ALGORITMY A DATOVÉ STRUKTURY II

Ivana Černá

Jaro 2019

Fakulta informatiky MU

George Pólya: How to Solve It?

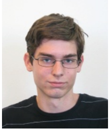
What is the difference between method and device?
A method is a device which you used twice.

Andrew Hamilton, rektor Oxford University, Respekt 7/2015

Je nutné mít na paměti, že my studenty nepřipravujeme na konkrétní povolání, ale trénujeme jejich mysl, aby byli co nejlépe připraveni na změny, jež nás nevyhnutelně čekají, a které budou nejspíš dosti dramatické.

CONTENTS

ADMINISTRATIVE STUFF

Interactive syllabi

`https://is.muni.cz/auth/el/1433/jaro2019/IV003/index.qwarp`

follow discussion groups in IS

LITERATURE

recommended textbooks

- J. Kleinberg, E. Tardos: *Algorithm Design*. Addison-Wesley, 2006.
- T. Cormen, Ch. Leiserson, R. Rivest, C. Stein: *Introduction to Algorithms*. 3rd Edition. MIT Press, 2009.
- S. Dasgupta, Ch. Papadimitriou, U. Vazirani:*Algorithms*. McGraw Hill, 2007.

slides and demo

http://www.cs.princeton.edu/~wayne/kleinberg-tardos/

Part I

Algorithmic Complexity

OVERVIEW

Complexity of Problems and Algorithms

Algorithm Complexity Analysis

- Recursive algorithms
- Iterative algorithms

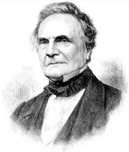
Amortized Complexity

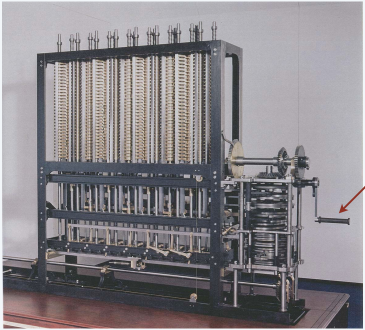
- Amortized analysis

COMPLEXITY OF PROBLEMS AND ALGORITHMS

A strikingly modern thought

" *As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time?* " — *Charles Babbage (1864)*



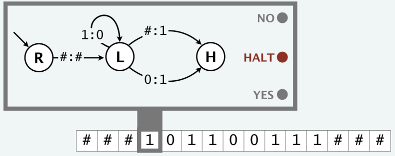


how many times do you have to turn the crank?

Analytic Engine

Models of computation: Turing machines

Deterministic Turing machine. Simple and idealistic model.



Running time. Number of steps.

Memory. Number of tape cells utilized.

Caveat. No random access of memory.

- Single-tape TM requires $\geq n^2$ steps to detect n-bit palindromes.
- Easy to detect palindromes in $\leq cn$ steps on a real computer.

Models of computation: word RAM

Word RAM.

assume $w \geq \log_2 n$

- Each memory location and input/output cell stores a w-bit integer.
- Primitive operations: arithmetic/logic operations, read/write memory, array indexing, following a pointer, conditional branch, ...

constant-time C-style operations
($w = 64$)

Running time. Number of primitive operations.

Memory. Number of memory cells utilized.

Caveat. At times, need more refined model (e.g., multiplying n-bit integers).

Brute force

Brute force. For many nontrivial problems, there is a natural brute-force search algorithm that checks every possible solution.

- Typically takes 2^n steps (or worse) for inputs of size n.
- Unacceptable in practice.

Ex. Stable matching problem: test all $n!$ perfect matchings for stability.

Polynomial running time

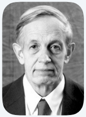
Desirable scaling property. When the input size doubles, the algorithm should slow down by at most some constant factor C.

Def. An algorithm is poly-time if the above scaling property holds.

There exist constants $c > 0$ and $d > 0$ such that,
for every input of size n, the running time of the algorithm
is bounded above by $c\,n^d$ primitive computational steps.

choose $C = 2^d$

von Neumann
(1953)

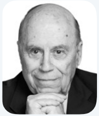


Nash
(1955)

Gödel
(1956)

Cobham
(1964)

Edmonds
(1965)



Rabin
(1966)

Polynomial running time

We say that an algorithm is efficient if it has a polynomial running time.

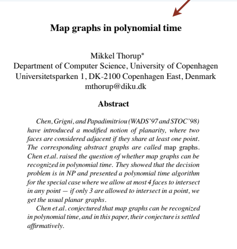
Theory. Definition is (relatively) insensitive to model of computation.

Practice. It really works!

- The poly-time algorithms that people develop have both small constants and small exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions. Some poly-time algorithms in the wild have galactic constants and/or huge exponents.

n^{120}



Map graphs in polynomial time

Mikkel Thorup*
Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
mthorup@diku.dk

Abstract

Chen, Grigni, and Papadimitriou (WADS'97 and STOC'98) have introduced a modified notion of planarity, where two faces are considered adjacent if they share at least one point. The corresponding abstract graphs are called map graphs. *Chen et.al. raised the question of whether map graphs can be recognized in polynomial time. They showed that the decision problem is in NP and presented a polynomial time algorithm for the special case where we allow at most 4 faces to intersect in any point — if only 3 are allowed to intersect in a point, we get the usual planar graphs.*

Chen et.al. conjectured that map graphs can be recognized in polynomial time, and in this paper, their conjecture is settled affirmatively.

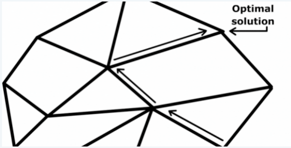
Q. Which would you prefer: $20\,n^{120}$ or $n^{1 + 0.02 \ln n}$?

Worst-case analysis

Worst case. Running time guarantee for any input of size n.

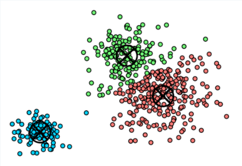
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Exceptions. Some exponential-time algorithms are used widely in practice because the worst-case instances don't arise.



simplex algorithm

Linux grep



k-means algorithm

Other types of analyses

Probabilistic. Expected running time of a randomized algorithm.
Ex. The expected number of compares to quicksort n elements is $\sim 2n \ln n$.

Amortized. Worst-case running time for any sequence of n operations.
Ex. Starting from an empty stack, any sequence of n push and pop operations takes $O(n)$ primitive computational steps using a resizing array.

Also. Average-case analysis, smoothed analysis, competitive analysis, ...

PROBLEM COMPLEXITY

algorithm complexity versus problem complexity

- lower bound of the problem complexity
  - proof techniques
- upper bound of the problem complexity
  - complexity of the particular algorithm solving the problem
- problem complexity
  - given by a lower and upper bound
  - *tight bounds*

LOWER BOUND PROOF TECHNIQUES

- information-theoretic arguments
- decision tree
- problem reduction
- adversary arguments

INFORMATION-THEORETIC ARGUMENTS

based on counting the number of items in the problems' input that must be processed and the number of output items that need to be produced

list all permutations of n-elements sequence

the number of permutation is $n!$; lower bound is $\Omega(n!)$; problem complexity is $\Theta(n!)$

evaluate a polynomial of degree n at a given point x

lower bound is $\Omega(n)$; problem complexity is $\Theta(n)$

product of two n-by-n matrices

lower bound is $\Omega(n^2)$ *(size of the result)*; upper bound is $\mathcal{O}(n^{\log_2 7})$; problem complexity is ???

Traveling Salesperson

lower bound is $\Omega(n^2)$ *(number of graph edges)*; upper bound is exponential; problem complexity is ???

DECISION TREES

- many algorithms work by comparing items of their inputs
- we study performance of such algorithms with a device called the *decision tree*
- decision tree represents computations on inputs of length n
- each internal node represents a key comparison
- node's degree is proportional to the number of possible answers and node's subtrees contain information about subsequent comparisons
- each leaf represents a possible outcome; the number of leaves must be at least as large as the number of possible outcomes
- the algorithms's work on a particular input of size n can be traced by a path from the root to a leaf

DECISION TREES AND LOWER BOUNDS

- a tree with a given number of leaves has to be tall enough
- a tree with degree k and l leaves has depth at least $\lceil \log_k l \rceil$
- $\Omega(\log l(n))$ is the lower bound for problem complexity
  ($l(n)$ is the number of possible outputs for inputs of length n)

DECISION TREES FOR SEARCHING A SORTED ARRAY

Input sorted sequence of numbers $(x_1, \ldots, x_n)$, number x
Output index i such that $x_i = x$ or NONE

- key comparisons
- $n + 1$ possible outcomes

- lower bound $\Omega(\log n)$
- lower bound is *tight*

DECISION TREES FOR SORTING ALGORITHMS

Input sequence of pairwise different numbers $(x_1, \dots, x_n)$
Output permutation Π such that $x_{\Pi(1)} < x_{\Pi(2)} < \dots < x_{\Pi(n)}$

- key comparisons: $x_i < x_j$
- $n!$ possible outcomes *(number of permutations)*

- lower bound $\Omega(\log n!)$
- $\log n! \in \Omega(n \log n)$ *(viz Stirling formulae)*
- lower bound is *tight*

PROBLEM REDUCTION

if	we know a lower bound for problem Q
	and problem Q reduces to problem P
then	lower bound for Q is as well a lower bound for P

Q element uniqueness problem in $(x_1, \ldots, x_n)$?
P Euclidean minimum spanning tree problem

lower bound for the element uniqueness problem is $\Omega(n \log n)$

reduction graph with vertices $(x_1, 0), \ldots (x_n, 0)$
checking whether the minimum spanning tree contains a zero-length edge answers the question about uniqueness of the given numbers

claim lower bound for the Euclidean minimum spanning tree problem is $\Omega(n \log n)$

ADVERSARY ARGUMENTS

The idea is that an all-powerful malicious adversary pretends to choose an input for the algorithm. When the algorithm asks a question about the input, the adversary answers in whatever way will make the algorithm do the most work. If the algorithm does not ask enough queries before terminating, then there will be several different inputs, each consistent with the adversary's answers, that should result in different outputs. In this case, whatever the algorithm outputs, the adversary can 'reveal' an input that is consistent with its answers, but contradists the algorithm's output, an then claim that that was the input that he was using all along.

ADVERSARY FOR THE MAXIMUM PROBLEM

The adversary originally pretends that $x_i = i$ for all i, and answers all comparison queries accordingly. Whenever the adversary reveals that $x_i < x_j$, he marks x_i as an item that the algorithm knows (or should know) is not the maximum element. At most one element x_i is marked after each comparison. Note that x_n is never marked. If the algorithm does less than $n-1$ comparisons before it terminates, the adversary must have at least one other unmarked element $x_k \neq x_n$. In this case, the adversary can change the value of x_k from k to $n+1$ making x_k the largest element, without being inconsistent with any of the comparisons that the algorithm has performed. However, x_n is the maximum element in the original input, and x_k is the largest element in the modified input, so the algorithm cannot possibly give the correct answer for both cases.

Any comparison-based algorithm solving the maximum element problem must perform at least $n-1$ comparisons.

The decision tree model gives lower bound $\log_2 n$.

ADVERSARY FOR THE MAXIM. AND MINIM. PROBLEM

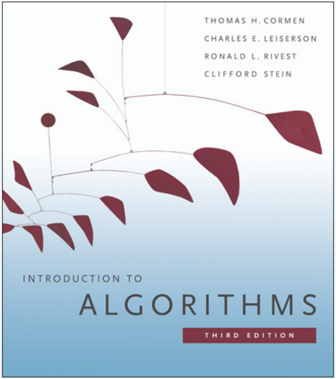
Similar arguments as for the maximum problem. Whenever the adversary reveals that $x_i < x_j$, he marks x_i as an item that the algorithm knows is not the maximum element, and he marks x_j as an item that the algorithm knows is not the minimum element. Whenever two already marked elements are compared, at most one new mark can be added. If the algorithm does less than $\lfloor n/2 \rfloor + n - 2$ comparisons before it terminates, the adversary must have at least two elements that can be both the maximum or both minimum, so the algorithm cannot possibly give the correct answer .

Any comparison-based algorithm solving the maximum and minimum element problem must perform at least $\lfloor n/2 \rfloor + n - 2$ comparisons.

ALGORITHM COMPLEXITY ANALYSIS

ALGORITHM COMPLEXITY ANALYSIS

RECURSIVE ALGORITHMS



SECTION 9.3

5. DIVIDE AND CONQUER

- mergesort
- counting inversions
- randomized quicksort
- *median and selection*
- closest pair of points

Median and selection problems

Selection. Given n elements from a totally ordered universe, find k^{th} smallest.

- Minimum: $k = 1$; maximum: $k = n$.
- Median: $k = \lfloor (n + 1) / 2 \rfloor$.
- $O(n)$ compares for min or max.
- $O(n \log n)$ compares by sorting.
- $O(n \log k)$ compares with a binary heap. ⟵ max heap with k smallest

Applications. Order statistics; find the "top k"; bottleneck paths, …

Q. Can we do it with $O(n)$ compares?

A. Yes! Selection is easier than sorting.

Randomized quickselect

- Pick a random pivot element $p \in A$.
- 3-way partition the array into L, M, and R.
- Recur in one subarray—the one containing the k^{th} smallest element.

QUICK-SELECT(A, k)

Pick pivot $p \in A$ uniformly at random.

