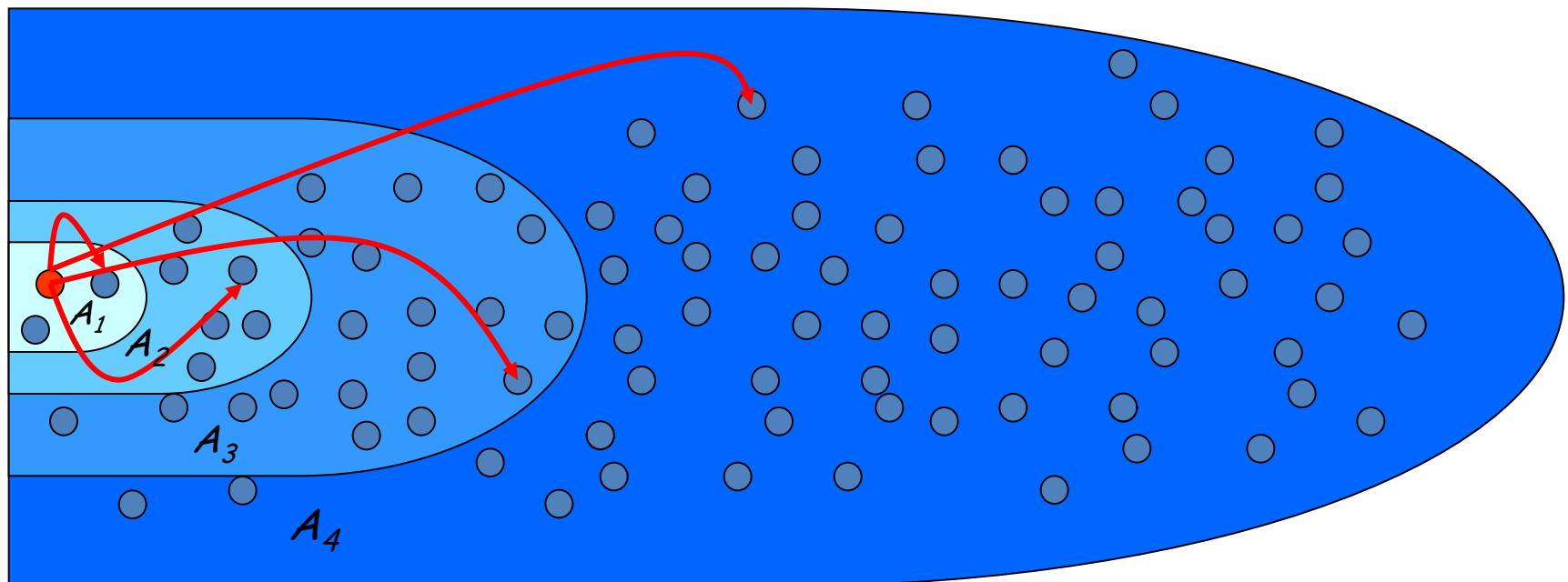
$$\frac{1}{x \ln N}$$



- No restrictions on peer-degree
- Implicit load balancing

Approximation of Kleinberg's model

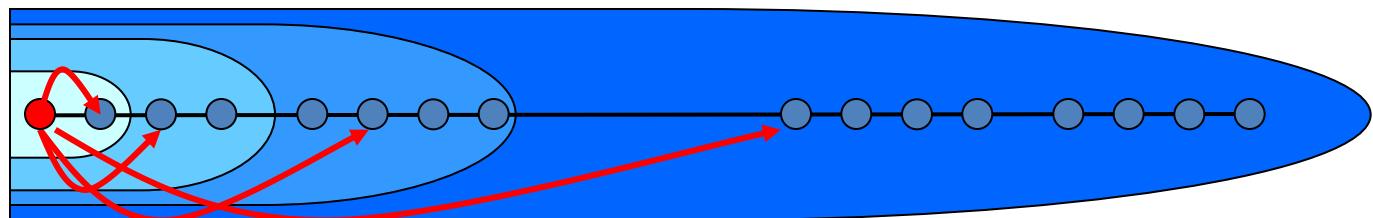
- Given node u if we can partition the remaining peers into sets $A_1, A_2, A_3, \dots, A_{\log N}$, where A_i consists of all nodes whose distance from u is between 2^{-i} and 2^{-i+1} .
 - Then given $r=\dim$ each long range contact of u is nearly equally likely to belong to any of the sets A_i*
 - When $q=\log N$ – on average each node will have a link in each set of A_i*



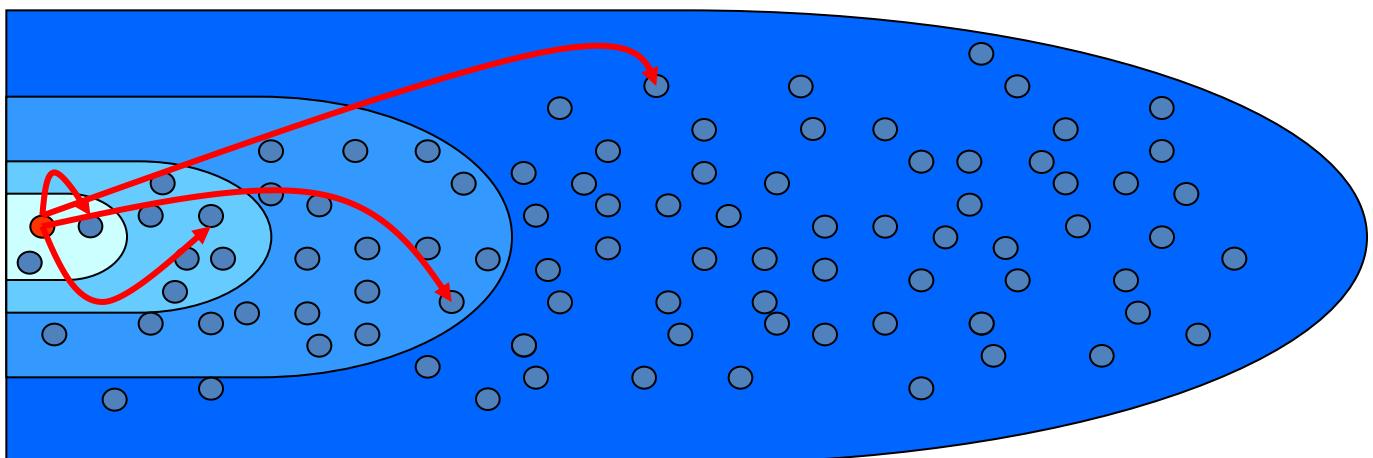
Traditional DHTs and Kleinberg model

- Most of the structured P2P systems are similar to Kleinberg's model and are called logarithmic-like approaches. E.g.
 - Chord (randomized version) $q=\log N$, $r=1$
 - Gnutella $q=5$, $r=0$

- Randomized Chord's model



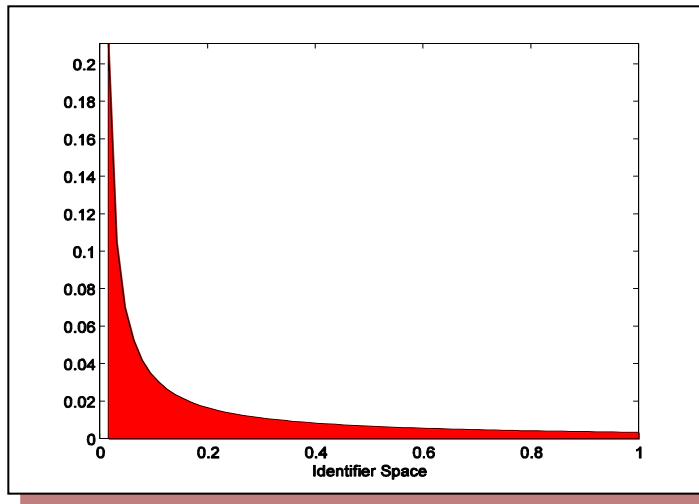
- Kleinberg's model



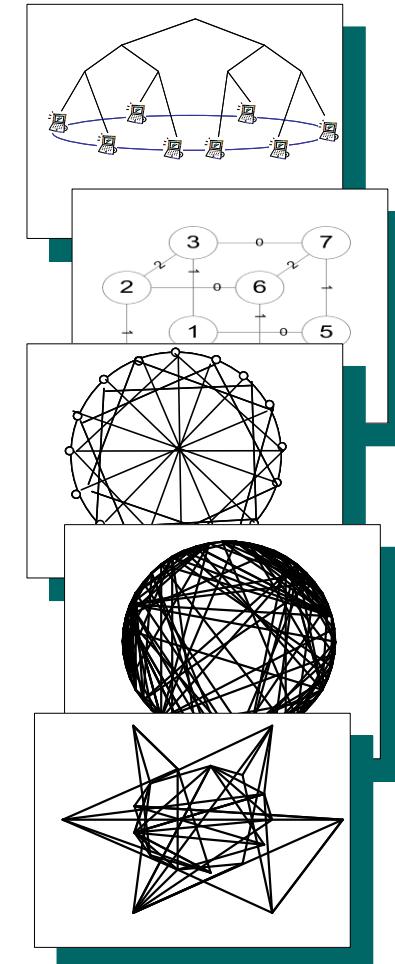
Similarity to other P2P construction techniques

- Many existing P2P are just the “special cases” of Kleinberg’s navigable Small-World

- Ring based
 - Tree based
 - Hypercube
 - Torus
 - Etc.,



- Kleinberg’s Small-World
 - Randomized construction
 - No restrictions on peer-degree
 - “choice-of-two” possibility
 - Effective nonGreedy routing



What to take away from the small-world tour?

- How can we characterize P2P overlay networks such that we can study them using graph-theoretic approaches?
- What is the main difference between a random graph and a SW graph?
- What is the main difference between Watts/Strogatz and Kleinberg models?
- What is the relationship between structured overlay networks and small world graphs?
- What are possible variations of the small world graph model?
- How does it relate to our social networks?

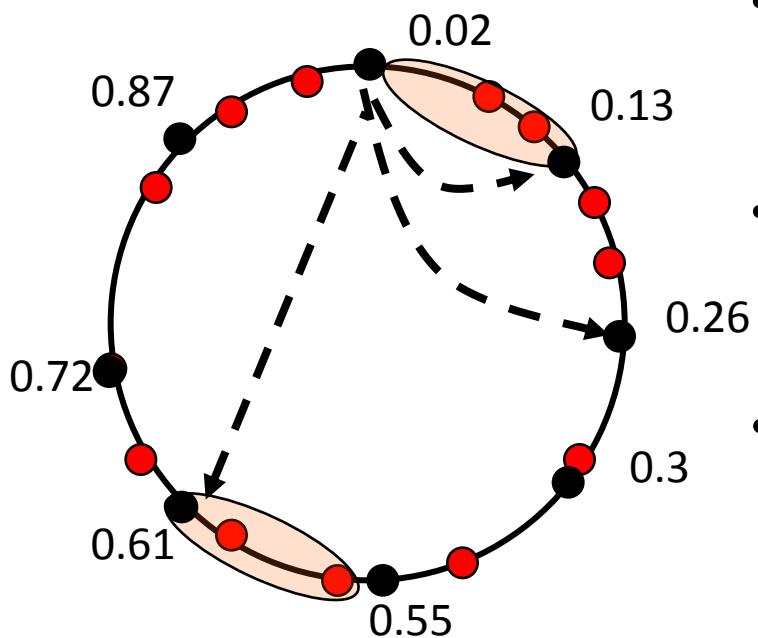


Non-Uniform Structured Overlays

Structured overlay

- Build a routing table
 - Each peer has a well-defined neighbourhood and information about its immediate neighbours (in contrast to unstructured topologies)
- The search operation is performed efficiently (in contrast to unstructured)

Data on a Ring-Structured Overlay



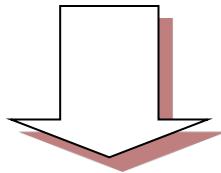
- Peers mapped onto an ID the **ring identifier space**
 - Uniform hash function (e.g., SHA-1)
- Resources mapped on to an ID the **ring identifier space**
 - Uniform hash function
- Peers are **responsible** for a **range** of the **ring identifier space**
- **Connectivity establishment**
 - E.g., Symphony [Manku et al. 2003]
- **Uniform** peer key (id) distribution
 - Implicit load balancing

Problems with range queries

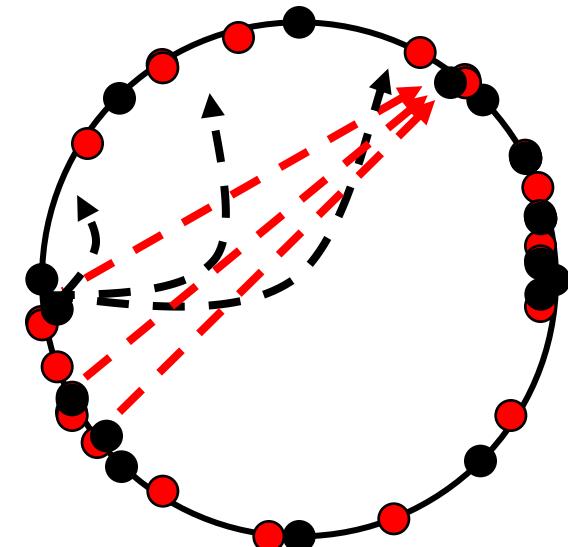
- Point queries
 - E.g. “ABBA Waterloo”
 - “ABBA Mamma Mia”
- Range queries:
 - What about all the files with prefix “ABBA..”?
 - Uniform hashing assigns all the files to random locations on the ring
 - Uniform distribution of IDs (keys)
 - **Inefficient lookup!**
- Order preserving (Lexicographic) hashing
 - If $a > b$ then $\text{id}(a) > \text{id}(b)$ (in uniform hashing $\text{id}(a) = \text{rand}$; $\text{id}(b) = \text{rand}$)
 - Non-uniform distribution (depends on the data)!

Problems with uniform key distributions

- Order preserving hash functions (e.g. Lexicographical ordering)
- Peer clustering in key space
- Etc.

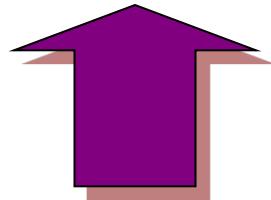


- For skewed key distributions
 - How do we make SW long range links?

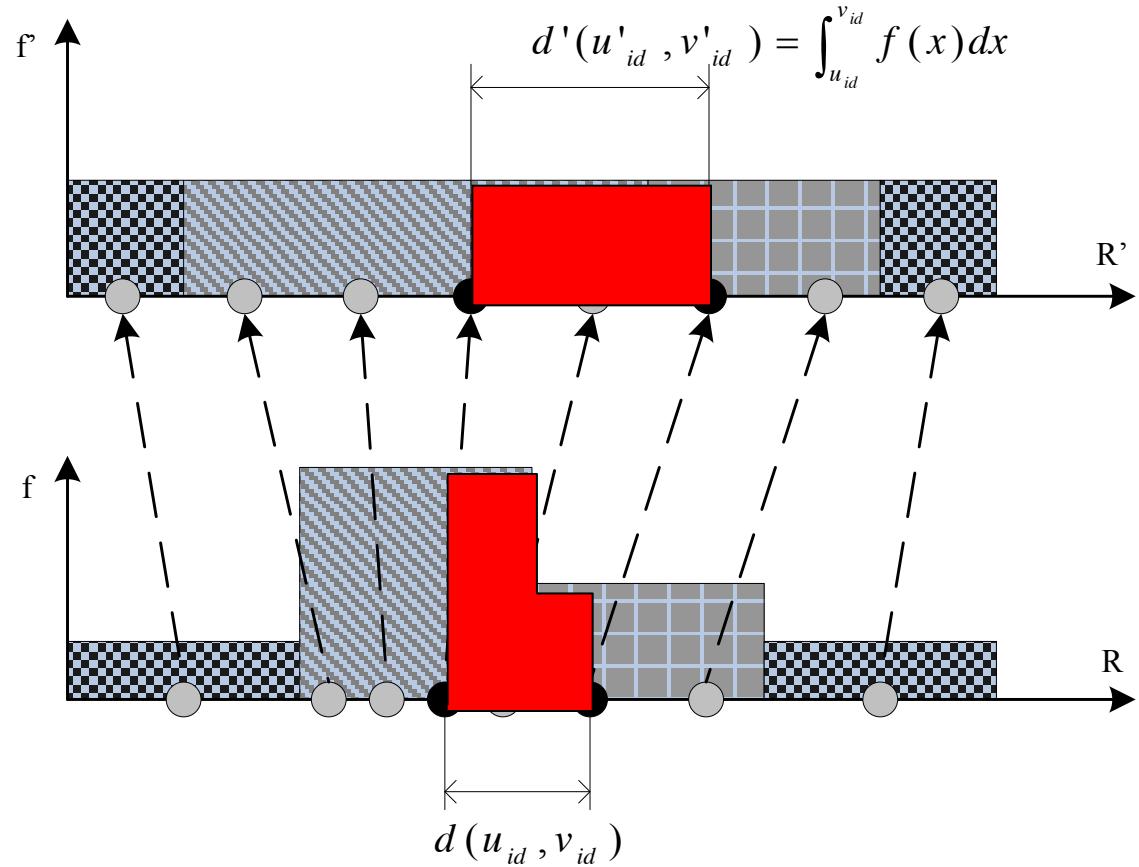


Extending Kleinberg's model

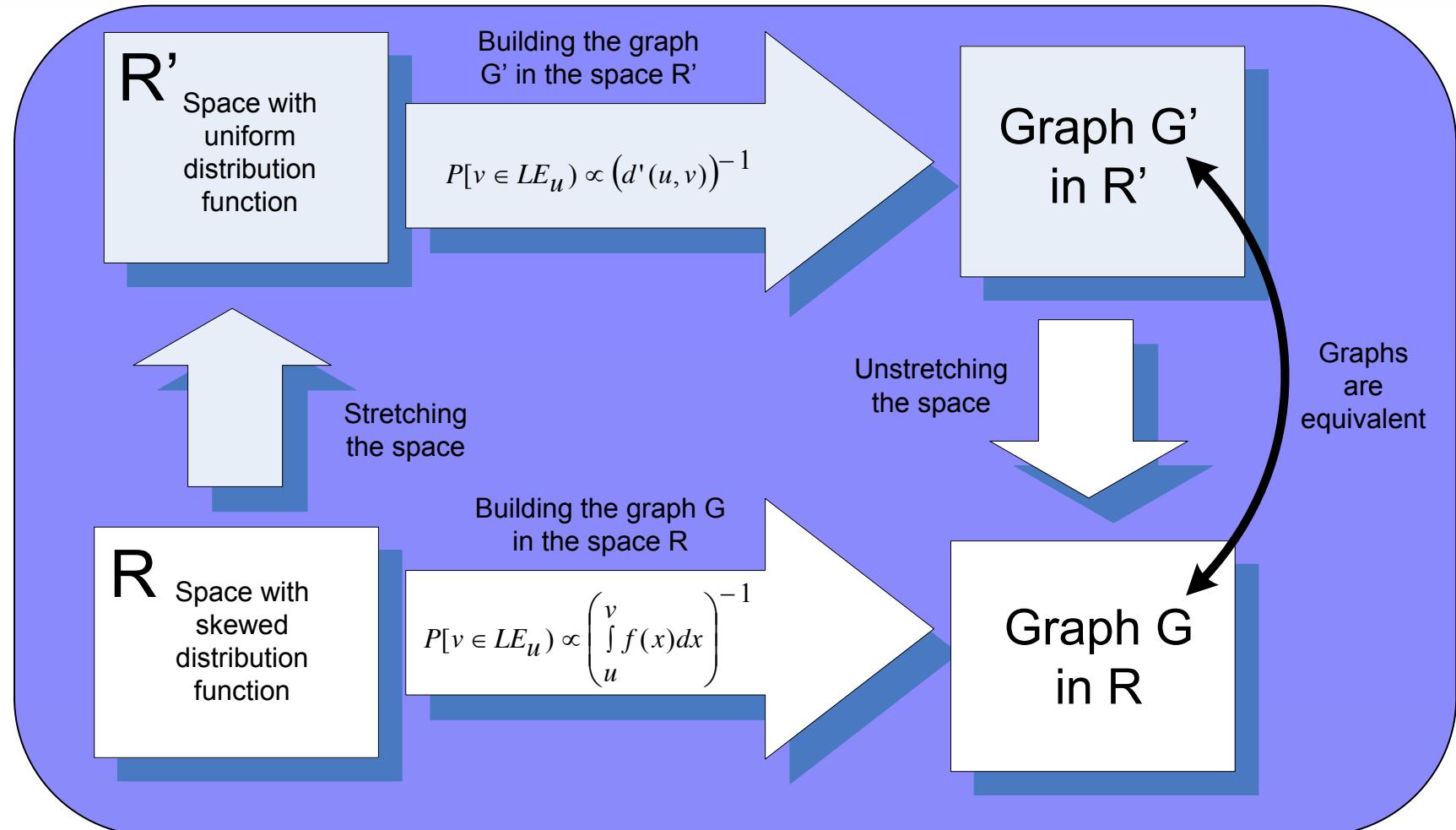
Stretched space



Original space



Extension of the “uniform model” to nonuniform case (1)



Small-World P2P in non-uniform spaces

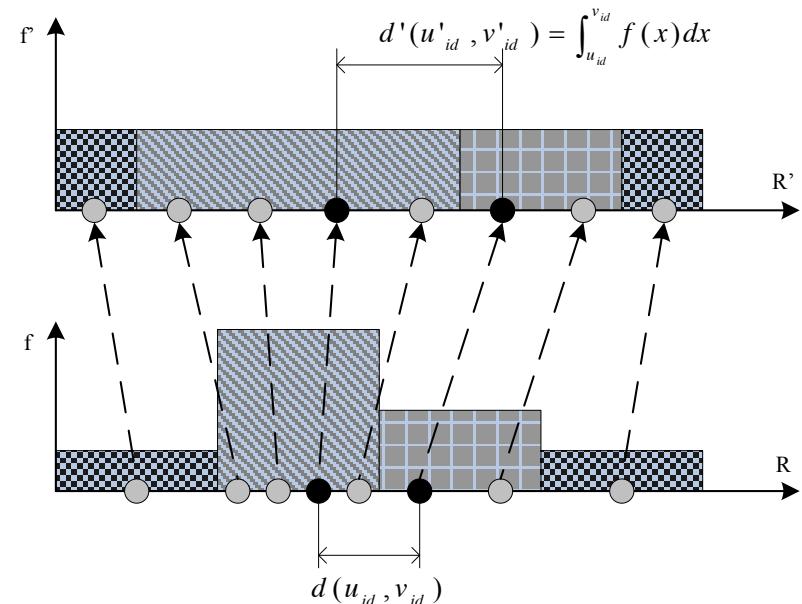
- $f(x)$ – probability density function of the peer keys.
- Long range neighbours are chosen inversely proportional to the integral of $f(x)$ between the two nodes

$$P[v \leftrightarrow u] \sim \frac{1}{\int_{u_{id}}^{v_{id}} f(x) dx}$$

Expected routing cost in such a network using greedy routing is $O(\log N)$ when the network degree is $O(\log N)$.

Acquiring Peer Key Distribution

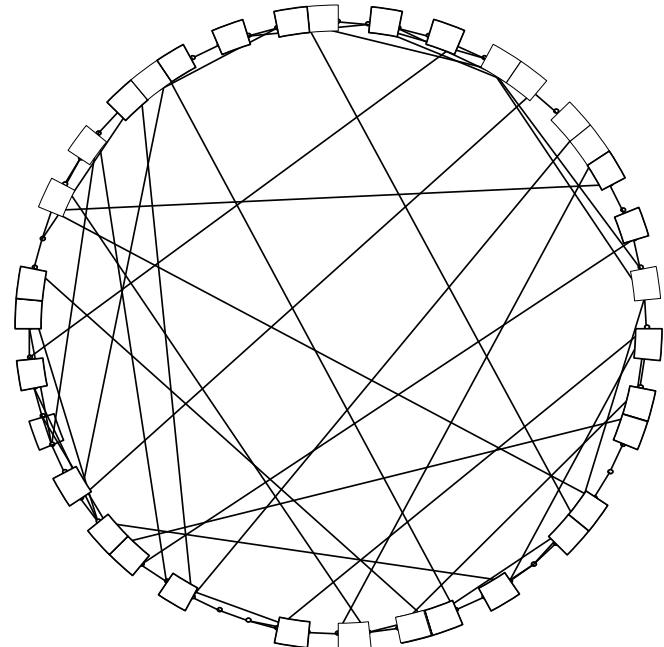
- Problem:
 - Need to acquire the **global** key distribution function **locally**.



- Uniform sampling
(Mercury [Bharambe et al., 2004])
 - $\log^2 N$ samples
 - Cannot cope with complex distributions! ☹
- Non-uniform (Scalable Sampling)
 - OSCAR (Overlays using SCALable sampling of Realistic distributions)
 - [DBISP2P06, ICDE07, TAAS10]

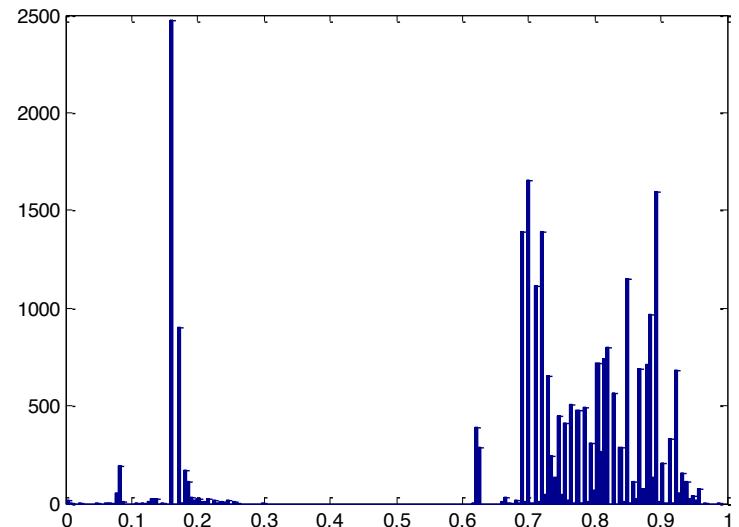
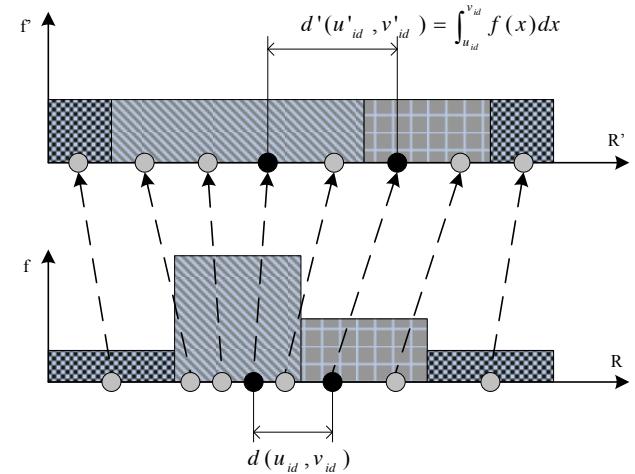
Sampling by random walks

- Bharambe et al. 2004
“Mercury: supporting scalable multi-attribute range queries”
- Sampling by random walks
 - A random walk with TTL at least $\log N$ ends up in a uniform random node (on expander graphs)



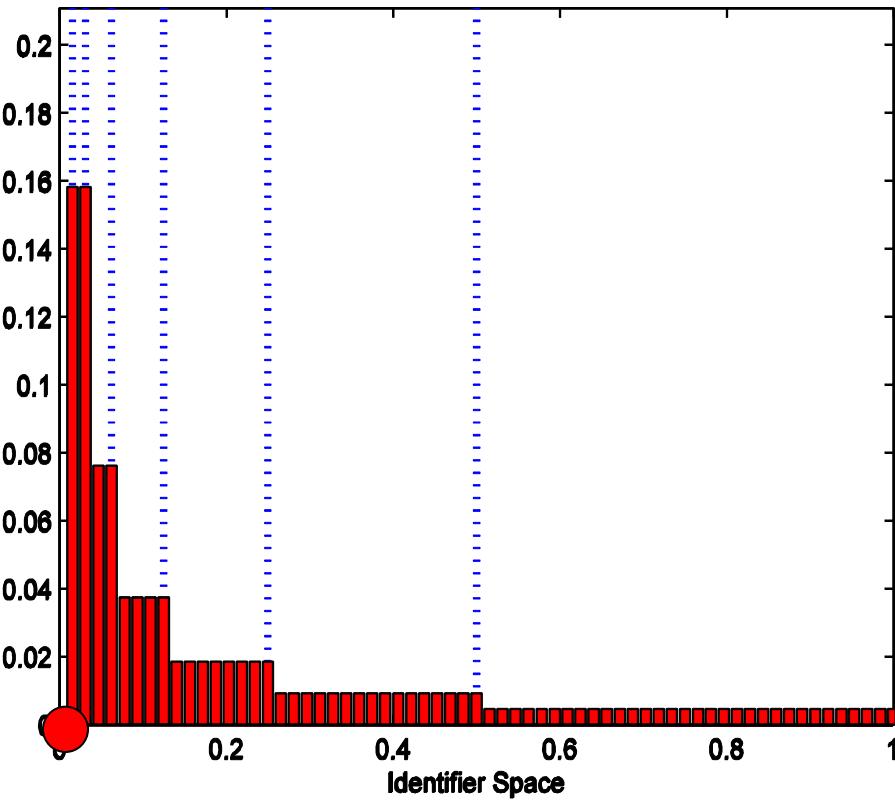
Sampling in Mercury

- Every node periodically issues k_1 samples
 - The sampled nodes return its ID and their own collected samples (k_2)
 - It is suggested $k_1 = \log N$
- Over time the ID distribution histogram is built.
- Real world distributions are far more complex!

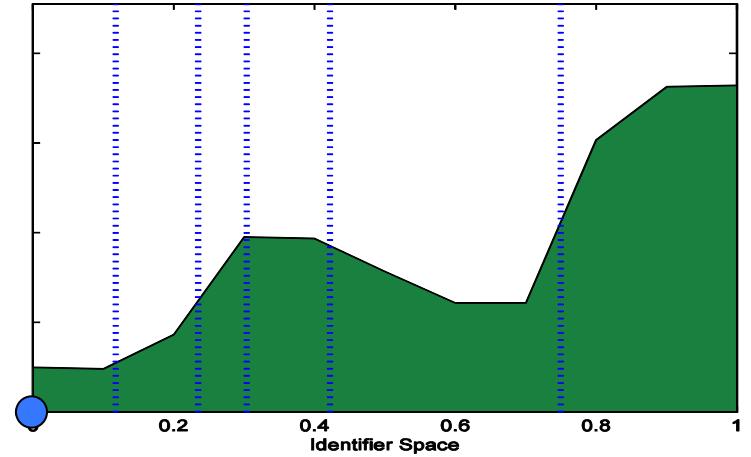
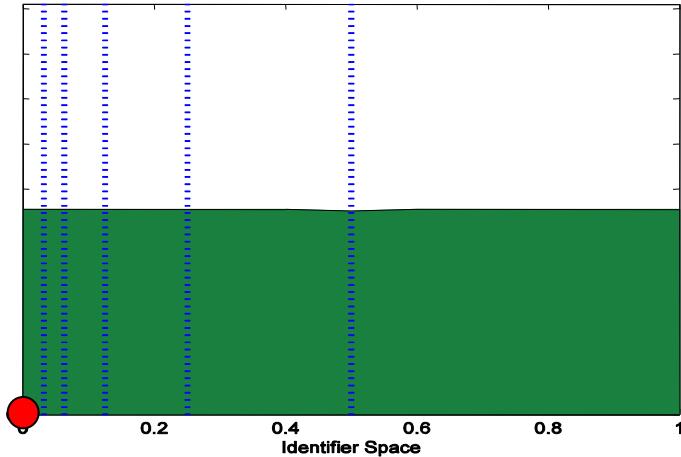


Oscar: Modifying Kleinberg's method

- Choosing long-range link:
 - 1) u.a.r. choose a partition
 - 2) u.a.r. choose a peer within that partition
- It can be proven that **search cost** remains $O(\log^2 N)$
 $O(\log N)$ with $O(\log N)$ degree

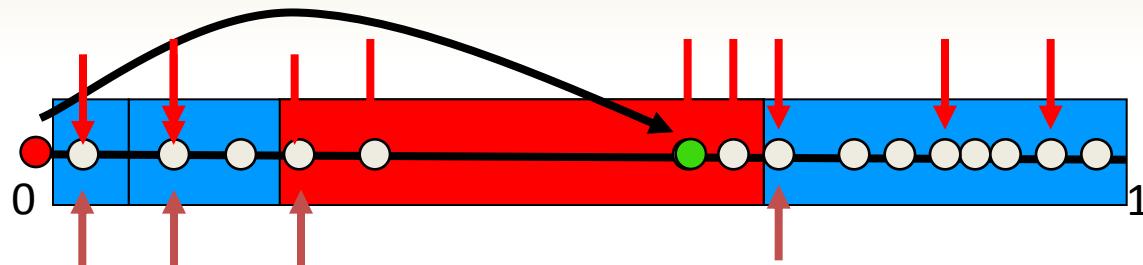


Oscar: Dealing with Skewed Spaces



- Find the boundaries between partitions!
- Uniform sampling by random walks.
- k samples for each boundary.

Oscar: an Example



- $O(k \cdot \log N)$ samples is needed to construct a routing efficient network.
- Does not depend on the complexity of the distribution
- “The view” can be copied from a ring-neighbour
(by contacting median peers and requesting their ring-neighbour ids)

Recap of non-uniform structured overlays

- What is the relationship between the resource placement and the network structure in structured P2P overlays?
- What is a challenge for enabling range queries for structured overlays?
- What is the main difference between a regular Kleinberg's model Small-World model for non-uniform id spaces?
- How does random sampling work?
- What are the possible solutions to solve the non-uniform resource placement problem?

Acknowledgements:

Some slides were derived from the lecture notes of K. Aberer (EPFL, Switzerland) and A. Datta (NTU, Singapore)

Text search

Stanislav Protasov

Agenda

- Text indexing
 - Inverted index
 - Text embedding
- ANNS
 - Clustering
 - Proximity graphs

Text indexing: inverted index

Inverted index: terms

- **case folding:** London = london; Лев = лев
- **Stemming:**
 - compress = compression = compressed
 - лев = льва = львом
- ignore **stop words:** to, the, it, be, or, ...
- Problems arise when search on “To be or not to be” or “the month of May”
- **Thesaurus:** fast = rapid; лев = лёвшка
 - handbuilt clustering

Inverted index: tokenization

Lexemes (distinct objects of the language) are produced by **scanner**.

token = (lexeme, token_type)

Program, converting stream of characters into a stream of **tokens** is called **lexical analyzer, lexer, tokenizer**.

E.g. for Java:

T1 = (“Lion123”, “identifier”)

T2 = (“+”, “binary operator”)

T3 = (“++”, “unary operator”)

Search and recommender systems idea

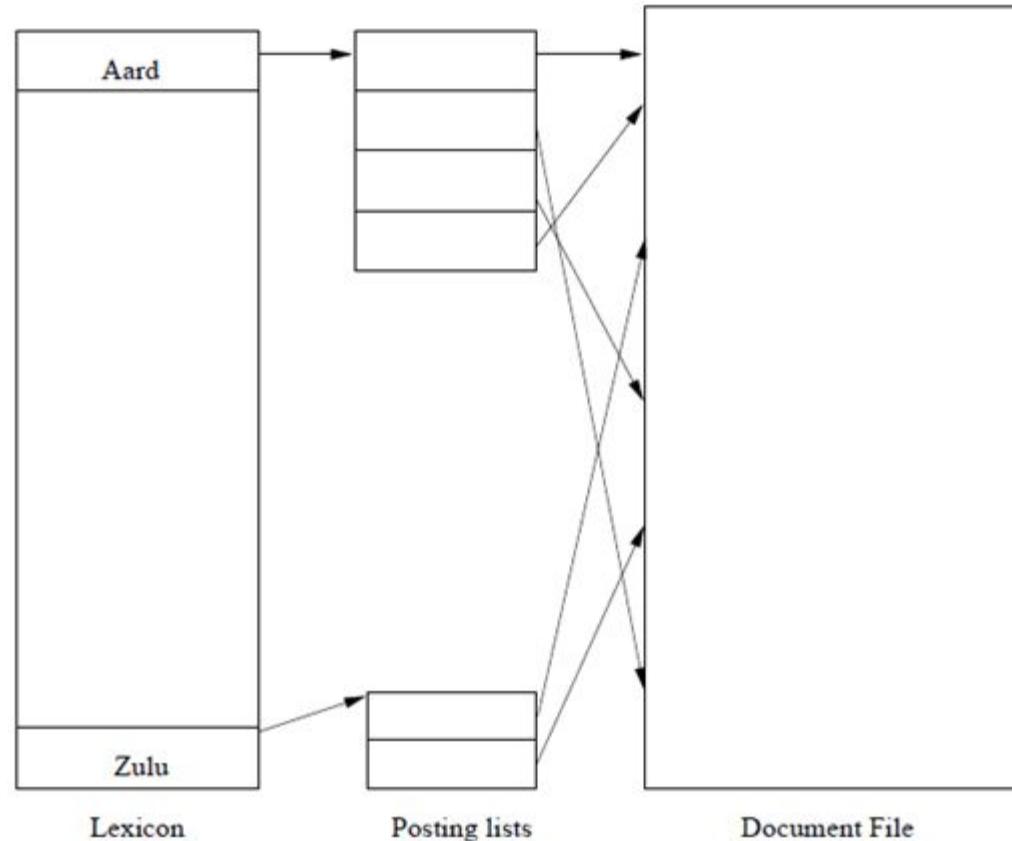
Exact match, exact NN search or exact range queries can be too **expensive**.

Pre-select (pre-ranking sets, approximate NN ...) which is done fastly (e.g. $O(\log(N))$) selects enough to catch [almost] all relevant elements.

Select (ranking, exact match) is done on a smaller set.

Inverted index: one, two, three

- 1) Build a **lexicon** for the whole database
- 2) For each word of lexicon build a **posting list** (set of pointers)
- 3) Search **queries**:
 - a) Get posting lists for each word
 - b) Intersect these lists ([skip-lists can help to intersect in \$O\(m+n\)\$](#))
 - c) Re-check selected documents hold expected substring



Text indexing: embedding

Lexicon as a vector space

- Consider common lexicon of a database as a **vector** space (each *word* is a *dimension*):
 - Binary vector:** 0 – if document has no word, 1 – if has at least one.
 - Compute **Euclidean** distance
 - Size problem
 - Weighted vector (TF-IDF):**
 - How do we run the query?
 - **Dot product** is faster
 - But wait, **20K+ dimensions...**

$$w_{d,t} = \log_2(N/f_t) \log_2(1 + f_{d,t})$$

Weight # of docs # of docs
 with this word # of occurrences

$$\text{similarity} = \cos(\text{doc}, \text{query}) = \frac{\overrightarrow{\text{doc}} \cdot \overrightarrow{\text{query}}}{\|\overrightarrow{\text{doc}}\| * \|\overrightarrow{\text{query}}\|}$$

Lexicon as a vector space: reduce dimensions!

- Compression approach #1:

```
max_size = N  
doc_compressed[i % max_size] += doc[i]
```

- Compression approach #2:

- [Random projection](#)
- Or even randomly remove some dimensions!

- Compression approach #3 - [latent semantic analysis](#):

- [LDA](#), [PCA](#), GDA, ...
- Embedding using encoder networks (BERT, doc2vec, DSSM, ...)

NN search

Approximation for k-NN search

Pre-select $k*c$ elements from approximate neighbourhood. Then select relevant

- Locality sensitive hashing
- Search trees and supporting data structures ([cover tree](#))
- Vector compression, clustering
- [Proximity graphs](#)

Approach #1. Hierarchical clustering

Why do we cluster?

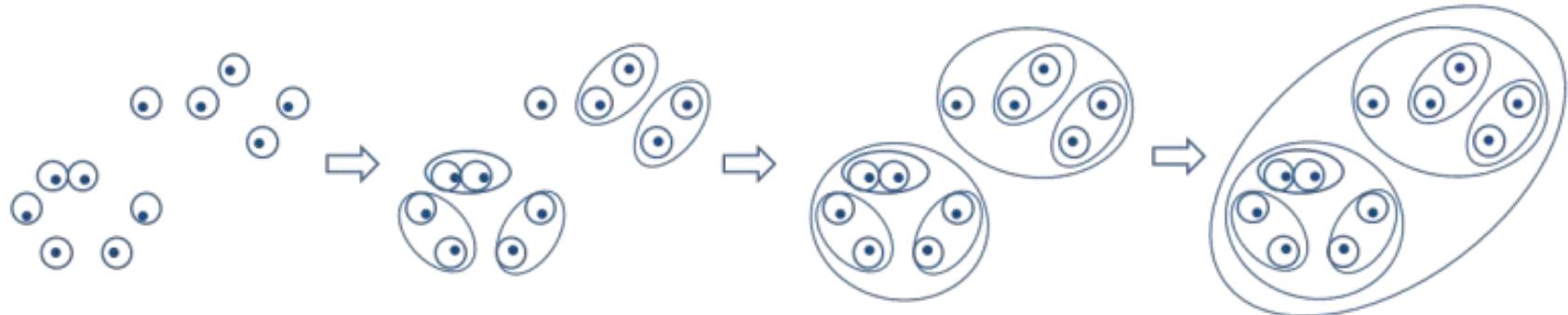
For a flat list we run $O(N)$ comparisons to find kNN

For \sqrt{N} similar* clusters we can pick one closest**
for $O(\sqrt{N})$ and find k NN*** in $O(\sqrt{N})$.

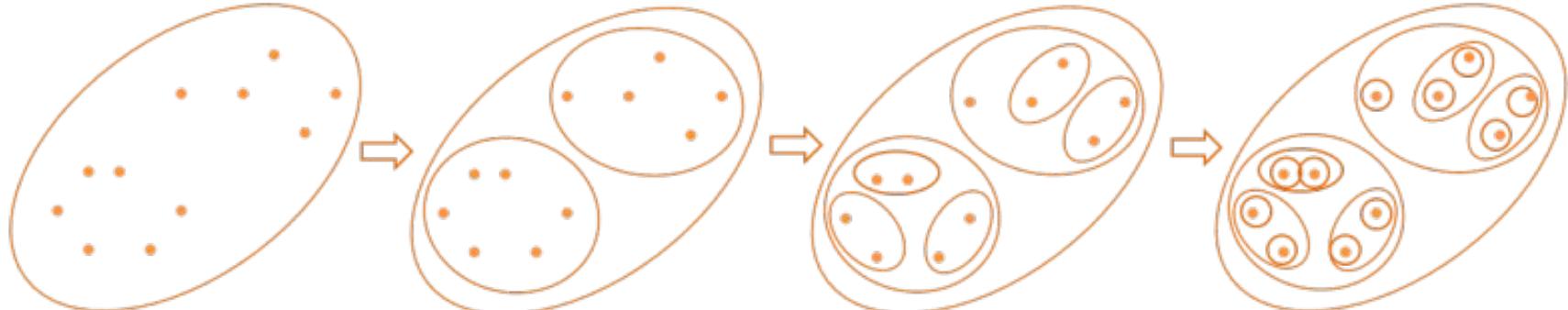
For two layers of $^3\sqrt{N}$...

How to we cluster?

Agglomerative Hierarchical Clustering



Divisive Hierarchical Clustering



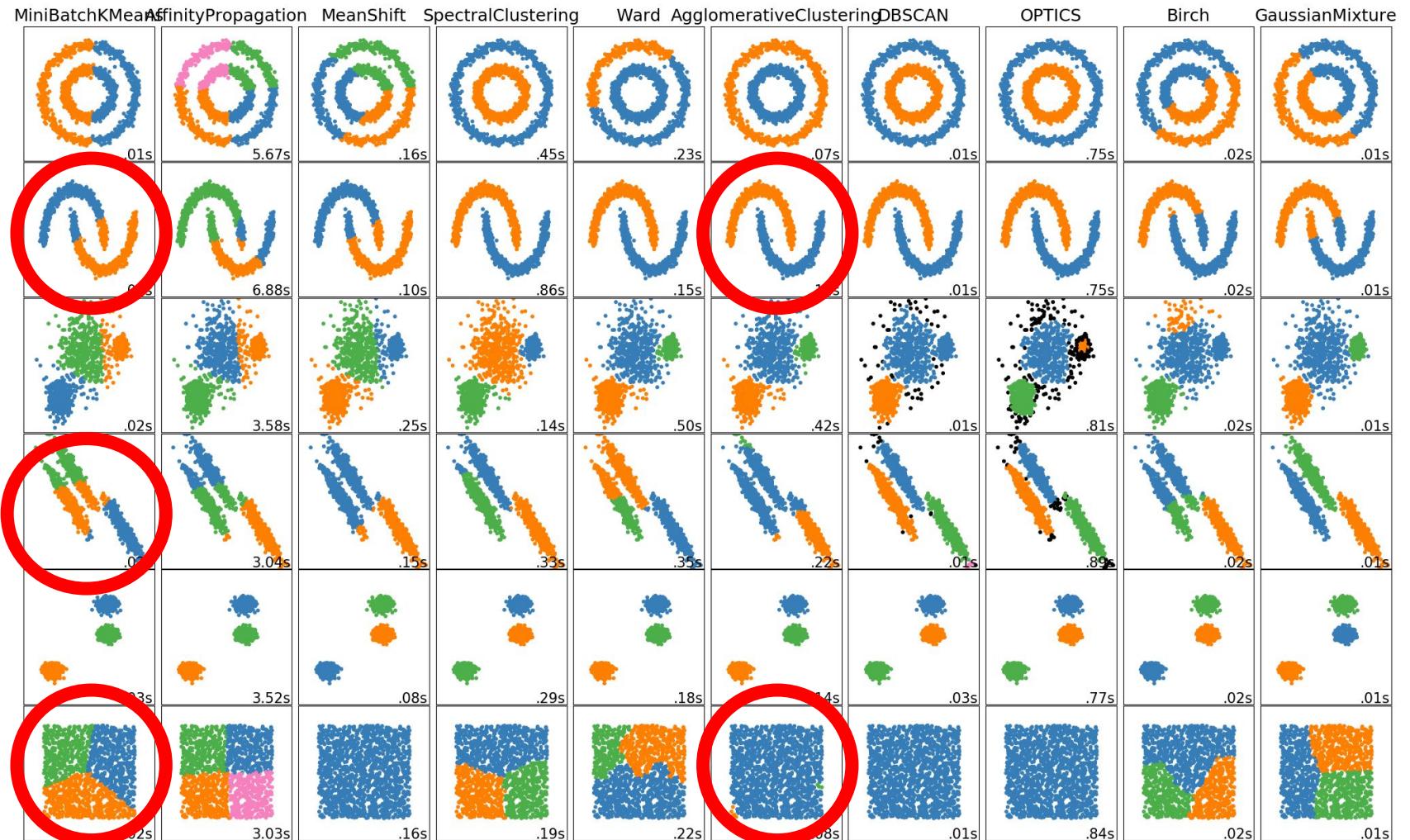
Linkage criteria

Single linkage (smallest distance) ~ DBSCAN

Complete linkage (maximum distance)

Minimum energy (variance grows slowly in we merge)

Average distance and centroid-based approaches



Offtopic: remember our matrix topic?

Matrices are helping to cluster: spectral analysis

Spectrum of a matrix = list of eigenvalues

Spectral matrix analysis - utilizing eigenvalues and eigenvectors to find interesting matrix properties

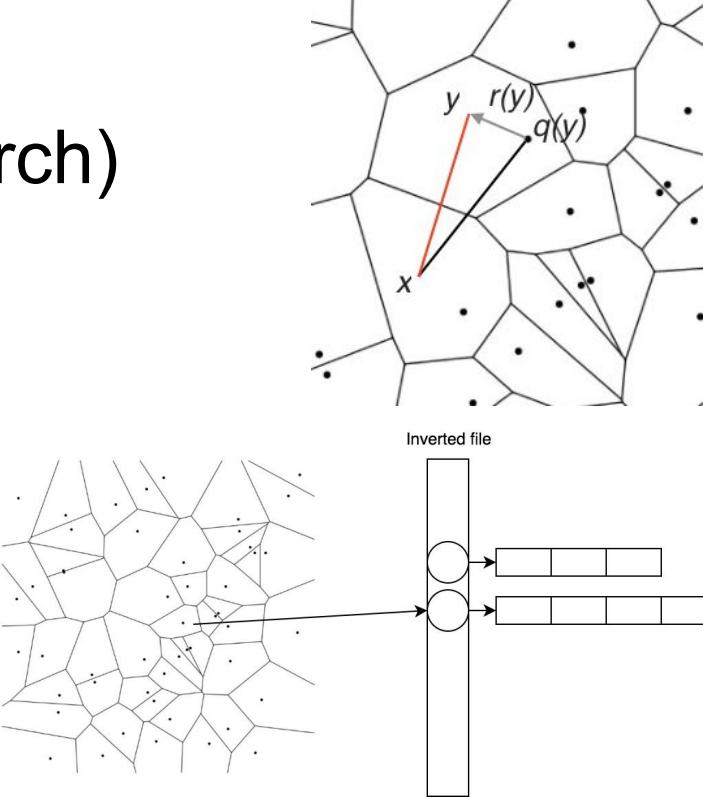
Adjacency matrix and Laplacian adjacency matrix are used to work with cuts:

- Cheeger constant $h(G) = \min_{0 < |S| \leq \frac{n}{2}} \frac{|\partial(S)|}{|S|}$, can be estimated using adjacency matrix second eigenvalue
- Shi-Malik algorithm uses cuts to segment images

PS see also how Checkanovski clustering (rus) utilize transitive closure ~ single linkage

FAISS (Facebook AI similarity search)

- Use [Voronoi diagram](#) clusters. Vectors are approximated with **centroids** (ADC - asymmetric distance computation)
- Build **inverted index** for points in clusters
- Vector compression: product quantizer
 - Split \mathbb{R}^{128} into 8 groups of 16 floats
 - Perform 256-means clustering of these “sub-vectors” and encode with 1 byte each



Approach #2. Proximity graphs

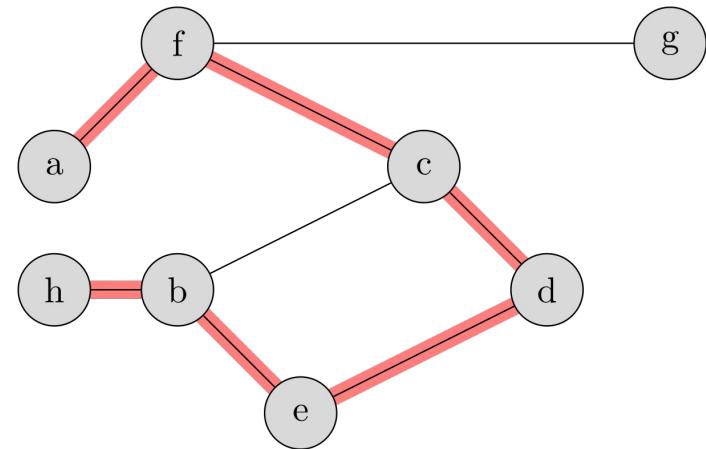
Graphs cheat sheet

Graph - $G = (V, E)$, can be weighted, directed, finite

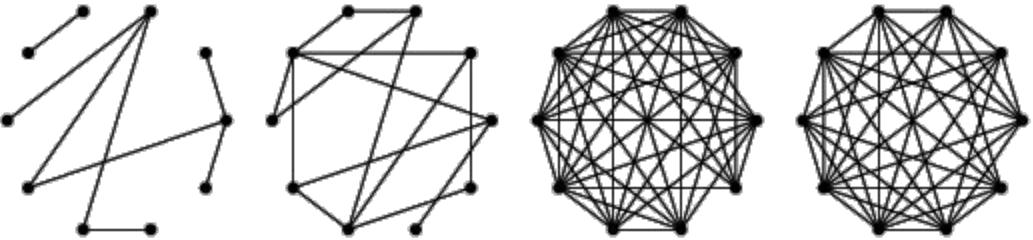
[Simple] **path** - sequence of vertices and edges

Degree of vertex - number of incident edges

Graph diameter - longest shortest path between a pair of vertices



Random graph



Some random process (uniform, Gaussian, ...) generates edges.

Almost every graph in the world. *Previously* considered as a model for social networks.

Small average shortest path - which is **good** for search.

Small clustering coefficient (defines how close are neighborhoods to cliques) - which is **bad** for **NN search**.

$$C(v) = \frac{e(v)}{\deg(v)(\deg(v)-1)/2}$$

$$\tilde{C} = \frac{1}{N} \sum_{i=1}^N C(i)$$

Regular graphs

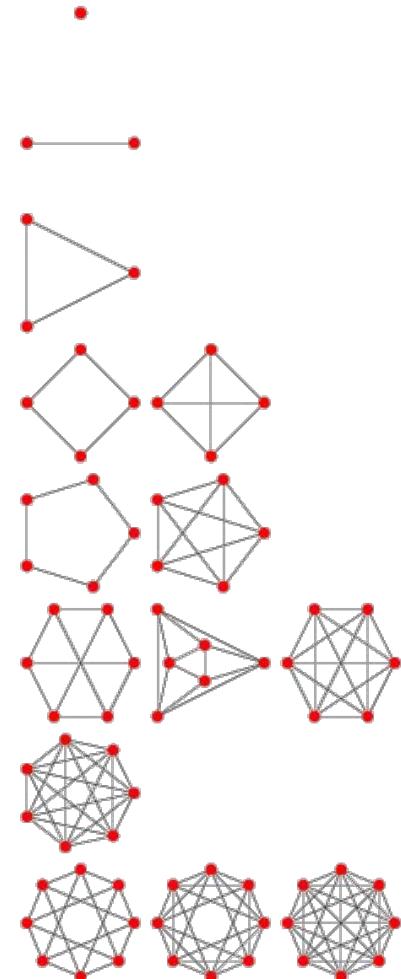
K -regular graph is a graph with $\deg(v) = K$ for any v .

Used to model big homogeneous networks.

Can also be random (as there are multiple K -regular graphs on the same size)

Big diameter - which is **bad** for **search**

Big clustering coefficient - which is **good** for **NN search**



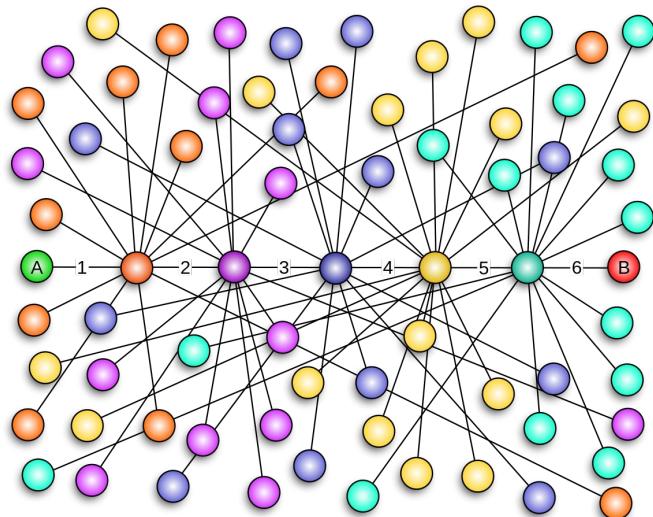
Small World experiment by Stanley Milgram, 1967

Initially it was considered, that social graph is kind of regular.

Experiment discovered (even with some questions to method) that even graph is **highly clustered, average path length is small.**

Was a basis for 6 handshakes rule.

New type of graphs was suggests:
small world networks.



Small world network

Most vertices are not neighbours (small degree means *sparse* graph).

Nevertheless, small number of hops needed to reach any other node.

Typical path length L between 2 random nodes (of N): $L \propto \log N$

Many real world networks are like this: internet, wiki, social graphs, power grids, brain cells. Although not all real networks like SW: many-generation networks, classmate graphs.

[Watts–Strogatz model](#) and [Kleinberg model](#) are how we describe and build SW networks

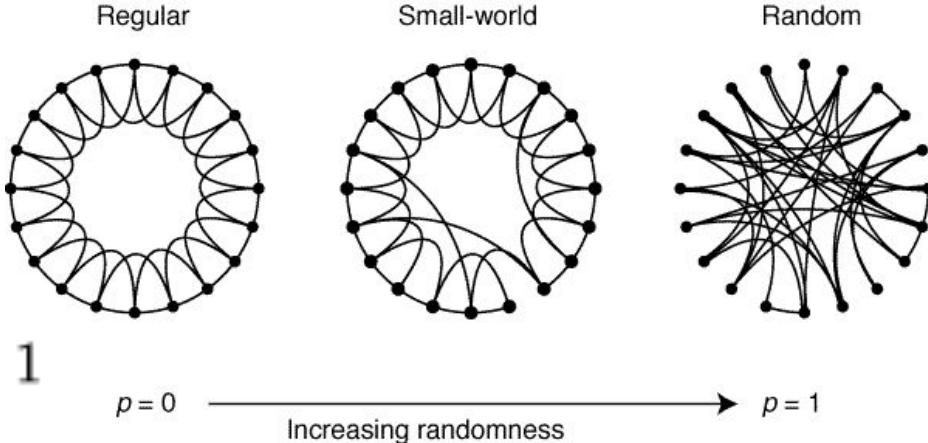
Watts–Strogatz model

Given N nodes and K -”regularity”

(average degree K) $N \gg K \gg \ln N \gg 1$

Given parameter p from $[0, 1]$.

- 1) Construct a regular ring lattice.
- 2) take every edge connecting **vertex** to its **$K/2$ rightmost neighbors**, and rewire it with probability p . Rewiring is done by **replacing destination** with vertex k (chosen **uniformly** at random from all possible nodes while avoiding self-loops and duplication).



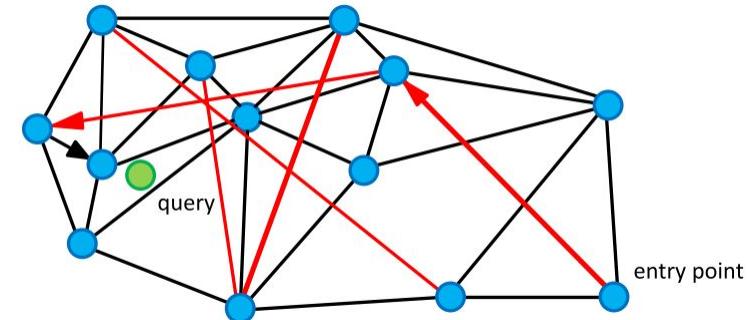
Navigable small world networks

Idea is similar to skip-lists.

We can also measure **distance** (e.g. dot product, Euclidian, L_k -norm, Hamming, Levenshtein, ...) between query and current vertex. Originally Delaunay graph needed to converge for exact search, but ANNS allows other small-world graphs.

Search:

1. Perform greedy search. Move to the neighbour vertex **closest to query**
2. Update NN set on each step until it converges



Hierarchical navigable small world (github)

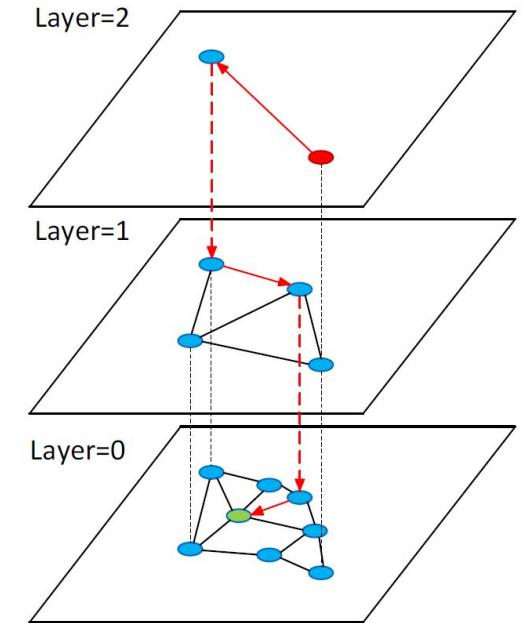
Layer 0 holds complete NSW network

Ideas:

- Better **start search** from a node with **high degree**
- Higher layer has longer links (skip-list!)
- Decrease layer size exponentially

Highlight:

- 1) **building [H]NSW requires no information about values and semantics of nodes**
- 2) **search procedure requires only dist(u, v) function**
- 3) **No embedding, hyperplanes, centroids of whatever needed**



Quad Trees A Data Structure for Retrieval on Composite Keys

R. A. Finkel and J. L. Bentley

Received April 8, 1974

Summary. The quad tree is a data structure appropriate for storing information to be retrieved on composite keys. We discuss the specific case of two-dimensional retrieval, although the structure is easily generalised to arbitrary dimensions. Algorithms are given both for straightforward insertion and for a type of balanced insertion into quad trees. Empirical analyses show that the average time for insertion is logarithmic with the tree size. An algorithm for retrieval within regions is presented along with data from empirical studies which imply that searching is reasonably efficient. We define an optimized tree and present an algorithm to accomplish optimization in $n \log n$ time. Searching is guaranteed to be fast in optimized trees. Remaining problems include those of deletion from quad trees and merging of quad trees, which seem to be inherently difficult operations.

Introduction

One way to attack the problem of retrieval on composite keys is to consider records arranged in a several-dimensional space, with one dimension for every attribute. Then a query concerning the presence or absence of records satisfying given criteria becomes a specification of some (possibly disconnected) subset of that space. All records which lie in that subset are to be returned as the response to the query.

The retrieval of information on only one key has been well studied. Experience has shown binary trees serve as a good data structure for representing linearly ordered data, and that balanced binary trees provide a guaranteed fast structure (Knuth, 6.2.3).

This paper will discuss a generalization of the binary tree for the treatment of data with inherently two-dimensional structure. One clear example of such records is that of cities on a map. A sample query might be: "Find all the cities which are within 300 miles of Chicago or north of Seattle." The data structure we propose to handle such queries is called a quad tree. It will be obvious that the basic concepts involved are easily generalized to records of any dimensionality.

Definitions and Notation

The location of records with two-dimensional keys will be stored in a tree with out-degree four at each node. Each node will store one record and will have up to four sons, each a node. The root of the tree divides the universe into four quadrants, namely NE, NW, SW, and SE (using the map analogy). Let us call these quadrants one, two, three and four, respectively. Fig. 1 shows the correspondence between a simple tree and the records it represents.

The convention we use for points which lie directly on one of the quadrant lines emanating from a node is as follows: Quadrants one and three are closed,

and quadrants two and four are open. Thus a point on the line due east of a node is in quadrant one of that node. Actually, the convention assumed is of little import to the basic idea behind the tree; the one we have chosen is nice for the purpose of deciding quickly in which quadrant of the root a given node lies.

Notice that we do not take into account the possibility of nonunique records (collisions). If collisions are indeed valid in a particular application, an extra pointer could be placed in each node for a linear list of similar records, or some more sophisticated structure could be used to store the collisions.

Given two records, say A and B , we define $\text{COMPARE}(A, B)$ to be an integer representing which quadrant of A holds B . We will define $\text{COMPARE}(A, A)$ to be 0.

It is occasionally convenient to be able to discuss the direction directly opposite to a given direction. We will use the notation $\text{CONJUGATE}(\text{DIRECTION})$ to represent the direction in question. For example, $\text{conjugate}(3) = 1$. A simple formula for conjugate is $\text{conjugate}(N) = ((N + 1) \bmod 4) + 1$.

It is not necessary to specify any exact implementation for nodes; many alternatives are possible, and the choice made will almost certainly be language-dependent. However, we need some notation for the purpose of discussing the algorithms. Therefore, let us agree to these conventions: There is a datatype NODE; the entire tree is a node. To denote a subtree of a node, say the third subtree of the node called ELM, we will write $\text{ELM}[3]$. NULL is the empty node. By convention, the zeroth subtree (say $\text{ELM}[0]$) is always NULL.

The algorithms in this paper will be presented in an ad-hoc version of ALGOL.

Insertion

Insertion of new records into quad trees is based on the same philosophy that governs insertions into binary trees; at each node, a comparison is made and the correct subtree is chosen for the next test; upon falling out of the tree, the algorithm knows where to insert the new record.

Here is an algorithmic description of the process:

```

PROCEDURE INSERT (NODE VALUE K, R);
BEGIN
  COMMENT: Inserts record K in the tree whose root is R;
  INTEGER DIRECTION; COMMENT: Direction from father to
    son, i.e. 1, 2, 3, or 4;
  DIRECTION ← COMPARE (R, K);
  WHILE R[DIRECTION] ≠ NULL DO
    BEGIN
      COMMENT: Each iteration dives one level deeper;
      R ← R[DIRECTION];
      DIRECTION ← COMPARE (R, K);
    END;
  IF DIRECTION = 0 THEN RETURN; COMMENT: Node
    already exists;
  R[DIRECTION] ← K;
END

```

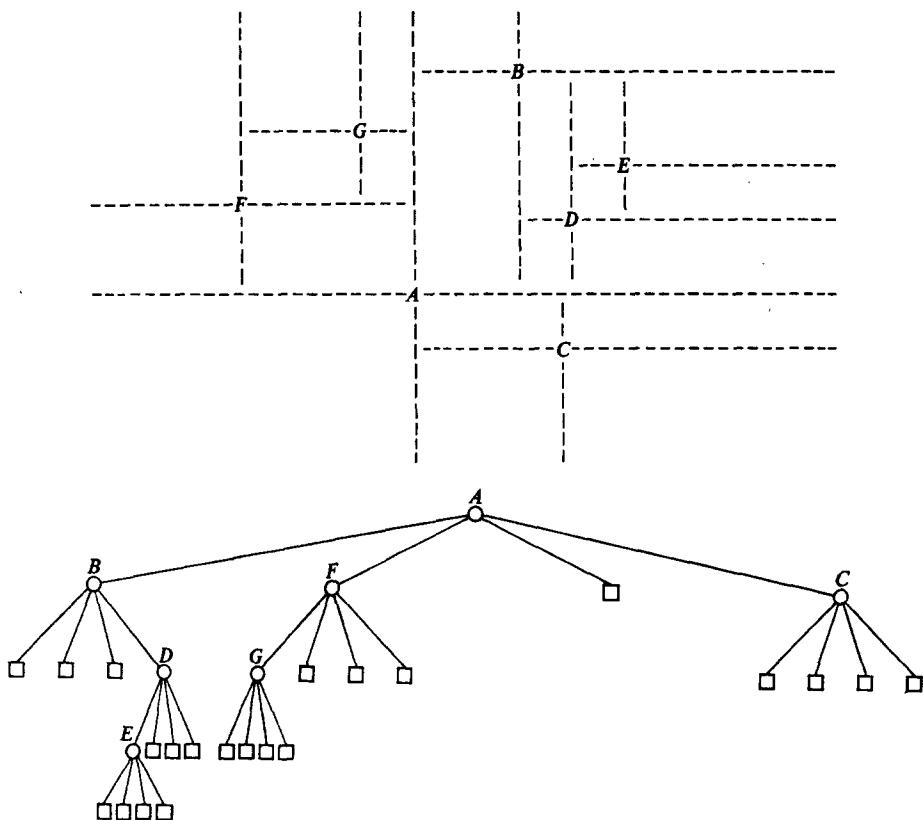


Fig. 1. Correspondence of a quad tree to the records it represents. RECORDS A, B, C, D, E, F, G . Null subtrees are indicated by boxes, but they do not appear explicitly in computer memory

This algorithm has been used to construct many trees of various sizes in order to collect statistics on its performance. The nodes were uniformly distributed in both coordinates, with key values ranging from 0 to $2^{31} - 1$. Table 1 is a summary of the results.

Here X is the quantity $(\text{AVERAGE TPL})/(n \log n)$, where n is the number of nodes, and the natural logarithm is used. The Low and High X are one standard

Table 1. Data on straightforward insertion

Number of nodes	Total path length		X	Low X	High X	Number of trees
	AVG	STD DEV				
25	67.21	8.699	0.8352	0.7271	0.9433	300
50	168.4	17.34	0.8608	0.7722	0.9495	300
100	403.5	30.68	0.8763	0.8096	0.9429	150
1000	6288	325.6	0.9103	0.8632	0.9575	30
10000	84705	2882	0.9197	0.8884	0.9510	10

deviation on either side of X . These are of mild interest; they show that for random nodes the algorithm seldom gets much worse than the average.

The most interesting aspect of Table 1 is the column marked X ; its slowly growing nature implies that the TPL of a quad tree under random insertion is roughly proportional to $N \log N$, where N is the number of nodes. Therefore, point searching in a quad tree can be expected to take about $\log N$ probes.

It should be noted, however, that the extreme case is much worse. If each successive node gets placed as a son of the currently lowest node in the tree, then the resulting tree will have TPL of $n(n - 1)/2$. Thus worst-case insertion and searching are order n^2 operations.

More Sophisticated Insertion

In an effort to lessen the TPL and to find some analog to balancing of binary trees, a simple balancing algorithm was developed. It does not aim to keep the height of sons of a given node within some fixed distance of each other, but rather takes note of the fact that some simple situations can be represented in two ways, one of which has lower TPL. Two examples are shown in Fig. 2 and 3; the former demonstrates what is termed a single balance; the latter a double balance (performed if $\text{COMPARE}(B, C)$ is the conjugate of $\text{COMPARE}(A, B)$).

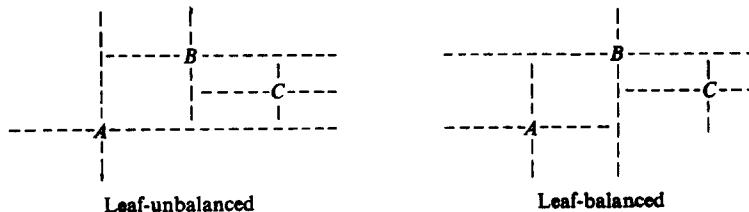


Fig. 2. Single balance

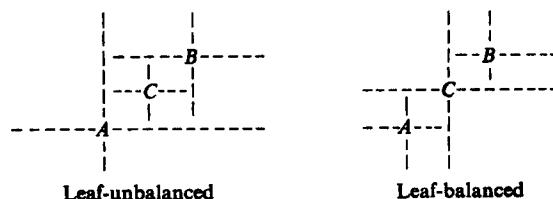


Fig. 3. Double balance

Table 2. Data on leaf-balanced insertion

Number of nodes	Total path length		X	Low X	High X	Number of trees
	AVG	STD DEV				
25	60.57	5.703	0.7526	0.6818	0.8235	300
50	152.9	12.37	0.7818	0.7185	0.8450	300
100	367.6	24.13	0.7982	0.7458	0.8506	150
1000	5812	247.0	0.8414	0.8057	0.8772	30
10000	78020	1960	0.8471	0.8258	0.8684	10

Table 2 gives statistics for the algorithm which performs these leaf balances during insertion. The nodes used were identical to those used in the tests of straightforward insertion. Of interest are the about 10% lower values of X and the slightly lower standard deviations. It should be remarked, however, that at times a tree built by the leaf-balancing algorithm has higher TPL than a tree built by the straightforward insertion algorithm out of the same nodes, arriving in the same order.