$(L, M, R) \leftarrow$ PARTITION-3-WAY(A, p). ⟵ $\Theta(n)$

IF $(k \le |L|)$ RETURN QUICK-SELECT(L, k). ⟵ $T(i)$

ELSE IF $(k > |L| + |M|)$ RETURN QUICK-SELECT$(R, k - |L| - |M|)$ ⟵ $T(n - i - 1)$

ELSE IF $(k = |L|)$ RETURN p.

Randomized quickselect analysis

Intuition. Split candy bar uniformly $\Rightarrow$ expected size of larger piece is ¾.

$$T(n) \le T(3n/4) + n \quad \Rightarrow \quad T(n) \le 4n$$

not rigorous: can't assume
$\mathbf{E}[T(i)] \le T(\mathbf{E}[i])$

Def. $T(n, k)$ = expected # compares to select k^{th} smallest in array of length $\le n$.

Def. $T(n) = \max_k T(n, k)$.

Proposition. $T(n) \le 4n$.

Pf. [by strong induction on n]

- Assume true for $1, 2, \ldots, n-1$.
- $T(n)$ satisfies the following recurrence:

can assume we always recur of larger of two subarrays since $T(n)$ is monotone non-decreasing

$$\begin{aligned} T(n) &\le n + 1/n\,[\; 2T(n/2) + \ldots + 2T(n-3) + 2T(n-2) + 2T(n-1)\;] \\ &\le n + 1/n\,[\; 8(n/2) + \ldots + 8(n-3) + 8(n-2) + 8(n-1)\;] \\ &\le n + 1/n\,(3n^2) \\ &= 4n. \quad \blacksquare \end{aligned}$$

inductive hypothesis

tiny cheat: sum should start at $T(\lfloor n/2 \rfloor)$

Selection in worst-case linear time

Goal. Find pivot element p that divides list of n elements into two pieces so that each piece is guaranteed to have $\leq$ $7/10\ n$ elements.

Q. How to find approximate median in linear time?
A. Recursively compute median of sample of $\leq$ $2/10\ n$ elements.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(7/10\ n) + T(2/10\ n) + \Theta(n) & \text{otherwise} \end{cases}$$

two subproblems
of different sizes!

$$\Rightarrow T(n) = \Theta(n)$$

we'll need to show this

Choosing the pivot element

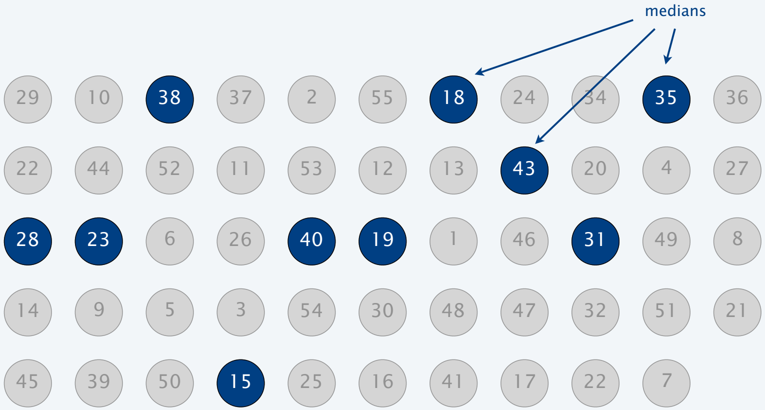
- Divide n elements into $\lfloor n / 5 \rfloor$ groups of 5 elements each (plus extra).

29	10	38	37	2	55	18	24	34	35	36
22	44	52	11	53	12	13	43	20	4	27
28	23	6	26	40	19	1	46	31	49	8
14	9	5	3	54	30	48	47	32	51	21
45	39	50	15	25	16	41	17	22	7	

n = 54

Choosing the pivot element

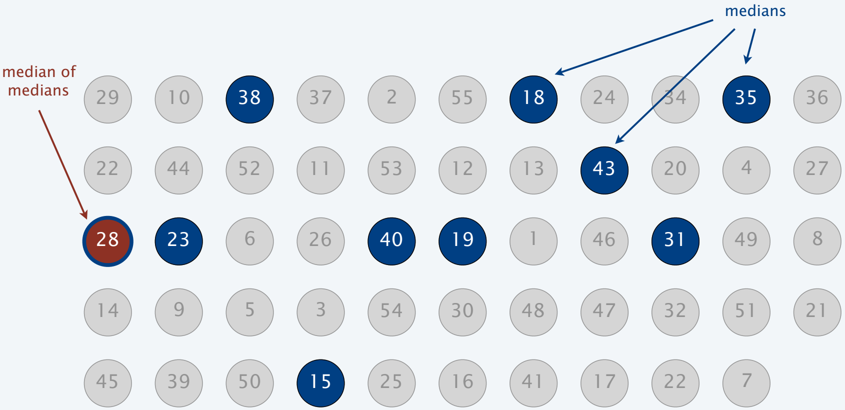
- Divide n elements into $\lfloor n / 5 \rfloor$ groups of 5 elements each (plus extra).
- Find median of each group (except extra).



n = 54

Choosing the pivot element

- Divide n elements into $\lfloor n / 5 \rfloor$ groups of 5 elements each (plus extra).
- Find median of each group (except extra).
- Find median of $\lfloor n / 5 \rfloor$ medians recursively.
- Use median-of-medians as pivot element.



n = 54

Median-of-medians selection algorithm

MOM-SELECT(A, k)

$n \leftarrow |A|$.

IF $(n < 50)$

  RETURN k^{th} smallest of element of A via mergesort.

Group A into $\lfloor n / 5 \rfloor$ groups of 5 elements each (ignore leftovers).

$B \leftarrow$ median of each group of 5.

$p \leftarrow$ MOM-SELECT$(B, \lfloor n / 10 \rfloor)$ ⟵ median of medians

$(L, M, R) \leftarrow$ PARTITION-3-WAY(A, p).

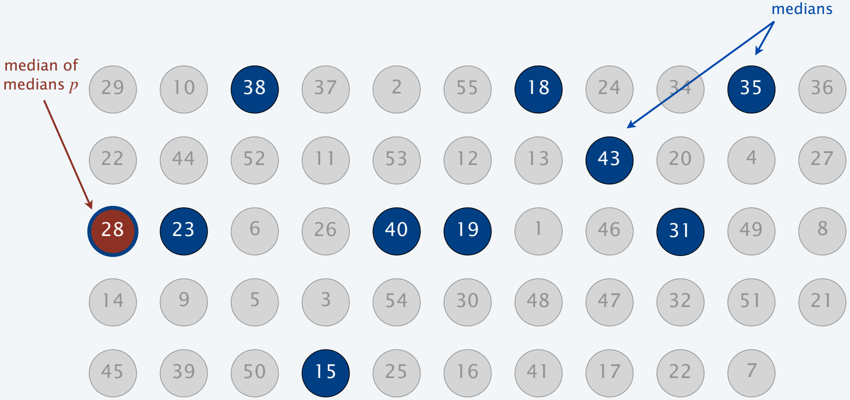
IF $(k \leq |L|)$ RETURN MOM-SELECT(L, k).

ELSE IF $(k > |L| + |M|)$ RETURN MOM-SELECT$(R, k - |L| - |M|)$

ELSE RETURN p.

Analysis of median-of-medians selection algorithm

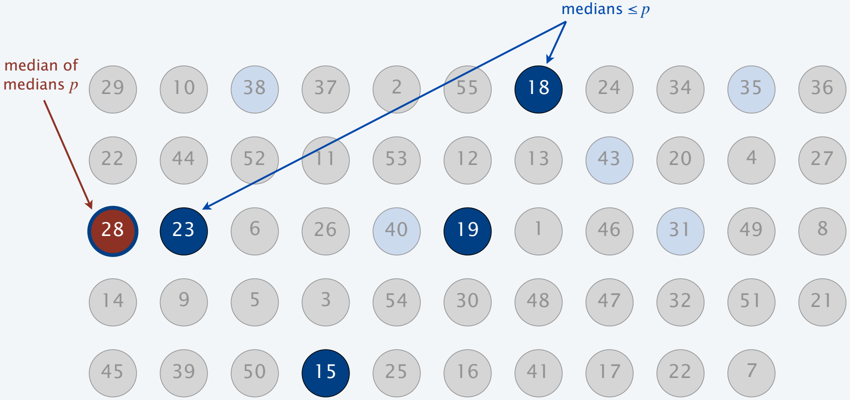
- At least half of 5-element medians ≤ p.



n = 54

Analysis of median-of-medians selection algorithm

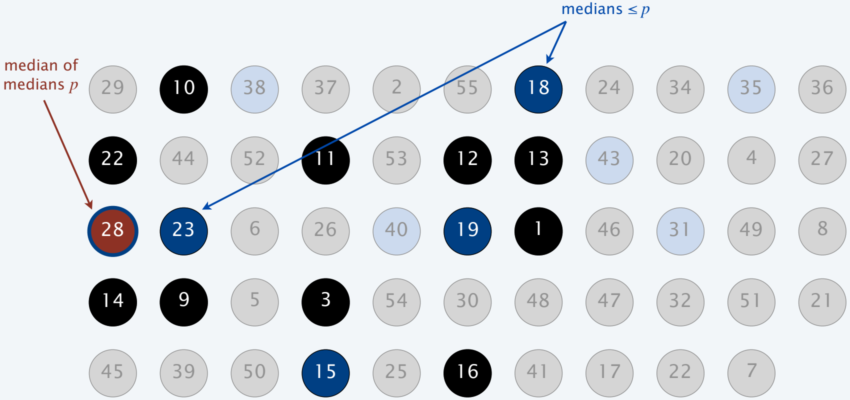
- At least half of 5-element medians $\leq$ p.
- At least $\lfloor \lfloor n / 5 \rfloor / 2 \rfloor = \lfloor n / 10 \rfloor$ medians $\leq$ p.



n = 54

Analysis of median-of-medians selection algorithm

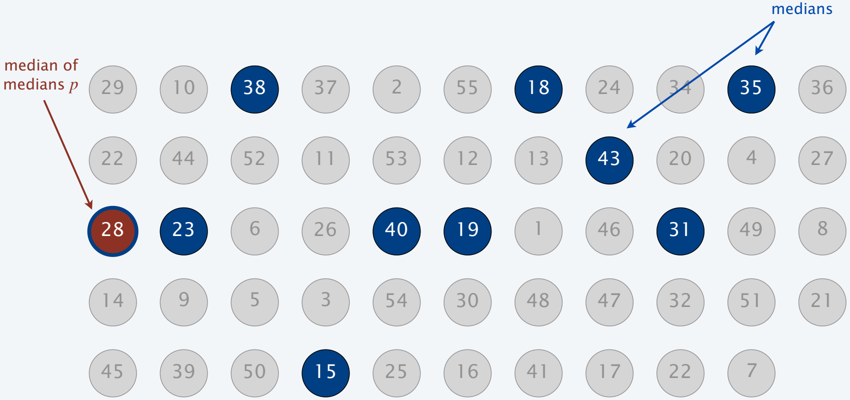
- At least half of 5-element medians $\leq p$.
- At least $\lfloor \lfloor n / 5 \rfloor / 2 \rfloor = \lfloor n / 10 \rfloor$ medians $\leq p$.
- At least $3 \lfloor n / 10 \rfloor$ elements $\leq p$.



n = 54

Analysis of median-of-medians selection algorithm

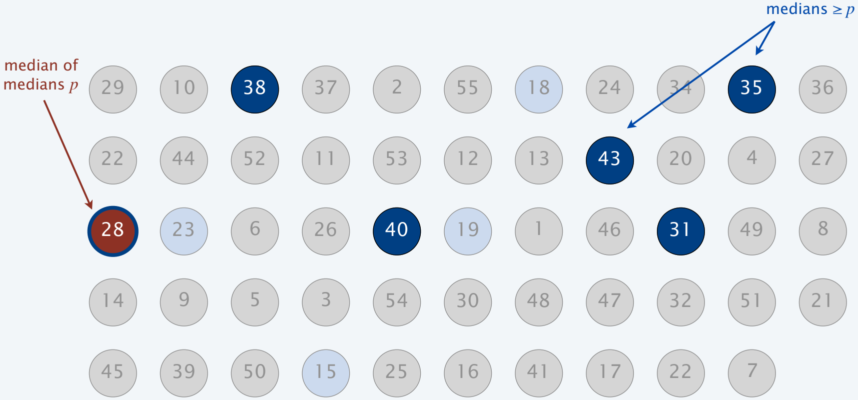
- At least half of 5-element medians ≥ p.



n = 54

Analysis of median-of-medians selection algorithm

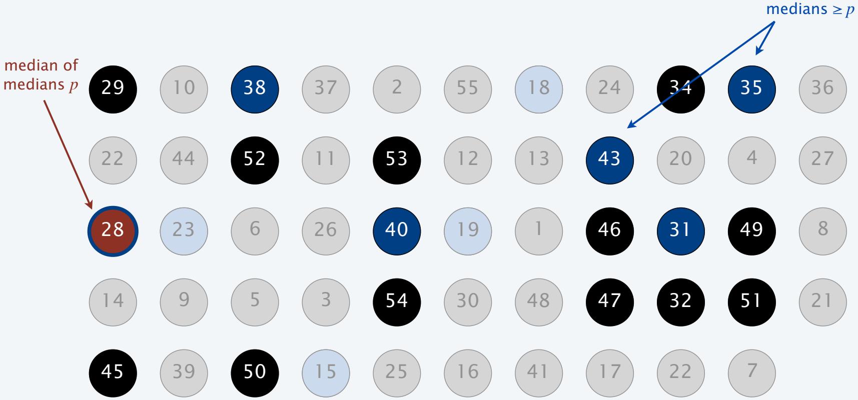
- At least half of 5-element medians $\geq p$.
- At least $\lfloor \lfloor n / 5 \rfloor / 2 \rfloor = \lfloor n / 10 \rfloor$ medians $\geq p$.



n = 54

Analysis of median-of-medians selection algorithm

- At least half of 5-element medians ≥ p.
- At least $\lfloor \lfloor n / 5 \rfloor / 2 \rfloor = \lfloor n / 10 \rfloor$ medians ≥ p.
- At least $3 \lfloor n / 10 \rfloor$ elements ≥ p.



n = 54

Median-of-medians selection algorithm recurrence

Median-of-medians selection algorithm recurrence.