Searching

There are two general classes of searches facilitated by quad trees. The first is a point search for a single record, invoked by a query like "Is the point (37, 18) in the data structure, and what is the address of the node representing it?" The second is a region search, invoked by something like "What are all the points within a circle of center (15, 31) and radius 7?" or (if the quad tree's two dimensions represent age and annual income) "Who are all the people between the ages of 21 and 28 who earn between \$ 12000 and \$ 14000 per year?" Our definition of point search corresponds to Knuth's "simple query" (Knuth, 6.5) and our region search encompasses both a "range query" and a "boolean query" in Knuth's terminology.

Point searching is accomplished by an algorithm much like that used for insertion. The average time required to do a point search for a node in the tree is proportional to

$$\text{TPL}/(\text{number of nodes in tree}).$$

This is the algorithm used to perform a region search:

```

PROCEDURE REGIONSEARCH (NODE VALUE P; REAL VALUE L, R, B, T);
BEGIN
  COMMENT: Recursively searches all subtrees of P to find all nodes within
          window bounded by L(left), R(right), B(bottom), and T(top) which
          are in region in question;
  REAL XC, YC; COMMENT: The X and Y coordinates of P;
  XC ← X(P);
  YC ← Y(P);
  IF IN_REGION (XC, YC) THEN FOUND (P);
  IF P[1] ≠ NULL AND RECTANGLE_OVERLAPS_REGION
    (XC, R, YC, T)
    THEN REGIONSEARCH (P[1], XC, R, YC, T);
  IF P[2] ≠ NULL AND RECTANGLE_OVERLAPS_REGION
    (L, XC, YC, T)
    THEN REGIONSEARCH (P[2], L, XC, YC, T);
  IF P[3] ≠ NULL AND RECTANGLE_OVERLAPS_REGION
    (L, XC, B, YC)
    THEN REGIONSEARCH (P[3], L, XC, B, YC);
  IF P[4] ≠ NULL AND RECTANGLE_OVERLAPS_REGION
    (XC, R, B, YC)
    THEN REGIONSEARCH (P[4], XC, R, B, YC);
END

```

This algorithm must be supplied with two procedures to be used during searching and one procedure that is informed of each node found in the region. Procedure IN-REGION is a boolean procedure which is passed the x and y coordinates of a node and returns TRUE if and only if the node is in the region. Procedure RECTANGLE_OVERLAPS-REGION is passed the left, right, bottom, and top extreme values of a rectilinearly oriented rectangle. (For example, the parameters $(4, 6, 1, 3)$ represent the rectangle bounded by the lines $x=4$, $x=6$, $y=1$, $y=3$). RECTANGLE_OVERLAPS-REGION returns TRUE if the region being searched overlaps the described rectangle and FALSE otherwise. Procedure FOUND is given each node found in the region to allow processing.

The algorithm itself is recursive and is initially invoked by the statement

REGIONSEARCH (ROOT, MIN_X, MAX_X, MIN_Y, MAX_Y)

where ROOT is the root node of the tree, and the rest of the parameters are defined such that for all x and y values represented in the quad tree,

$$(MIN_X \leq x \leq MAX_X) \text{ and } (MIN_Y \leq y \leq MAX_Y).$$

Here are the auxiliary procedures IN-REGION and RECTANGLE_OVERLAPS-REGION necessary to search for all points in the rectilinearly oriented rectangle defined by $(BP \leq y \leq TP)$ and $(LP \leq x \leq RP)$:

BOOLEAN PROCEDURE IN-REGION (REAL VALUE X, Y);

COMMENT: Returns TRUE if (X, Y) is in region;

RETURN $((LP \leq X) \wedge (X \leq RP) \wedge (BP \leq Y) \wedge (Y \leq TP))$;

BOOLEAN PROCEDURE RECTANGLE_OVERLAPS-REGION

(REAL VALUE L, R, B, T);

COMMENT: Returns TRUE if region overlaps rectangle

bounded by L, R, B , and T ;

RETURN $((L \leq RP) \wedge (R \geq LP) \wedge (B \leq TP) \wedge (T \geq BP))$;

Similar procedures can be defined to search for points in any connected geometric figure. After these basic procedures have been defined, the logical operators AND, OR and NOT can be used to search within unions, intersections, and complements of regions. Thus, the specification of the region of interest is highly flexible.

The versatility of this algorithm makes a formal analysis very difficult, so we have conducted some empirical tests to investigate its efficiency. We let the x and y coordinates of all points in the quad tree be real-valued numbers in $[0, 1]$. The regions of search were randomly located, rectilinearly oriented squares having a given edge size. For each of the tree sizes for which we gathered data, we randomly generated four quad trees. For all of the edge sizes presented we searched each of the four trees 25 times; a total of 100 searches per edge size. In Table 3 we show for every combination of tree and edge size the number of nodes visited in the 100 searches (Visited), of those visited the number that were actually in the region (Found), and significant ratios of these data. For example, there were four 250-node quad trees tested. On each of these trees, 25 searches were made for all the points within squares of edge 0.125 located at random

positions within the unit square. In these 100 searches a total of 1820 nodes were visited. Of the 1820 visited, 387 were in the particular squares being searched. The ratio of those visited to the number of searches was 18.20, the ratio of the number of nodes found to the number of searches was 3.87, and the ratio of those visited to those found was 4.70.

The two most significant figures in the table are the values Visited/Found and Visited/Search. Visited/Found is indicative of the amount of total work the algorithm has to do to find a particular node in the region it is searching. Visited/Search shows the amount of work the algorithm does to find all nodes in a certain region. It is pleasant to note in Table 3 that as Visited/Search increases, Visited/Found decreases, implying one is going to be low as the other is high. It is likely that the lower measure will be of more concern to the user; when one is gathering a large amount of data from a certain region one is more concerned about the cost per collected node, and when one is making a large number of small searches one is more concerned about the cost of each search.

Table 3. Region-search data

Number of nodes	Edge Size	Visited	Found	Number of Searches	Visited/ Search	Found/ Search	Visited/ Found
125	0.03125	598	13	100	5.98	0.13	46.00
	0.0625	789	41	100	7.89	0.41	19.24
	0.125	1218	206	100	12.18	2.06	5.91
	0.25	2195	799	100	21.95	7.99	2.75
	0.5	5188	3130	100	51.88	31.30	1.66
250	0.03125	777	22	100	7.77	0.22	35.32
	0.0625	1074	103	100	10.74	1.03	10.43
	0.125	1820	387	100	18.20	3.87	4.70
	0.25	3562	1536	100	35.62	15.36	2.32
	0.5	9550	6390	100	95.50	63.90	1.50
500	0.03125	975	55	100	9.75	0.55	17.73
	0.0625	1493	205	100	14.93	2.05	7.28
	0.125	2641	754	100	26.41	7.54	3.50
	0.25	6248	3163	100	62.48	31.63	1.97
	0.5	17453	12860	100	174.53	128.60	1.36
1000	0.03125	1316	99	100	13.16	0.99	13.29
	0.0625	2144	376	100	21.44	3.76	5.70
	0.125	4246	1611	100	42.46	16.11	2.64
	0.25	10100	6172	100	101.00	61.72	1.64
	0.5	31845	24880	100	318.45	248.80	1.28
2000	0.03125	1619	183	100	16.19	1.83	8.85
	0.0625	2906	786	100	29.06	7.86	3.70
	0.125	6803	3106	100	68.03	31.06	2.19
	0.25	18347	12627	100	183.47	126.27	1.45
	0.5	60581	49943	100	605.81	499.43	1.21
4000	0.03125	2407	397	100	24.07	3.97	6.06
	0.0625	4369	1504	100	43.69	15.04	2.90
	0.125	11096	6218	100	110.96	62.18	1.78
	0.25	33133	24610	100	331.33	246.10	1.35
	0.5	114767	99875	100	1147.67	998.75	1.15

Deletion

It turns out to be very difficult to perform deletions from quad trees. The difficulty lies in deciding what to do with the subtrees that were attached to the deleted node. It is necessary to merge them into the rest of the tree, but merging is not an easy process. In fact, it seems that one cannot do better than to reinsert all of the stranded nodes, one by one, into the new tree. This answer is not very satisfactory, and it is a matter of some interest whether there exists any merging algorithm that works faster than $n \log n$, where n is the total number of nodes in the two trees to be merged.

Some attempt has been made to take advantage of the hope that not all subtrees of a newly-merged node need be disjointed, that some of them (which ones depending on in what direction the newly-merged son lies with respect to its father) can be left intact. This has so far not been found to be of much help.

An Optimization Algorithm

At times the simple balancing mentioned earlier is not sufficient; the nodes might be presented in a very lopsided order, or the tree might be needed frequently for point searches and never updated. These conditions make it worthwhile to consider expending extra effort at the outset to optimize the tree.

By an optimized tree we will mean a quad tree such that every node K has this property: No subtree of K accounts for more than one half of the nodes in the tree whose root is K . The first step in building an optimized quad tree from a collection of records is to order the nodes lexicographically primarily by the x coordinate and secondarily by the y coordinate. A simple recursive algorithm to complete optimization is this: Given a collection of lexicographically ordered records, we will first find one, R , which is to serve as the root of the collection, and then we will regroup the nodes into four subcollections which will be the four subtrees of R . The process will be called recursively on each subcollection. The root that is then found for each subcollection is linked in as an appropriate son of R . To find the root R , select the median element of the ordered list. The reason this works is that all records falling before R in the ordered list will lie in quadrants 3 and 4; those falling after it will lie in quadrants 1 and 2. Thus the condition is met: No subtree can possibly contain more than half the total number of nodes. The maximum path length in an optimized tree of n nodes is therefore $\lfloor \log_2(n) \rfloor$ and the maximum TPL is

$$\sum_{1 \leq i \leq n} \lfloor \log_2(i) \rfloor.$$

The running time for this algorithm is on the order of $n \log n$: The ordering step on n elements will take $n \log n$. On each level of recursion the selection of the median requires n , the regrouping of the records takes n and can be done in such a way that each group is still ordered. The depth of recursion will be the maximum path length which is bounded by $\log n$, so the time for optimization will also be $n \log n$.

Empirical studies with optimized trees of random nodes have shown that the TPL of trees after the treatment is roughly 15% lower than that obtained by

straightforward insertion. The measures of searching effectiveness (nodes visited per nodes found and nodes visited per search) remain roughly the same.

Conclusions

The quad tree seems to be an efficient means of storage for two-dimensional data. The most straightforward insertion algorithm yields $n \log n$ performance when given random keys. Region searching is quite efficient.

Unfortunately, deletion from quad trees and merging of two quad trees is not easy.

The basic concepts involved can easily be generalized to an arbitrary number of dimensions. (In m dimensions, each node has 2^m sons.)

References

Knuth, D. E.: The art of computer programming, vol. 3: Sorting and Searching.
Reading (Mass.): Addison-Wesley 1973

R. Finkel
Computer Science Department
Stanford University
Stanford, Calif. 94305/U.S.A.

J. Bentley
Computer Science Department
University of North Carolina
Chapel Hill, North Carolina 27514/U.S.A.

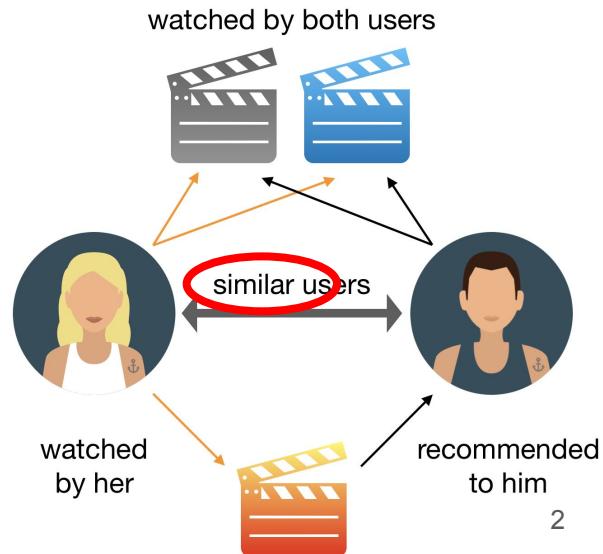
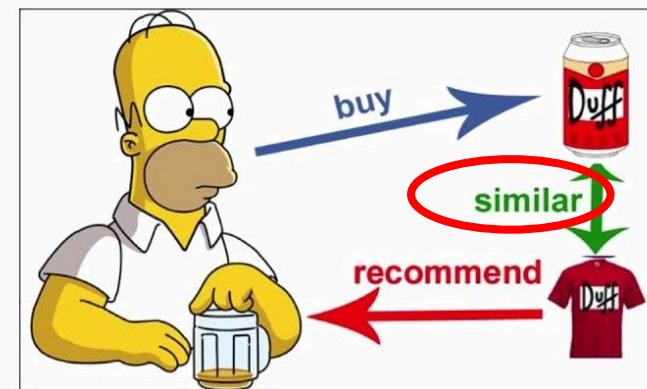
Forests of search trees

Stanislav Protasov



Problem statement

1. Number **N** of users and content items in today's services exceed 10^{10}
 - a. $\sqrt{N} = 10000$, $\log_{10}(N) = 10$
2. Content delivery in major services (Google, Facebook, Yandex, Netflix, markets, ...) is now managed by **recommendations**, not **subscription feeds**
3. Major recommendation systems utilize **vector representations** and **similarity functions**
4. Thus, recommendation to query is a **search** of **K** most similar object (often vectors) out of **N**



Hierarchical navigable small world (github)

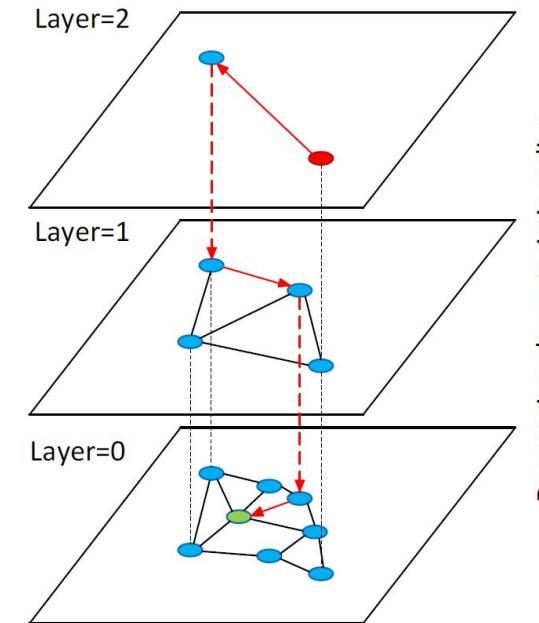
Layer 0 holds complete NSW network

Ideas:

- Better **start search** from a node with **high degree**
- Higher layer has longer links (see skip-list)
- Decrease layer size exponentially

Highlights:

- 1) Building [H]NSW requires no information about values and semantics of nodes
- 2) Search procedure requires only dist(u, v) function
- 3) No embedding, hyperplanes, centroids or whatever needed



Problem statement

Most frequently discussed search types:

- **Exact search (query is in the dataset)**
- Range search (all items satisfy a condition)
- **[Approximate] K nearest neighbours search (ANNS)**
 - Find K objects in the dataset, closest to a query (also object) with respect to some metric

Agenda

Trees - space subdivision

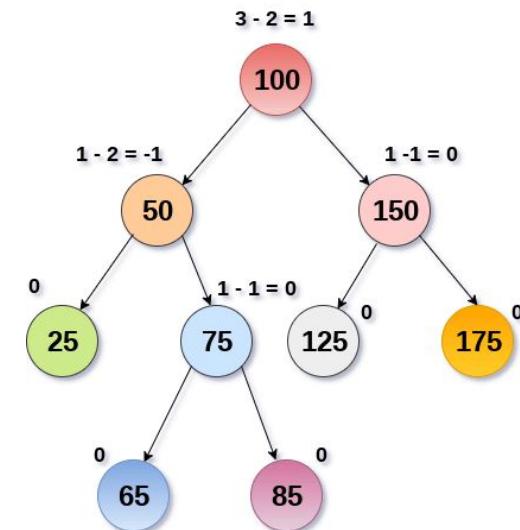
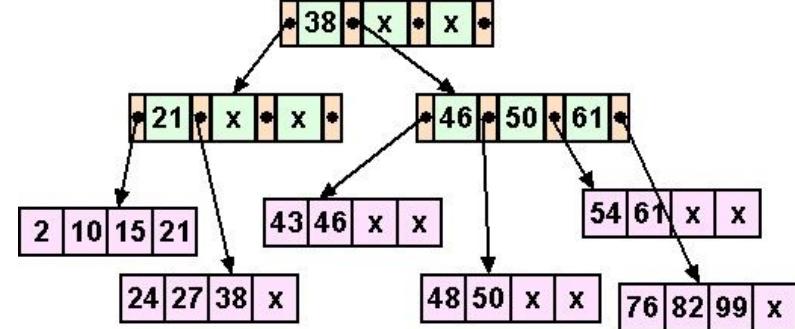
Trees - order in space

...

Trees - best for queries

Refresher for [B]ST

- K-ary (usually binary) trees
- Built upon comparable keys (scalars)
- Similar search procedure
- Preserved balance property, ensures $O(\log(N))$ max path length
- Can be *homogeneous* (AVL) and not (*B+* tree)



AVL Tree

But what if we have vectors?

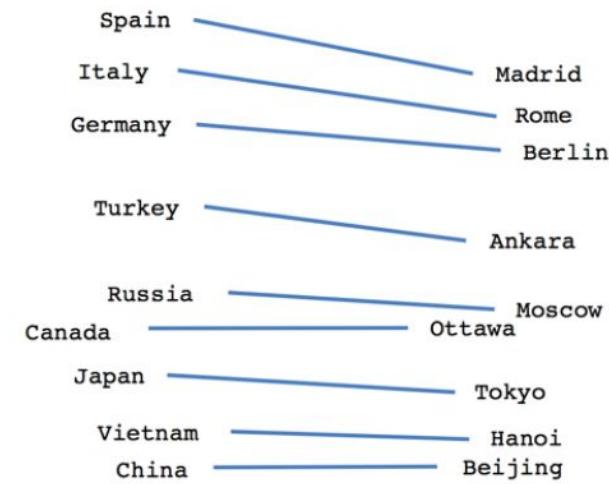
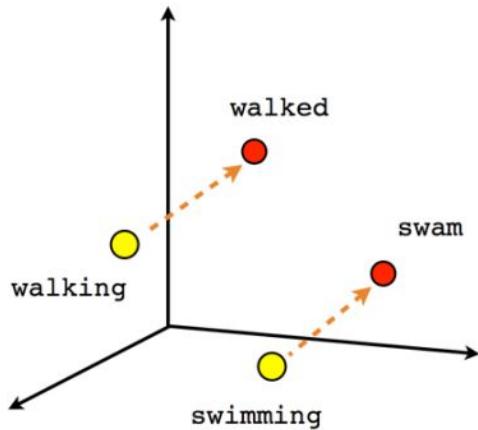
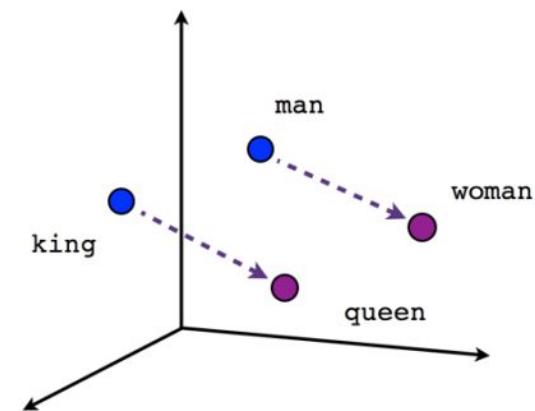
Latent [semantic] space

Hypothesis:

There exists a R^n space, where human understanding of object similarity and relations can be encoded with arithmetic operations ([latent space arithmetics](#))

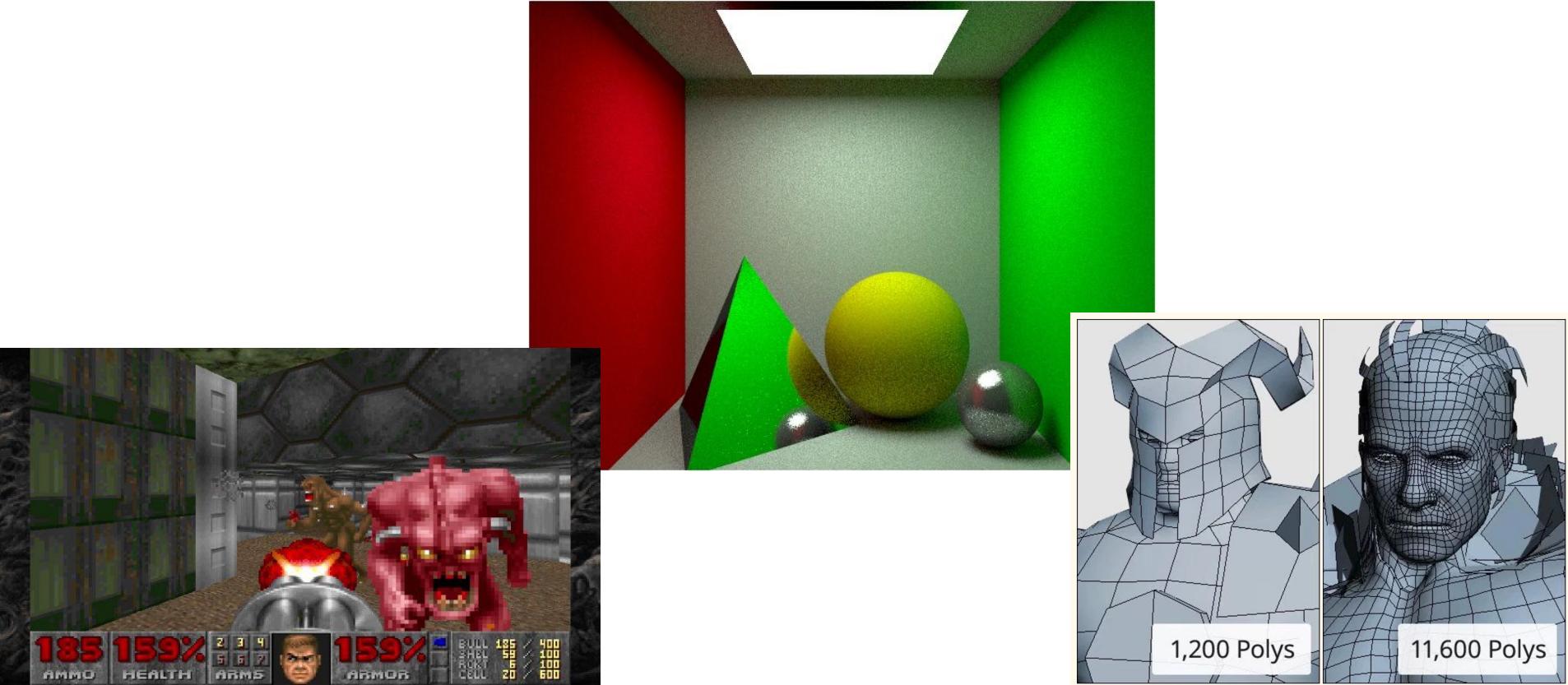
E.g.

‘King’ - ‘Queen’ + ‘Woman’ = ‘Man’



Country-Capital

And of course computer graphics!!!

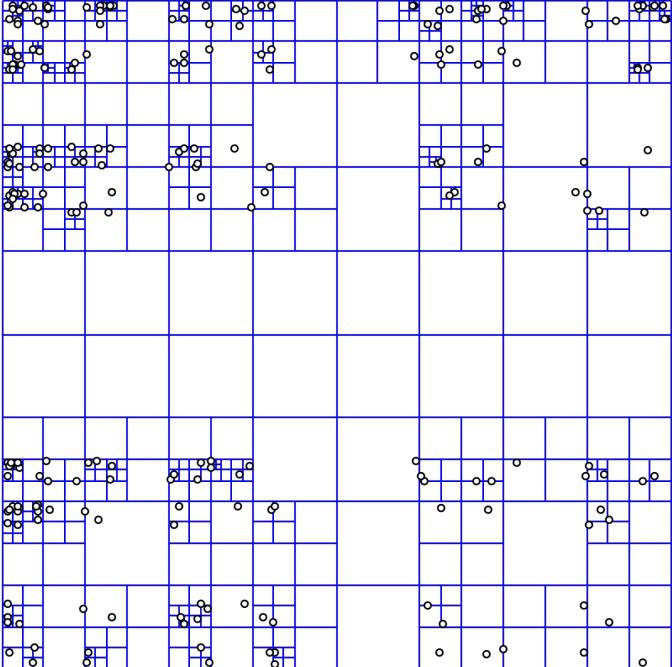


Trivial case: vector is a scalar

- Binary search trees:
 - Splay, RB, AVL trees are best for RAM
- N-ary search trees:
 - B-trees, LSM-trees are used with hard drives
- Search:
 - Exact search is $O(\log(N))$
 - K nearest neighbour search $O(\log(N) + K)$

QuadTree (1974)

- Forms of Quad Trees:
 - Region
 - Point
 - Edge
 - Polygon
- All forms of quadtrees share some common features:
 - decompose space into **adaptable** cells
 - Each cell (or bucket) has a **maximum capacity**.
When maximum capacity is reached, the bucket **splits**



QuadTree search

```
function queryRange(range) {  
    pointsInRange = [];  
    if (!this.boundary.intersects(range))  
        return pointsInRange;  
  
    for (int p = 0; p < this.points.size; p++) {  
        if (range.containsPoint(this.points[p]))  
            pointsInRange.append(this.points[p]);  
    }  
    if (this.northWest == null) // no children  
        return pointsInRange;  
  
    pointsInRange.appendArray(this.northWest->queryRange(range));  
    pointsInRange.appendArray(this.northEast->queryRange(range));  
    pointsInRange.appendArray(this.southWest->queryRange(range));  
    pointsInRange.appendArray(this.southEast->queryRange(range));  
    return pointsInRange;  
}
```

QuadTree insertion #1

```
function insert(p) {  
    if (!this.boundary.containsPoint(p))  
        return false; // object cannot be added  
    if (this.points.size < QT_NODE_CAPACITY && northWest == null) {  
        this.points.append(p);  
        return true;  
    }  
    if (this.northWest == null) this.subdivide();  
  
    if (this.northWest->insert(p)) return true;  
    if (this.northEast->insert(p)) return true;  
    if (this.southWest->insert(p)) return true;  
    if (this.southEast->insert(p)) return true;  
}
```

QuadTree insertion #2

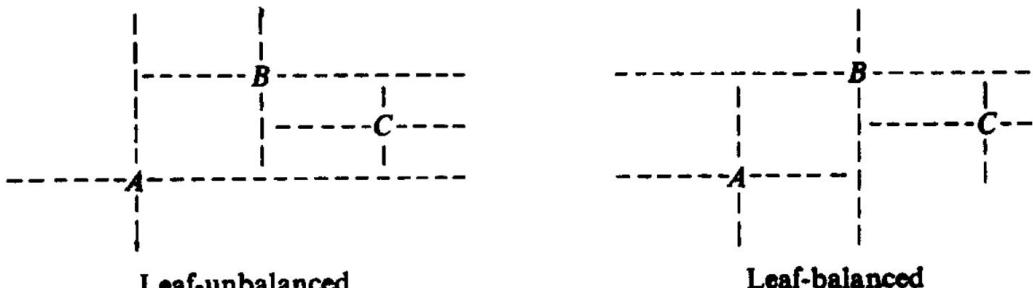


Fig. 2. Single balance

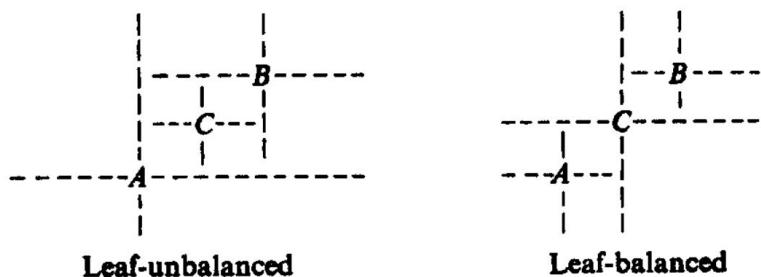


Fig. 3. Double balance

Table 2. Data on leaf-balanced insertion

QuadTree deletion

<<... In fact, it seems that **one cannot do better than to reinsert all of the stranded nodes, one by one, into the new tree. This answer is not very satisfactory, and it is a matter of some interest whether there exists any merging algorithm that works faster than $n \log n$, where n is the total number of nodes in the two trees to be merged...>>**

QuadTree optimization

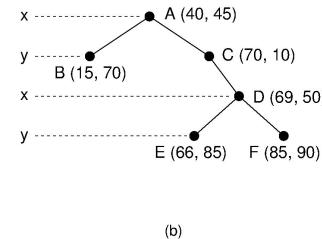
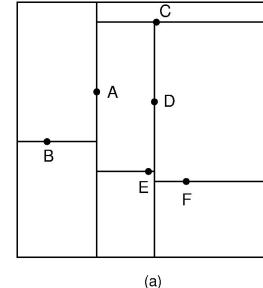
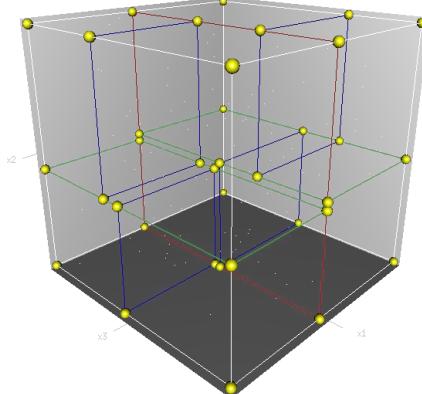
By an **optimized tree** we will mean a quad tree such that every node K has this property: **No subtree of K accounts for more than $\frac{1}{2}$** of the nodes in the tree whose root is K.

A simple recursive algorithm to complete optimization is this: Given a collection of **lexicographically ordered records**, we will first find one, R, which is to serve as the root of the collection, and then we will regroup the nodes into 4 subcollections which will be the four subtrees of R. The process will be called recursively on each subcollection... No subtree can possibly contain more than half the total number of nodes

Can you see any suboptimality?