- Select called recursively with $\lfloor n / 5 \rfloor$ elements to compute MOM p.
- At least $3 \lfloor n / 10 \rfloor$ elements $\leq p$.
- At least $3 \lfloor n / 10 \rfloor$ elements $\geq p$.
- Select called recursively with at most $n - 3 \lfloor n / 10 \rfloor$ elements.

Def. $C(n)$ = max # compares on any array of n elements.

$$C(n) \leq \underbrace{C(\lfloor n/5 \rfloor)}_{\text{median of medians}} + \underbrace{C(n - 3\lfloor n/10 \rfloor)}_{\text{recursive select}} + \tfrac{11}{5}n$$

computing median of 5 (≤ 6 compares per group)

partitioning (≤ n compares)

Intuition.

- $C(n)$ is going to be at least linear in $n \Rightarrow C(n)$ is super-additive.
- Ignoring floors, this implies that

$$\begin{aligned} C(n) &\leq C(n/5 + n - 3n/10) + 11/5\, n \\ &= C(9n/10) + 11/5\, n \\ &\Rightarrow C(n) \leq 22n. \end{aligned}$$

Median-of-medians selection algorithm recurrence

Median-of-medians selection algorithm recurrence.

- Select called recursively with $\lfloor n / 5 \rfloor$ elements to compute MOM p.
- At least $3 \lfloor n / 10 \rfloor$ elements $\le p$.
- At least $3 \lfloor n / 10 \rfloor$ elements $\ge p$.
- Select called recursively with at most $n - 3 \lfloor n / 10 \rfloor$ elements.

Def. $C(n)$ = max # compares on any array of n elements.

$$C(n) \le \underbrace{C(\lfloor n/5 \rfloor)}_{\text{median of medians}} + \underbrace{C(n - 3\lfloor n/10 \rfloor)}_{\text{recursive select}} + \tfrac{11}{5}n$$

computing median of 5 (≤ 6 compares per group)

partitioning (≤ n compares)

Now, let's solve given recurrence.

- Assume n is both a power of 5 and a power of 10 ?
- Prove that $C(n)$ is monotone non-decreasing.

Consider the following recurrence

$$C(n) \;=\; \begin{cases} 0 & \text{if } n \le 1 \\ C(\lfloor n/5 \rfloor) + C(n - 3\lfloor n/10 \rfloor) + \frac{11}{5}n & \text{if } n > 1 \end{cases}$$

Is $C(n)$ monotone non-decreasing?

A. Yes, obviously.

B. Yes, but proof is tedious.

C. Yes, but proof is hard.

D. No.

Median-of-medians selection algorithm recurrence

Analysis of selection algorithm recurrence.

- $T(n)$ = max # compares on any array of $\leq n$ elements.
- $T(n)$ is monotone non-decreasing, but $C(n)$ is not!

$$T(n) \;\leq\; \begin{cases} 6n & \text{if } n < 50 \\ \max\{\, T(n-1),\; T(\lfloor n/5 \rfloor) + T(n - 3\lfloor n/10 \rfloor) + \tfrac{11}{5}n) \,\} & \text{if } n \geq 50 \end{cases}$$

Claim. $T(n) \leq 44\,n$.

Pf. [by strong induction]

- Base case: $T(n) \leq 6\,n$ for $n < 50$ (mergesort).
- Inductive hypothesis: assume true for $1, 2, \ldots, n-1$.
- Induction step: for $n \geq 50$, we have either $T(n) \leq T(n-1) \leq 44\,n$ or

$$\begin{aligned} T(n) &\leq T(\lfloor n/5 \rfloor) + T(n - 3\lfloor n/10 \rfloor) + 11/5\,n \\ &\leq 44\,(\lfloor n/5 \rfloor) + 44\,(n - 3\lfloor n/10 \rfloor) + 11/5\,n \\ &\leq 44\,(n/5) + 44\,n - 44\,(n/4) + 11/5\,n \\ &= 44\,n. \quad \blacksquare \end{aligned}$$

inductive hypothesis

for $n \geq 50,\; 3\lfloor n/10 \rfloor \geq n/4$

Divide-and-conquer: quiz 5

Suppose that we divide n elements into $\lfloor n / r \rfloor$ groups of r elements each, and use the median-of-medians of these $\lfloor n / r \rfloor$ groups as the pivot. For which r is the worst-case running time of select $O(n)$?

A. $r = 3$

B. $r = 7$

C. Both A and B.

D. Neither A nor B.

Linear-time selection retrospective

Proposition. [Blum–Floyd–Pratt–Rivest–Tarjan 1973] There exists a compare-based selection algorithm whose worst-case running time is $O(n)$.

Time Bounds for Selection*

MANUEL BLUM, ROBERT W. FLOYD, VAUGHAN PRATT,
RONALD L. RIVEST, AND ROBERT E. TARJAN

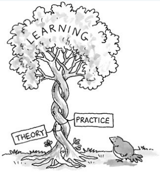
Department of Computer Science, Stanford University, Stanford, California 94305

Received November 14, 1972

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm—PICK. Specifically, no more than $5.430\dot{5}\, n$ comparisons are ever required. This bound is improved for extreme values of i, and a new lower bound on the requisite number of comparisons is also proved.

Theory.

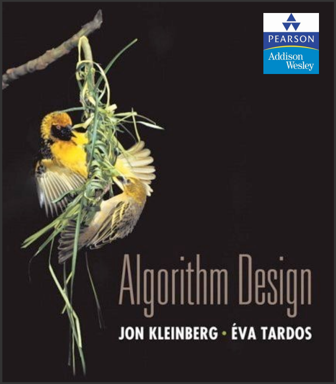
- Optimized version of BFPRT: $\le 5.4305\, n$ compares.
- Upper bound: [Dor–Zwick 1995] $\le 2.95\, n$ compares.
- Lower bound: [Dor–Zwick 1999] $\ge (2 + 2^{-80})\, n$ compares.



Practice. Constants too large to be useful.

ALGORITHM COMPLEXITY ANALYSIS

ITERATIVE ALGORITHMS

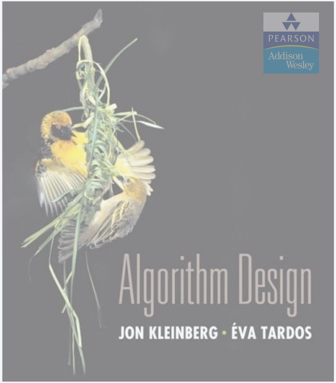


1. Stable Matching

- *stable matching problem*
- *Gale–Shapley algorithm*
- *hospital optimality*
- *context*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos



SECTION 1.1

1. STABLE MATCHING

- *stable matching problem*
- *Gale–Shapley algorithm*
- *hospital optimality*
- *context*

Matching med-school students to hospitals

Goal. Given a set of preferences among hospitals and med-school students, design a self-reinforcing admissions process.

Unstable pair. Hospital h and student s form an unstable pair if both:

- h prefers s to one of its admitted students.
- s prefers h to assigned hospital.

Stable assignment. Assignment with no unstable pairs.

- Natural and desirable condition.
- Individual self-interest prevents any hospital–student side deal.

Stable matching problem: input

Input. A set of n hospitals H and a set of n students S.

- Each hospital $h \in H$ ranks students.
- Each student $s \in S$ ranks hospitals.

one student per hospital (for now)

favorite — least favorite

	1st	2nd	3rd
Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

hospitals' preference lists

favorite — least favorite

	1st	2nd	3rd
Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

students' preference lists

Perfect matching

Def. A matching M is a set of ordered pairs h–s with $h \in H$ and $s \in S$ s.t.

- Each hospital $h \in H$ appears in at most one pair of M.
- Each student $s \in S$ appears in at most one pair of M.

Def. A matching M is perfect if $|M| = |H| = |S| = n$.

	1st	2nd	3rd
Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

	1st	2nd	3rd
Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

a perfect matching M = { A–Z, B–Y, C–X }

Unstable pair

Def. Given a perfect matching M, hospital h and student s form an unstable pair if both:

- h prefers s to matched student.
- s prefers h to matched hospital.

Key point. An unstable pair h–s could each improve by joint action.

	1st	2nd	3rd
Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

	1st	2nd	3rd
Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

A–Y is an unstable pair for matching M = { A–Z, B–Y, C–X }

Which pair is unstable in the matching { A–X, B–Z, C–Y } ?

A. A–Y.

B. B–X.

C. B–Z.

D. None of the above.

	1st	2nd	3rd
Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

	1st	2nd	3rd
Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

Stable matching problem

Def. A stable matching is a perfect matching with no unstable pairs.

Stable matching problem. Given the preference lists of n hospitals and n students, find a stable matching (if one exists).

	1st	2nd	3rd
Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

	1st	2nd	3rd
Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

a stable matching M = { A–X, B–Y, C–Z }

Stable roommate problem

Q. Do stable matchings always exist?

A. Not obvious a priori.

Stable roommate problem.

- $2n$ people; each person ranks others from 1 to $2n - 1$.
- Assign roommate pairs so that no unstable pairs.

	1st	2nd	3rd
A	B	C	D
B	C	A	D
C	A	B	D
D	A	B	C

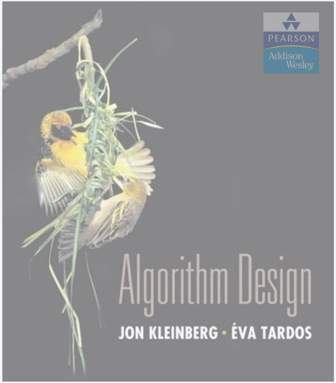
no perfect matching is stable

A–B, C–D $\Rightarrow$ B–C unstable

A–C, B–D $\Rightarrow$ A–B unstable

A–D, B–C $\Rightarrow$ A–C unstable

Observation. Stable matchings need not exist.



SECTION 1.1

1. STABLE MATCHING

- stable matching problem
- Gale–Shapley algorithm
- hospital optimality
- context

Gale–Shapley deferred acceptance algorithm

An intuitive method that guarantees to find a stable matching.

GALE–SHAPLEY (*preference lists for hospitals and students*)

INITIALIZE M to empty matching.

WHILE (some hospital h is unmatched and hasn't proposed to every student)

$s \leftarrow$ first student on h's list to whom h has not yet proposed.

IF (s is unmatched)

Add h–s to matching M.

ELSE IF (s prefers h to current partner h')

Replace h'–s with h–s in matching M.

ELSE

s rejects h.

RETURN stable matching M.

Proof of correctness: termination

Observation 1. Hospitals propose to students in decreasing order of preference.

Observation 2. Once a student is matched, the student never becomes unmatched; only "trades up."

Claim. Algorithm terminates after at most n^2 iterations of WHILE loop.
Pf. Each time through the WHILE loop, a hospital proposes to a new student. Thus, there are at most n^2 possible proposals. ▪

	1st	2nd	3rd	4th	5th
A	V	W	X	Y	Z
B	W	X	Y	V	Z
C	X	Y	V	W	Z
D	Y	V	W	X	Z
E	V	W	X	Y	Z

	1st	2nd	3rd	4th	5th
V	B	C	D	E	A
W	C	D	E	A	B
X	D	E	A	B	C
Y	E	A	B	C	D
Z	A	B	C	D	E

n(n−1) + 1 proposals

Proof of correctness: perfect matching

Claim. Gale–Shapley outputs a matching.

Pf.

- Hospital proposes only if unmatched. $\Rightarrow$ matched to ≤ 1 student
- Student keeps only best hospital. $\Rightarrow$ matched to ≤ 1 hospital

Claim. In Gale–Shapley matching, all hospitals get matched.

Pf. [by contradiction]

- Suppose, for sake of contradiction, that some hospital $h \in H$ is unmatched upon termination of Gale–Shapley algorithm.
- Then some student, say $s \in S$, is unmatched upon termination.
- By Observation 2, s was never proposed to.
- But, h proposes to every student, since h ends up unmatched. ※

Claim. In Gale–Shapley matching, all students get matched.

Pf. [by counting]

- By previous claim, all n hospitals get matched.
- Thus, all n students get matched. ▪

Proof of correctness: stability

Claim. In Gale–Shapley matching M^*, there are no unstable pairs.

Pf. Consider any pair h–s that is not in M^*.