K-d trees (1975)

Ideas:



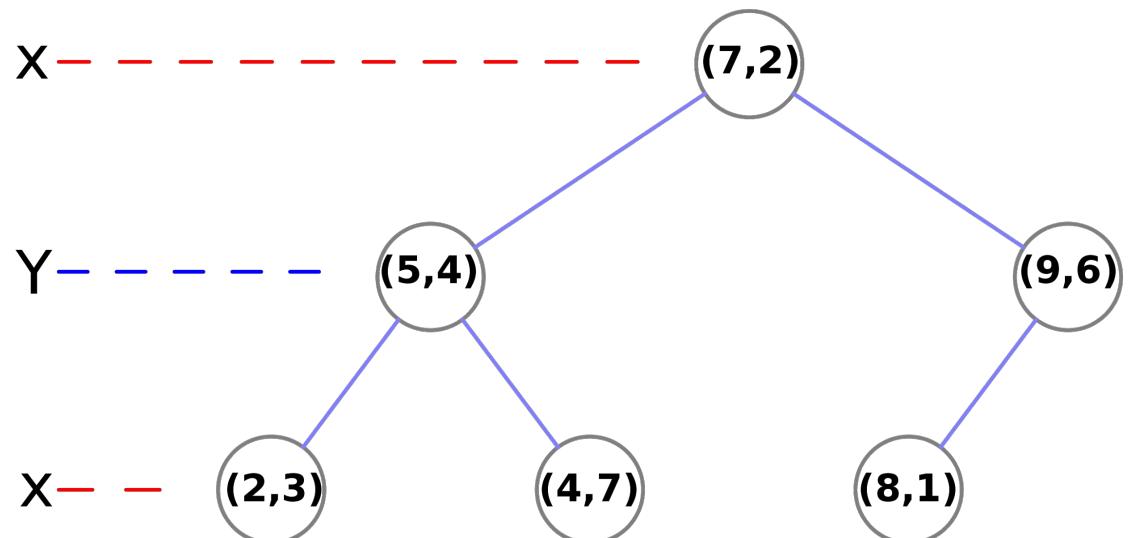
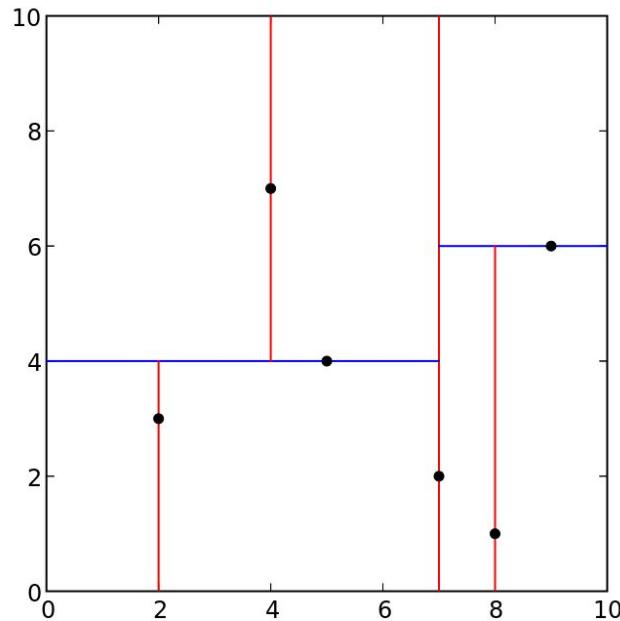
- Split points in 2 **equal** subspaces, not 4
- Use **alternating coordinates** at each level
 $(x, y, z, x, y, z, \dots)$
 - Thus, we need 2 levels to encode quadrants, but they are **equal**
 - And yes, this allows us to have **more than 2 dimensions**

K-d trees

Construction (“homogeneous”):

```
def buildKDTree(vectors, dim=0):
    if not vectors:
        return None      # stop condition, e.g.
    if len(vectors) == 1:
        return Node(vectors[0])
    vectors.sort(key = lambda x: x[dim]) # or Selection alg for O(N)
    med = len(vectors) // 2
    left, med, right = vectors[:med], vectors[med], vectors[med+1:]
    node = Node(med)
    node.left = buildKDTree(left, (dim + 1) % K)
    node.right = buildKDTree(right, (dim + 1) % K)
    return node
```

K-d trees: building example



K-d trees characteristics

Is built in $O(n(k + \log(n)))$ time

Requires $O(kn)$ memory (at most node for a point)

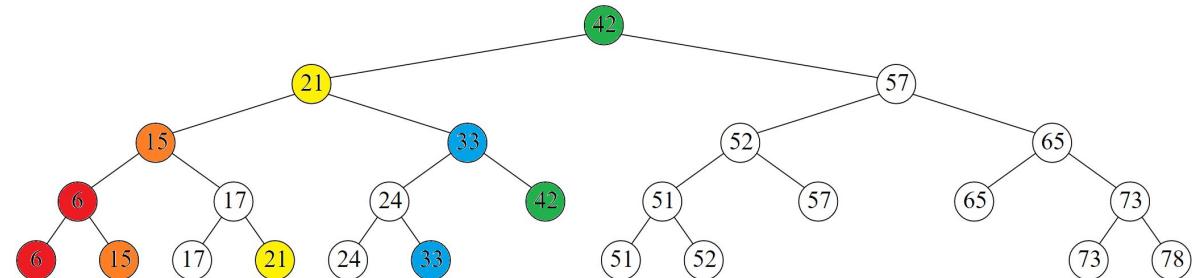
Runs range search for $O(n^{1-\frac{1}{k}} + a)$ where a — result size

Runs 1-NN search in $O(\log(n))$ time

To build hyperplanes it requires vector representation of keys

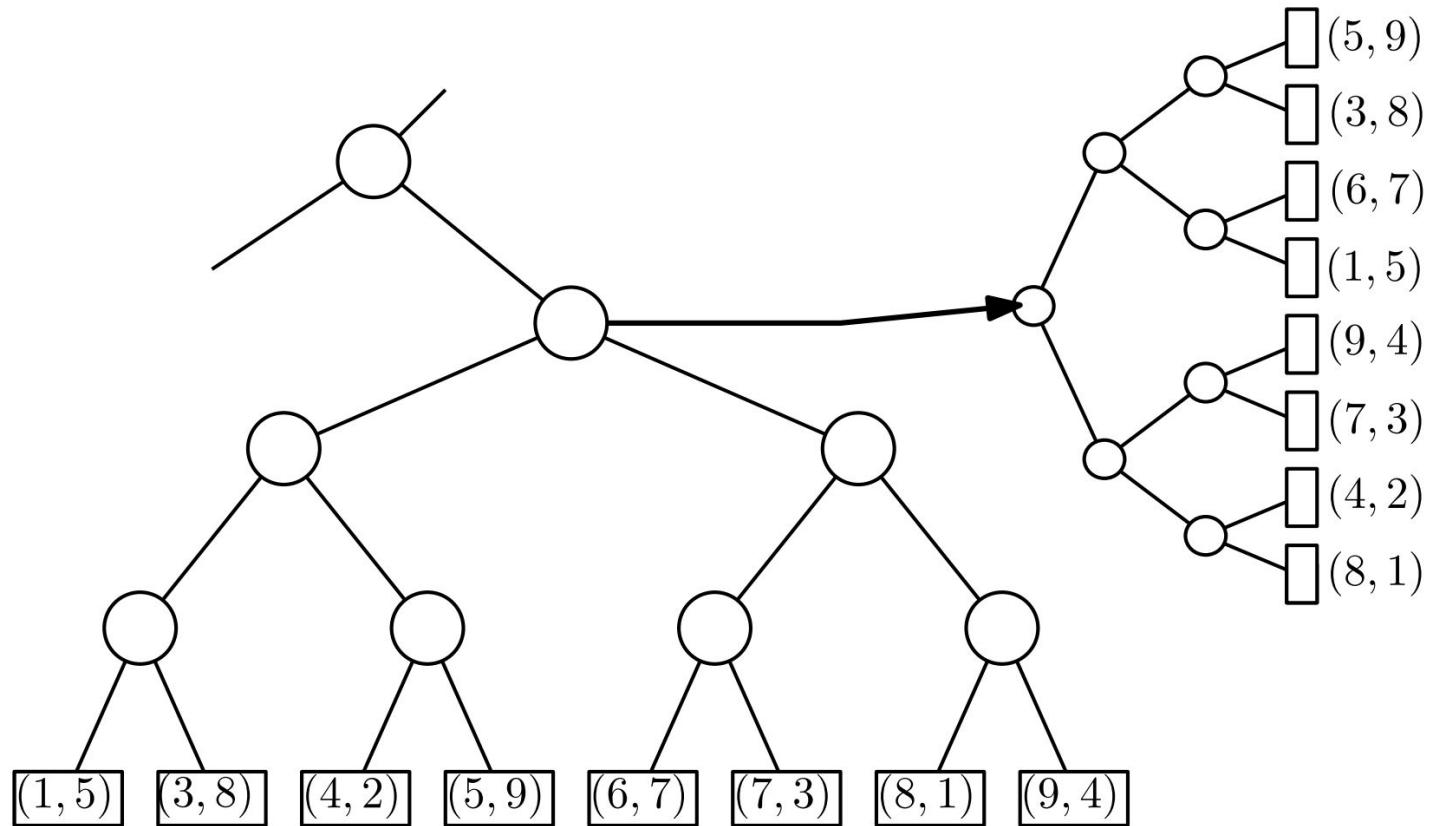
Faster range queries - range trees

A range tree on a set of **1-dimensional** points is a **balanced non-homogeneous binary search tree** on those points. Internal nodes store predecessors (largest to the left)



Range trees in **higher dimensions** are constructed recursively by constructing a balanced binary **search tree on the first coordinate** of the points, and then, for each vertex **v** in this tree, constructing a **(d-1)-dimensional range tree** on the points contained in the **subtree of v**

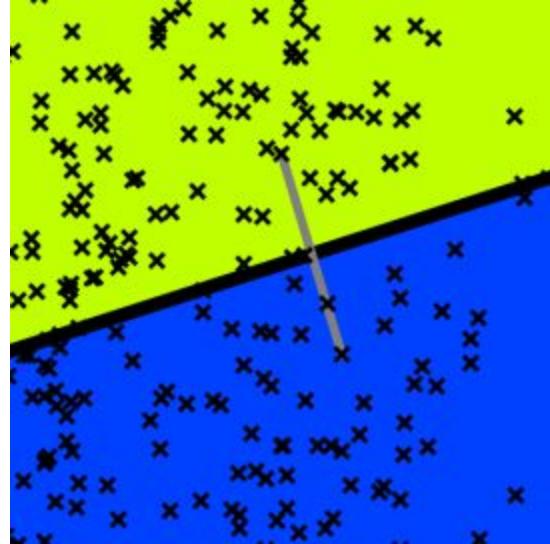
Image source link



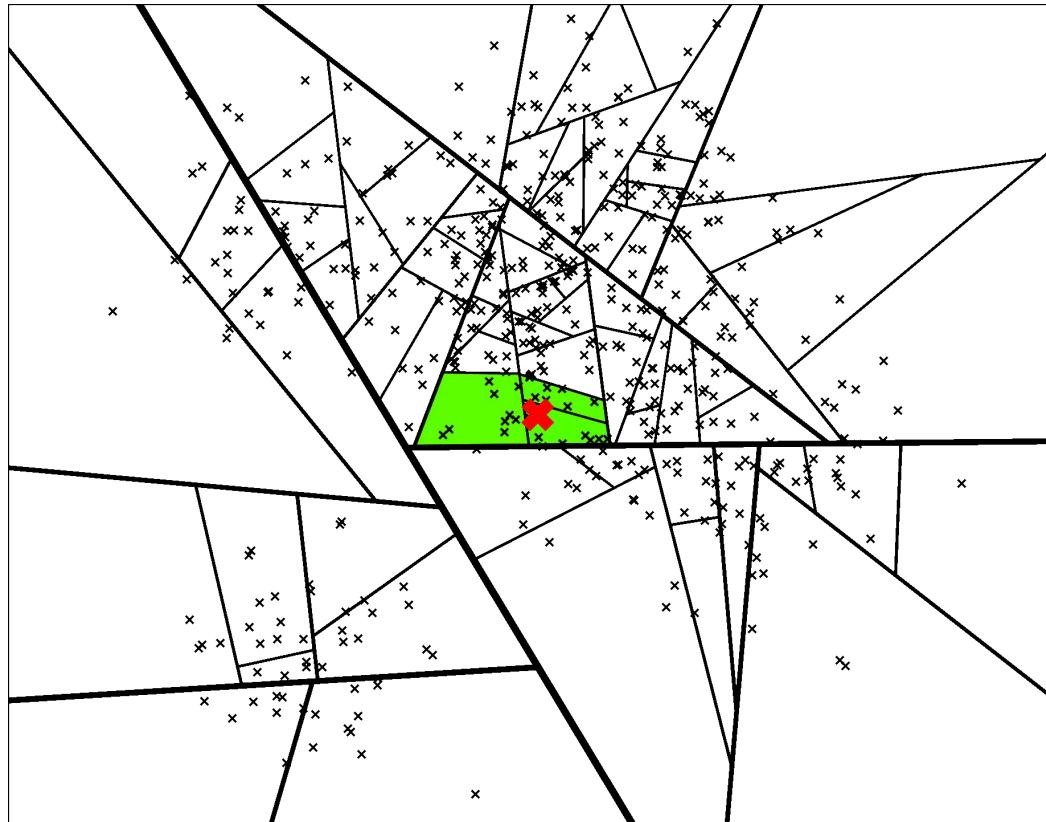
Sorting and looking for median is soooo
boring...

Annoy from Spotify (2015)

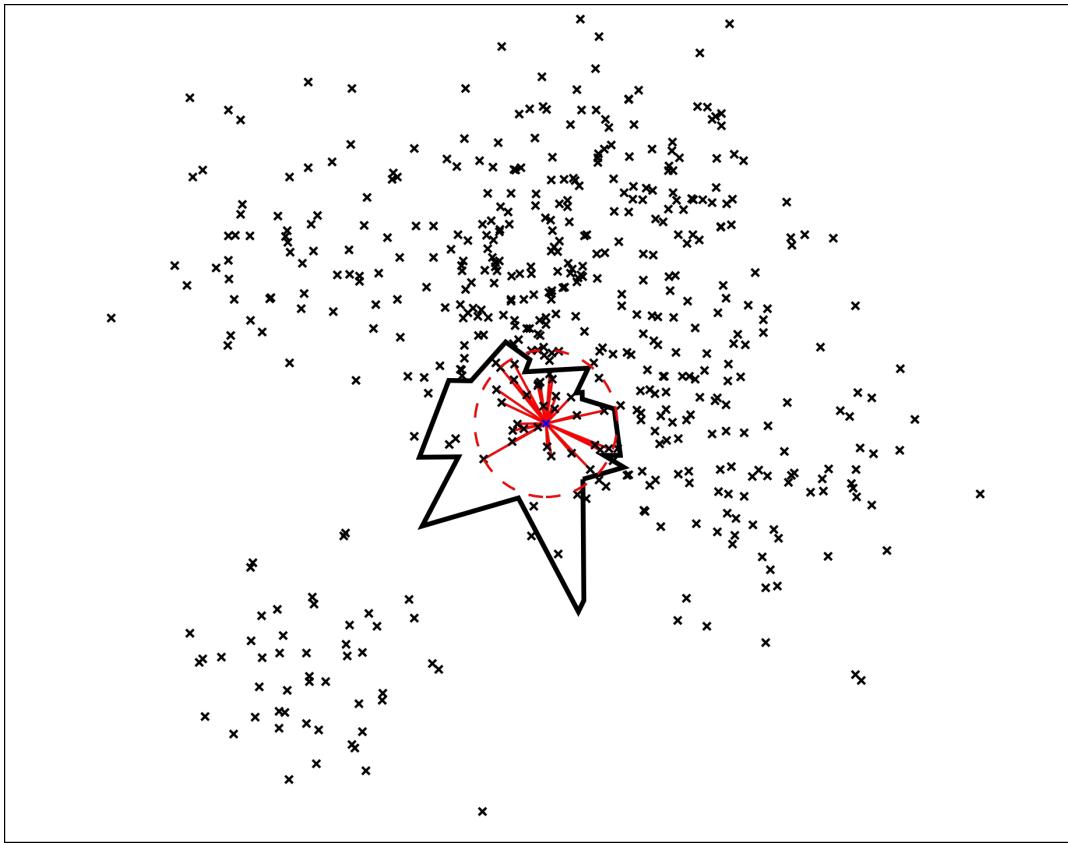
1. Instead of looking for a median, **select equidistant hyperplane for 2 random points** - then split is done in linear time ([random projection](#))
2. Use “**soft threshold**” that allows traversing “wrong” branches for ANNS
3. Build **multiple search trees** over the same dataset (*compare to multiple searches in NSW*)
4. Generalization of binary space partitioning ([BSP-tree](#)) used in CG (Doom, Quake, ...) for visibility sorting.



Multiple trees ([animation](#))



ANNS results with Annoy



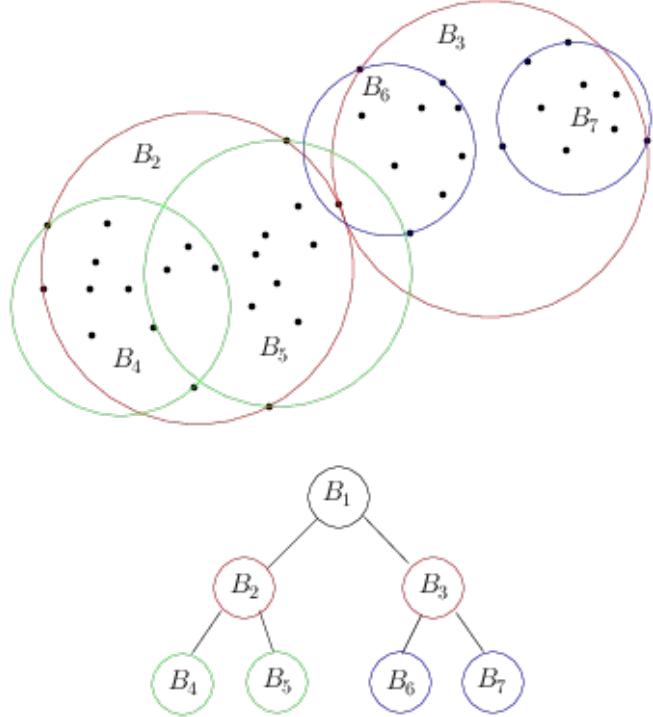
Oh no, I don't have vectors!

Vantage-point (VP) trees (1991)

Instead of dividing space by a plane, we can divide it by a **sphere** (or nested spheres, recursively). **Sphere** requires only **center** (one of dataset points) and **radius** (which can be estimated in any **metric space**). Radius is selected to split points into equal parts.

```
function Select_vp(S)
    P := Random sample of S;
    best_spread := 0;
    for p ∈ P
        D := Random sample of S;
        mu := Mediand ∈ D d(p, d);
        spread := 2nd-Momentd ∈ D (d(p, d) - mu);
        if spread > best_spread
            best_spread := spread; best_p := p;
    return best_p;
```

```
function Make_vp_tree(S)
    if S = ∅ then return ∅
    new(node);
    node↑.p := Select_vp(S);
    node↑.mu := Medians ∈ S d(p, s);
    L := {s ∈ S - {p} | d(p, s) < mu};
    R := {s ∈ S - {p} | d(p, s) ≥ mu};
    node↑.left := Make_vp_tree(L);
    node↑.right := Make_vp_tree(R)
    return node;
```



SEARCH

Ok, you must be lost...

All those trees recursively split the space into similar size parts

Quad Tree - works in R^2 only. Each node splits space into 4 non equal quadrants.

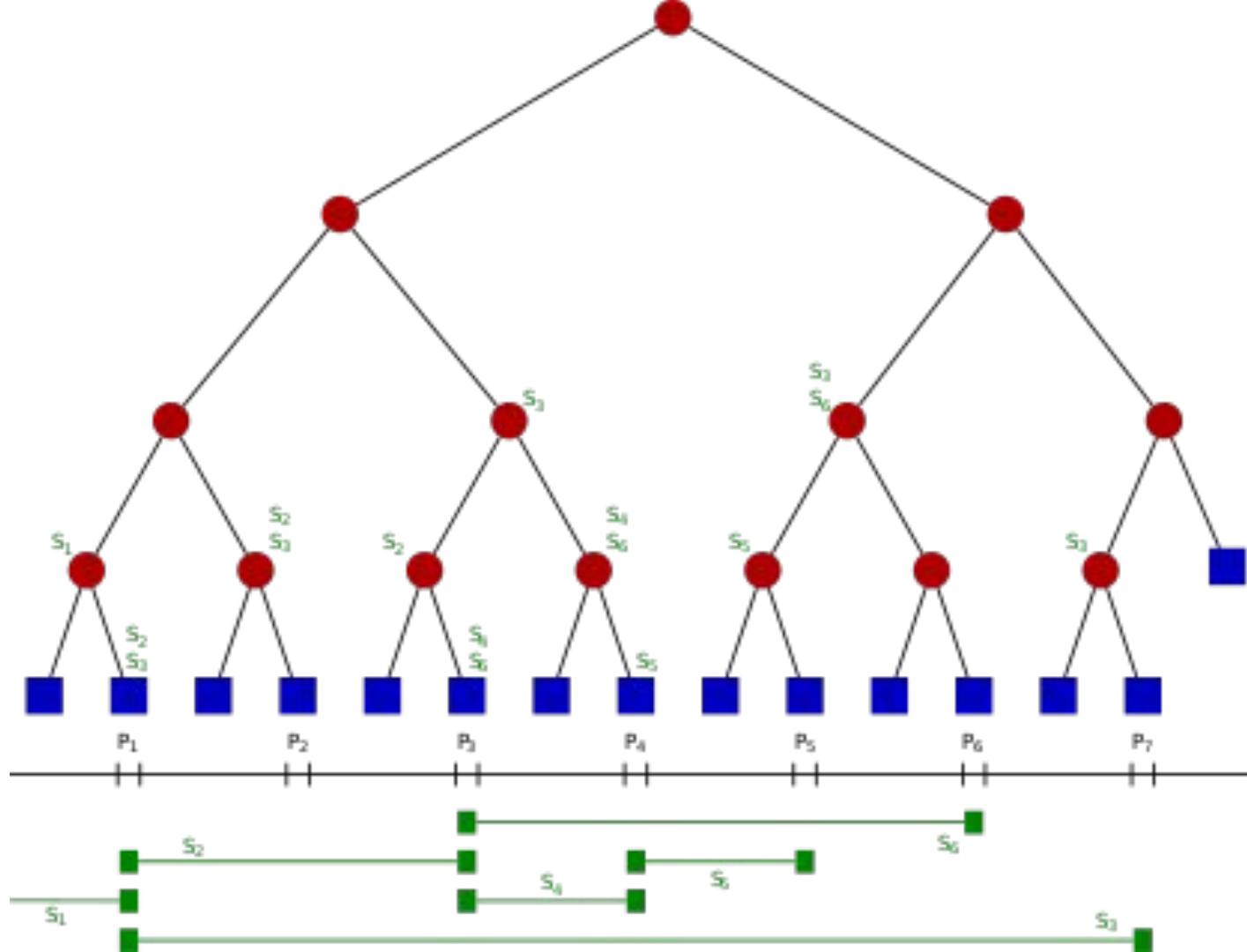
K-d Tree - works in R^K . Each node splits space into 2 equal parts.

Annoy - works in R^K . Instead of sorting and finding median - uses random separating hyperplanes. But compensate with multiple trees

Vantage-point tree - works for any metric space. Instead of hyperplanes uses spheres.

Offtopic: interval and BSP trees.
When object is not a point

Interval tree



Interval tree

Tree that **holds intervals** and allows to search fastly which of them overlap the query (point or interval).

Construction(L):

1. You have a list of intervals L .
2. By X_{center} split all intervals into “left”, “intersecting”, “right” lists.
3. Store “intersecting” in current node in 2 lists (sorted by start and by end).
4. Run Construction(“left”) and Construction(“right”) intervals.

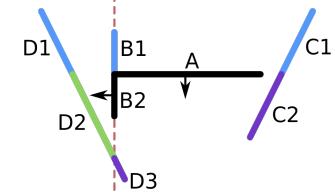
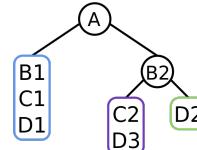
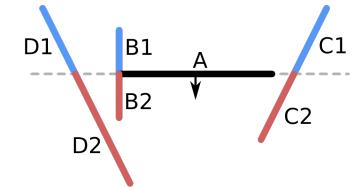
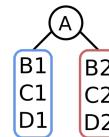
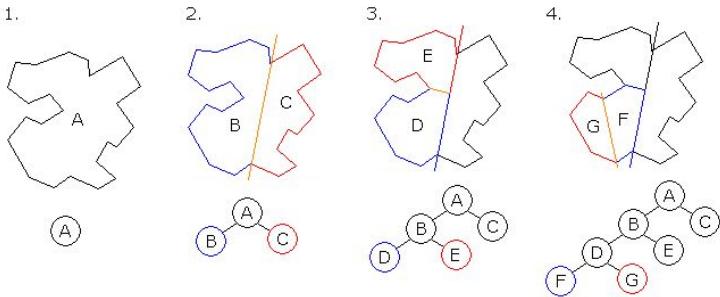
Search(p , node):

1. Compare p to $node.X_{center}$
2. Use sorted list in node to find intersecting
3. Go Search(p , $node.[left|right]$) with respect to [1]

BSP-tree

To store polygons in a list:

- Choose a polygon P from a list L .
- Make a node N , and add P to the list of N .
- For each other polygon Q in the list:
 - If Q is in front of P plane, move Q to the list L_F “**in front of P** ”.
 - If Q is behind P plane, move Q to the list L_B “**behind P** ”.
 - If Q intersects P plane, **split** it into two polygons and move them to the respective lists.
 - If that polygon lies in the plane containing P , add it to the **list of N** .
- Apply this algorithm to L_F and L_B .



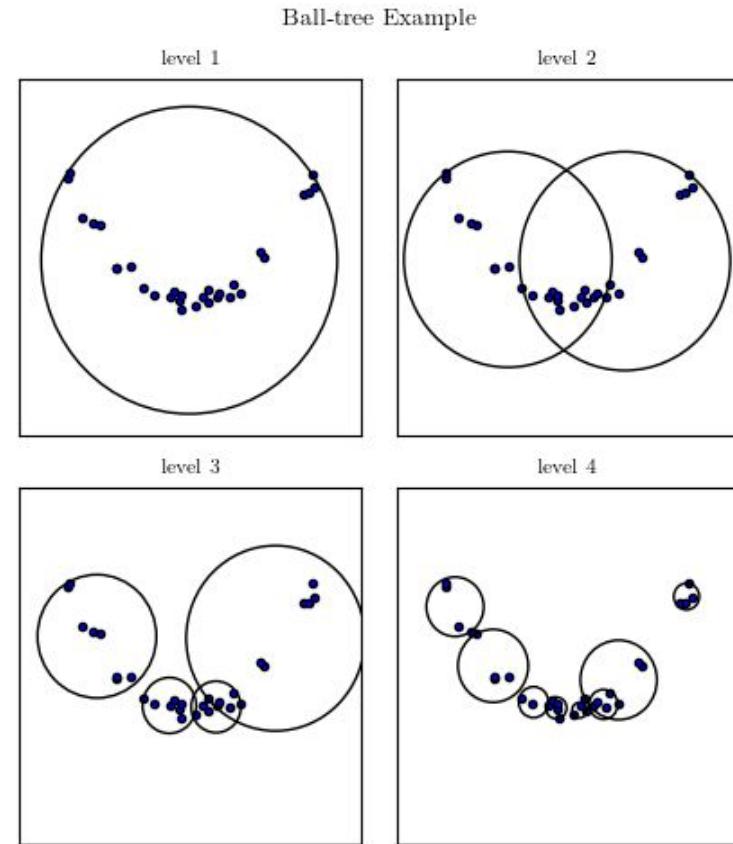
Any other trees left?
Yes, and lots of!

Ball-trees (1989)

1. Select a dimension of biggest variance
2. Split by pivot element (median)
3. Construct ball tree for “lefts” and “rights”.

Branch-and-bound powered:

Algorithm is searching the data structure with a test point t , and has already seen some point p that is closest to t among the points encountered so far, then any ***subtree whose ball is further*** from t than p can be ignored for the rest of the search.



See also

M-trees

R-trees and R*-trees

Octree

...



Full-text indexes
(not the same as before!)



What is a full-text index (in this lecture)?

- ▶ Text index = a data structure built from a given text (string, sequence) that supports certain type of (typically pattern look-up) queries
- ▶ genomic sequences don't have word structure (cf *inverted indexes*) and query size is arbitrary (cf *hash tables*)
- ▶ other examples:
 - ▶ Chinese, Korean languages
 - ▶ agglutinative/inflectional languages (Finnish, German, ...): looking for particles
 - ▶ MIDI files, audio signals, program code, numerical sequences, ...
- ▶ **full-text indexes** allow a search for patterns of *any length* occurring at *any position* of the text



Indexes

- ▶ “Classical” indexes
 - ▶ Suffix trees
 - ▶ "suffix-tree-like" data structures: Directed Acyclic Word Graph (DAWG), position heap
 - ▶ Suffix arrays
- ▶ Succinct (compressed) indexes
 - ▶ Burrows-Wheeler transform and BWT-index

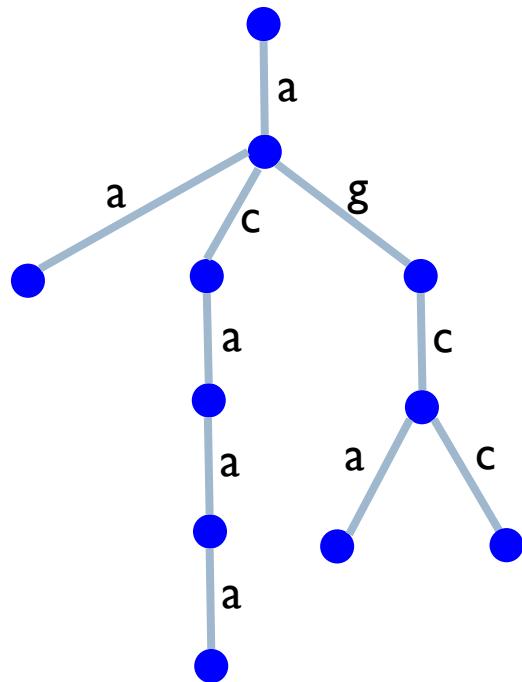
“Ideal” index structure for $T[1..n]$

- ▶ Takes space $O(n)$
- ▶ Can be constructed in time $O(n)$
- ▶ All occurrences of a query pattern P can be reported in time $O(|P|+occ)$

Suffix tree

Trie (aka digital tree)

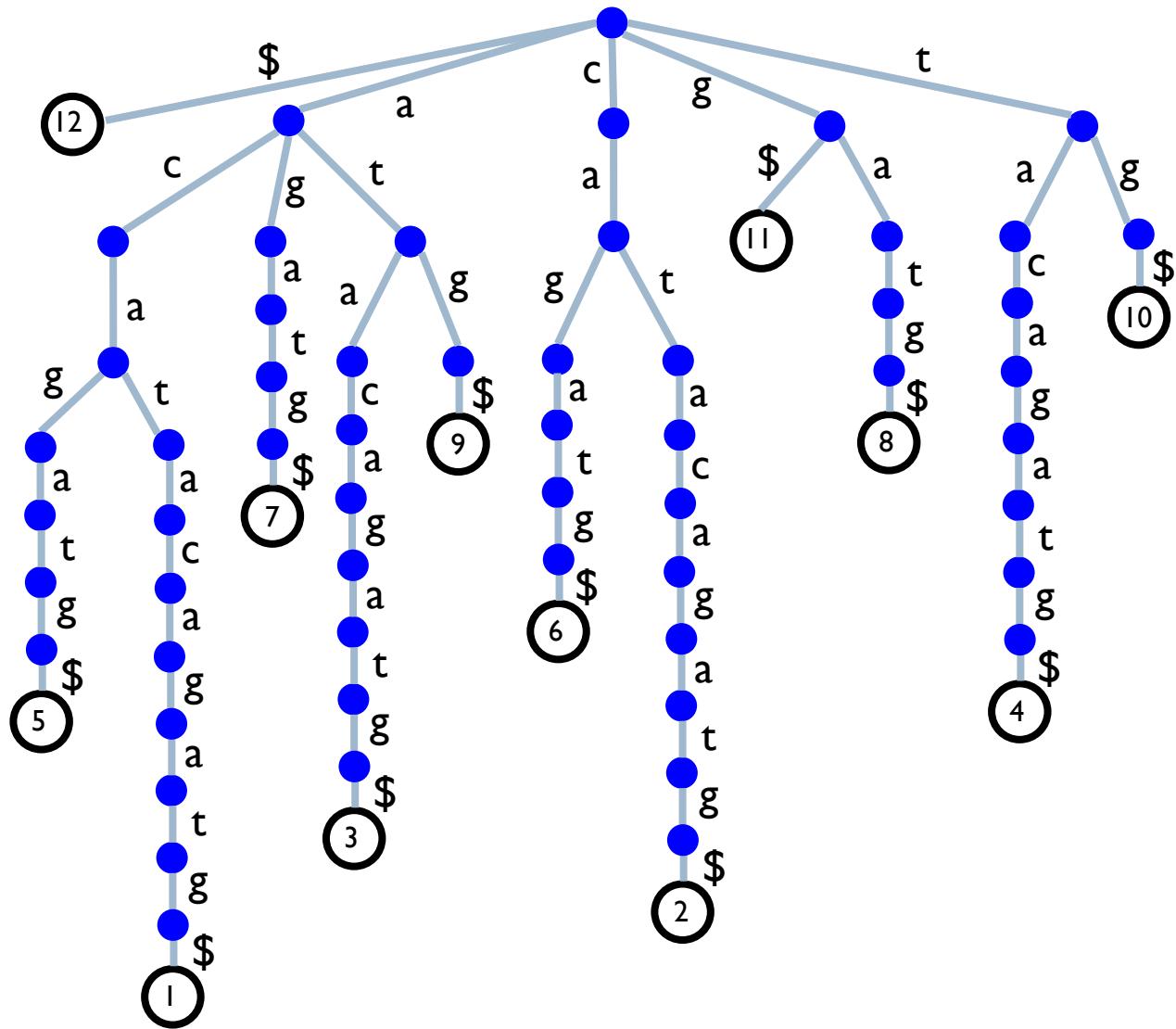
trie for {agca, acaa, agcc, aa}



- every string of the set is “spelled” starting from root
- edges outgoing from a node are labeled by different characters
- trie can be viewed as an automaton recognizing the given set of strings (or all their prefixes)

Suffix trie

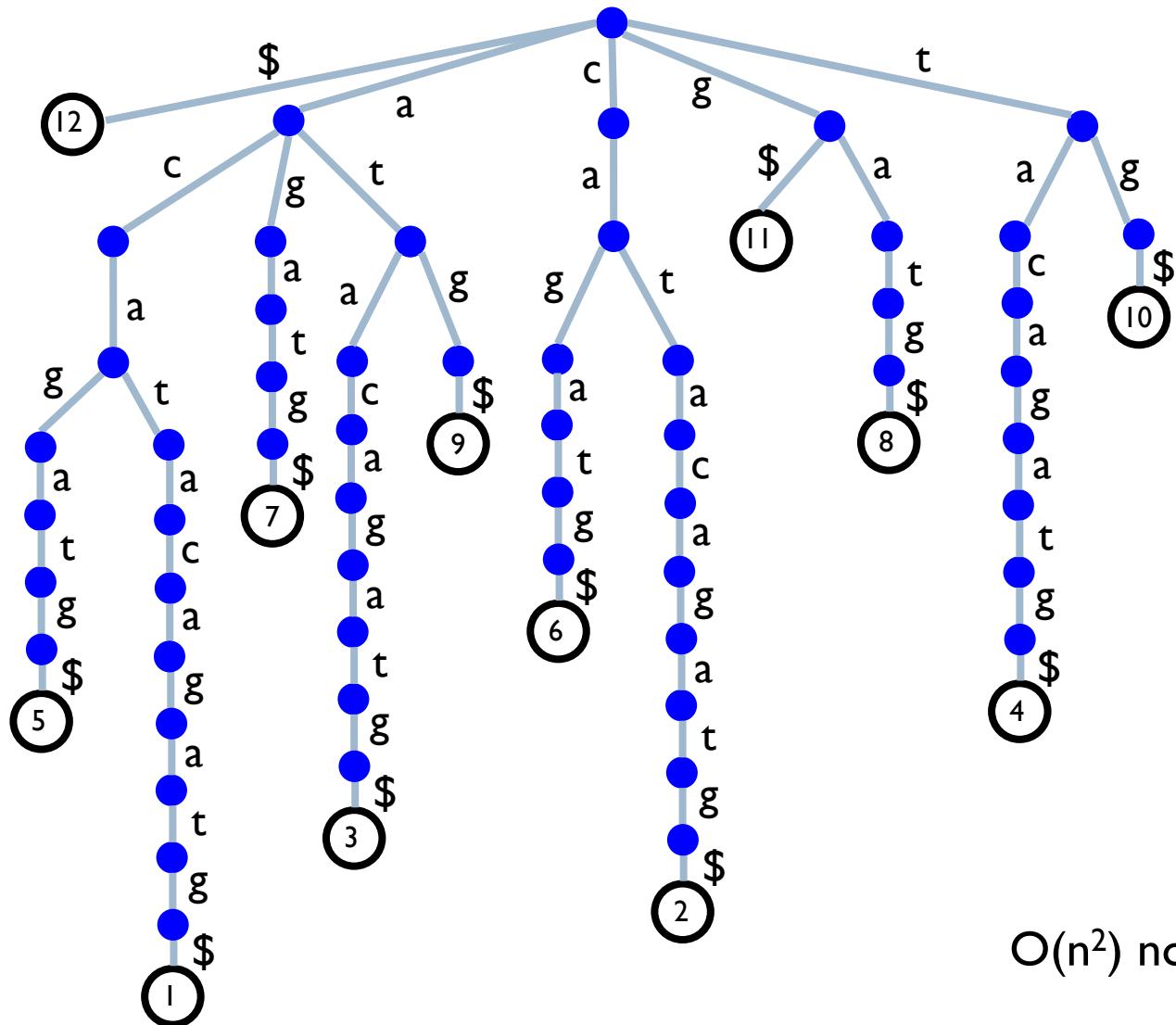
T=acatacacagatg\$



acatacacagatg\$	1
catacacagatg\$	2
atacacagatg\$	3
tacagatg\$	4
acagatg\$	5
cagatg\$	6
agatg\$	7
gatg\$	8
atg\$	9
tg\$	10
g\$	11
\$	12