- Case 1: h never proposed to s.
  - $\Rightarrow$ h prefers its Gale–Shapley partner s' to s. ← hospitals propose in decreasing order of preference
  - $\Rightarrow$ h–s is not unstable.
- Case 2: h proposed to s.
  - $\Rightarrow$ s rejected h (either right away or later)
  - $\Rightarrow$ s prefers Gale–Shapley partner h' to h.
  - $\Rightarrow$ h–s is not unstable.

students only trade up

- In either case, the pair h–s is not unstable. ▪

$h - s'$

$h' - s$

$\vdots$

Gale–Shapley matching M*

Summary

Stable matching problem. Given n hospitals and n students, and their preference lists, find a stable matching if one exists.

Theorem. [Gale–Shapley 1962] The Gale–Shapley algorithm guarantees to find a stable matching for any problem instance.

COLLEGE ADMISSIONS AND THE STABILITY OF MARRIAGE

D. GALE* AND L. S. SHAPLEY, Brown University and the RAND Corporation

1. Introduction. The problem with which we shall be concerned relates to the following typical situation: A college is considering a set of n applicants of which it can admit a quota of only q. Having evaluated their qualifications, the admissions office must decide which ones to admit. The procedure of offering admission only to the q best-qualified applicants will not generally be satisfactory, for it cannot be assumed that all who are offered admission will accept. Accordingly, in order for a college to receive q acceptances, it will generally have to offer to admit more than q applicants. The problem of determining how many and which ones to admit requires some rather involved guesswork. It may not be known (a) whether a given applicant has also applied elsewhere; if this is known it may not be known (b) how he ranks the colleges to which he has applied; even if this is known it will not be known (c) which of the other colleges will offer to admit him. A result of all this uncertainty is that colleges can expect only that the entering class will come reasonably close in numbers to the desired quota, and be reasonably close to the attainable optimum in quality.

Stable matching: quiz 2

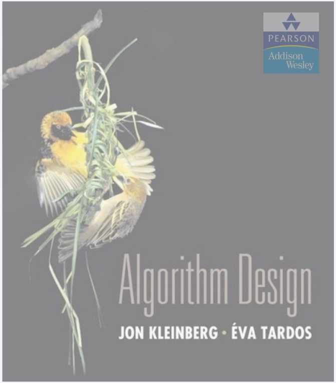
Do all executions of Gale–Shapley lead to the same stable matching?

A. No, because the algorithm is nondeterministic.

B. No, because an instance can have several stable matchings.

C. Yes, because each instance has a unique stable matching.

D. Yes, even though an instance can have several stable matchings and the algorithm is nondeterministic.



SECTION 1.1

1. STABLE MATCHING

- *stable matching problem*
- *Gale–Shapley algorithm*
- *hospital optimality*
- *context*

Understanding the solution

For a given problem instance, there may be several stable matchings.

	1st	2nd	3rd
A	X	Y	Z
B	Y	X	Z
C	X	Y	Z

	1st	2nd	3rd
X	B	A	C
Y	A	B	C
Z	A	B	C

an instance with two stable matchings: S = { A–X, B–Y, C–Z } and S′ = { A–Y, B–X, C–Z }

Understanding the solution

Def. Student s is a valid partner for hospital h if there exists any stable matching in which h and s are matched.

Ex.

- Both X and Y are valid partners for A.
- Both X and Y are valid partners for B.
- Z is the only valid partner for C.

	1st	2nd	3rd
A	X	Y	Z
B	Y	X	Z
C	X	Y	Z

	1st	2nd	3rd
X	B	A	C
Y	A	B	C
Z	A	B	C

an instance with two stable matchings: S = { A–X, B–Y, C–Z } and S′ = { A–Y, B–X, C–Z }

Stable matching: quiz 3

Who is the best valid partner for W in the following instance?

A.

B.

C.

D.

6 stable matchings

{ A–W, B–X, C–Y, D–Z }

{ A–X, B–W, C–Y, D–Z }

{ A–X, B–Y, C–W, D–Z }

{ A–Z, B–W, C–Y, D–X }

{ A–Z, B–Y, C–W, D–X }

{ A–Y, B–Z, C–W, D–X }

	1st	2nd	3rd	4th
A	Y	Z	X	W
B	Z	Y	W	X
C	W	Y	X	Z
D	X	Z	W	Y

	1st	2nd	3rd	4th
W	D	A	B	C
X	C	B	A	D
Y	C	B	A	D
Z	D	A	B	C

Understanding the solution

Def. Student s is a valid partner for hospital h if there exists any stable matching in which h and s are matched.

Hospital-optimal assignment. Each hospital receives best valid partner.

- Is it a perfect matching?
- Is it stable?

Claim. All executions of Gale–Shapley yield hospital-optimal assignment.
Corollary. Hospital-optimal assignment is a stable matching!

Hospital optimality

Claim. Gale–Shapley matching M^* is hospital-optimal.

Pf. [by contradiction]

- Suppose a hospital is matched with student other than best valid partner.
- Hospitals propose in decreasing order of preference.
  $\Rightarrow$ some hospital is rejected by a valid partner during Gale–Shapley
- Let h be first such hospital, and let s be the first valid partner that rejects h.
- Let M be a stable matching where h and s are matched.
- When s rejects h in Gale–Shapley, s forms (or re-affirms) commitment to a hospital, say h'.
  $\Rightarrow$ $\boxed{s \text{ prefers } h' \text{ to } h.}$
- Let s' be partner of h' in M.
- h' had not been rejected by any valid partner (including s') at the point when h is rejected by s. ← because this is the first rejection by a valid partner
- Thus, h' had not yet proposed to s' when h' proposed to s.
  $\Rightarrow$ $\boxed{h' \text{ prefers } s \text{ to } s'.}$
- Thus, h'–s is unstable in M, a contradiction. ▪

$h - s$

$h' - s'$

$\vdots$

stable matching M

Student pessimality

Q. Does hospital-optimality come at the expense of the students?
A. Yes.

Student-pessimal assignment. Each student receives worst valid partner.

Claim. Gale–Shapley finds student-pessimal stable matching M^*.
Pf. [by contradiction]

- Suppose h–s matched in M^* but h is not the worst valid partner for s.
- There exists stable matching M in which s is paired with a hospital, say h', whom s prefers less than h.
  $\Rightarrow$ $\boxed{s \text{ prefers } h \text{ to } h'.}$
- Let s' be the partner of h in M.
- By hospital-optimality, s is the best valid partner for h.
  $\Rightarrow$ $\boxed{h \text{ prefers } s \text{ to } s'.}$
- Thus, h–s is an unstable pair in M, a contradiction. ▪

$h' - s$

$h - s'$

$\vdots$

stable matching M

Stable matching: quiz 4

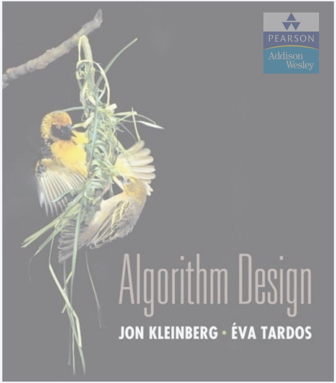
Suppose each agent knows the preference lists of every other agent before the hospital propose-and-reject algorithm is executed. Which is true?

A. No hospital can improve by falsifying its preference list.

B. No student can improve by falsifying their preference list.

C. Both A and B.

D. Neither A nor B.



SECTION 1.1

1. STABLE MATCHING

- *stable matching problem*
- *Gale–Shapley algorithm*
- *hospital optimality*
- *context*

Extensions

Extension 1. Some agents declare others as unacceptable.

med-school student unwilling to work in Cleveland

Extension 2. Some hospitals have more than one position.

Extension 3. Unequal number of positions and students.

≥ 43K med-school students; only 31K positions

Def. Matching M is unstable if there is a hospital h and student s such that:

- h and s are acceptable to each other; and
- Either s is unmatched, or s prefers h to assigned hospital; and
- Either h does not have all its places filled, or h prefers s to at least one of its assigned students.

Theorem. There exists a stable matching.

Pf. Straightforward generalization of Gale–Shapley algorithm.

Historical context

National resident matching program (NRMP).

- Centralized clearinghouse to match med-school students to hospitals.
- Began in 1952 to fix unraveling of offer dates.

hospitals began making offers earlier and earlier, up to 2 years in advance

- Originally used the "Boston Pool" algorithm.
- Algorithm overhauled in 1998.
  - med-school student optimal
  - deals with various side constraints (e.g., allow couples to match together)

stable matching no longer guaranteed to exist

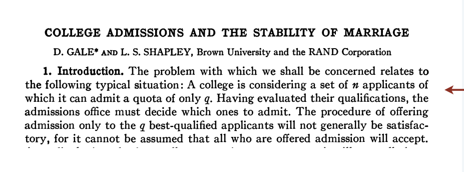
The Redesign of the Matching Market for American Physicians: Some Engineering Aspects of Economic Design

By Alvin E. Roth and Elliott Peranson*

We report on the design of the new clearinghouse adopted by the National Resident Matching Program, which annually fills approximately 20,000 jobs for new physicians. Because the market has complementarities between applicants and between positions, the theory of simple matching markets does not apply directly. However, computational experiments show the theory provides good approximations. Furthermore, the set of stable matchings, and the opportunities for strategic manipulation, are surprisingly small. A new kind of "core convergence" result explains this; that each applicant interviews only a small fraction of available positions is important. We also describe engineering aspects of the design process. (JEL C78, B41, J44)

2012 Nobel Prize in Economics

Lloyd Shapley. Stable matching theory and Gale–Shapley algorithm.



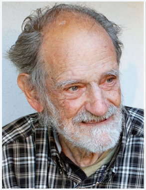
COLLEGE ADMISSIONS AND THE STABILITY OF MARRIAGE

D. GALE* AND L. S. SHAPLEY, Brown University and the RAND Corporation

1. Introduction. The problem with which we shall be concerned relates to the following typical situation: A college is considering a set of n applicants of which it can admit a quota of only q. Having evaluated their qualifications, the admissions office must decide which ones to admit. The procedure of offering admission only to the q best-qualified applicants will not generally be satisfactory, for it cannot be assumed that all who are offered admission will accept.

original applications:
college admissions and
opposite-sex marriage

Alvin Roth. Applied Gale–Shapley to matching med-school students with hospitals, students with schools, and organ donors with patients.



Lloyd Shapley **Alvin Roth**

New York City high school match

8th grader. Ranks top-5 high schools.

High school. Ranks students (and limit).

Goal. Match 90K students to 500 high school programs.

How Game Theory Helped Improve New York City's High School Application Process

By TRACY TULLIS DEC. 5, 2014

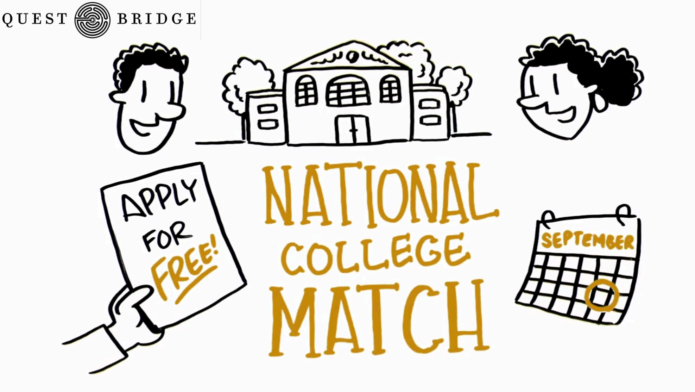
Tuesday was the deadline for eighth graders in New York City to submit applications to secure a spot at one of 426 public high schools. After months of school tours and tests, auditions and interviews, 75,000 students have entrusted their choices to a computer program that will arrange their school assignments for the coming year. The weeks of research and deliberation will be reduced to a fraction of a second of mathematical calculation: In just a couple of hours, all the sorting for the Class of 2019 will be finished.

Questbridge national college match

Low-income student. Ranks colleges.

College. Ranks students willing to admit (and limit).

Goal. Match students to colleges.



A modern application

Content delivery networks. Distribute much of world's content on web.

User. Preferences based on latency and packet loss.
Web server. Preferences based on costs of bandwidth and co-location.
Goal. Assign billions of users to servers, every 10 seconds.

Algorithmic Nuggets in Content Delivery

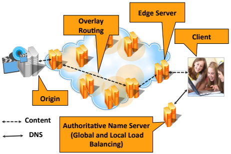
Bruce M. Maggs
Duke and Akamai
bmm@cs.duke.edu

Ramesh K. Sitaraman
UMass, Amherst and Akamai
ramesh@cs.umass.edu

This article is an editorial note submitted to CCR. It has NOT been peer reviewed.
The authors take full responsibility for this article's technical content. Comments can be posted through CCR Online.

ABSTRACT

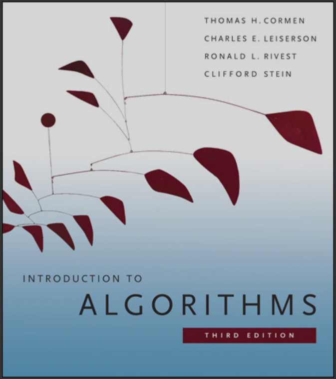
This paper "peeks under the covers" at the subsystems that provide the basic functionality of a leading content delivery network. Based on our experiences in building one of the largest distributed systems in the world, we illustrate how sophisticated algorithmic research has been adapted to balance the load between and within server clusters, manage the caches on servers, select paths through an overlay routing network, and elect leaders in various contexts. In each instance, we first explain the theory underlying the algorithms, then introduce practical considerations not captured by the theoretical models, and finally describe what is implemented in practice. Through these examples, we highlight the role of algorithmic research in the design of complex networked systems. The paper also illustrates the close synergy that exists between research and industry where research ideas cross over into products and product requirements drive future research.



AMORTIZED COMPLEXITY

AMORTIZED COMPLEXITY

AMORTIZED ANALYSIS



Data structures I, II, III, and IV

I. Amortized Analysis

II. Binary and Binomial Heaps

III. Fibonacci Heaps

IV. Union–Find

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

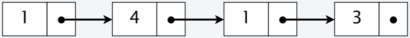
Last updated on 2/2/18 5:50 AM

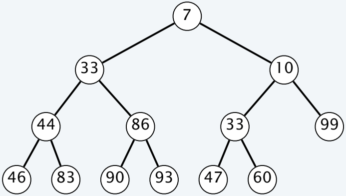
Data structures

Static problems. Given an input, produce an output.
Ex. Sorting, FFT, edit distance, shortest paths, MST, max-flow, ...

Dynamic problems. Given a sequence of operations (given one at a time), produce a sequence of outputs.
Ex. Stack, queue, priority queue, symbol table, union–find, ...

Algorithm. Step-by-step procedure to solve a problem.
Data structure. Way to store and organize data.
Ex. Array, linked list, binary heap, binary search tree, hash table, ...





Appetizer

Goal. Design a data structure to support all operations in $O(1)$ time.

- INIT(n): create and return an initialized array (all zero) of length n.
- READ(A, i): return element i in array.
- WRITE(A, i, $value$): set element i in array to $value$.

Assumptions.

true in C or C++, but not Java

- Can MALLOC an uninitialized array of length n in $O(1)$ time.
- Given an array, can read or write element i in $O(1)$ time.

Remark. An array does INIT in $\Theta(n)$ time and READ and WRITE in $\Theta(1)$ time.

Appetizer

Data structure. Three arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer k.

- $A[i]$ stores the current value for READ (if initialized).
- k = number of initialized entries.
- $C[j]$ = index of j^{th} initialized element for $j = 1, \ldots, k$.
- If $C[j] = i$, then $B[i] = j$ for $j = 1, \ldots, k$.

Theorem. $A[i]$ is initialized iff both $1 \leq B[i] \leq k$ and $C[B[i]] = i$.

Pf. Ahead.

	1	2	3	4	5	6	7	8
A[ ]	?	22	55	99	?	33	?	?
B[ ]	?	3	4	1	?	2	?	?
C[ ]	4	6	2	3	?	?	?	?

k = 4

A[4]=99, A[6]=33, A[2]=22, and A[3]=55 initialized in that order

Appetizer

INIT (A, n)

$k \leftarrow 0.$

$A \leftarrow$ MALLOC(n).

$B \leftarrow$ MALLOC(n).

$C \leftarrow$ MALLOC(n).

READ (A, i)

IF (IS-INITIALIZED $(A[i])$)

  RETURN $A[i]$.

ELSE

  RETURN 0.

WRITE $(A, i, value)$

IF (IS-INITIALIZED $(A[i])$)

  $A[i] \leftarrow value.$

ELSE

  $k \leftarrow k + 1.$

  $A[i] \leftarrow value.$

  $B[i] \leftarrow k.$

  $C[k] \leftarrow i.$

IS-INITIALIZED (A, i)

IF $(1 \le B[i] \le k)$ and $(C[B[i]] = i)$

  RETURN *true*.

ELSE

  RETURN *false*.

Appetizer

Theorem. $A[i]$ is initialized iff both $1 \le B[i] \le k$ and $C[B[i]] = i$.

Pf. $\Rightarrow$

- Suppose $A[i]$ is the j^{th} entry to be initialized.
- Then $C[j] = i$ and $B[i] = j$.
- Thus, $C[B[i]] = i$.

	1	2	3	4	5	6	7	8
A[ ]	?	22	55	99	?	33	?	?
B[ ]	?	3	4	1	?	2	?	?
C[ ]	4	6	2	3	?	?	?	?

k = 4

A[4]=99, A[6]=33, A[2]=22, and A[3]=55 initialized in that order

Appetizer

Theorem. $A[i]$ is initialized iff both $1 \le B[i] \le k$ and $C[B[i]] = i$.

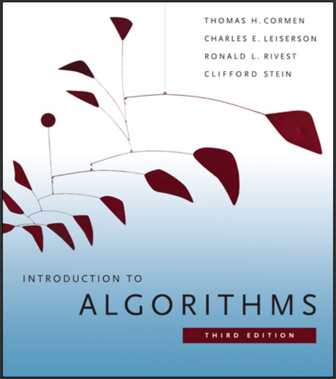
Pf. $\Leftarrow$

- Suppose $A[i]$ is uninitialized.
- If $B[i] < 1$ or $B[i] > k$, then $A[i]$ clearly uninitialized.
- If $1 \le B[i] \le k$ by coincidence, then we still can't have $C[B[i]] = i$ because none of the entries $C[1..k]$ can equal i. ▪

	1	2	3	4	5	6	7	8
A[ ]	?	22	55	99	?	33	?	?
B[ ]	?	3	4	1	?	2	?	?
C[ ]	4	6	2	3	?	?	?	?

k = 4

A[4]=99, A[6]=33, A[2]=22, and A[3]=55 initialized in that order



Amortized Analysis

- *binary counter*
- *multi-pop stack*
- *dynamic table*

Lecture slides by Kevin Wayne
http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Amortized analysis

Worst-case analysis. Determine worst-case running time of a data structure operation as function of the input size n.

can be too pessimistic if the only way to encounter an expensive operation is when there were lots of previous cheap operations

Amortized analysis. Determine worst-case running time of a sequence of n data structure operations.

Ex. Starting from an empty stack implemented with a dynamic table, any sequence of n push and pop operations takes $O(n)$ time in the worst case.

Amortized analysis: applications

- Splay trees.
- Dynamic table.
- Fibonacci heaps.
- Garbage collection.
- Move-to-front list updating.
- Push–relabel algorithm for max flow.
- Path compression for disjoint-set union.
- Structural modifications to red–black trees.
- Security, databases, distributed computing, ...

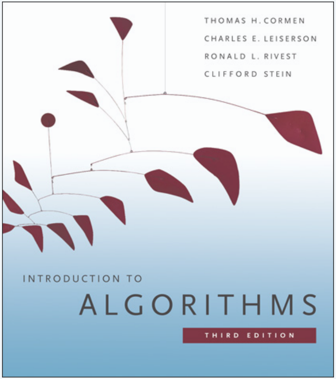
SIAM J. ALG. DISC. METH.
Vol. 6, No. 2, April 1985

AMORTIZED COMPUTATIONAL COMPLEXITY*

ROBERT ENDRE TARJAN†

Abstract. A powerful technique in the complexity analysis of data structures is *amortization*, or averaging over time. Amortized running time is a realistic but robust complexity measure for which we can obtain surprisingly tight upper and lower bounds on a variety of algorithms. By following the principle of designing algorithms whose amortized complexity is low, we obtain "self-adjusting" data structures that are simple, flexible and efficient. This paper surveys recent work by several researchers on amortized complexity.

ASM(MOS) subject classifications. 68C25, 68E05



CHAPTER 17

AMORTIZED ANALYSIS

- *binary counter*
- *multi-pop stack*
- *dynamic table*

Binary counter

Goal. Increment a k-bit binary counter (mod 2^k).

Representation. $a_j = j^{th}$ least significant bit of counter.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

Cost model. Number of bits flipped.

Binary counter

Goal. Increment a k-bit binary counter $(\bmod 2^k)$.

Representation. $a_j = j^{th}$ least significant bit of counter.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

Theorem. Starting from the zero counter, a sequence of n INCREMENT operations flips $O(n\,k)$ bits. ← overly pessimistic upper bound

Pf. At most k bits flipped per increment. ▪

Aggregate method (brute force)

Aggregate method. Analyze cost of a sequence of operations.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Binary counter: aggregate method

Starting from the zero counter, in a sequence of n INCREMENT operations:

- Bit 0 flips n times.
- Bit 1 flips $\lfloor n / 2 \rfloor$ times.
- Bit 2 flips $\lfloor n / 4 \rfloor$ times.
- ...

Theorem. Starting from the zero counter, a sequence of n INCREMENT operations flips $O(n)$ bits.

Pf.

- Bit j flips $\lfloor n / 2^j \rfloor$ times.
- The total number of bits flipped is $\displaystyle\sum_{j=0}^{k-1} \left\lfloor \frac{n}{2^j} \right\rfloor \; < \; n \sum_{j=0}^{\infty} \frac{1}{2^j}$

$$= \; 2n \quad ▪$$

Remark. Theorem may be false if initial counter is not zero.

Accounting method (banker's method)

Assign (potentially) different charges to each operation.

- D_i = data structure after i^{th} operation.
- c_i = actual cost of i^{th} operation.
- $\hat{c}_i$ = amortized cost of i^{th} operation = amount we charge operation i.
- When $\hat{c}_i > c_i$, we store credits in data structure D_i to pay for future ops; when $\hat{c}_i < c_i$, we consume credits in data structure D_i.
- Initial data structure D_0 starts with 0 credits.

can be more or less than actual cost

Credit invariant. The total number of credits in the data structure ≥ 0.

$$\sum_{i=1} \hat{c}_i - \sum_{i=1} c_i \geq 0$$

our job is to choose suitable amortized costs so that this invariant holds

Accounting method (banker's method)

Assign (potentially) different charges to each operation.

- D_i = data structure after i^{th} operation.
- c_i = actual cost of i^{th} operation.
- $\hat{c}_i$ = amortized cost of i^{th} operation = amount we charge operation i.
- When $\hat{c}_i > c_i$, we store credits in data structure D_i to pay for future ops; when $\hat{c}_i < c_i$, we consume credits in data structure D_i.
- Initial data structure D_0 starts with 0 credits.

can be more or less than actual cost

Credit invariant. The total number of credits in the data structure ≥ 0.

$$\sum_{i=1} \hat{c}_i - \sum_{i=1} c_i \geq 0$$

Theorem. Starting from the initial data structure D_0, the total actual cost of any sequence of n operations is at most the sum of the amortized costs.

Pf. The amortized cost of the sequence of n operations is: $\displaystyle\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$. ▪

credit invariant

Intuition. Measure running time in terms of credits (time = money).

Binary counter: accounting method

Credits. One credit pays for a bit flip.

Invariant. Each 1 bit has one credit; each 0 bit has zero credits.

Accounting.

- Flip bit j from 0 to 1: charge 2 credits (use one and save one in bit j).

increment

Binary counter: accounting method

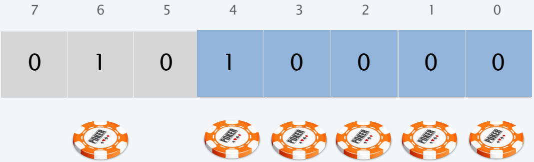
Credits. One credit pays for a bit flip.

Invariant. Each 1 bit has one credit; each 0 bit has zero credits.

Accounting.

- Flip bit j from 0 to 1: charge 2 credits (use one and save one in bit j).
- Flip bit j from 1 to 0: pay for it with the 1 credit saved in bit j.

increment



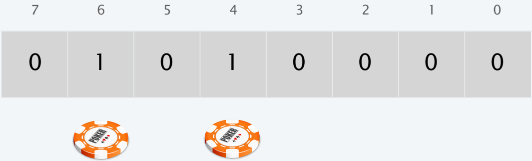
Binary counter: accounting method

Credits. One credit pays for a bit flip.

Invariant. Each 1 bit has one credit; each 0 bit has zero credits.

Accounting.

- Flip bit j from 0 to 1: charge 2 credits (use one and save one in bit j).
- Flip bit j from 1 to 0: pay for it with the 1 credit saved in bit j.



Binary counter: accounting method

Credits. One credit pays for a bit flip.

Invariant. Each 1 bit has one credit; each 0 bit has zero credits.

Accounting.

- Flip bit j from 0 to 1: charge 2 credits (use one and save one in bit j).
- Flip bit j from 1 to 0: pay for it with the 1 credit saved in bit j.

Theorem. Starting from the zero counter, a sequence of n INCREMENT operations flips $O(n)$ bits.

Pf.

the rightmost 0 bit
(unless counter overflows)

- Each INCREMENT operation flips at most one 0 bit to a 1 bit, so the amortized cost per INCREMENT ≤ 2.
- Invariant $\Rightarrow$ number of credits in data structure ≥ 0.
- Total actual cost of n operations $\leq$ sum of amortized costs $\leq 2n$. ▪

accounting method theorem

Potential method (physicist's method)

Potential function. $\Phi(D_i)$ maps each data structure D_i to a real number s.t.:

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each data structure D_i.

Actual and amortized costs.

- c_i = actual cost of i^{th} operation.
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ = amortized cost of i^{th} operation.

Potential method (physicist's method)

Potential function. $\Phi(D_i)$ maps each data structure D_i to a real number s.t.:

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each data structure D_i.

Actual and amortized costs.

- c_i = actual cost of i^{th} operation.
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ = amortized cost of i^{th} operation.

Theorem. Starting from the initial data structure D_0, the total actual cost of any sequence of n operations is at most the sum of the amortized costs.