Suffix trie

$T = \text{acatacagatg\$}$

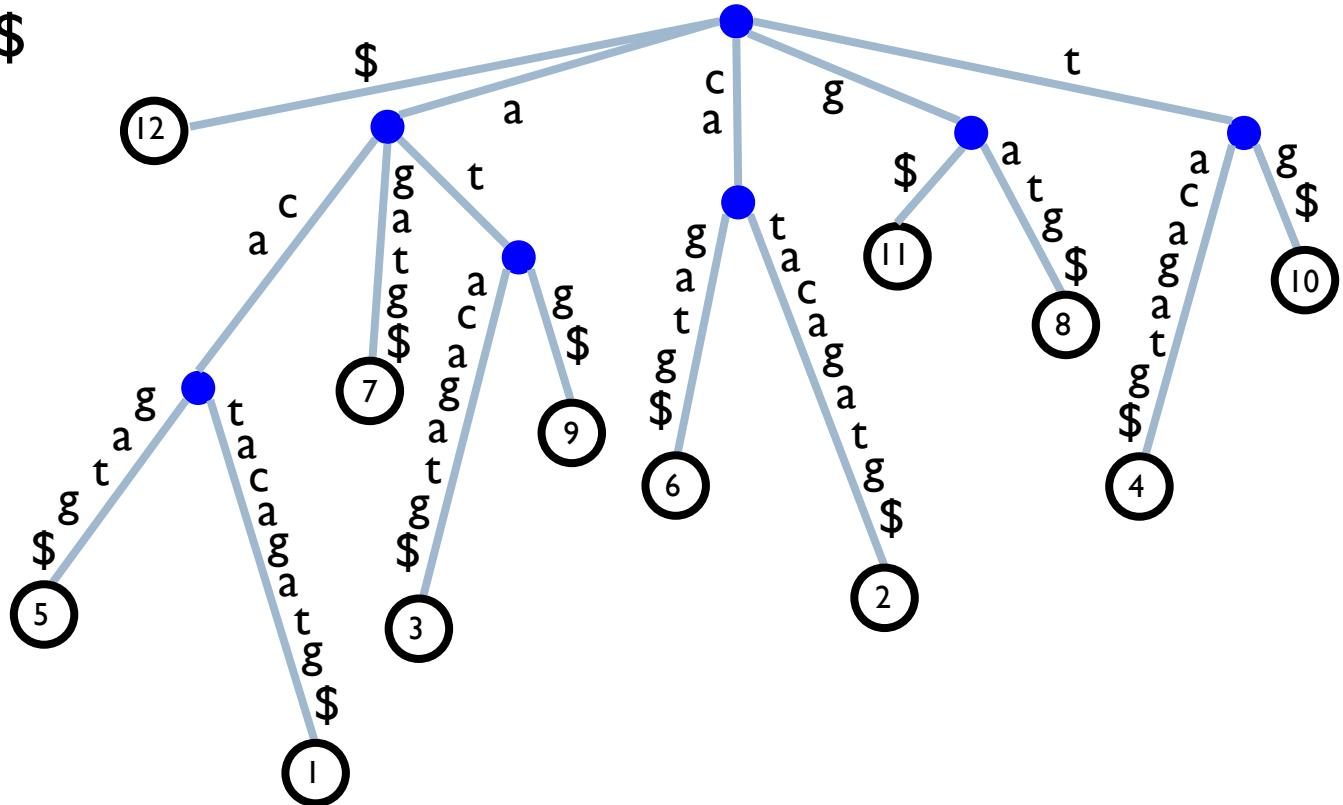


acatacagatg\$	1
catacagatg\$	2
atacagatg\$	3
tacagatg\$	4
acagatg\$	5
cagatg\$	6
agatg\$	7
gatg\$	8
atg\$	9
tg\$	10
g\$	11
\$	12

$O(n^2)$ nodes $a^n b^n$

Suffix tree

$T = \text{acatacagatg\$}$



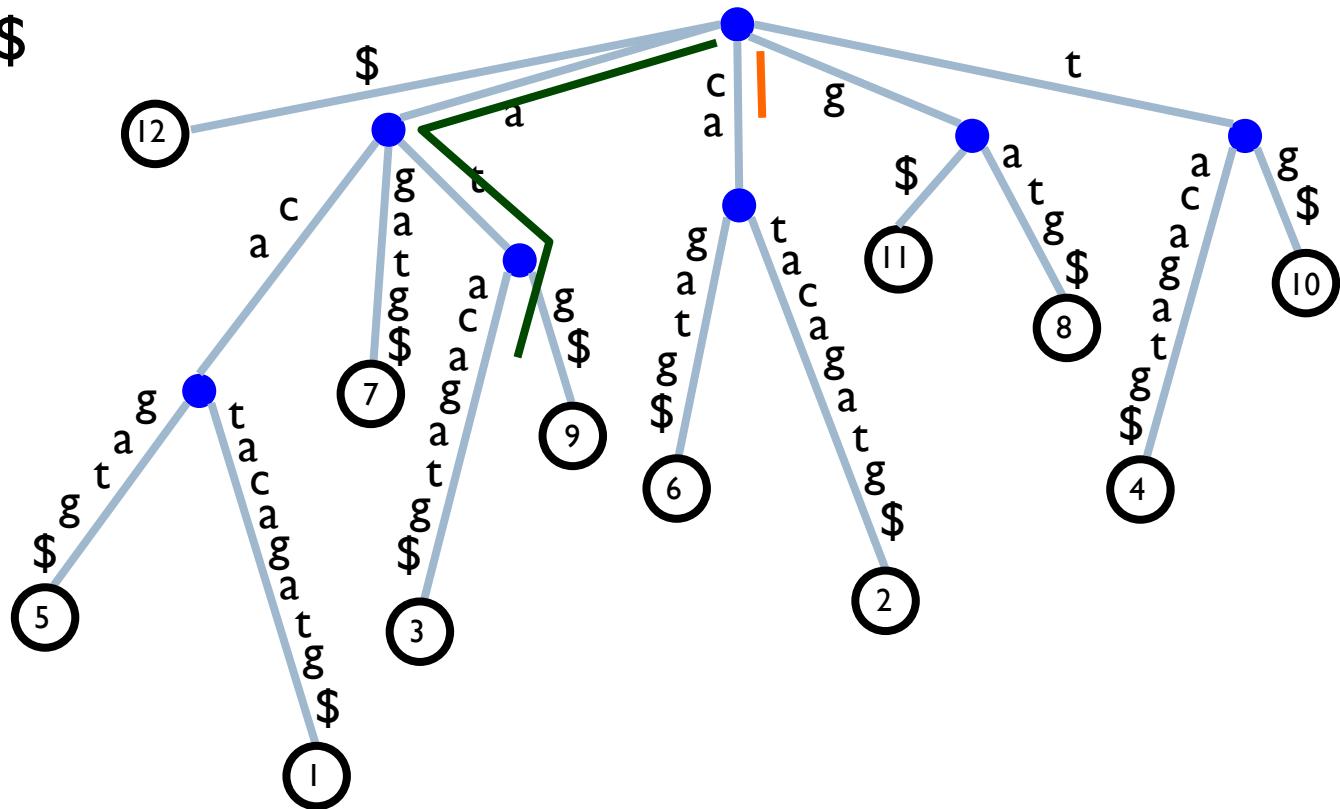
- explicit vs implicit nodes
- an edge label is start and end positions of corresponding substring (rather than substring itself)
- takes space $O(n)$

Exercise

- ▶ Construct the suffix tree for string 010110011100\$

Suffix tree: applications

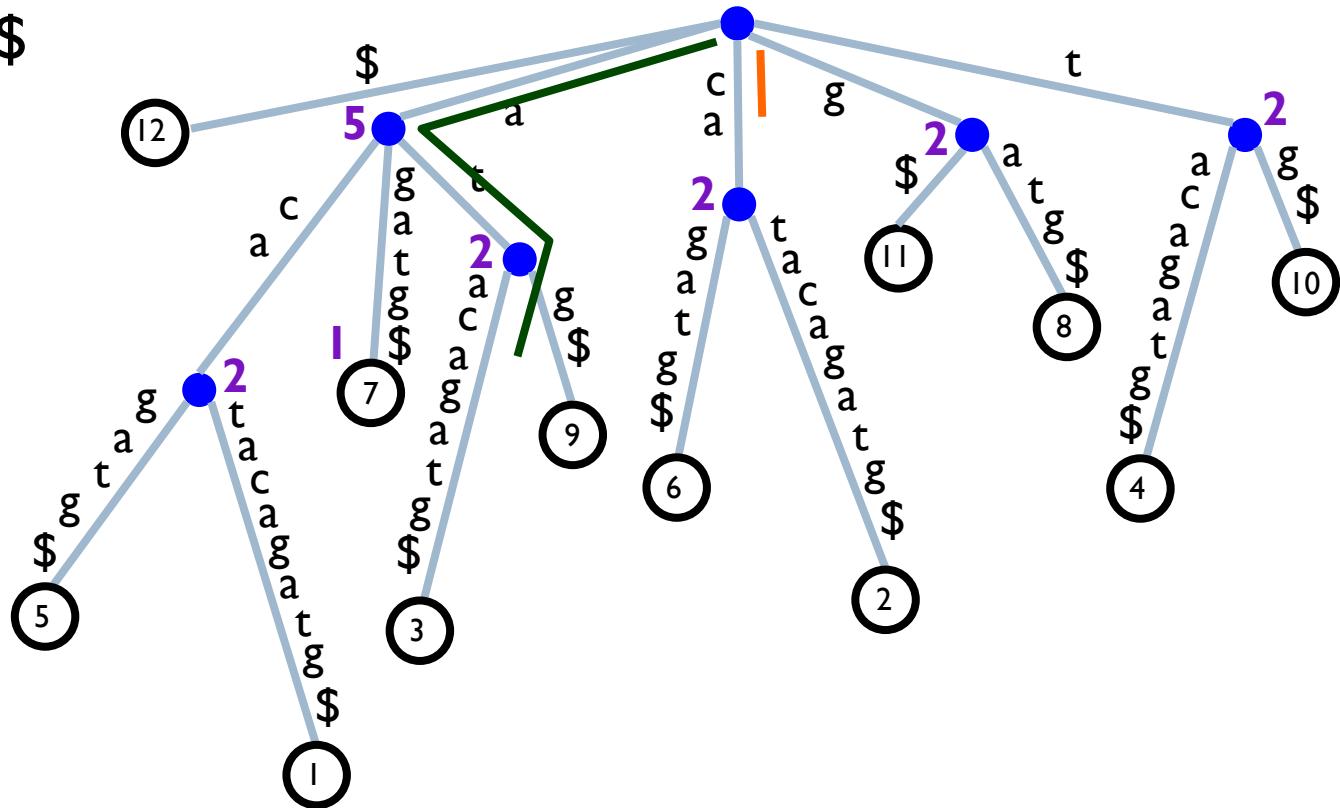
$T = \text{acatacacagatg\$}$



- check if a pattern P occurs in T in time $O(|P|)$. Ex: $P_1 = \text{atac}$ $P_2 = \text{c}$

Suffix tree: applications

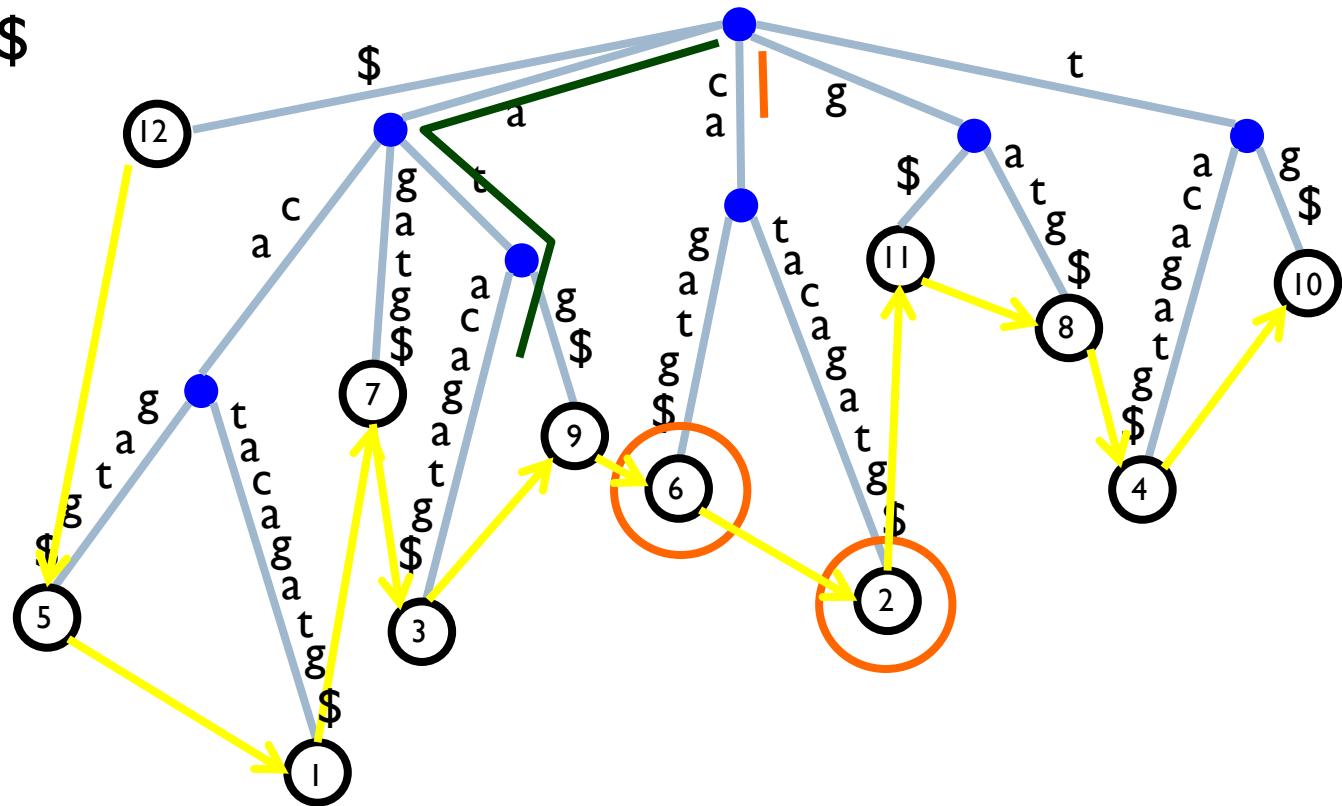
$T = acatacacagatg\$$



- check if a pattern P occurs in T in time $O(|P|)$. Ex: $P_1 = \text{atac}$ $P_2 = \text{c}$
- report the number of occurrences in $O(|P|) \Rightarrow$ preprocess nb of leaves

Suffix tree: applications

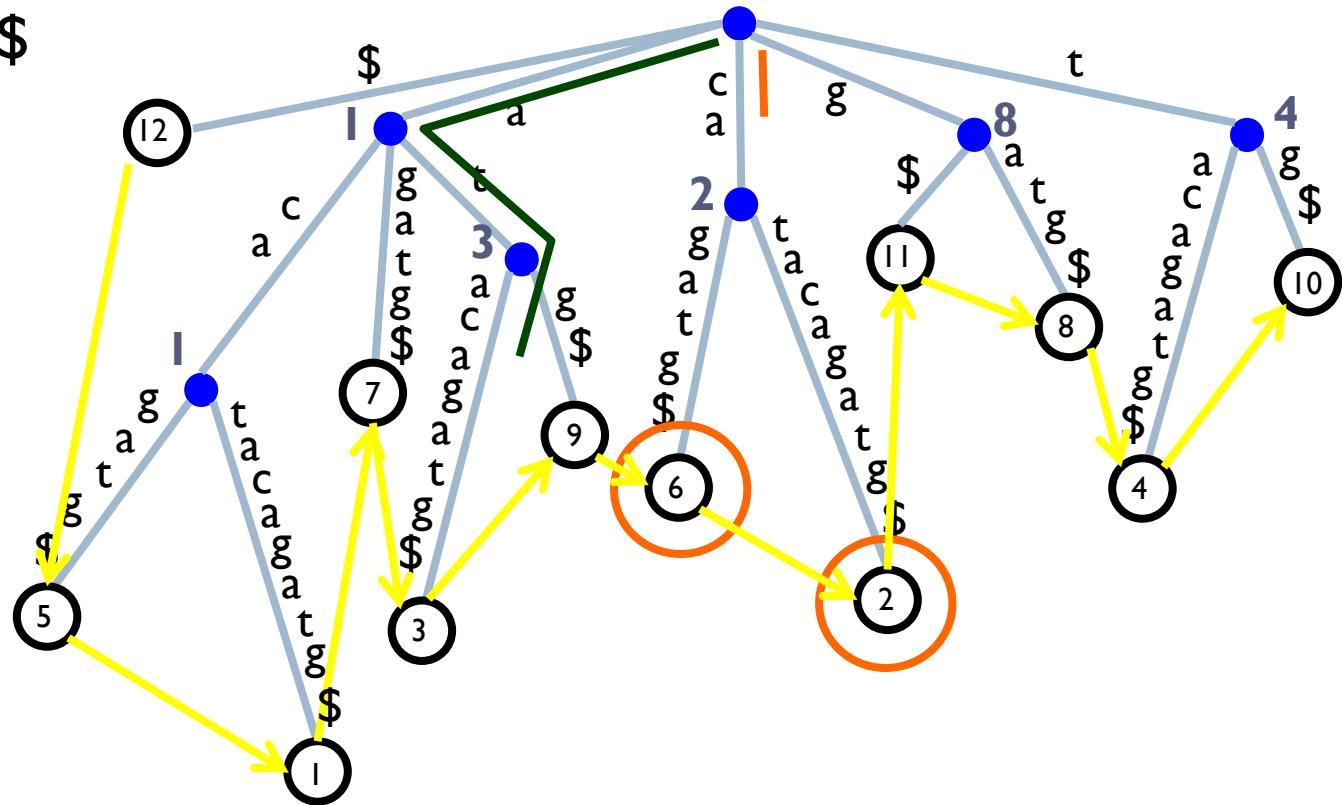
$T = \text{acatacagatg\$}$



- check if a pattern P occurs in T in time $O(|P|)$. Ex: $P_1 = \text{atac}$ $P_2 = \text{c}$
- report the number of occurrences in $O(|P|) \Rightarrow$ preprocess nb of leaves
- report all occurrences in $O(|P| + \text{occ}) \Rightarrow$ chain leaves and preprocess leftmost and rightmost leaves

Suffix tree: applications

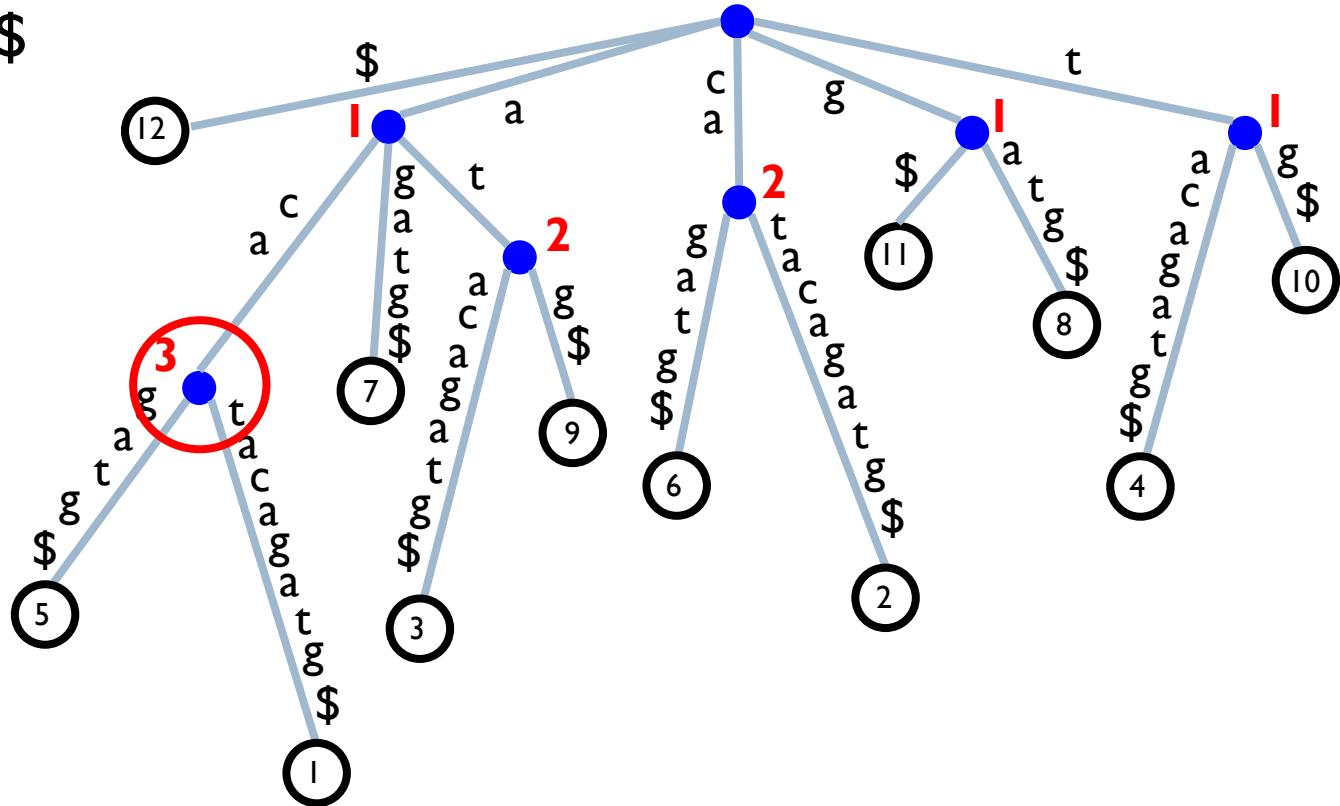
$T = acatacagatg\$\$$



- check if a pattern P occurs in T in time $O(|P|)$. Ex: $P_1 = \text{atac}$ $P_2 = \text{c}$
- report the number of occurrences in $O(|P|) \Rightarrow$ preprocess nb of leaves
- report all occurrences in $O(|P| + \text{occ}) \Rightarrow$ chain leaves and preprocess leftmost and rightmost leaves
- report the first (leftmost) occurrence in $O(|P|) \Rightarrow$ preprocess minimal leaf label

Suffix tree: applications

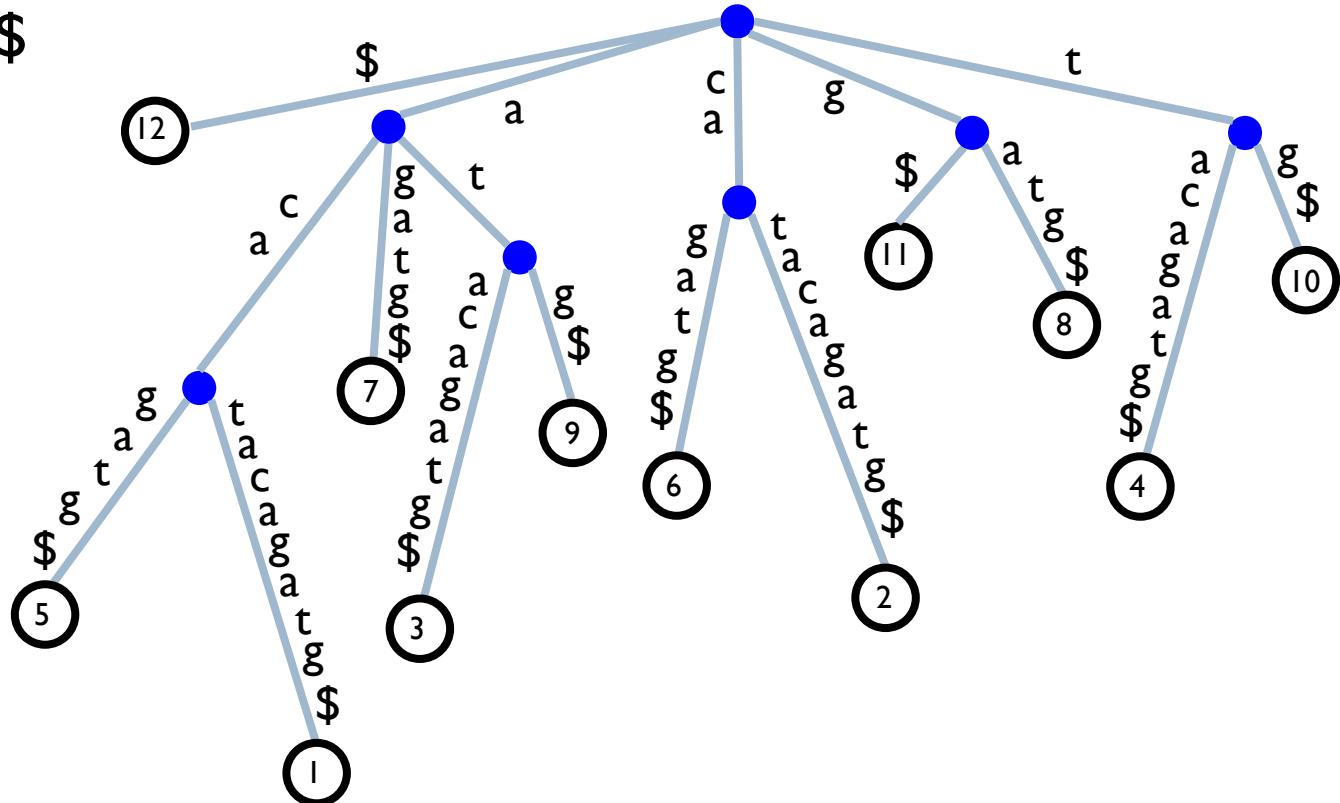
$T = acatacacagatg\$\$$



- longest repeated substring \Rightarrow deepest (w.r.t. string depth) internal node in the suffix tree (**aca**)

Suffix tree: applications

$T = acatacacatg\$$



- longest extension queries: given two positions i, j , output the length of the longest common substring starting at i, j
- reduces to lowest common ancestor (lca) queries
- lca queries can be answered in $O(1)$ time after linear-time preprocessing of the tree [Harel,Tarjan 84], [**Bender,Farach-Colton 00**]

Exercise

- ▶ Compute maximal repeats

A *maximal repeat* of T is a substring α that has (at least) two occurrences followed by distinct letters and preceded by distinct letters

Suffix tree: some history

- ▶ Suffix tree can be constructed in time $O(n)$
- ▶ Weiner 1973: right-to-left construction

LINEAR PATTERN MATCHING ALGORITHMS

Peter Weiner

The Rand Corporation, Santa Monica, California *

Abstract

In 1970, Knuth, Pratt, and Morris [1] showed how to do basic pattern matching in linear time. Related problems, such as those discussed in [4], have previously been solved by efficient but sub-optimal algorithms. In this paper, we introduce an interesting data structure called a bi-tree. A linear time algorithm for obtaining a compacted version of a bi-tree associated with a given string is presented. With this construction as the basic tool, we indicate how to solve several pattern matching problems, including some from [4], in linear time.

I. Introduction

In 1970, Knuth, Morris, and Pratt [1-2] showed how to match a given pattern into another given string in time proportional to the sum of the lengths of the pattern and string. Their algorithm was derived from a result of Cook [3] that the 2-way deterministic pushdown languages are recognizable on a random access machine in time $O(n)$. Since 1970, attention has been given to several related problems in pattern matching [4-6], but the algorithms developed in these investigations usually run in time which is slightly worse than linear, for example $O(n \log n)$. It is of considerable interest to either establish that there exists a non-linear lower bound on the run time of all algorithms which solve a given pattern matching problem, or to exhibit an algorithm whose run time is of $O(n)$.

In the following sections, we introduce an interesting data structure, called a bi-tree, and show how an efficient calculation of a bi-tree can be applied to

for giving a formal definition of a bi-tree, we review basic definitions and terminology concerning t-ary trees. (See Knuth [7] for further details.)

A t -ary tree T over $\Sigma = \{\sigma_1, \dots, \sigma_t\}$ is a set of nodes N which is either empty or consists of a root, $n_0 \in N$, and t ordered, disjoint t-ary trees.

Clearly, every node $n \in N$ is the root of some t-ary tree T^i which itself consists of n_i and t ordered, disjoint t-ary trees, $T_1^i, T_2^i, \dots, T_t^i$. We call the tree T_j^i a sub-tree of T^i , n_j^i a full subtree of T^i if T_j^i are considered to be sub-trees of T^i . It is natural to associate with a tree T a successor function

$$S: N \times \Sigma \rightarrow (N - \{n_0\}) \cup \{\text{NIL}\}$$

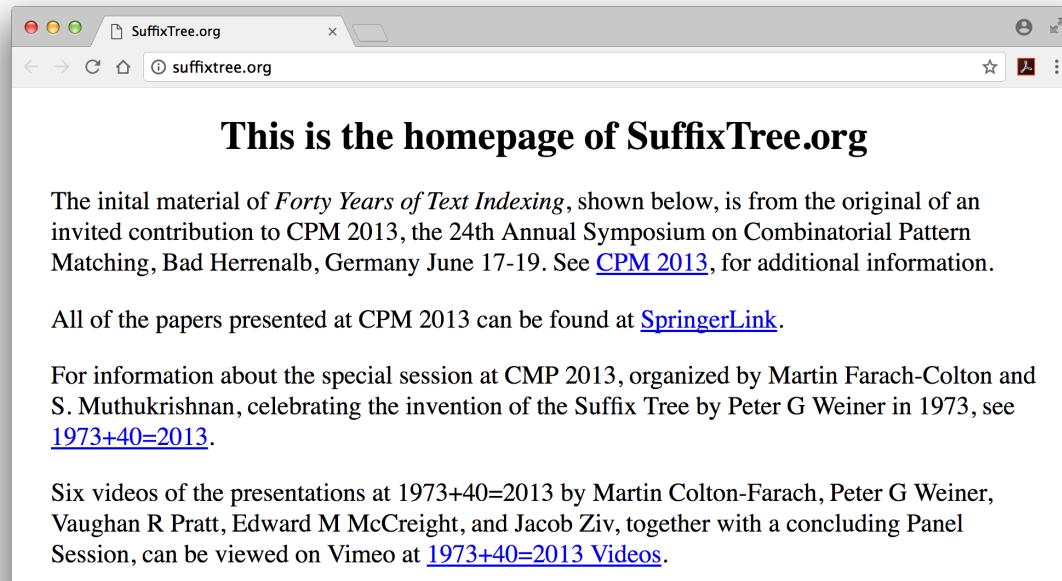
defined for all $n_i \in N$ and $\sigma_j \in \Sigma$ by

$$n^i \quad \text{the root of } T^i \text{ if } T^i \text{ is non-empty}$$

Algorithm of the Year 1973
Donald Knuth

Suffix tree: some history

- ▶ Suffix tree can be constructed in time $O(n)$
 - ▶ Weiner 1973: right-to-left construction
 - ▶ McCreight 1976: left-to-right
 - ▶ Ukkonen 1995: left-to-right and *online*
 - ▶ Farach 1997: for integer alphabets



This is the homepage of SuffixTree.org

The initial material of *Forty Years of Text Indexing*, shown below, is from the original of an invited contribution to CPM 2013, the 24th Annual Symposium on Combinatorial Pattern Matching, Bad Herrenalb, Germany June 17-19. See [CPM 2013](#), for additional information.

All of the papers presented at CPM 2013 can be found at [SpringerLink](#).

For information about the special session at CPM 2013, organized by Martin Farach-Colton and S. Muthukrishnan, celebrating the invention of the Suffix Tree by Peter G Weiner in 1973, see [1973+40=2013](#).

Six videos of the presentations at 1973+40=2013 by Martin Colton-Farach, Peter G Weiner, Vaughan R Pratt, Edward M McCreight, and Jacob Ziv, together with a concluding Panel Session, can be viewed on Vimeo at [1973+40=2013 Videos](#).

review articles

Tracing the first four decades in the life
of suffix trees, their many incarnations,
and their applications.

BY ALBERTO APOSTOLICO, MAXIME CROCHEMORE,
MARTIN FARACH-COLTON, ZVI GALIL, AND S. MUTHUKRISHNAN

40 Years of Suffix Trees

WHEN WILLIAM LEGRAND finally decrypted the string, it did not seem to make much more sense than it did before.

53#(305))6*,48264‡,4z);806",48†8P60))85;1‡
(;‡#8183(88)5*,46;88*96?;8)‡;(485);5*2‡;‡
(4956*2§(5*N4)8P8";4069285);6‡8)4‡;1‡(9;48081;8;
8‡;1;4885;4)85†528806*81(ddag9;48;8§;4(‡34;
48)4‡;161;‡;188;‡;‡;

The decoded message read: "A good glass in the bishop's hostel in the devil's seat forty-one degrees and thirteen minutes northeast and by north main branch seventh lime east side shroo from the left eye of the death's-head a bee line from the tree through the shot fifty feet out." But at least it did sound more like natural language, and eventually guided the main character of Edgar Allan Poe's "The Gold-Bug"³⁶ to discover the treasure he had been after. Legrand solved a substitution cipher using symbol frequencies.

He first looked for the most frequent symbol and changed it into the most frequent letter of English, then similarly inferred the most frequent word, then punctuation marks, and so forth. Before and after 1843, the natural impulse when faced with some mysterious message has been to count frequencies of individual tokens or subassemblies in search of a clue. This is a little bit of a naive and fascinating subject for this kind of scrutiny but have been biosequences. As soon as some such sequences became available, statistical analysts tried to efficiently analyze them or characters relevant biological functions. With the early examples of whole genomes emerging in the mid-1990s, it seemed natural to count the occurrences of each k-mers of length 1, 2, and so on, up to any desired length, looking for statistical characteristics of coding regions, promoter regions, among others.

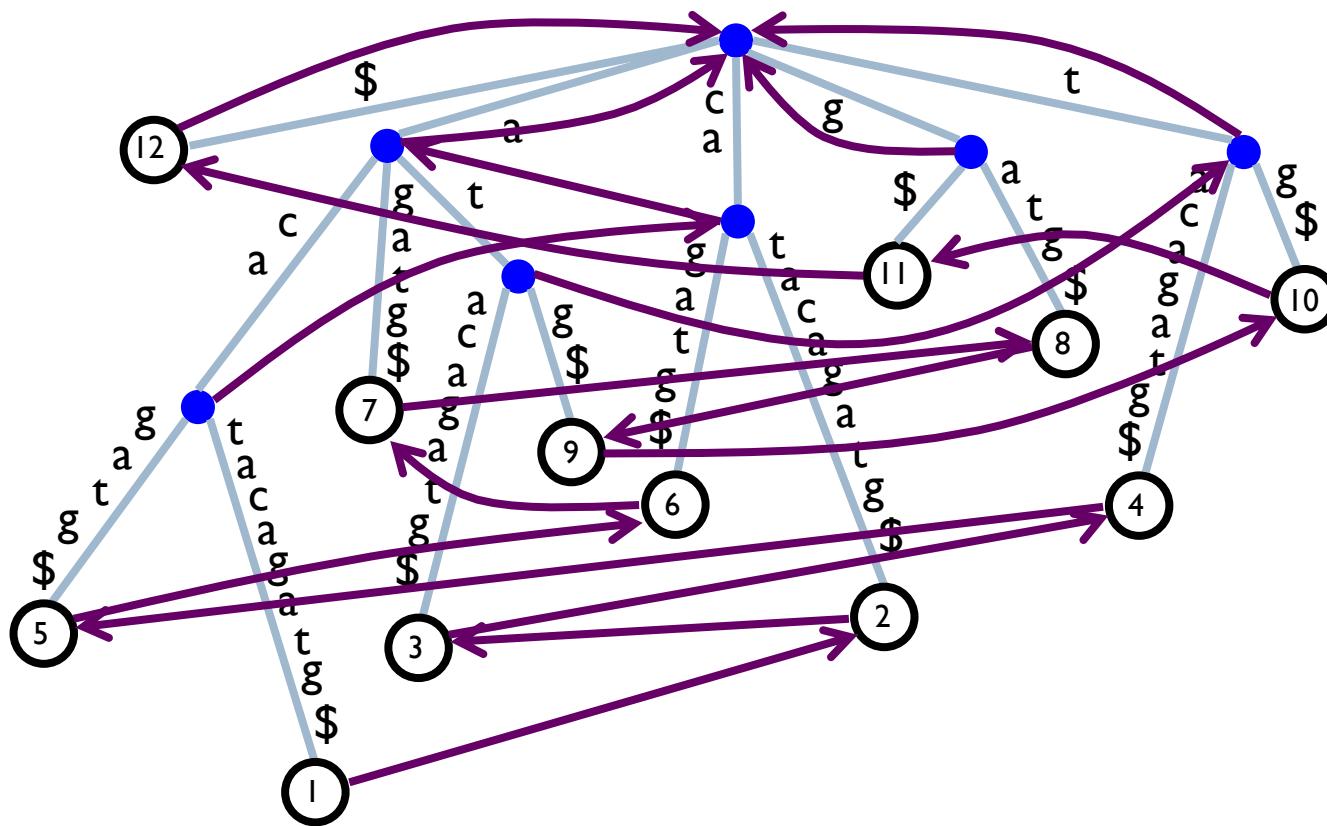
This article is not about cryptology, nor about string matching and its variants, and the many surprising and useful features it carries. Among these is the fact that, to set up a statistical table of occurrences for all substrings of a given portion of any string, it only takes time and space linear in the length of the text string. While nobody would be so foolish as to solve the problem by first generating all exponentially many possible strings and then counting their occurrences one by one, a test string may still contain $\Theta(n^k)$ distinct substrings, so that tabulating all of them in linear space, never mind linear time, already seems puzzling.