Pf. The amortized cost of the sequence of operations is:

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \\ &\geq \sum_{i=1}^{n} c_i \quad \blacksquare\end{aligned}$$

Binary counter: potential method

Potential function. Let $\Phi(D)$ = number of 1 bits in the binary counter D.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each D_i.

increment

7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	1

Binary counter: potential method

Potential function. Let $\Phi(D)$ = number of 1 bits in the binary counter D.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each D_i.

increment

7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	0

Binary counter: potential method

Potential function. Let $\Phi(D)$ = number of 1 bits in the binary counter D.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each D_i.

7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	0

Binary counter: potential method

Potential function. Let $\Phi(D)$ = number of 1 bits in the binary counter D.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

Theorem. Starting from the zero counter, a sequence of n INCREMENT operations flips $O(n)$ bits.

Pf.

- Suppose that the i^{th} INCREMENT operation flips t_i bits from 1 to 0.
- The actual cost $c_i \le t_i + 1$. ← operation flips at most one bit from 0 to 1 (no bits flipped to 1 when counter overflows)
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

$$\le c_i + 1 - t_i$$

← potential decreases by 1 for t_i bits flipped from 1 to 0 and increases by 1 for bit flipped from 0 to 1

$$\le 2.$$

- Total actual cost of n operations $\le$ sum of amortized costs $\le 2n$. ▪

↑ potential method theorem

Famous potential functions

Fibonacci heaps. $\Phi(H) \;=\; 2\, trees(H) \;+\; 2\, marks(H)$

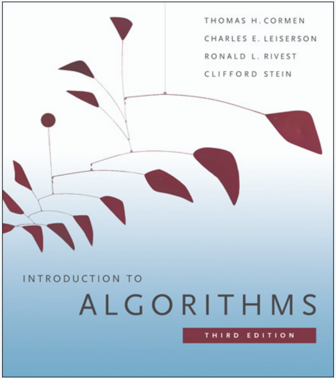
Splay trees. $\Phi(T) \;=\; \sum_{x \in T} \lfloor \log_2 size(x) \rfloor$

Move-to-front. $\Phi(L) \;=\; 2\, inversions(L, L^*)$

Preflow–push. $\Phi(f) \;=\; \sum_{v \,:\, excess(v) \,>\, 0} height(v)$

Red–black trees. $\Phi(T) \;=\; \sum_{x \in T} w(x)$

$$w(x) \;=\; \begin{cases} 0 & \text{if } x \text{ is red} \\ 1 & \text{if } x \text{ is black and has no red children} \\ 0 & \text{if } x \text{ is black and has one red child} \\ 2 & \text{if } x \text{ is black and has two red children} \end{cases}$$



SECTION 17.4

AMORTIZED ANALYSIS

- *binary counter*
- *multi-pop stack*
- *dynamic table*

Multipop stack

Goal. Support operations on a set of elements:

- PUSH(S, x): add element x to stack S.
- POP(S): remove and return the most-recently added element.
- MULTI-POP(S, k): remove the most-recently added k elements.

```
MULTI-POP(S, k)
FOR i = 1 TO k
    POP(S).
```

Exceptions. We assume POP throws an exception if stack is empty.

Multipop stack

Goal. Support operations on a set of elements:

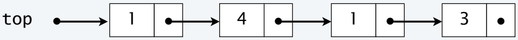
- PUSH(S, x): add element x to stack S.
- POP(S): remove and return the most-recently added element.
- MULTI-POP(S, k): remove the most-recently added k elements.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n^2)$ time.

overly pessimistic upper bound

Pf.

- Use a singly linked list.
- POP and PUSH take $O(1)$ time each.
- MULTI-POP takes $O(n)$ time. ▪



Multipop stack: aggregate method

Goal. Support operations on a set of elements:

- PUSH(S, x): add element x to stack S.
- POP(S): remove and return the most-recently added element.
- MULTI-POP(S, k): remove the most-recently added k elements.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf.

- An element is popped at most once for each time that it is pushed.
- There are $\leq n$ PUSH operations.
- Thus, there are $\leq n$ POP operations (including those made within MULTI-POP). ▪

Multipop stack: accounting method

Credits. 1 credit pays for either a PUSH or POP.

Invariant. Every element on the stack has 1 credit.

Accounting.

- PUSH(S, x): charge 2 credits.
  - use 1 credit to pay for pushing x now
  - store 1 credit to pay for popping x at some point in the future
- POP(S): charge 0 credits.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf.

- Invariant $\Rightarrow$ number of credits in data structure ≥ 0.
- Amortized cost per operation ≤ 2.
- Total actual cost of n operations $\leq$ sum of amortized costs $\leq 2n$. ▪

accounting method theorem

Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of elements currently on the stack.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each D_i.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf. [Case 1: push]

- Suppose that the i^{th} operation is a PUSH.
- The actual cost $c_i = 1$.
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.

Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of elements currently on the stack.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each D_i.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf. [Case 2: pop]

- Suppose that the i^{th} operation is a POP.
- The actual cost $c_i = 1$.
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$.

Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of elements currently on the stack.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf. [Case 3: multi-pop]

- Suppose that the i^{th} operation is a MULTI-POP of k objects.
- The actual cost $c_i = k$.
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - k = 0$. ▪

Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of elements currently on the stack.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each D_i.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf. [putting everything together]

- Amortized cost $\hat{c}_i \leq 2$. ⟵ 2 for push; 0 for pop and multi-pop
- Sum of amortized costs $\hat{c}_i$ of the n operations $\leq 2n$.
- Total actual cost $\leq$ sum of amortized cost $\leq 2n$. ▪

potential method theorem

Part II

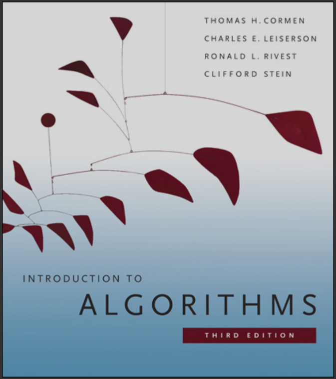
Data stuctures

OVERVIEW

Heaps

Disjoint-sets Data Structures

HEAPS



FIBONACCI HEAPS

- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

Lecture slides by Kevin Wayne
http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on 7/25/17 11:07 AM

Priority queues performance cost summary

operation	linked list	binary heap	binomial heap	Fibonacci heap †
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

† amortized

Ahead. $O(1)$ INSERT and DECREASE-KEY, $O(\log n)$ EXTRACT-MIN.

Fibonacci heaps

Theorem. [Fredman–Tarjan 1986] Starting from an empty Fibonacci heap, any sequence of m INSERT, EXTRACT-MIN, and DECREASE-KEY operations involving n INSERT operations takes $O(m + n \log n)$ time.

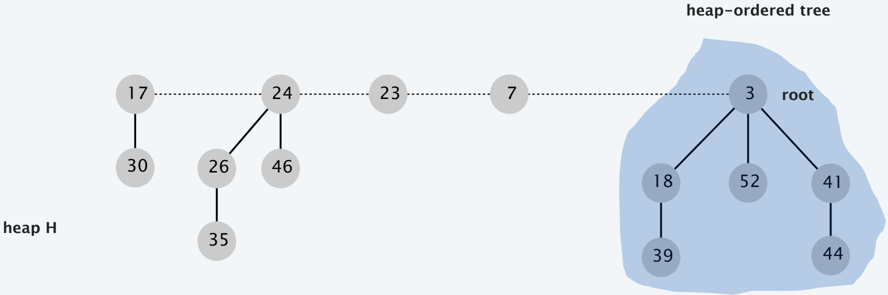
History.

- Ingenious data structure and application of amortized analysis.
- Original motivation: improve Dijkstra's shortest path algorithm from $O(m \log n)$ to $O(m + n \log n)$.
- Also improved best-known bounds for all-pairs shortest paths, assignment problem, minimum spanning trees.

Fibonacci heap: structure

- Set of heap-ordered trees.

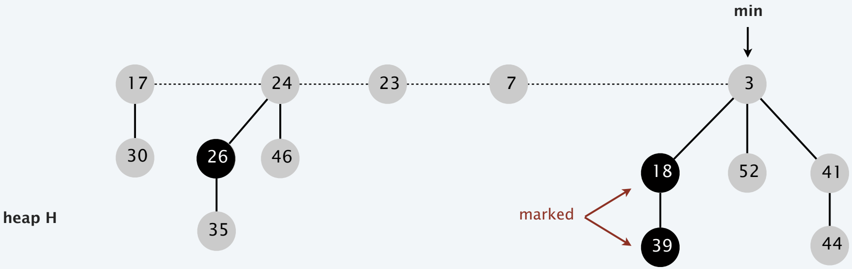
each child no smaller
than its parent



Fibonacci heap: structure

- Set of heap-ordered trees.
- Set of marked nodes.

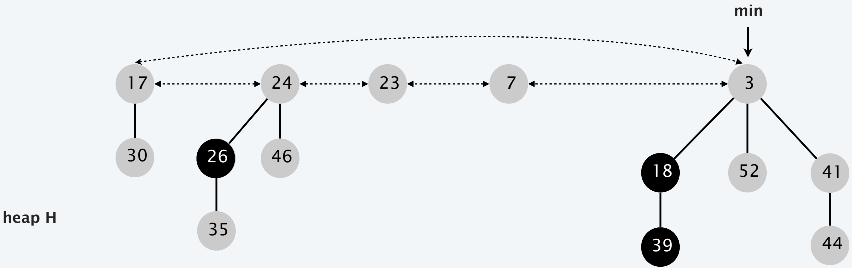
used to keep trees bushy
(stay tuned)



Fibonacci heap: structure

Heap representation.

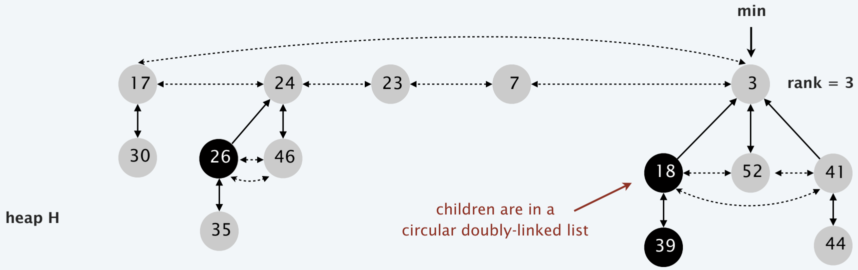
- Store a pointer to the minimum node.
- Maintain tree roots in a circular, doubly-linked list.



Fibonacci heap: representation

Node representation. Each node stores:

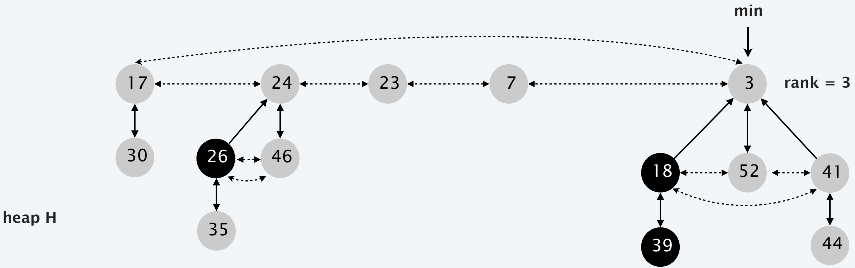
- A pointer to its parent.
- A pointer to any of its children.
- A pointer to its left and right siblings.
- Its rank = number of children.
- Whether it is marked.



Fibonacci heap: representation

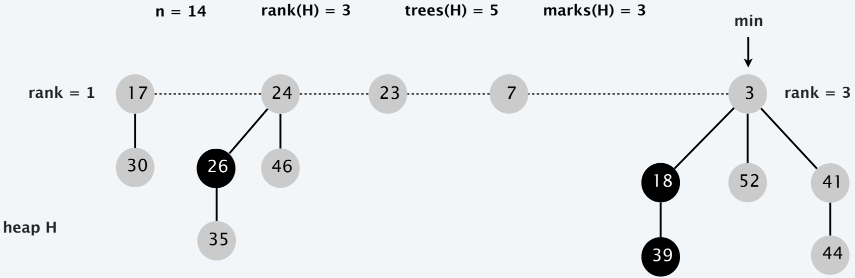
Operations we can do in constant time:

- Determine rank of a node.
- Find the minimum element.
- Merge two root lists together.
- Add or remove a node from the root list.
- Remove a subtree and merge into root list.
- Link the root of a one tree to root of another tree.



Fibonacci heap: notation

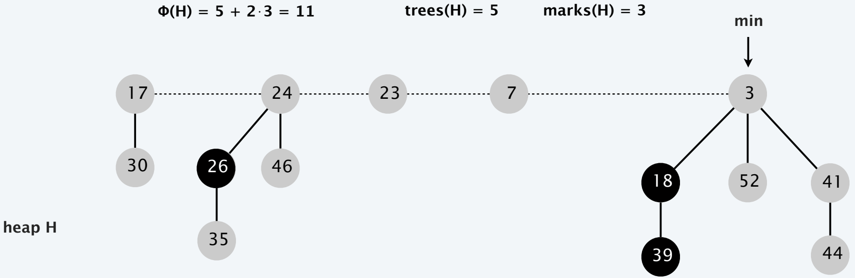
notation	meaning
n	number of nodes
$rank(x)$	number of children of node x
$rank(H)$	max rank of any node in heap H
$trees(H)$	number of trees in heap H
$marks(H)$	number of marked nodes in heap H

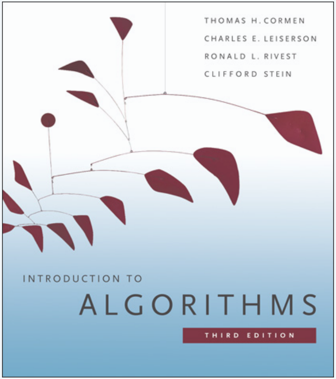


Fibonacci heap: potential function

Potential function.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$





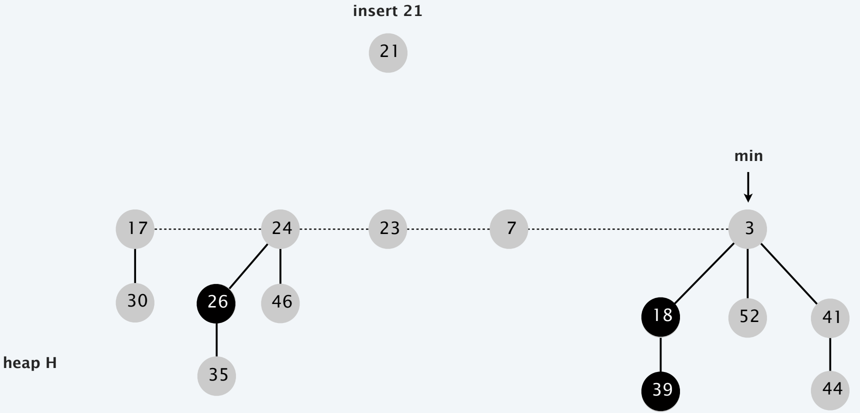
SECTION 19.2

FIBONACCI HEAPS

- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

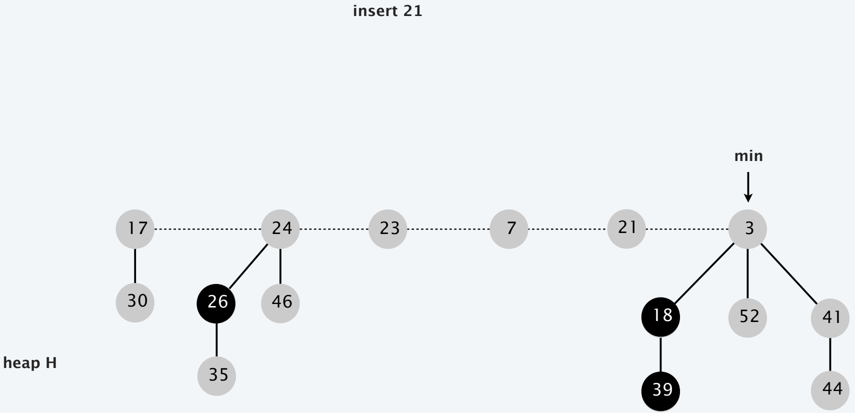
Fibonacci heap: insert

- Create a new singleton tree.
- Add to root list; update min pointer (if necessary).



Fibonacci heap: insert

- Create a new singleton tree.
- Add to root list; update min pointer (if necessary).



Fibonacci heap: insert analysis

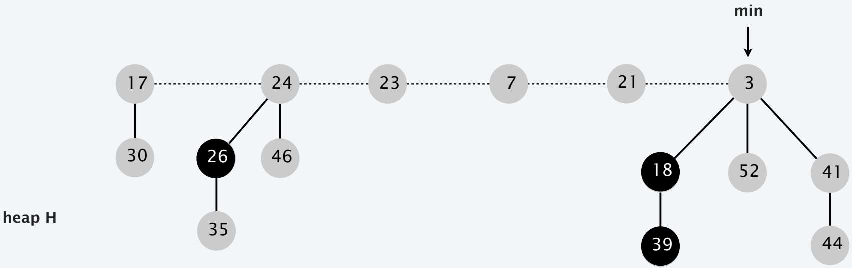
Actual cost. $c_i = O(1)$.

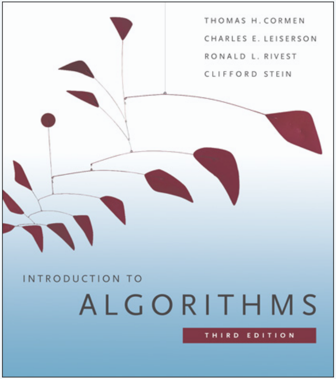
Change in potential. $\Delta\Phi = \Phi(H_i) - \Phi(H_{i-1}) = +1$.

one more tree;
no change in marks

Amortized cost. $\hat{c}_i = c_i + \Delta\Phi = O(1)$.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$





SECTION 19.2

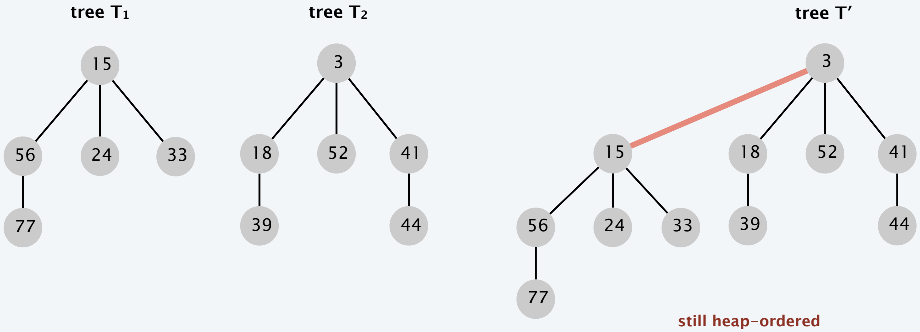
FIBONACCI HEAPS

- *preliminaries*
- *insert*
- ***extract the minimum***
- *decrease key*
- *bounding the rank*
- *meld and delete*

Linking operation

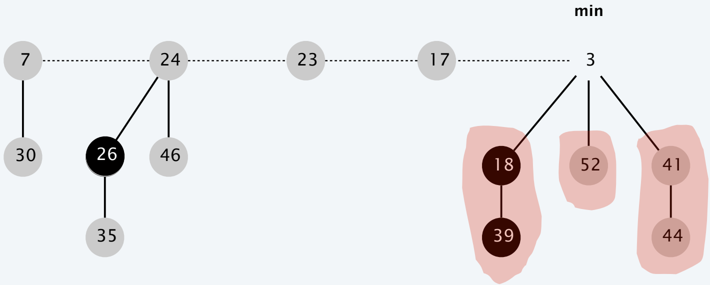
Useful primitive. Combine two trees T_1 and T_2 of rank k.

- Make larger root be a child of smaller root.
- Resulting tree T' has rank $k + 1$.



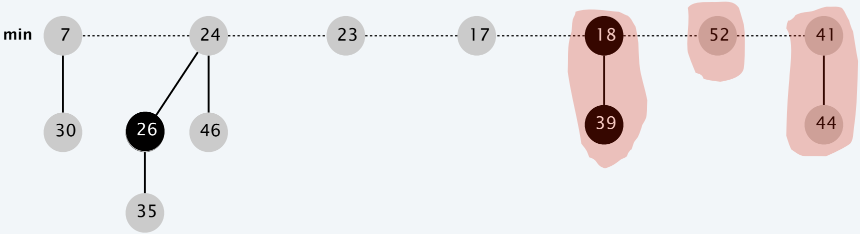
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

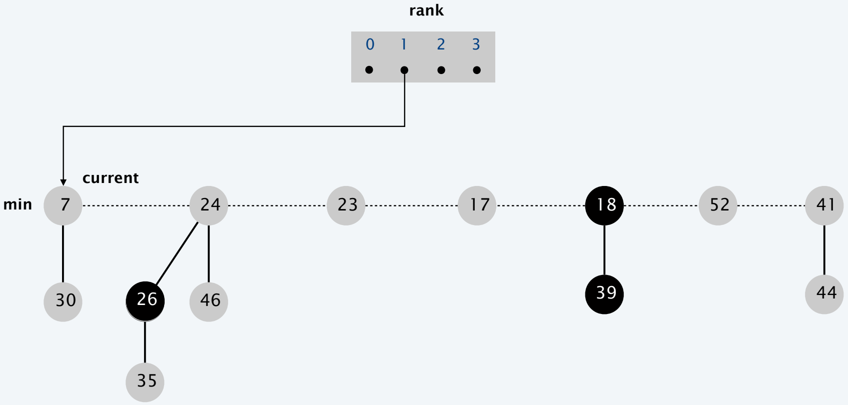


Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

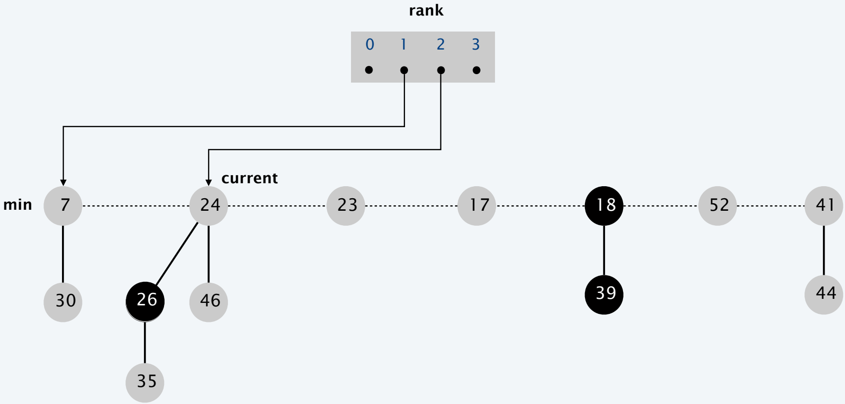
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



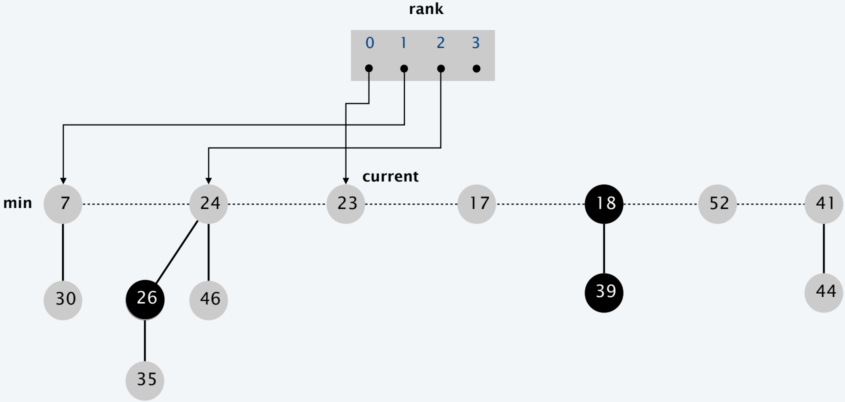
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



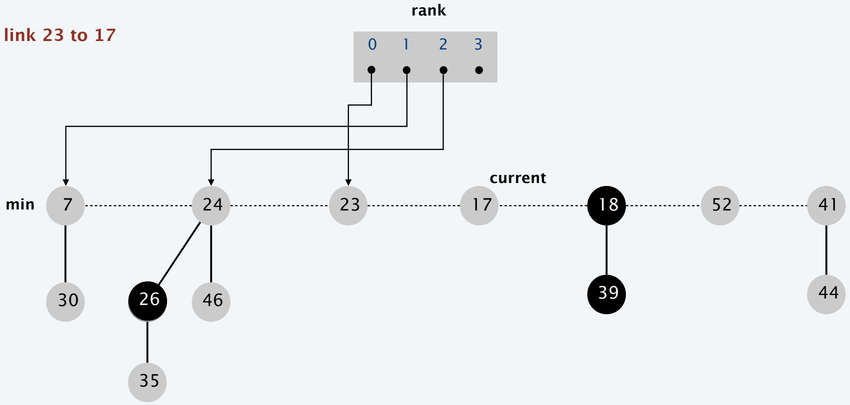
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



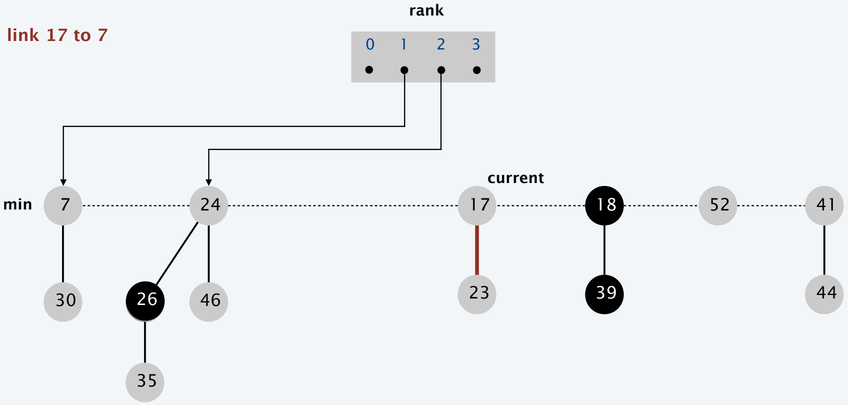
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