We dedicate this article to our friend and colleague, Alberto Apostolico (1948–2015), who passed away on July 20. He was a major figure in the development of algorithms on strings.

Suffix tree augmented with suffix links

McCreight and Ukkonen use *suffix links*

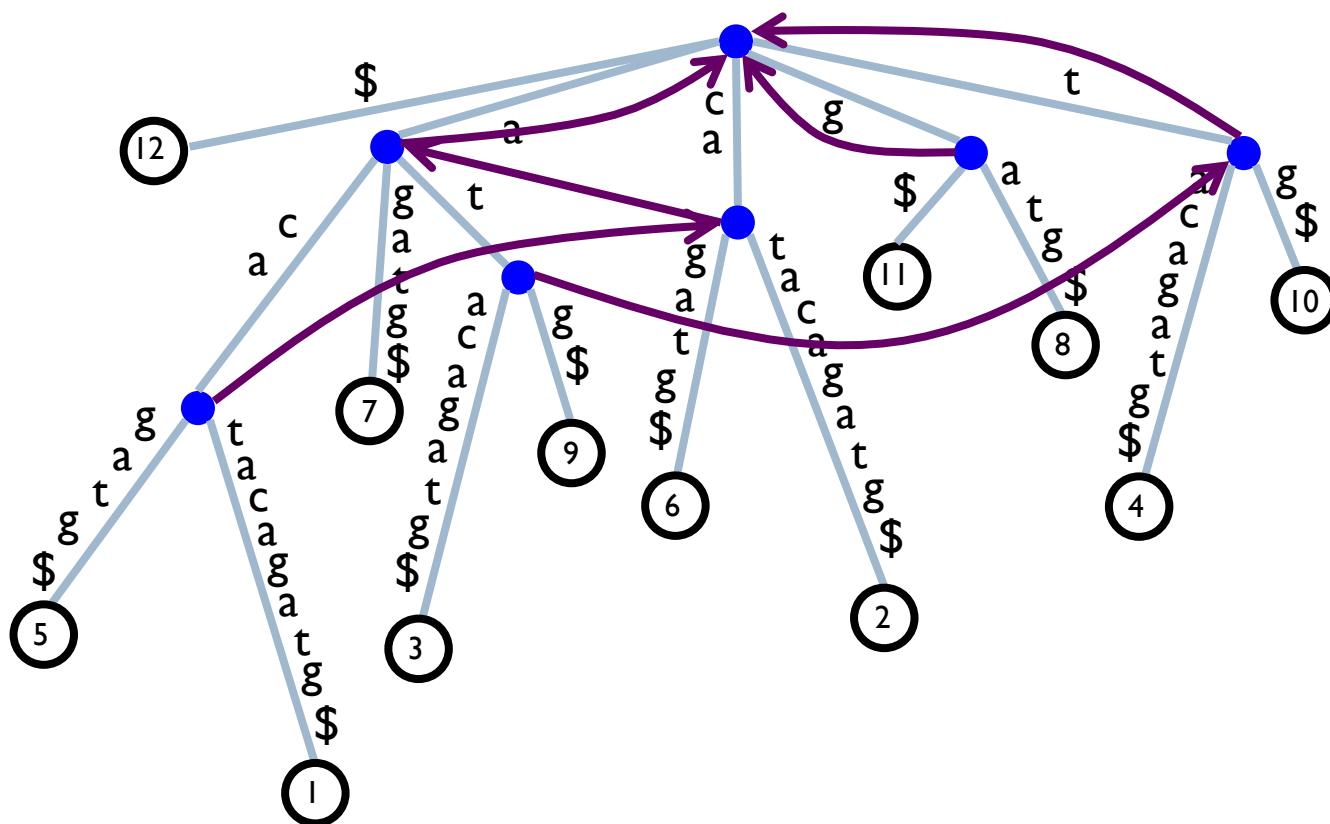
$\text{suf-link}(\bar{a}u) = \bar{u}$ for explicit nodes $\bar{a}u$



Suffix tree augmented with suffix links

McCreight and Ukkonen use *suffix links*

$\text{suf-link}(\bar{a}u) = \bar{u}$ for explicit nodes $\bar{a}u$



Lempel-Ziv encoding

- ▶ used in compression algorithms (compress, gzip, gif, ...)

abaabaaaabababaabb...

a b a abaa aba baba ab b...

a b (1,1) (1,4) (1,3) (9,4) (1,2) b

- ▶ **Definition:** $T = f_1 \dots f_{i-1} f_i \dots$, where f_i is defined by
 - ▶ if the letter a following $f_1 \dots f_{i-1}$ does not occur earlier, then $f_i = a$
 - ▶ otherwise f_i is the longest substring that occurs earlier (with possible overlap)

Lempel-Ziv encoding (cont)

- ▶ used in compression algorithms (compress, gzip, gif, ...)

abaabaaaabababaabb...

a b a abaa aba baba ab b...

a b (1,1) (1,4) (1,3) (9,4) (1,2) b

- ▶ compute LZ-encoding (offline) with suffix tree:
 - ▶ build suffix tree for T
 - ▶ annotate internal nodes by smallest position of a descendant leaf
 - ▶ to compute a new phrase (i,l) starting at position p , match $T[p..]$ against the suffix tree as long as the min position is smaller than p

Modifications and extensions of suffix tree

- ▶ *Generalized suffix tree*: suffix tree for several strings (dictionaries)
- ▶ *Sparse suffix tree*: suffix tree for a fraction of suffixes
- ▶ *Affix trees*: suffix tree for a string and its inverse
- ▶ *Order-preserving suffix trees*: suffix trees for integer sequences allowing “order-matching search”
- ▶ ...

String algorithms

“Stringology”

Exact string matching

Pattern search in a sequence

- ▶ $T[1..n]$ sequence (text, string)
- ▶ $P[1..m]$ pattern
- ▶ *Problem:* locate all occurrences of P in T (*variants:* check if P occurs in T , count the number of occurrences)

Naïve algorithm: $O(n \cdot m)$

$T = aaaaaa...aa = a^n$

$P = aa...ab = a^{m-1}b$

Pattern search in a sequence

T = a b a b a a a b a b a a b a ...

P = a b a a b

Pattern search in a sequence



$T = a \ b \ a \ b \ a \ a \ a \ b \ a \ b \ a \ a \ b \ a \ a \ b \ a \ ...$

$P = a \ b \ a \ a \ b$

Pattern search in a sequence



$T = a \ b \ a \ b \ a \ a \ a \ b \ a \ b \ a \ b \ a \ a \ b \ a \ ...$

$P = a \ b \ a \ a \ b$

Pattern search in a sequence



T = a b a b a a a b a b a a b a ...

P = a b a a b

Pattern search in a sequence



T = a b a **b** a a a b a b a a b a ...

P = a b a a b

Pattern search in a sequence



$T = a \ b \ a \ b \ a \ a \ a \ b \ a \ b \ a \ b \ a \ a \ b \ a \ ...$

$P = a \ b \ a \ a \ b$

a b a a b

Pattern search in a sequence



$T = a \ b \ a \ b \ a \ a \ a \ b \ a \ b \ a \ a \ b \ a \ a \ b \ a \ ...$

$P = a \ b \ a \ a \ b$

a b a a b

Pattern search in a sequence



T = a b a b a a a b a b a a b a ...

P = a b a a b

a b a a b

Pattern search in a sequence



T = a b a b a a **a** b a b a a b a ...

P = a **b** a a **b**
a b a a b

Pattern search in a sequence



$T = a \ b \ a \ b \ a \ a \ a \ b \ a \ b \ a \ a \ b \ a \ ...$

$P = a \ b \ a \ a \ b$

a b a a b

a b a a b

Pattern search in a sequence



T = a b a b a a a b a b a a b a ...

P = a b a a b

a b a a b

a b a a b

Pattern search in a sequence



T = a b a b a a a b a b a a b a ...

P = a b a a b

a b a a b

a b a a b

Pattern search in a sequence



T = a b a b a a a b a **b** a a b a ...

P = a **b** a a b

a b a a b

a b a a b

Pattern search in a sequence



T = a b a b a a a b a b a a b a ...

P = a b a a b

a b a a b

a b a a b

a b a a b

Pattern search in a sequence



T = a b a b a a a b a b a a b a ...

P = a b a a b

a b a a b

a b a a b

a b a a b

Pattern search in a sequence

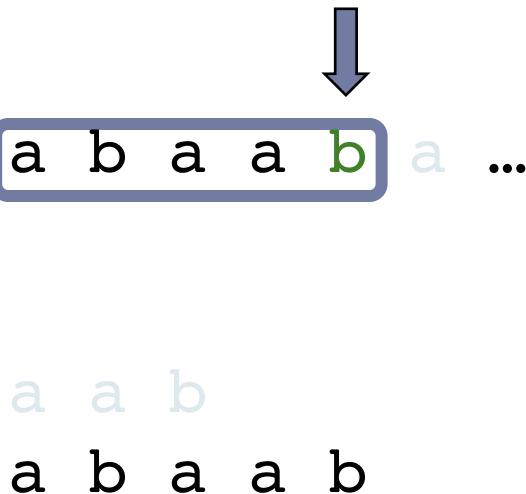
↓

$T = a \ b \ a \ b \ a \ a \ a \ b \ a \ b \ a \ a \ b \ a \ ...$

$P = \begin{matrix} a & b & a & a & b \\ & a & b & a & a & b \\ & & a & b & a & a & b \\ & & & a & b & a & a & b \\ & & & & a & b & a & a & b \end{matrix}$

Pattern search in a sequence

T = a b a b a a a b a b a a b a ...
P = a b a a b
 a b a a b
 a b a a b
 a b a a b



Pattern search in a sequence

T = a b a b a a a b a b a a b a ...
P = a b a a b
 a b a a b
 a b a a b
 a b a a b
 a b a a b



Pattern search in a sequence

T = a b a b a a a b a b a a b a ...

P = a b a a b

a b a a b

a b a a b

a b a a b

a b a a b



Pattern search in a sequence

Observations:

- at each step we move on by one letter in the text (i.e. each letter of the text is compared once)
- the state of the search is defined by a position in the text and a position in the pattern
- the shift of the pattern is defined depending on the failure position in the pattern and the text letter produced the failure

T =	...	*	*	*	*	*	*	*	*	*	...
P =	*	*	*	*	*	*	=	=	=	≠	
	*	*	*	*	*						

Pattern search in a sequence

Observations:

- at each step we move on by one letter in the text (i.e. each letter of the text is compared once)
- the state of the search is defined by a position in the text and a position in the pattern
- the shift of the pattern is defined depending on the failure position in the pattern and the text letter produced the failure

T =	...	*	*	*	*	*	*	*	*	*	...
P =		=	=	=	≠						
		*	*	*	*	*	*				

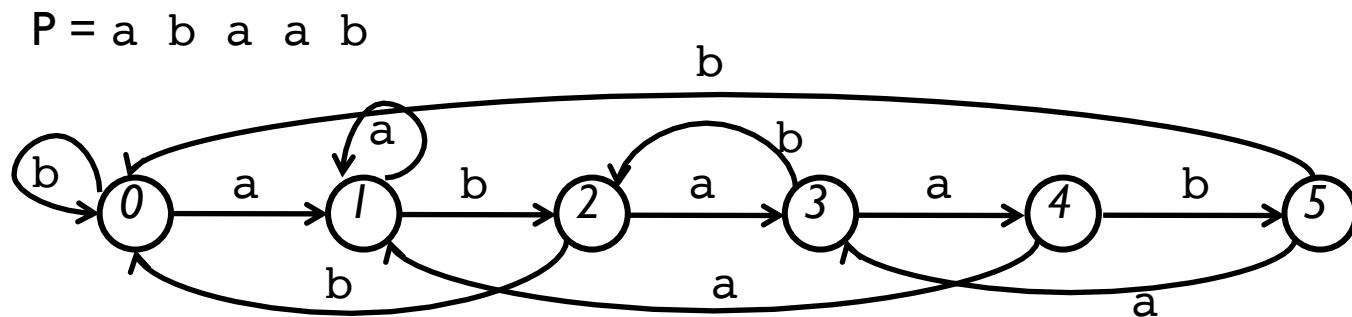
⇒ Finite automaton!

Pattern search in a sequence

$P \Rightarrow$ finite automaton

state \equiv prefix of P (position) $q \equiv P_q = P[1..q]$

$$\sigma(q, a) = \begin{cases} q+1, & \text{si } a = P[q+1] \\ \max\{ i \mid P_i \text{ is a suffix of } P_q a \}, & \text{otherwise} \end{cases}$$



Invariant: P_q is the longest prefix of P which is a suffix of the prefix of T read so far

Pattern search in a sequence

Resulting algorithm:

1. *Pre-processing*: compute the automaton $O(m \cdot |A|)$ (time and space)
2. *Search*: run the automaton on the text $O(n \cdot \log(|A|))$

Goal: design an algorithm that does not depend on the alphabet size

Knuth-Morris-Pratt algorithm

T = ... a b a b *	*	*	*	*	...
= = = ≠					
P =	b a b a b				

What are possible shifts of the pattern?

Failure function:

$f(q) = \max\{ k \mid k < q \text{ et } P_k = P[1..k] \text{ is a suffix of } P_q \}$

$P = a b a b a c a$

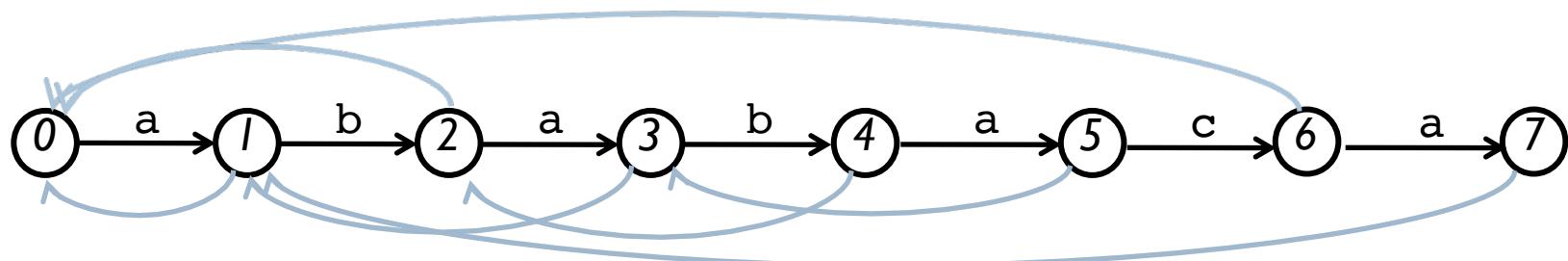
q	0	1	2	3	4	5	6	7
$f(q)$	-1	0	0	1	2	3	0	1

Knuth-Morris-Pratt algorithm

$P = a \ b \ a \ b \ a \ c \ a$

q	0	1	2	3	4	5	6	7
$f(q)$	-1	0	0	1	2	3	0	1

Knuth-Morris-Pratt "automaton"



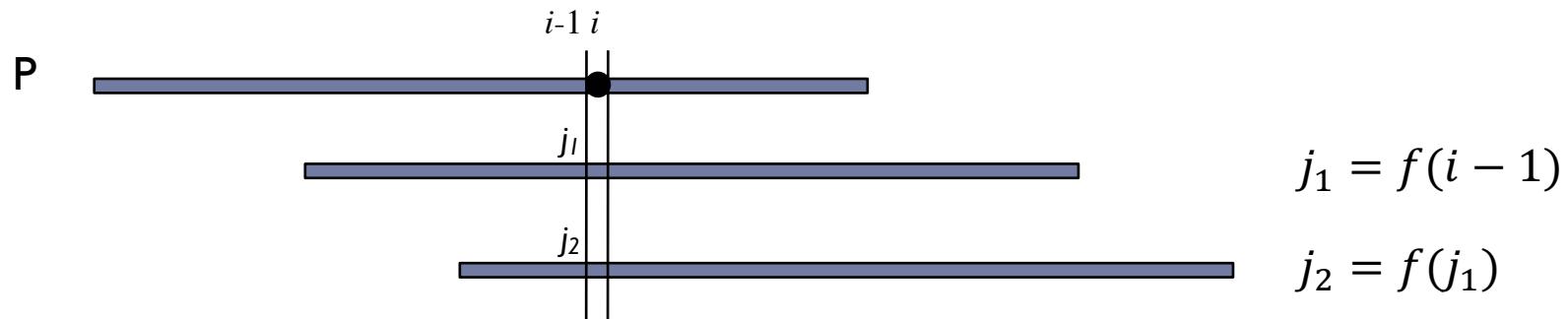
Knuth-Morris-Pratt algorithm

Once failure function f is computed ...

```
KMP(T[1..n],f)
    j=0 /* pointer in P */
    for i=1 to n do
        while j≥0 and P[j+1]≠T[i] do
            j=f(j) endwhile
        j=j+1
        if j==m then
            output(occurrence of P at position (i-m))
            j=f(j) endif
    endfor
```

Knuth-Morris-Pratt algorithm

How to compute the failure function



Shift the pattern against itself until $P[j_q+1]=P[i]$

Knuth-Morris-Pratt algorithm

Computation of the failure function

```
FF(P[1..m])
  f[0]=-1
  f[1]=0
  k=0
  for j=2 to m do
    while k≥0 and P[k+1]≠P[j] do
      k=f(k) endwhile
    k=k+1
    f(j)=k
  endfor
```

Knuth-Morris-Pratt algorithm

Optimized version (KMP vs MP)

T = ... a b a * * * * ...
= = ≠

P = a b a b a c a
a b a b a c a

⇐ useless shift

$h(3)=0$ h optimized failure function

q	0	1	2	3	4	5	6	7
$f(q)$	-1	0	0	1	2	3	0	1
$h(q)$	-1	0	0	0	0	3	0	1

Knuth-Morris-Pratt algorithm

Computation of the **optimized** failure function h

OFF($P[1..m]$)

$h[0]=-1$

$h[1]=0$

$k=0$

for $j=2$ **to** m **do**

while $k \geq 0$ **and** $P[k+1] \neq P[j]$ **do**

$k=h(k)$ **endwhile**

$k=k+1$

~~$f(j)=k$~~ — **if** $P[j] \neq P[k]$ **then** $h(j)=h(k)$ **else** $h(j)=k$

endfor

Knuth-Morris-Pratt algorithm

Difference with automaton approach: the algorithm can stay at the same position of the text during several steps

at most $\log_\phi m$ shifts at the same position, where $\phi=(1+\sqrt{5})/2$ is the golden ratio

Amortized time complexity

$O(n)$ for the search (KMP)

$O(m)$ for the pre-processing (FE)

History: Morris-Pratt (1970), Knuth-Morris-Pratt (1976),
Matiyasevich (1971)

In practice: Boyer-Moore algorithm, $O(n/|A|)$ time on average

Exercise

- ▶ Give a linear-time algorithm to determine if a string T is a *cyclic rotation (conjugate)* of another string T' . That is, $T=uv$ where $T'=vu$. For example,

arc – car

louche – chelou

lenver (l'envers) – verlen (Verlan)

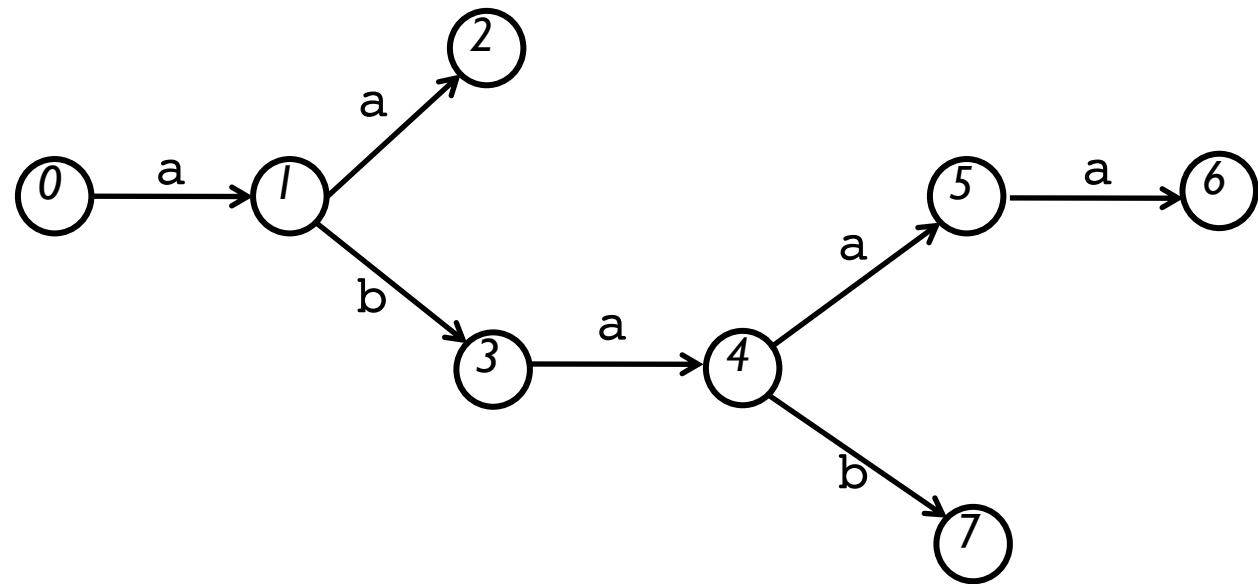
окорок – рококо

Aho-Corasick algorithm (1984)

- ▶ Ideas of the Knuth-Morris-Pratt algorithm can be generalized to several patterns ⇒ Aho-Corasick algorithm (1974)

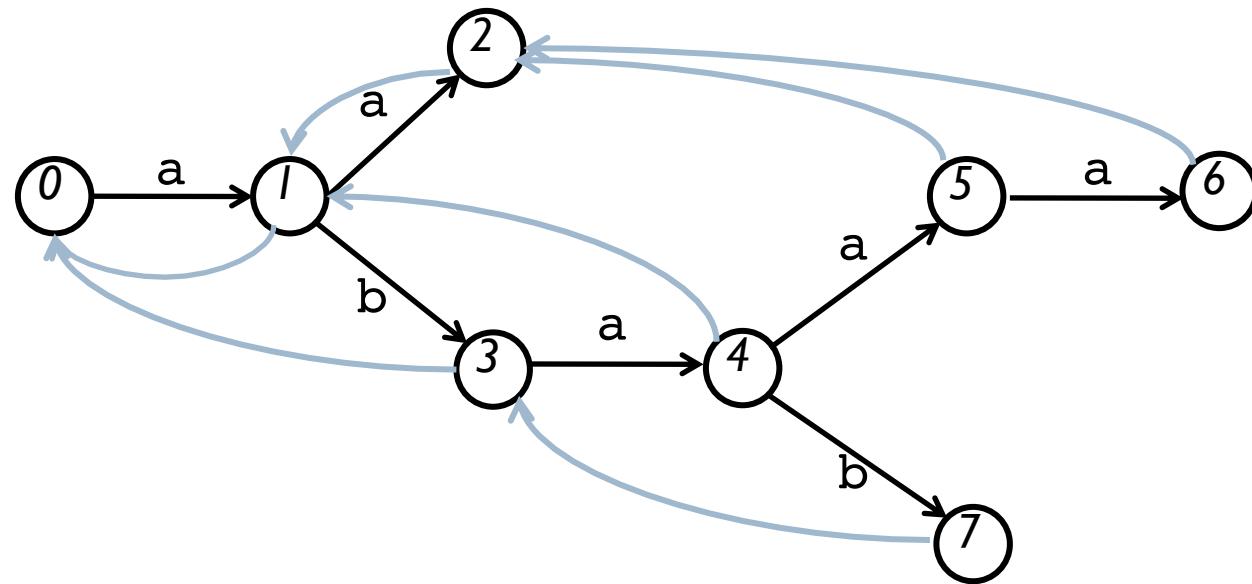
Aho-Corasick algorithm

- ▶ $S = \{aa, abaaa, abab\}$
- ▶ Construct the *trie* of S



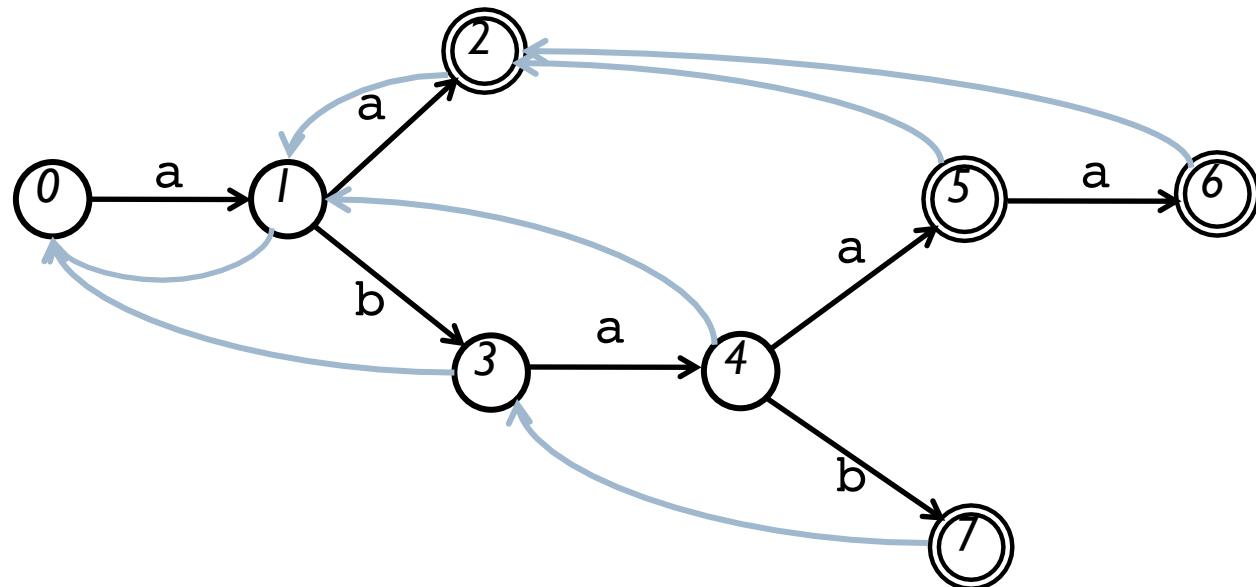
Aho-Corasick algorithm

- ▶ $S = \{aa, abaaa, abab\}$
- ▶ Construct the *trie* of S , compute the failure function



Aho-Corasick algorithm

- ▶ $S = \{aa, abaaa, abab\}$
- ▶ Construct the *trie* of S , compute the failure function, identify final states



Can be constructed in time $O(m)$, where m is the **total** size of patterns in S

Karp-Rabin algorithm (1987)

- ▶ Use hashing for filtering!

Karp-Rabin algorithm (1987)

- ▶ Use hashing for filtering!
- ▶ let $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- ▶ encode pattern $P[1..m]$:

$$p = P[1] \cdot 10^{m-1} + P[2] \cdot 10^{m-2} + \dots + P[m-1] \cdot 10 + P[m]$$

- ▶ p can be computed in time $O(m)$ with Horner's rule :
$$p = P[m] + 10 \cdot (P[m-1] + 10 \cdot (P[m-2] + \dots + 10 \cdot (P[2] + 10 \cdot P[1]) \dots))$$

- ▶ idea:
 - ▶ iteratively compare p with encodings t_i of $T[i..i+m-1]$, for $i=1..n-m+1$
 - ▶ encoding of t_{i+1} computed from the encoding of t_i in constant time :
$$t_{i+1} = 10 \cdot (t_i - 10^{m-1} \cdot T[i]) + T[i+m]$$
 (assuming 10^{m-1} is pre-computed)
- ▶ **Problem:** encodings can be very large \Rightarrow compute them modulo a prime number q

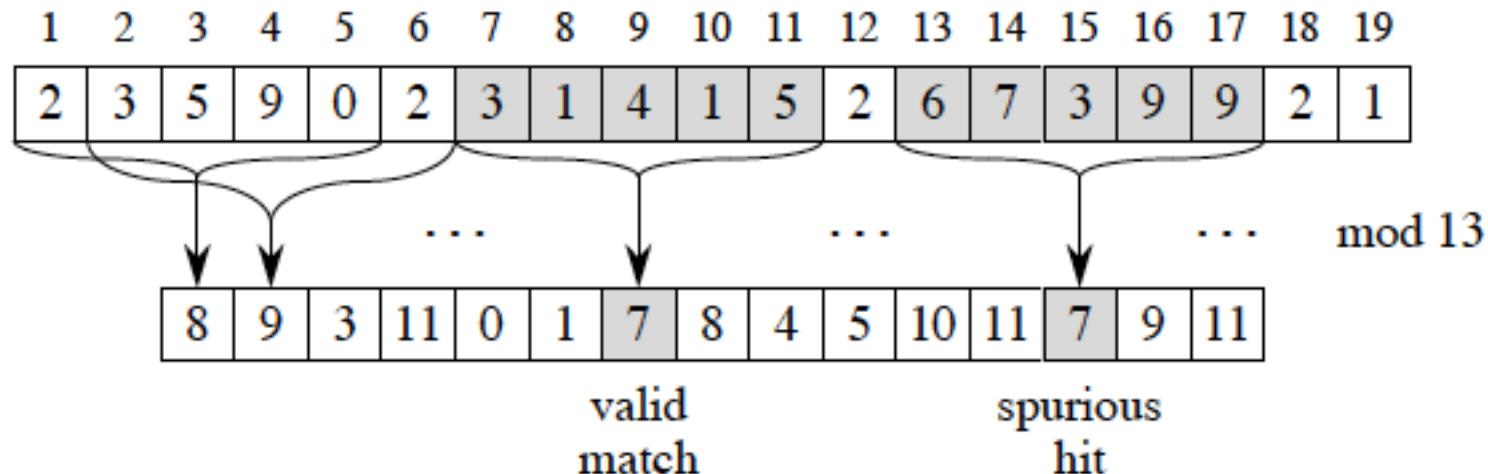
Karp-Rabin algorithm

- we then have

$$t_{i+1} = 10 \cdot (t_i - h \cdot T[i]) + T[i+m] \bmod q, \text{ where } h = 10^{m-1} \bmod q$$

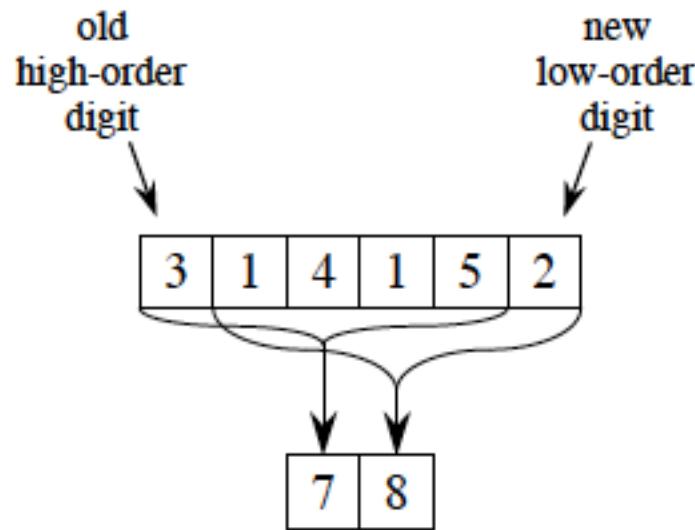
- Problem: we can have false positives!*

- Ex: $P=31415, p=31415 \bmod 13 = 7$



Karp-Rabin algorithm

- ▶ computing hash of t_i from hash of t_{i+1} (illustration):



$$\begin{aligned} 14152 &\equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13} \\ &\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13} \\ &\equiv 8 \pmod{13} \end{aligned}$$

The diagram illustrates the rolling hash computation. It shows the old high-order digit (3) being shifted out and the new low-order digit (2) being shifted in. The calculation for the new hash value is shown as:

- ▶ once a candidate ($T[i..i+m-1]$ with the same hash) is found, we verify it by comparing P and $T[i..i+m-1]$ letter-by-letter

Karp-Rabin hash function for strings

- ▶ $T = T[1..n]$
- ▶ Family of hash functions:
 - ▶ fix a large prime number p . In practice, $p = 2^{31} - 1$ for 32-bit and $p = 2^{61} - 1$ for 64-bit numbers
 - ▶ for $x \in [1..p - 1]$, define
$$h_x(T) = (T[1] \cdot x^{n-1} + T[2] \cdot x^{n-2} + \dots + T[n]) \bmod p$$
- ▶ Family $\{h_x\}$ is n -universal, i.e. for two strings T and S of length n , $P[h_x(T) = h_x(S)] = n/p$, where probability is taken over a random choice of x
- ▶ ⇒ excellent simple, practical and “almost universal” hash functions for strings

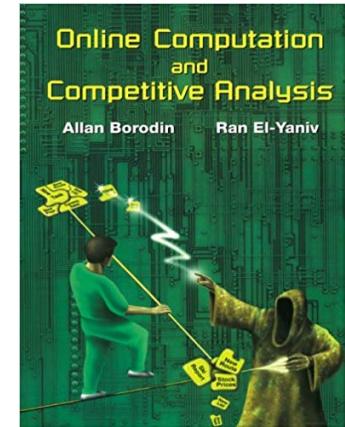
Online/streaming algorithms

Offline vs online

- ▶ ***Offline***: all input data is available beforehand
- ▶ ***Online***: data is input by units in a stream
 - ▶ a unit should be processed (or stored) “immediately” (otherwise it cannot be accessed later)
 - ▶ no information available about items to come
 - ▶ *additional requirement (optional)*: at any moment, the algorithm has computed the result for the data that has been received up to now
 - ▶ *real time*: $O(1)$ worst case time spent on each unit
- ▶ ***Examples***: cache management, scheduling, traffic routing in networks, ...

Competitive analysis

- ▶ σ : sequence of *requests*
- ▶ An online algorithm alg is *c-competitive* iff its cost (e.g. time) $alg(\sigma) \leq c \cdot offline(\sigma)$, for all sequences σ , where $offline(\sigma)$ is the cost of an optimal offline algorithm
- ▶ Remarks:
 - ▶ σ is the only input
 - ▶ we are interested in the worst-case (over all inputs)
 - ▶ concept of *adversary*



Ski rental problem

- ▶ renting skis costs 10\$/day, buying skis costs 100\$
- ▶ you don't know how many days you are going to ski.
Rent or buy?

Ski rental problem

- ▶ renting skis costs 10\$/day, buying skis costs 100\$
- ▶ you don't know how many days you are going to ski.
Rent or buy?
- ▶ *Online algorithm*: rent for 10 days then buy
- ▶ this algorithm is 2-competitive
- ▶ Can we do better?

Ski rental problem

- ▶ renting skis costs 10\$/day, buying skis costs 100\$
- ▶ you don't know how many days you are going to ski.
Rent or buy?
- ▶ *Online algorithm*: rent for 10 days then buy
- ▶ this algorithm is 2-competitive
- ▶ Can we do better?
- ▶ **NO** with a deterministic algorithm (why?)
- ▶ **YES** with a probabilistic algorithm: there exists an expected 1.58-competitive online algorithm

Probabilistic ski rental: intuition

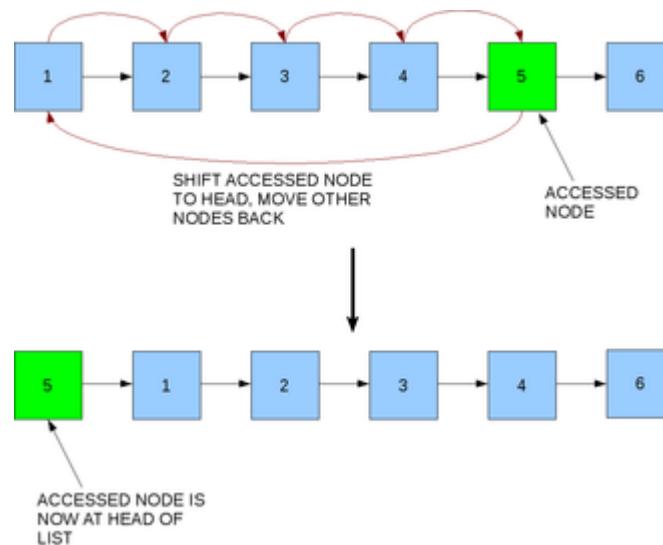
- ▶ If I buy skis after day j ($j \leq 10$), the worst scenario is that I ski only j days
 - ▶ However, for different j 's worst scenarios are different
 - ▶ If we choose j randomly, we avoid the impact of a single worst scenario
-
- ▶ *Example:* buy skis after 8 days with proba $\frac{1}{2}$ and after 10 days with proba $\frac{1}{2}$. Then expected competitiveness c is
 - ▶ if I ski < 8 days, then $c = 1$
 - ▶ if I ski 8 days, then $c = \frac{1}{2} \cdot \frac{80+100}{80} + \frac{1}{2} \cdot 1 = 1.625$
 - ▶ if I ski 9 days, then $c = \frac{1}{2} \cdot \frac{80+100}{90} + \frac{1}{2} \cdot 1 = 1.5$
 - ▶ if I ski ≥ 10 days, then $c = \frac{1}{2} \cdot \frac{80+100}{100} + \frac{1}{2} \cdot \frac{100+100}{100} = 1.9$

Probabilistic ski rental: general case

- ▶ Let
 - ▶ rental costs 1 per day
 - ▶ buying skis costs B
 - ▶ $T \geq 1$ be the # of days of skiing in the input
 - ▶ the algorithm buys skis after j days ($0 \leq j \leq B$), with proba p_j
- ▶ If $T < B$, then $c_{<B} = \sum_{i \leq T} p_i \cdot \frac{i+B}{T} + \sum_{T < i \leq B} p_i \cdot 1$
- ▶ If $T \geq B$, then $c_{\geq B} = \sum_{i \leq B} p_i \cdot \frac{i+B}{B}$
- ▶ $c_{opt} = \min\{\max_T\{c_{<B}, c_{\geq B}\} \mid \sum p_i = 1\} \approx \frac{e}{e-1} \approx 1.58$

Self-organizing lists [Sleator&Tarjan 1985]

- ▶ a set of elements is stored in a singly linked list
- ▶ a sequence of access operations given online
- ▶ accessing i -th element e in the list costs i
- ▶ after accessing e , we are allowed to move it (towards the head) *with no cost*
- ▶ *goal:* minimize the total cost of all accesses



Self-organizing lists: offline solution

- ▶ offline algorithm DF (*Decreasing Frequency*): sort elements in the non-increasing order of access frequency
- ▶ DF is optimal if no moves are allowed
- ▶ Is it also optimal if moves are allowed? In general not. (E.g. accessing each element twice)
- ▶ Computing the minimum cost is complicated (NP-complete)

Self-organizing lists: online algorithms

- ▶ MF (*Move-to-Front*): after accessing an element, move it to the head of the list
- ▶ T (*Transpose*): after accessing an element, exchange it with the immediately preceding item
- ▶ FC (*Frequency count*): Maintain a count of each element, initially 0, incremented by 1 when the element is accessed. Maintain the list in the non-increasing order of counts.

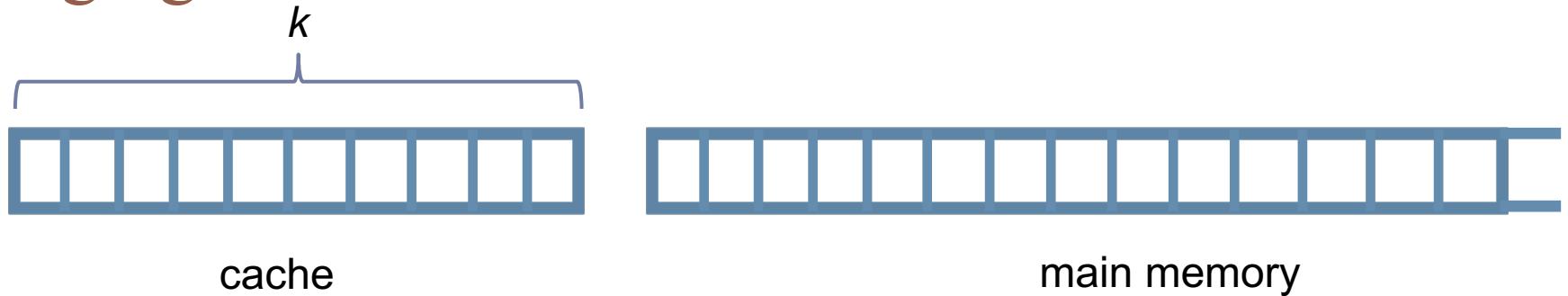
Self-organizing lists: results

- ▶ T does not have a constant competitive ratio
 - ▶ *Counter-example:* alternatively access the two last elements of the list
- ▶ FC does not have a constant competitive ratio
 - ▶ *Counter-example:* access the 1st element $k > n$ times, the second $(k-1)$ times, ... and the last $(k-n+1)$ time. The list is then never modified. The cost is $k+2(k-1)+3(k-2)+\dots+n(k-n+1) \geq (k-n)(1+2+3+\dots+n) = \Theta(kn^2)$. With MF the cost is $k+[2+(k-2)]+[3+(k-3)]+\dots=kn$
 - ▶ *Corollary:* MF can be a factor of $O(n)$ better than DF

Self-organizing lists: Move-to-Front

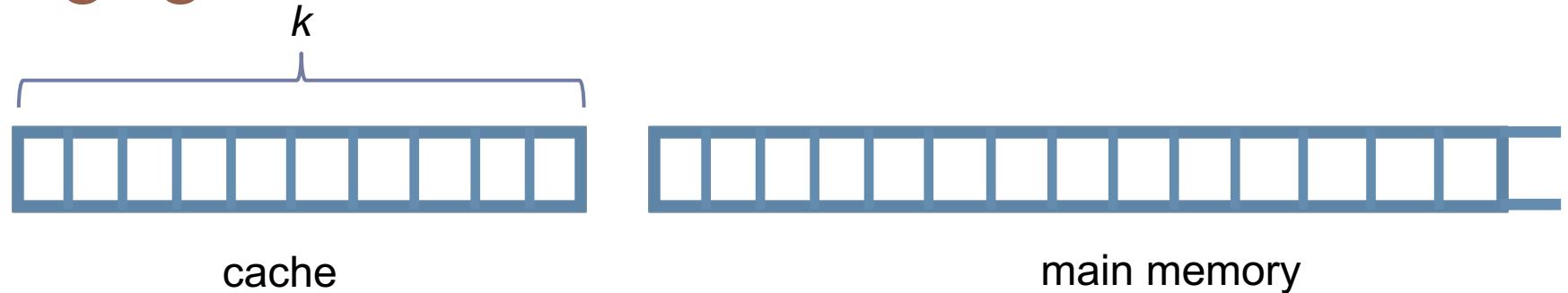
- ▶ *Theorem [Sleator&Tarjan 85]: MF is 2-competitive*
- ▶ *Remark:* The picture is different for the average-case complexity, when we access randomly according to access probabilities (p_1, p_2, \dots, p_n) . The measure is then the *expected cost*. In this case,
 - ▶ DF is an optimal algorithm
 - ▶ MF is no better than T [Rivest 1976]

Paging



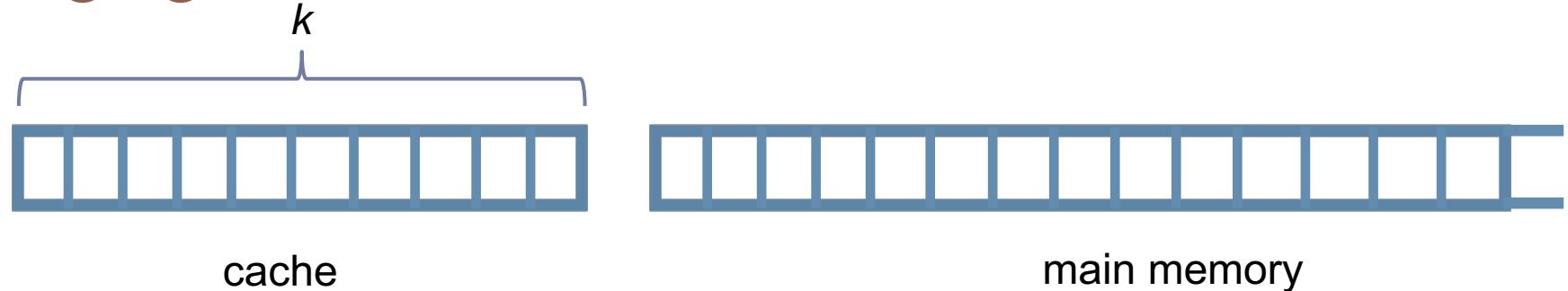
- ▶ requests: accesses to pages
 - ▶ access to cache costs 0
 - ▶ access to main memory requires a swap with cache and costs 1

Paging



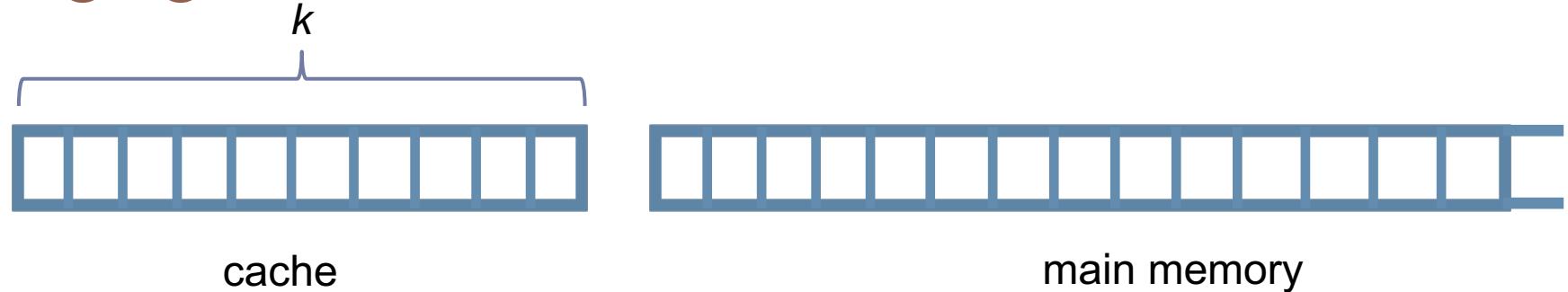
- ▶ Various online strategies can be applied:
 - ▶ *Least Recently Used (LRU)*: swap with the page with the earliest last access
 - ▶ *FIFO*: swap with the page that has entered the cache first
 - ▶ *Least Frequently Used (LFU)*: swap with the page that has been accessed the least
- ▶ ...

Paging: results



- ▶ *The following **offline** algorithm is optimal:* swap with the page that has the latest next access
- ▶ LRU and FIFO are k -competitive, and this is the best we can have with (deterministic) online algorithms
- ▶ LFU is *not* c -competitive for any constant c

Paging: results



- ▶ *The following **offline** algorithm is optimal:* swap with the page that has the latest next access
- ▶ LRU and FIFO are k -competitive, and this is the best we can have with (deterministic) online algorithms
- ▶ LFU is *not* c -competitive for any constant c
- ▶ in practice, LRU is better than FIFO, and both are c -competitive for $c \in [1.5..4]$ if requests are grouped (*locality of reference*) [Albers, Frascaria 2018]

Разборчивая невеста (Secretary problem)

- ▶ A princess chooses a husband
- ▶ n candidates are presented one-by-one (n is known to the princess)
- ▶ for any two candidates, the princess is able to say who is better
- ▶ the order of candidates is random
- ▶ each candidate is either definitely accepted or definitely rejected
- ▶ *goal:* maximize the probability to choose the best

Разборчивая невеста (Secretary problem)

- ▶ A princess chooses a husband
- ▶ n candidates are presented one-by-one (n is known to the princess)
- ▶ for any two candidates, the princess is able to say who is better
- ▶ the order of candidates is random
- ▶ each candidate is either definitely accepted or definitely rejected
- ▶ *goal:* maximize the probability to choose the best

...

Чтоб в одиночестве не кончить веку,
Красавица, пока совсем не отцвела,
За первого, кто к ней присватался,
пошла:
И рада, рада уж была,
Что вышла за калеку.

И.А.Крылов, Разборчивая невеста

Разборчивая невеста (Secretary problem)

- ▶ *NB:* not searching for best competitiveness
- ▶ *Optimal strategy:* reject $(r-1)$ candidates, then select the first one which is better than any of them
- ▶ Analysis shows that $r=1/e \cdot n \approx 0.368 \cdot n$ (rule of 37%)
- ▶ The probability of selecting the best candidate is $1/e$ as well

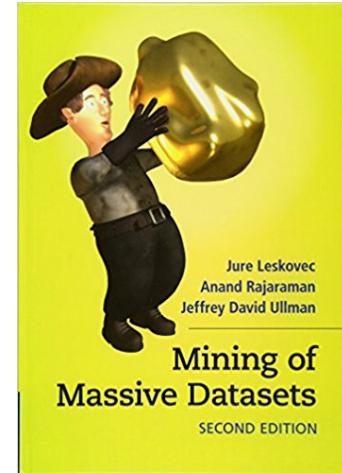


Mining big data streams

Examples of stream sources

- ▶ Sensor data
- ▶ Image data
- ▶ Internet and web traffic
- ▶ Phone calls
- ▶ Web searches
- ▶ streams of customers/purchased items/...
- ▶ ...

- ▶ Main issue: memory!
- ▶ See homeworks 2 (hashing), 4 (string alignment)



Counting distinct elements in a stream

- ▶ *Count-distinct problem*: count the number of distinct elements in a (very large) stream
- ▶ *Examples*:
 - ▶ unique users of a web site,
 - ▶ distinct IP addresses (routers, web servers, ...)
 - ▶ detecting DoS attacks
 - ▶ number of distinct words (k -mers) in a (streamed) text (DNA sequence)
 - ▶ estimating the cardinality for memory allocation (Bloom filter)
 - ▶ ...
- ▶ How can we beat the naïve solution?
 - ▶ **count approximately!**
 - ▶ **use probabilities!**

Approximate counting (Morris 1977)

- ▶ Robert Morris (Bell Labs): maintain the logs of a very large number of events in small registers
- ▶ Algorithm:
 - ▶ maintain K that stores (approx value of) $\log(n)$, i.e. size of K is $\log\log(n)$
 - ▶ initialize $K=0$
 - ▶ when a new event arrives, increment K **with probability 2^{-K}**
- ▶ K reaches a value k after $1+2+4+\dots+2^{k-1}=2^k-1$ steps, i.e. $E[2^k-1]=N$ (N is true count)
- ▶ $\sigma^2=N(N-1)/2$ i.e. $\sigma \approx N/\sqrt{2}$ (70% of N)

Exercise

- ▶ Implement this and run 10 times, acquire statistics:

$K=0$

for $i=1$ to 2^{30} do

 increment K with probability 2^{-K}

 print K // compare K with 30

How to improve?

- ▶ *Improvement 1:* change base of logarithms from 2 to some smaller b ($b > 1$), count $\log_b(N)$
 - ▶ increment K with probability b^{-K}
 - ▶ $E[b^K - 1] = (b-1)N$, $\sigma^2 = (b-1)N(N-1)/2$
 - ▶ price: space increased from $\log\log(n)$ to $\log\log_b(n)$
- ▶ *Improvement 2:* keep m counters instead of just one, then compute the average/median ...

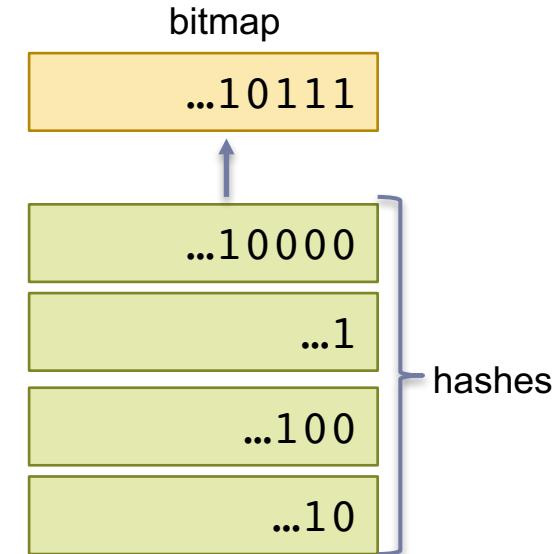
Counting distinct elements in a stream

[Flajolet & Martin 85] Main idea:

- ▶ hash elements into (binary) numbers from $[0..2^L-1]$ using a good hash function h
- ▶ hashes $\dots\text{xxx}1$ are expected to occur $\frac{1}{2}$ time, $\dots\text{xxx}10$ are expected $\frac{1}{4}$ time, $\dots\text{xxx}100$ $\frac{1}{8}$ time, etc.
- ▶ $\max_k[\text{hash } \dots 10^i \text{ are all observed for } 0 \leq i \leq k] + 1$ is a good indication of $\log(N)$ (N nb of distinct elements)

Flajolet & Martin algorithm: implementation

- ▶ maintain a bitmap of size L , set all bits to 0
- ▶ for each hash, compute position of the rightmost 1 and set the corresponding bit of bitmap to 1
- ▶ let i be the position (from right) of the rightmost 0 in bitmap
- ▶ then the nb N of unique elements is estimated as $2^{i-1}/\varphi$, $\varphi=0.77351..$



Intuition: if $i \gg \log(N)+1$ then i -th bit of bitmap is almost certainly 0; if $i \ll \log(N)+1$, then it is almost certainly 1

- ▶ Pb : big variance of results (accuracy within a factor of ~2)
- ▶ *Solution 1*: run the algo m times, then take average/median/combination (accuracy $O(1/\sqrt{m})$)
- ▶ *Solution 2 [FM]*: *stochastic averaging*
use first k bits of hash to dispatch elements into $m=2k$ bins, then average;
accuracy $\approx 0.78/\sqrt{m}$

Further improvements

- ▶ LogLog (Durand, Flajolet 03)
- ▶ HyperLogLog (Flajolet, Fusy, Gandouet, Meunier 07)
- ▶ (Kane, Nelson, Woodruff 10)

Sampling a stream

- ▶ General idea: sample a stream (e.g. consider 1/10 of the items) in hope that the sampled stream will have similar properties
- ▶ Cannot be done by simply sampling 1 over 10 items!
- ▶ Why? Assume we have a stream where s items occur once and d occur twice, and we want to estimate the fraction of repeated elements (right answer $\frac{d}{s+d}$).
- ▶ If we sample each element with $p = \frac{1}{10}$, then only $\frac{d}{100}$ items will occur twice and $\frac{2d}{10} - \frac{2d}{100} = \frac{18d}{100}$ will become unique
- ▶ The estimate will be $\frac{\frac{d}{100}}{\frac{s}{10} + \frac{18d}{100} + \frac{d}{100}} = \frac{d}{10s+19d}$

Sampling a stream (cont)

- ▶ How to solve this problem?

Sampling a stream (cont)

- ▶ How to solve this problem? Use hashing!
- ▶ Hash items to numbers, sample those whose hash ends with 0 (in decimal notation)

Frequent items (“heavy hitters”)

- ▶ Count-min sketch [Cormode&Muthukrishnan 2005]
- ▶ *Idea:*
 - ▶ allocate table $C[1..d, 1..k]$ (d, k will be defined later)
 - ▶ choose d hash functions $h_1, \dots, h_d: U \rightarrow [1..k]$
 - ▶ processing item x :
 - ▶ for $i = 1$ to d do $C[i, h_i(x)] = C[i, h_i(x)] + 1$
 - ▶ estimate the number of occurrences of x :
 - ▶ $\hat{f}(x) = \min_{1 \leq i \leq d} C[i, h_i(x)]$
- ▶ Analysis shows that if $t = \log_2 \frac{1}{\delta}$ and $k = \frac{2}{\varepsilon}$, then $P[\hat{f}(x) \geq f(x) + \varepsilon k] \leq \delta$, where $f(x)$ is the true count of x

Related problems on streams

- ▶ Filtering streams against a given sample
 - ▶ Bloom filters
- ▶ Counting ones in a window
 - ▶ given N , answer queries “how many ones are there in the last k bits?” for any $k \leq N$ [Datar, Gionis, Indyk, Motwani 2002]

NP-completeness

a glimpse into Structural Complexity

Polynomial-time algorithms

- ▶ All algorithms we have seen so far (and more) run in $O(n^c)$ time
 - ▶ graph traversals, Dijkstra, Bellman-Ford, all-pairs shortest paths, minimum spanning trees, minimum flow
 - ▶ search trees
 - ▶ heaps, sorting
 - ▶ interval scheduling, LCS, edit distance, sequence alignment
 - ▶ Viterbi algorithm in HMM

Polynomial-time algorithms

- ▶ All algorithms we have seen so far (and more) run in $O(n^c)$ time
 - ▶ graph traversals, Dijkstra, Bellman-Ford, all-pairs shortest paths, minimum spanning trees, minimum flow
 - ▶ search trees
 - ▶ heaps, sorting
 - ▶ interval scheduling, LCS, edit distance, sequence alignment
 - ▶ Viterbi algorithm in HMM
- ▶ ... but also
 - ▶ solving systems of linear equations over reals (Gauss elimination)
 - ▶ linear programming over reals (ellipsoid method by L.Khachiyan, 70s)
 - ▶ primality testing (Agrawal-Kayal-Saxena, 2002)
 - ▶ ...



"Efficiency assumption"

- ▶ Practical algorithms = polynomial-time algorithms
- ▶ Tractable problems = those which admit polynomial-time algorithms
(Cobham-Edmonds thesis)
- ▶ *Corollary*: untractable problems = those for which there is no polynomial-time algorithm

"Efficiency assumption"

- ▶ Practical algorithms = polynomial-time algorithms
- ▶ Tractable problems = those which admit polynomial-time algorithms
(Cobham-Edmonds thesis)
- ▶ *Corollary*: untractable problems = those for which there is no polynomial-time algorithm
- ▶ There are many problems for which we not to know polynomial-time algorithms, but for many of them, we cannot prove that there is none
- ▶ Provably untractable time problems exist, e.g. (generalized) chess, checkers or GO ($n \times n$ board)
- ▶ ... up to undecidable problems (e.g. Halting problem, Post correspondence problem, ...)

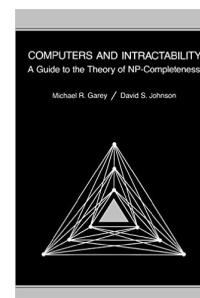
Polynomial vs exponential

in milliseconds

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Figure 1.2 Comparison of several polynomial and exponential time complexity functions.

Taken from :
Garey&Johnson, Computers and Intractability, 1979



Measuring the complexity of algorithms

- ▶ RAM = Random Access Machine (*unit-cost* or *log-cost*)
- ▶ TM = Turing machine

Measuring the complexity of algorithms

- ▶ RAM = Random Access Machine (*unit-cost* or *log-cost*)
- ▶ TM = Turing machine
- ▶ RAM and TM are "polynomially equivalent"

Simulated machine B	Simulating machine A		
	1TM	kTM	RAM
1-Tape Turing Machine (1TM)	—	$O(T(n))$	$O(T(n)\log T(n))$
k-Tape Turing Machine (kTM)	$O(T^2(n))$	—	$O(T(n)\log T(n))$
Random Access Machine (RAM)	$O(T^3(n))$	$O(T^2(n))$	—

Figure 1.6 Time required by machine A to simulate the execution of an algorithm of time complexity $T(n)$ on Machine B (for example, see [Hopcroft and Ullman, 1969] and [Aho, Hopcroft, and Ullman, 1974]).

from :
Garey&Johnson, Computers and Intractability, 1979

- ▶ ... provided (for unit-cost RAM) that all operands of arithmetic operations are polynomially bounded (i.e. fit a const number of computer words)

Measuring the complexity: input encoding

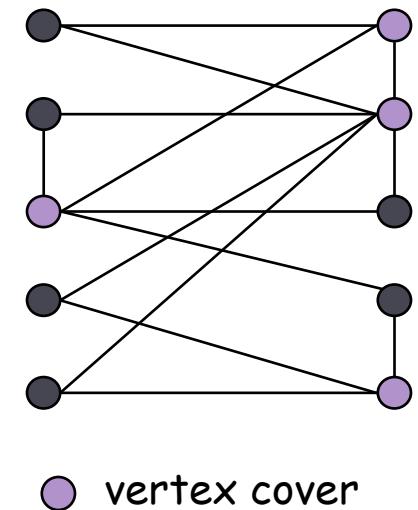
- ▶ Time complexity is a ***function*** of input data size
- ▶ For the Turing machine:
 - ▶ size of input = number of tape cells of the input encoding
 - ▶ time = number of steps (transitions)
- ▶ Inside the "polynomial world" the way of encoding the input is important
- ▶ Here we assume that all encodings are "polynomially equivalent" \Rightarrow numbers are ***not*** encoded in unary and the encoding of data is "compact"

Decision vs optimization problems

- ▶ Computational problems can be of different nature
- ▶ **Optimization problem:** Compute an object of optimum (minimum or maximum) "size" (e.g. minimum spanning tree, shortest path, etc.)
- ▶ **Decision problem:** Does there exist an object of "size" $\leq k$?
- ▶ Decision problems are formalized as language membership problem
- ▶ Obviously, solving an optimization problem implies solving the corresponding decision problem
- ▶ Very often, the inverse is true: an optimization problem is "polynomially reducible" (cf below) to the decision problem
- ▶ Decision and optimization problems are usually "polynomially equivalent". We focus on decision problems.

Example: vertex cover

- ▶ **Definition:** Given a graph $G = (V, E)$, a **vertex cover** of G is a subset of vertices $S \subseteq V$ such that for each edge, at least one of its endpoints is in S ?
 - ▶ **VERTEX COVER (decision problem):** Given a graph $G = (V, E)$ and an integer k , is there a vertex cover S such that $|S| \leq k$?
 - ▶ **VERTEX COVER (optimization problem):** Given a graph $G = (V, E)$, find a vertex cover S of minimum cardinality.
- To find min cardinality vertex cover:
- (Binary) search for cardinality k^* of min vertex cover
 - Find a vertex v such that $G - \{v\}$ has a vertex cover of size $\leq k^*-1$ (any vertex in any min vertex cover will have this property)
 - Include v in the vertex cover
 - Recursively find a min vertex cover in $G - \{v\}$
 - $T_{\text{opt}}(n) = \log(n) \cdot T_{\text{dec}}(n) + n^2 \cdot T_{\text{dec}}(n)$



Classes **P** and **NP**

- ▶ **Class P:**
 - ▶ *formal definition:* class of decision problems (languages) that can be solved on Turing machine in time $\leq p(n)$ for a fixed polynome p (n size of the problem)
 - ▶ *informal:* Class of problems that have polynomial time algorithms solving them
- ▶ **Class NP ("Non-deterministic P"):**
 - ▶ *formal definition:* class of decision problems that can be solved on a **non-deterministic** Turing machine in time $\leq p(n)$ for a fixed polynome p (n size of the problem)
 - ▶ *equivalent definition:* class of decision problems for which a solution can be **verified** in polynomial time (insures brute-force ('перебор') solutions)

A Survey of Russian Approaches to *Perebor* (Brute-Force Search) Algorithms

B. A. TRAKHTENBROT

*Concerns about computational problems requiring brute-force or exhaustive search methods have gained particular attention in recent years because of the widespread research on the “P = NP?” question. The Russian word for “brute-force search” is “*perebor*.” It has been an active research area in the Soviet Union for several decades. Disputes about approaches to *perebor* had a certain influence on the development, and developers, of complexity theory in the Soviet Union. This paper is a personal account of some events, ideas, and academic controversies that surrounded this topic and to which the author was a witness and—to some extent—a participant. It covers a period that started in the 1950s and culminated with the discovery and investigation of nondeterministic polynomial (NP)-complete problems independently by S. Cook and R. Karp in the United States and L. Levin in the Soviet Union.*

Categories and Subject Descriptors: I.2.8 [**Artificial Intelligence**]—graph and tree search strategies; K.2 [**History of Computing**]—people, software

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: brute-force search algorithms, *perebor*

Introduction

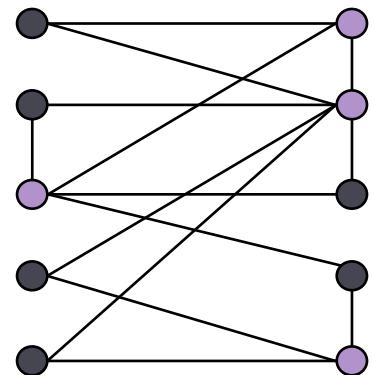
A *perebor* algorithm, or *perebor* for short, is Russian for what is called in English a “brute-force” or “exhaustive” search method. Other combinations of words also occur in translations from Russian, such as “successive trials,” “sequential searching,” and “thorough searching.” To keep the historical flavor, I

case of an affirmative answer to (1), an n -tuple should be produced.

The obvious *perebor* algorithm that solves both the existential and constructive versions of the problem considers all the n -tuples of truth values in some order (sav. lexicographical order). The first time an n -tuple

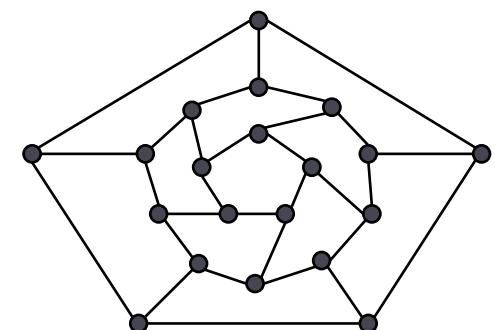
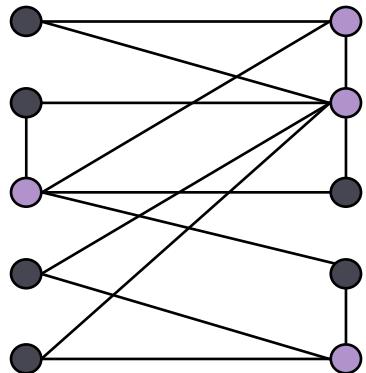
Examples of problems in NP

- ▶ **Vertex cover**
 - ▶ easy to verify if a given subset $S \subseteq V$ is a vertex cover



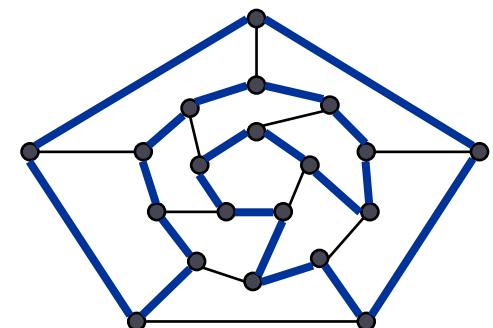
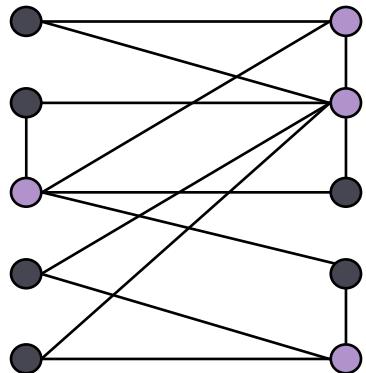
Examples of problems in NP

- ▶ **Vertex cover**
 - ▶ easy to verify if a given subset $S \subseteq V$ is a vertex cover
- ▶ **More examples:**
 - ▶ Hamiltonian circuit in a graph
 - ▶ largest clique in a graph
 - ▶ integer factorization $437669 = 541 \cdot 809$
 - ▶ longest common subsequence of multiple strings
 - ▶ multiset partition $\{3,1,1,2,2,1\} \rightarrow \{2,1,1,1\} \cup \{3,2\}$
(NB: numbers are encoded in binary! otherwise a pseudo-polynomial dynamic programming algorithm exists!)
 - ▶ ...



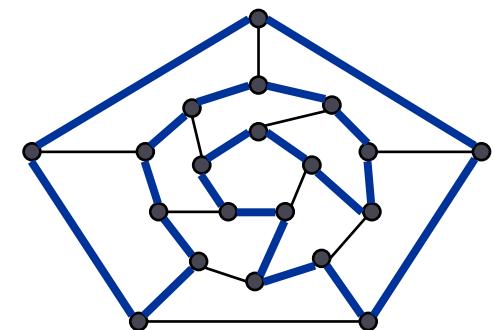
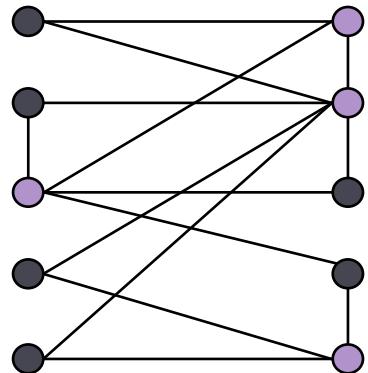
Examples of problems in NP

- ▶ **Vertex cover**
 - ▶ easy to verify if a given subset $S \subseteq V$ is a vertex cover
- ▶ **More examples:**
 - ▶ Hamiltonian circuit in a graph
 - ▶ largest clique in a graph
 - ▶ integer factorization $437669 = 541 \cdot 809$
 - ▶ longest common subsequence of multiple strings
 - ▶ multiset partition $\{3,1,1,2,2,1\} \rightarrow \{2,1,1,1\} \cup \{3,2\}$
(NB: numbers are encoded in binary! otherwise a pseudo-polynomial dynamic programming algorithm exists!)
 - ▶ ...



Examples of problems in NP

- ▶ Vertex cover
 - ▶ easy to verify if a given subset $S \subseteq V$ is a vertex cover
- ▶ More examples:
 - ▶ Hamiltonian circuit in a graph
 - ▶ largest clique in a graph
 - ▶ integer factorization $437669 = 541 \cdot 809$
 - ▶ longest common subsequence of multiple strings
 - ▶ multiset partition $\{3,1,1,2,2,1\} \rightarrow \{2,1,1,1\} \cup \{3,2\}$
(NB: numbers are encoded in binary! otherwise a pseudo-polynomial dynamic programming algorithm exists!)
 - ▶ ...
- ▶ $P \subseteq NP$



\$1,000,000 question

P=NP?

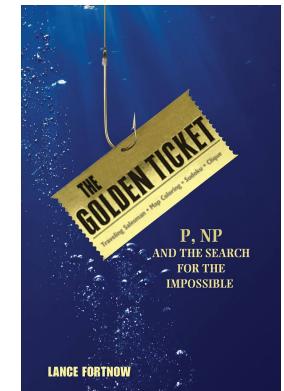
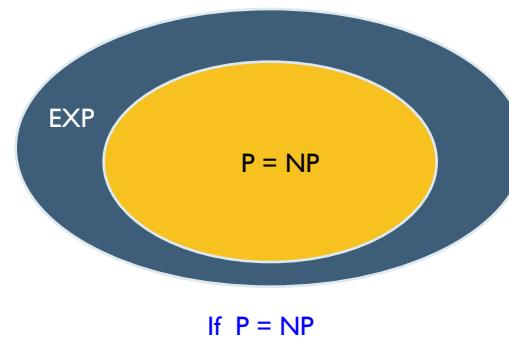
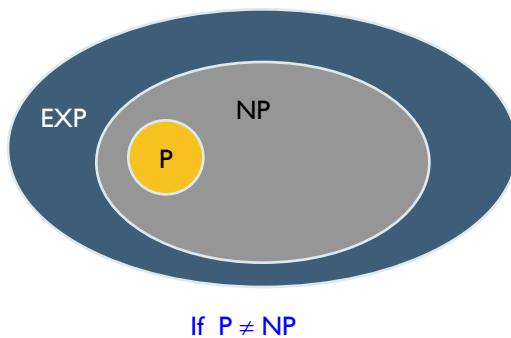
(or can '*перебор*' be eliminated?)

one of Millennium Prize Problems, Clay Mathematics Institute

Why is it so important?

- ▶ If yes: Efficient algorithms for 3-COLOR, TSP, FACTOR, SAT, ...
- ▶ If no: No efficient algorithms possible for 3-COLOR, TSP, SAT, ...

would break RSA cryptography
(and potentially collapse economy)



Consensus opinion on $P = NP$? Probably no.

about consequences of $P=NP$ read L.Fortnow, Golden ticket

Polynomial-Time Reduction

Idea: formalize "problem X is at least as hard as problem Y"

Suppose we could solve a problem X in polynomial-time. What else could we solve in polynomial time?

*Reduction**: Problem Y is polynomial-time reducible to problem X if arbitrary instances of problem Y can be solved using:

- ▶ Polynomial number of standard computational steps, and
- ▶ Polynomial number of calls to oracle that solves problem X

Notation: $Y \leq_P X$

 computational model supplemented by special piece of hardware that solves instances of Y in a single step

That is, if we have a code for X, we can obtain a code for Y with only polynomial overhead

(*) called "polynomial-time Turing reduction", or "Cook reduction" (as opposed to "Karp(-Levin) reduction" or "many-to-one reduction" or "polynomial transformation" which is more restricted)

Polynomial-Time Reduction

Goal: Classify problems according to **relative** difficulty.

Design algorithms: If $Y \leq_P X$ and X can be solved in polynomial-time, then Y can also be solved in polynomial time. That is, if X is tractable, so is Y .

↖
we have used this earlier in our course

Establish intractability: If $Y \leq_P X$ and Y cannot be solved in polynomial-time, then X cannot be solved in polynomial time. That is, if Y is hard, so is X .

Establish (polynomial-time) equivalence: If $X \leq_P Y$ and $Y \leq_P X$, we write $X \equiv_P Y$

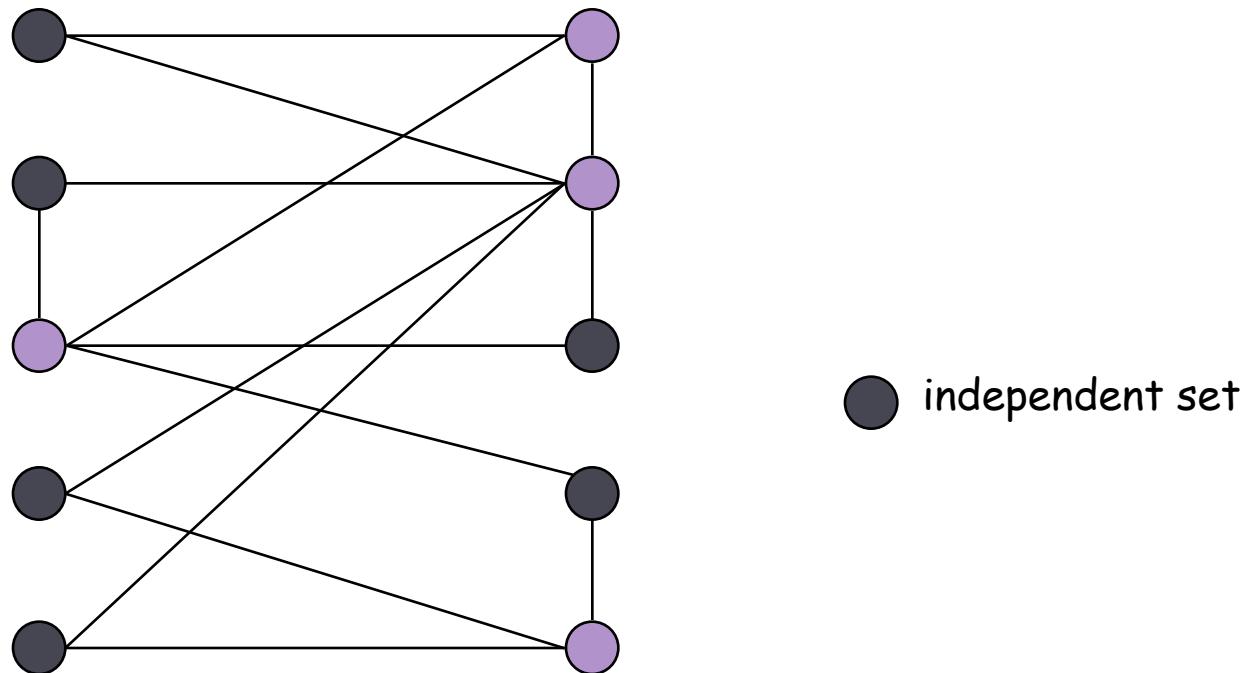
Reduction strategies

- ▶ Reduction by simple equivalence
- ▶ Reduction from special case to general case
- ▶ Reduction by encoding with gadgets

Independent set

INDEPENDENT SET: Given a graph $G = (V, E)$ and an integer k , is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S ?

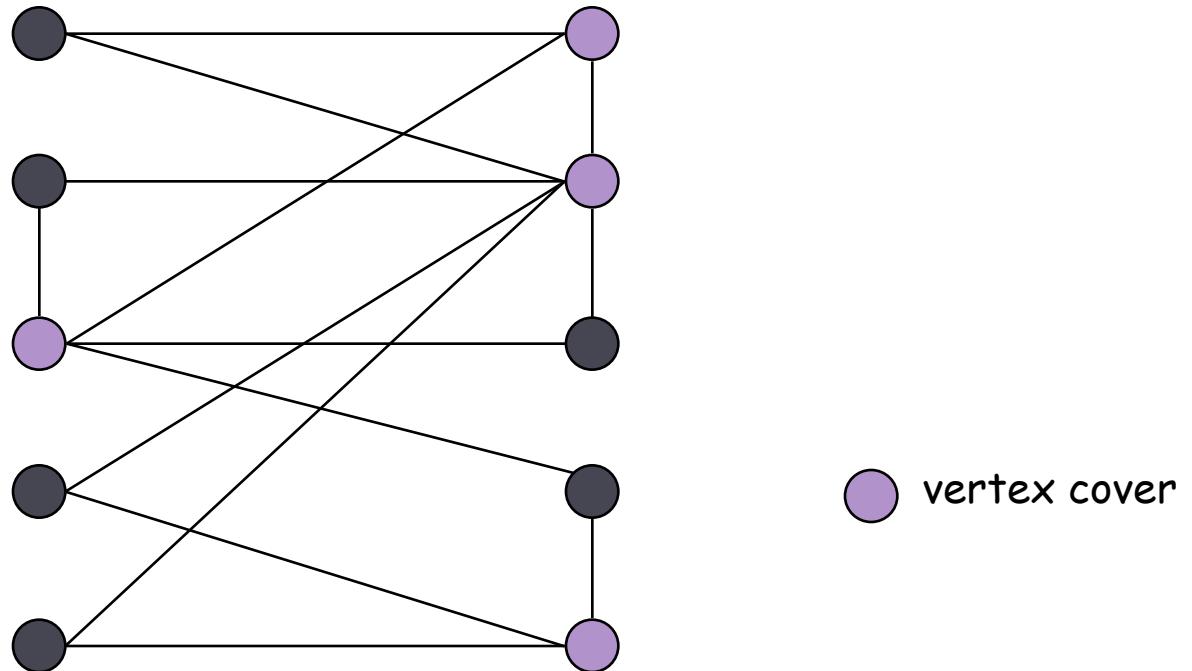
- ▶ *Example:* Is there an independent set of size ≥ 6 ? Yes.
- ▶ *Example:* Is there an independent set of size ≥ 7 ? No.



Vertex cover

VERTEX COVER: Given a graph $G = (V, E)$ and an integer k , is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge, at least one of its endpoints is in S ?

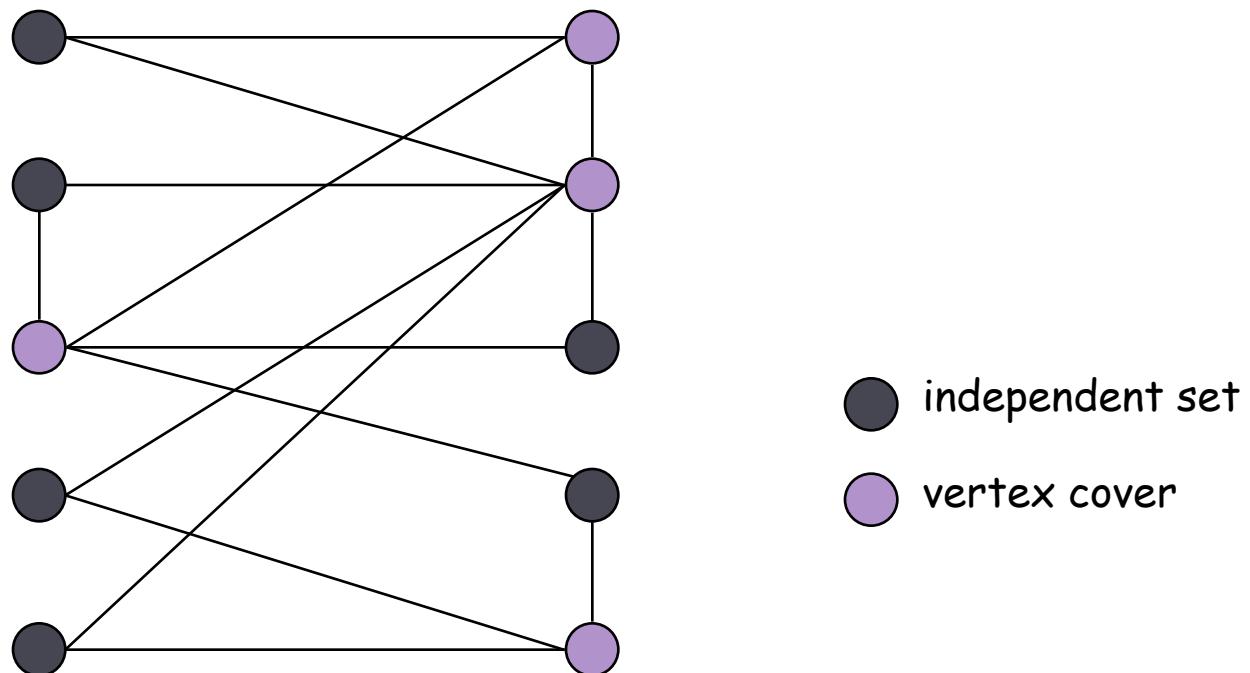
- ▶ *Example:* Is there a vertex cover of size ≤ 4 ? Yes.
- ▶ *Example:* Is there a vertex cover of size ≤ 3 ? No.



Vertex cover and independent set

Claim: VERTEX-COVER \equiv_P INDEPENDENT-SET

Proof: S is an independent set iff $V \setminus S$ is a vertex cover.



Clique

CLIQUE: Given a graph $G = (V, E)$ and an integer k , is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each pair (x, y) of points in S , (x, y) is an edge of E ?

Claim: CLIQUE \equiv_P INDEPENDENT-SET.

Proof: S is an independent set of G iff S is a clique of G' , where G' is the complement of G : $G' = (V, V^2 - E)$.

Reduction strategies

- ▶ Reduction by simple equivalence
- ▶ Reduction from special case to general case
- ▶ Reduction by encoding with gadgets

Set cover

SET COVER: Given a set U of elements, a collection S_1, S_2, \dots, S_m of subsets of U , and an integer k , does there exist a collection of $\leq k$ of these sets whose union is equal to U ?

► *Sample application:*

- m available pieces of software.
- Set U of n functionalities that we would like our system to have.
- The i^{th} piece of software provides the set $S_i \subseteq U$ of functionalities.
- Goal: achieve all n functionalities using fewest pieces of software.

► *Example:*

$$U = \{1, 2, 3, 4, 5, 6, 7\}$$

$$k = 2$$

$$S_1 = \{3, 7\} \quad S_4 = \{2, 4\}$$

$$S_2 = \{3, 4, 5, 6\} \quad S_5 = \{5\}$$

$$S_3 = \{1\} \quad S_6 = \{1, 2, 6, 7\}$$

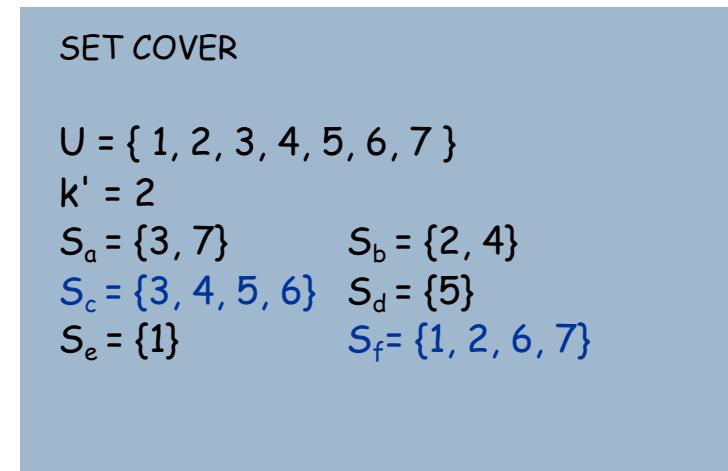
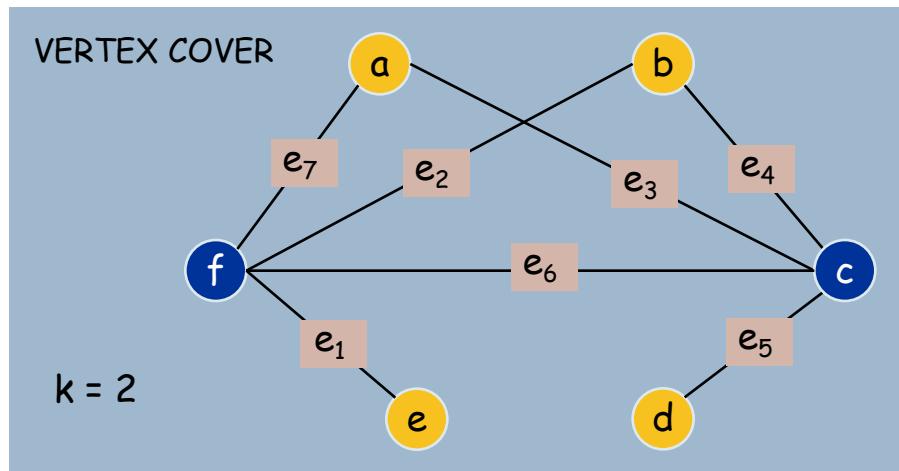
VERTEX-COVER \leq_p SET COVER

Claim: VERTEX-COVER \leq_p SET-COVER

Proof: Given a VERTEX-COVER instance $\langle G = (V, E), k \rangle$, we construct a SET-COVER instance $\langle U, S_1, S_2, \dots, S_m, k' \rangle$ such that the first instance is true **iff** the second instance is true.

Construction:

- ▶ Create SET-COVER instance:
 - ▶ $k' = k$, $U = E$, $S_v = \{e \in E : e \text{ incident to } v\}$
 - ▶ Set-cover of size $\leq k'$ **iff** vertex cover of size $\leq k$. ▀



Reduction strategies

- ▶ Reduction by simple equivalence
- ▶ Reduction from special case to general case
- ▶ Reduction by encoding with gadgets - wait a moment :)

NP-hard and NP-complete

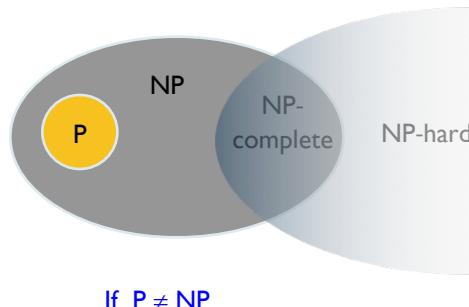
NP-hard: A problem X is NP-hard if, for every problem Y in NP, $Y \leq_p X$.

NP-complete: A problem X is NP-complete, if it is NP-hard and in NP.

Theorem: Suppose X is an NP-complete problem. Then X is solvable in poly-time iff $P = NP$.

Proof: \Leftarrow If $P = NP$ then X can be solved in poly-time since X is in NP
 \Rightarrow Suppose X can be solved in poly-time.

- Let Y be any problem in NP. Since $Y \leq_p X$, we can solve Y in poly-time. This implies $NP \subseteq P$.
- We already know $P \subseteq NP$. Thus $P = NP$. ■



Fundamental question

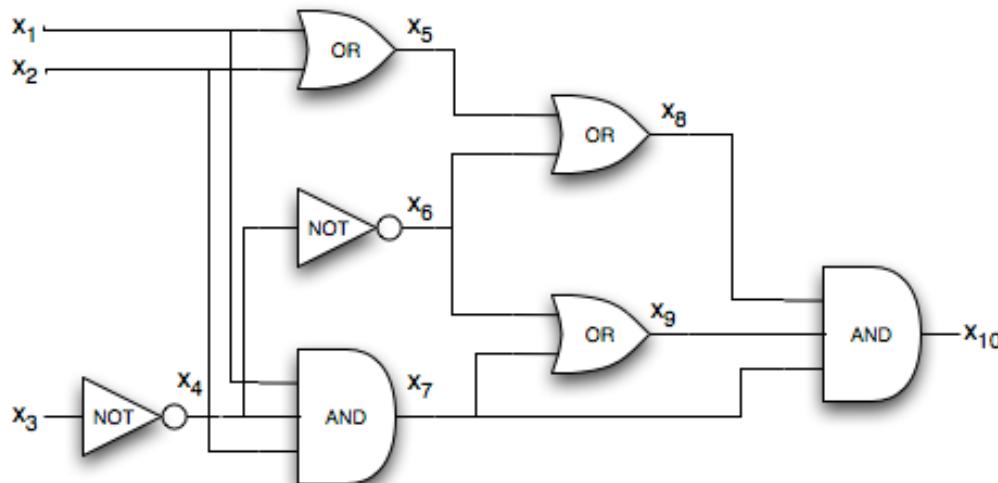
Do there exist "natural" NP-complete problems?

The "First" NP-Complete Problem

Theorem [Cook 71, Levin 73]: CIRCUIT-SAT is NP-complete.



CIRCUIT-SAT: Given a Boolean circuit built out of AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1?



How this was proved?

- ▶ from the definition of NP, that is
- ▶ by encoding an execution of a non-deterministic Turing machine by a boolean circuit
 - ▶ the input problem is solved by YES iff the circuit outputs 1
 - ▶ if the execution is polynomial-length then the circuit is polynomial-size

Establishing NP-Completeness

Remark: Once we established the first NP-complete problem, we can "bootstrap" and prove other problems NP-complete by reduction

Universal recipe to establish NP-completeness of problem X

- ▶ Step 1. Show that X is in NP.
- ▶ Step 2. Choose an NP-complete problem Y.
- ▶ Step 3. Prove that $Y \leq_p X$.

Justification: If Y is an NP-complete problem, and X is a problem in NP with the property that $Y \leq_p X$ then X is NP-complete.

Satisfiability

Literal: A Boolean variable or its negation.

x_i or $\overline{x_i}$

Clause: A disjunction of literals.

$C_j = x_1 \vee \overline{x_2} \vee x_3$

Conjunctive normal form: A propositional formula Φ that is the conjunction of clauses.

$\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

SAT: Given CNF formula Φ , does it have a satisfying truth assignment?

3-SAT: SAT where each clause contains exactly 3 literals.

↙
each corresponds to a different variable

Ex: $(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$

Yes: $x_1 = \text{true}$, $x_2 = \text{true}$ $x_3 = \text{false}$.

3-SAT is NP-Complete

Theorem: 3-SAT is NP-complete.

Proof: Suffices to show that CIRCUIT-SAT \leq_p 3-SAT since 3-SAT is in NP.

- ▶ Let K be any circuit.
- ▶ Create a 3-SAT variable x_i for each circuit element i.
- ▶ Make circuit compute correct values at each node:

- ▶ $x_2 = \neg x_3 \Rightarrow$ add 2 clauses:

$$x_2 \vee x_3, \quad \overline{x}_2 \vee \overline{x}_3$$

- ▶ $x_1 = x_4 \vee x_5 \Rightarrow$ add 3 clauses:

$$x_1 \vee \overline{x}_4, \quad x_1 \vee \overline{x}_5, \quad \overline{x}_1 \vee x_4 \vee x_5$$

- ▶ $x_0 = x_1 \wedge x_2 \Rightarrow$ add 3 clauses:

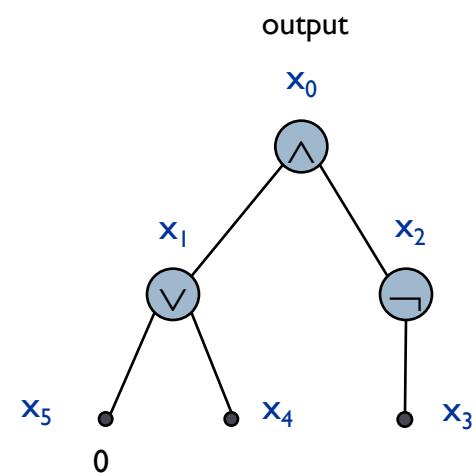
$$\overline{x}_0 \vee x_1, \quad \overline{x}_0 \vee x_2, \quad x_0 \vee \overline{x}_1 \vee \overline{x}_2$$

- ▶ Constant input values and output value (1)

- ▶ $x_5 = 0 \Rightarrow$ add 1 clause: \overline{x}_5

- ▶ $x_0 = 1 \Rightarrow$ add 1 clause: x_0

- ▶ Final step: turn clauses of length < 3 into clauses of length exactly 3 by introducing new variables. E.g. replace y by $(y \vee p \vee q) \wedge (y \vee \bar{p} \vee q) \wedge (y \vee p \vee \bar{q}) \wedge (y \vee \bar{p} \vee \bar{q})$



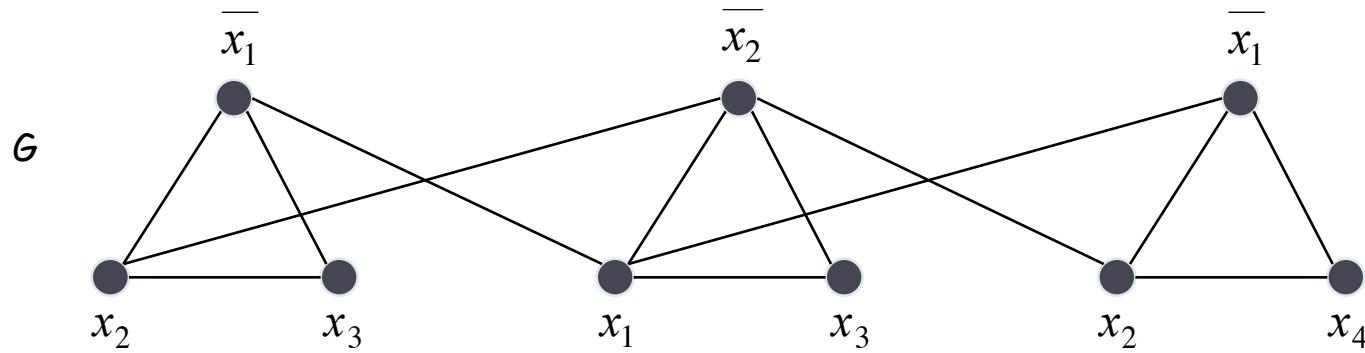
3-SAT Reduces to Independent Set

Theorem: 3-SAT \leq_p INDEPENDENT-SET.

Proof: Given an instance Φ of 3-SAT, we construct an instance (G, k) of INDEPENDENT-SET that has an independent set of size k iff Φ is satisfiable.

Construction.

- ▶ G contains 3 vertices for each clause, one for each literal.
- ▶ Connect 3 literals in a clause in a triangle.
- ▶ Connect literal to each of its negations.



$$k = 3$$

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

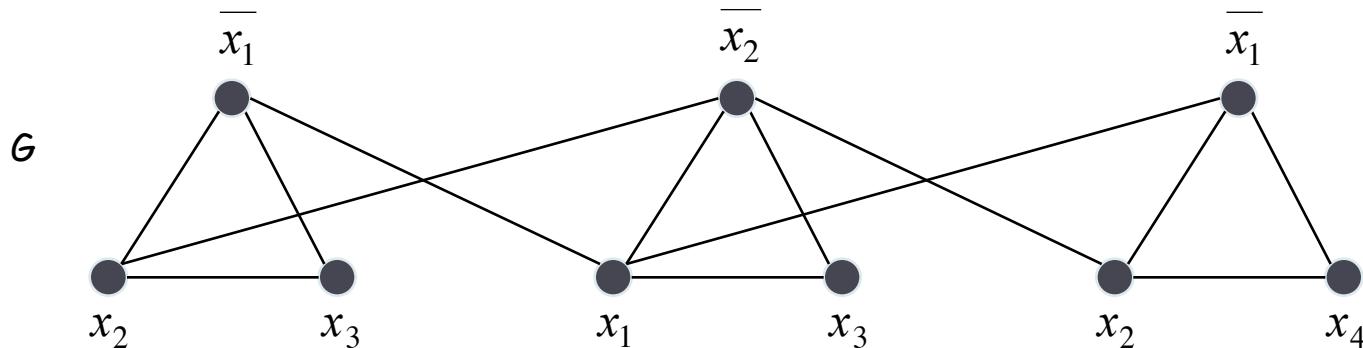
3-SAT Reduces to Independent Set

Claim: G contains independent set of size $k = |\Phi|$ iff Φ is satisfiable.

Proof: \Rightarrow Let S be independent set of size k.

- ▶ S must contain exactly one vertex in each triangle.
- ▶ Set these literals to true. ← and any other variables in a consistent way
- ▶ Truth assignment is consistent and all clauses are satisfied.

\Leftarrow Given satisfying assignment, select one true literal from each triangle. This is an independent set of size k. ▀



$$k = 3$$

$$\Phi = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4)$$

Review

- ▶ Basic reduction strategies.
 - ▶ Simple equivalence: $\text{INDEPENDENT-SET} \equiv_p \text{VERTEX-COVER}$
 - ▶ Special case to general case: $\text{VERTEX-COVER} \leq_p \text{SET-COVER}$
 - ▶ Encoding with gadgets: $\text{3-SAT} \leq_p \text{INDEPENDENT-SET}$

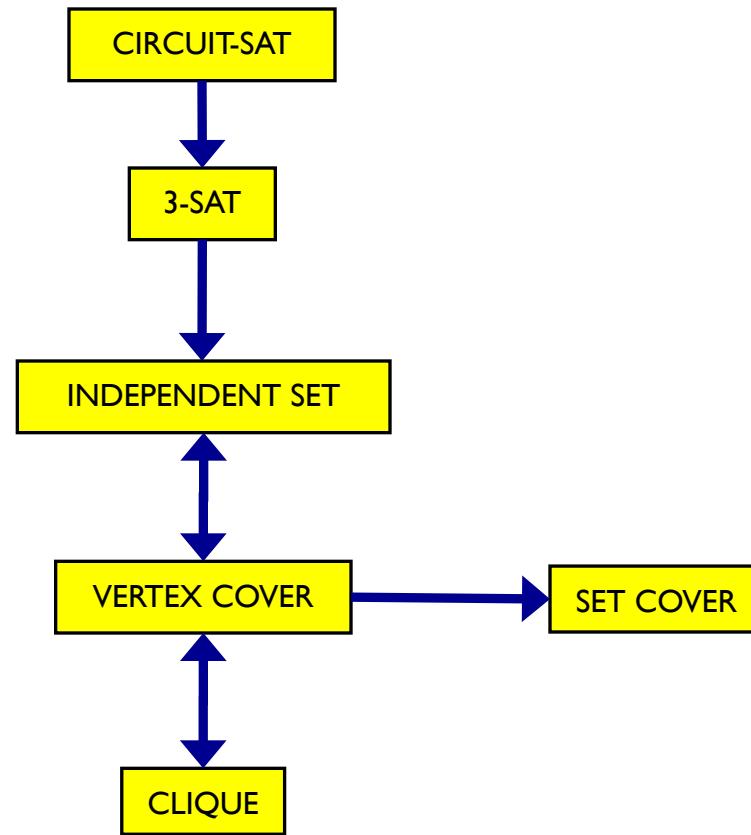
Transitivity: If $X \leq_p Y$ and $Y \leq_p Z$, then $X \leq_p Z$.

Proof idea: Compose the two algorithms.

Example: $\text{3-SAT} \leq_p \text{INDEPENDENT-SET} \leq_p \text{VERTEX-COVER} \leq_p \text{SET-COVER}$

What we showed so far

All problems below are NP-complete:



Some more NP-complete problems

- ▶ **SET-PACKING**: max number of mutually disjoint sets
- ▶ **INT-PROGRAMMING**: linear programming on integers
- ▶ **HAMILTONIAN-CYCLE**
- ▶ **TSP**: traveling salesman problem
- ▶ **3-COLOR**: coloring a graph (even planar!) NB: 2-color is in P
- ▶ **SUBGRAPH-ISOMORPHISM**: given two graphs, is the first one a subgraph of the second?
- ▶ **LCS of multiple strings**
- ▶ **MULTISET-PARTITION**: $\{3,1,1,2,2,1\} \rightarrow \{2,1,1,1\} \cup \{3,2\}$
- ▶ **KNAPSACK**: select a subset of "maximal value" fitting a knapsack
 - ▶ n objects, weights w_1, \dots, w_n , values v_1, \dots, v_n , knapsack weight capacity W
 - ▶ $\max_{S \subseteq 1..n} \{\sum_{i \in S} v_i \mid \sum_{i \in S} w_i \leq W\}$

Are there non-NP-complete problems that are not in P?

- ▶ Most of "natural" NP problems are either in P or NP-complete
- ▶ Notable exceptions:
 - ▶ INTEGER FACTORIZATION
 - ▶ GRAPH-ISOMORPHISM



“I can’t find an efficient algorithm, I guess I’m just too dumb.”

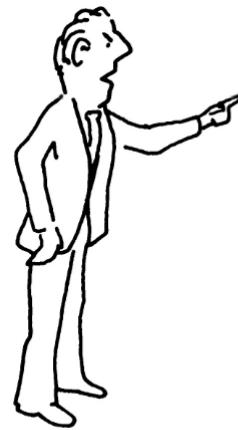


“I can’t find an efficient algorithm, because no such algorithm is possible!”

[Garey & Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.]



"I can't find an efficient algorithm, I guess I'm just too dumb."



"I can't find an efficient algorithm, because no such algorithm is possible!"



"I can't find an efficient algorithm, but neither can all these famous people."

[Garey & Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.]

NP

- ▶ A slight modification can transform a polynomial-time problem into NP-complete

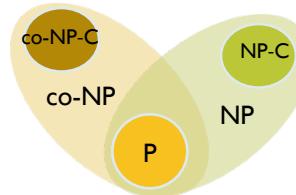
Polynomial	NP-complete
Shortest path	Longest path
2-SAT	3-SAT
2-colorability	3-colorability
Bipartite vertex cover	Vertex cover
Maximum graph matching	3D matching
Minimum cut	Maximum cut
Minimum spanning tree	Degree-constrained spanning tree

NP and co-NP

- ▶ **co-NP:** problems whose complement is in NP

- ▶ *Examples:*

- ▶ UNSATISFIABILITY (TAUTOLOGY)
- ▶ NO-HAMILTONIAN-CYCLE



If $P \neq NP$ and $NP\text{-complete} \neq co\text{-}NP\text{-complete}$

- ▶ **INTEGER FACTORIZATION:** in $NP \cap \text{co-NP}$ but not known to be in P

Coping with NP-completeness

- ▶ If a problem is NP-complete, you design an algorithm to do at most two of the following three things:
 1. Solve the problem exactly
 2. Guarantee to solve the problem in polynomial time
 3. Solve arbitrary instances of the problem
- ▶ **I+2:** solving only small instances (e.g. via pseudo-polynomial algorithms)
 - ▶ cf. fixed-parameter tractability
 - ▶ e.g. Vertex Cover can be solved in $2^k n^{O(1)}$ by simple exhaustive search, where k is the size of Minimum Vertex Cover. Cf <https://pacechallenge.org/2019/vc/>
- ▶ **I+3:** improved exponential-time algorithms, e.g.
 - ▶ 3-SAT can be solved in $O(1.48^n)$ instead of $O(2^n)$
 - ▶ 3-coloring can be solved in $O(1.3289^n)$
- ▶ **2+3: approximation algorithms, heuristics**
- ▶ Note that NP-completeness has also advantageous consequences (cryptography)

Approximation algorithms: examples

▶ VERTEX COVER

- ▶ has an easy 2-approximation algorithm
- ▶ There is no 1.3606-approximation algorithm unless P=NP [Dinur, Safra 2005]
- ▶ SET-COVER can be $\log(n)$ -approximated, where n is the size of the set to be covered (“universe”)
- ▶ KNAPSACK with a running time $O(n^3 / \varepsilon)$ such that the computed solution verifies $V \geq (1 - \varepsilon) \cdot V^*$, where V is the computed total value and V^* the optimal total value (FPTAS)