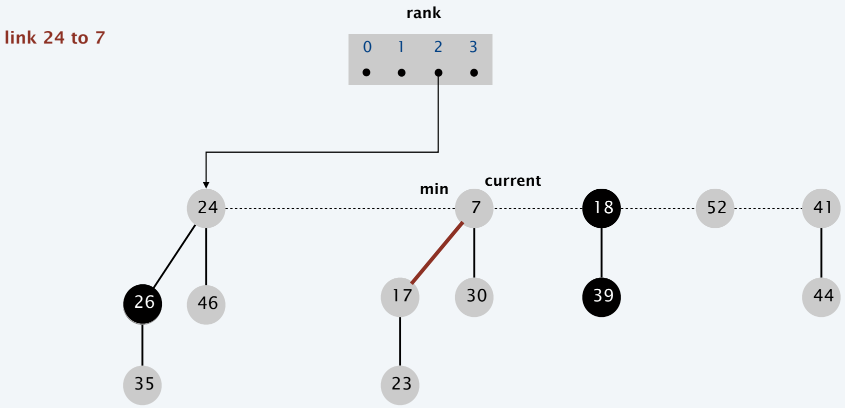
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



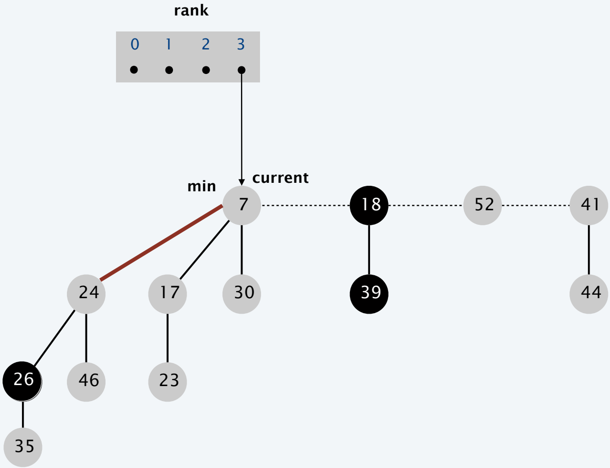
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



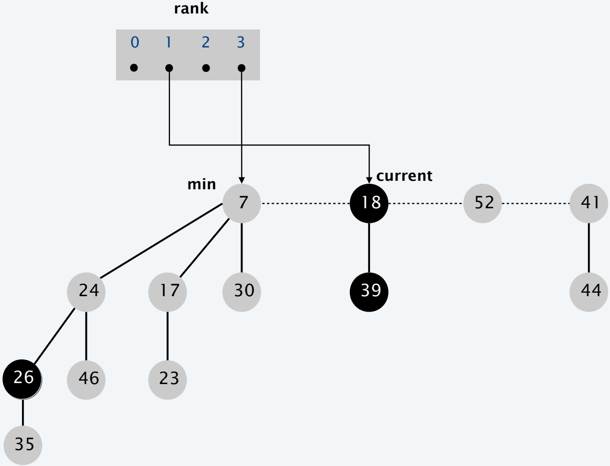
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



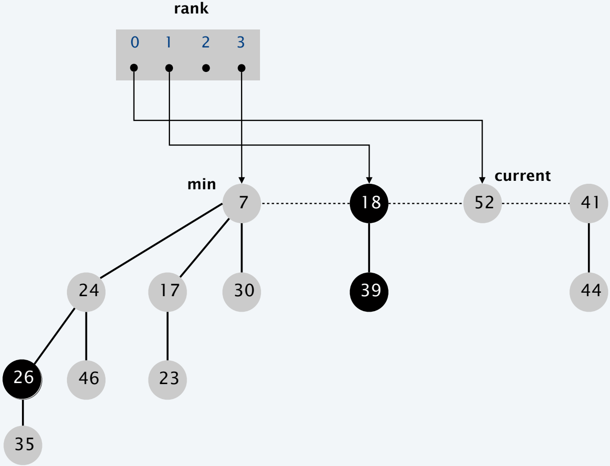
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

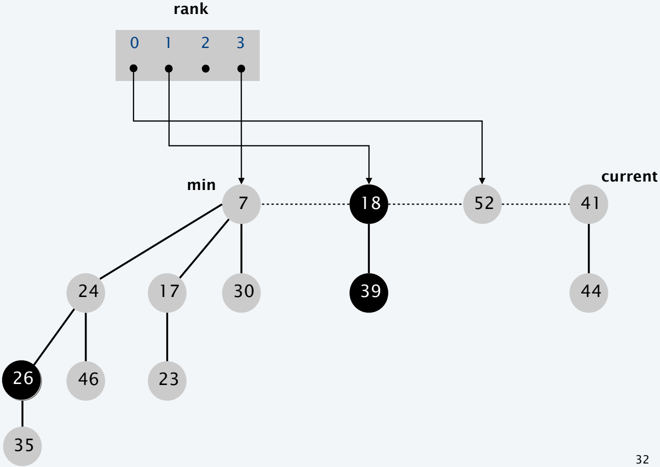
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

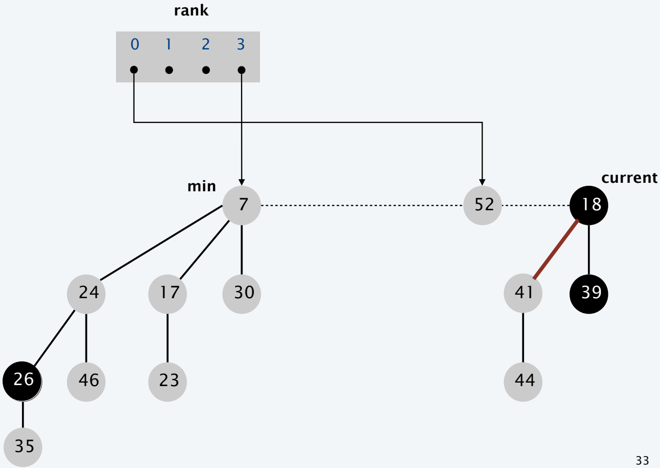
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

link 41 to 18



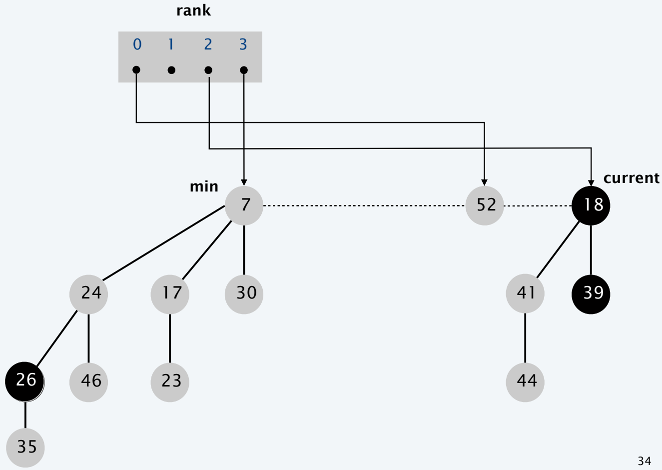
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

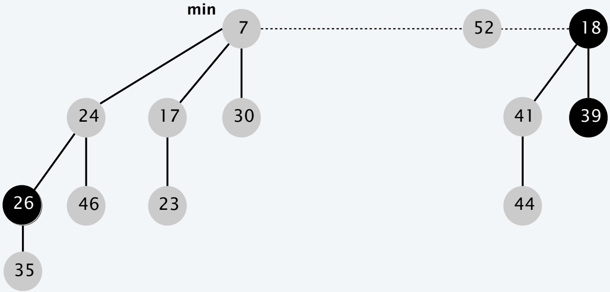
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

stop (no two trees have same rank)



Fibonacci heap: extract the minimum analysis

Actual cost. $c_i = O(rank(H)) + O(trees(H))$.

- $O(rank(H))$ to meld min's children into root list. ← ≤ rank(H) children
- $O(rank(H)) + O(trees(H))$ to update min. ← ≤ rank(H) + trees(H) − 1 root nodes
- $O(rank(H)) + O(trees(H))$ to consolidate trees. ← number of roots decreases by 1 after each linking operation

Change in potential. $\Delta\Phi \le rank(H') + 1 - trees(H)$.

- No new nodes become marked.
- $trees(H') \le rank(H') + 1$. ← no two trees have same rank after consolidation

Amortized cost. $O(\log n)$.

- $\hat{c}_i = c_i + \Delta\Phi = O(rank(H)) + O(rank(H'))$.
- The rank of a Fibonacci heap with n elements is $O(\log n)$.

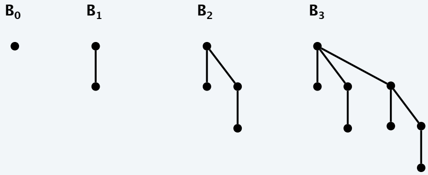
Fibonacci lemma (stay tuned)

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Fibonacci heap vs. binomial heaps

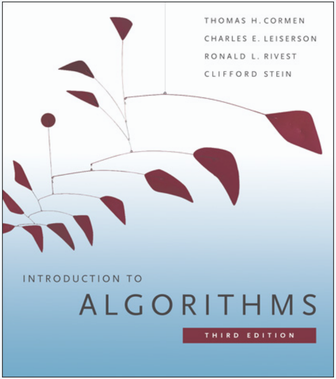
Observation. If only INSERT and EXTRACT-MIN operations, then all trees are binomial trees.

we link only trees of equal rank



Binomial heap property. This implies $rank(H) \leq \log_2 n$.

Fibonacci heap property. Our DECREASE-KEY implementation will not preserve this property, but we will implement it in such a way that $rank(H) \leq \log_\phi n$.



SECTION 19.3

FIBONACCI HEAPS

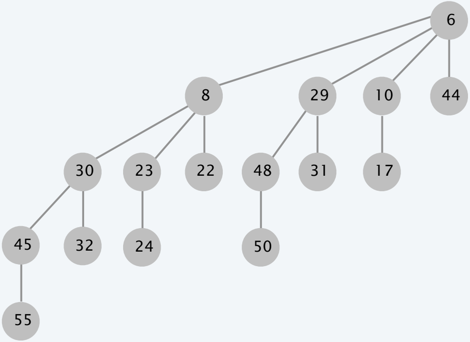
- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

Fibonacci heap: decrease key

Intuition for deceasing the key of node x.

- If heap-order is not violated, decrease the key of x.
- Otherwise, cut tree rooted at x and meld into root list.

decrease-key of x from 30 to 7

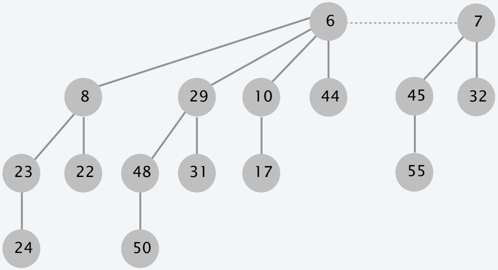


Fibonacci heap: decrease key

Intuition for deceasing the key of node x.

- If heap-order is not violated, decrease the key of x.
- Otherwise, cut tree rooted at x and meld into root list.

decrease-key of x from 23 to 5

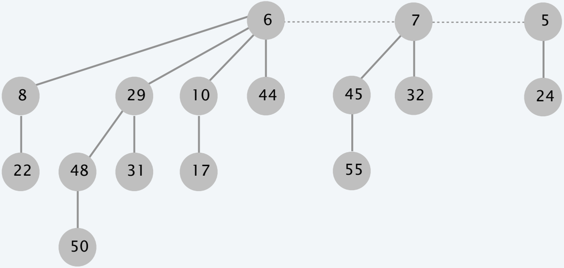


Fibonacci heap: decrease key

Intuition for deceasing the key of node x.

- If heap-order is not violated, decrease the key of x.
- Otherwise, cut tree rooted at x and meld into root list.

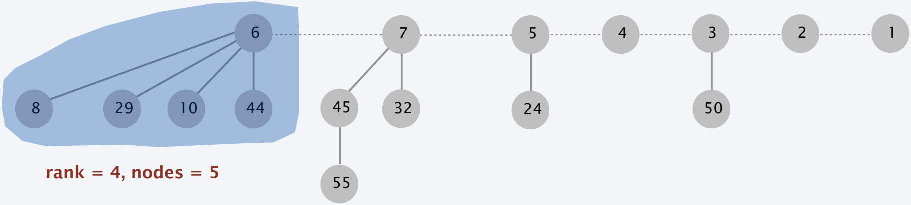
decrease-key of 22 to 4
decrease-key of 48 to 3
decrease-key of 31 to 2
decrease-key of 17 to 1



Fibonacci heap: decrease key

Intuition for deceasing the key of node x.

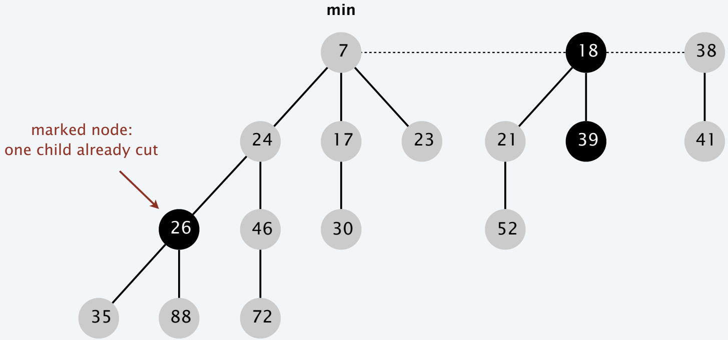
- If heap-order is not violated, decrease the key of x.
- Otherwise, cut tree rooted at x and meld into root list.
- Problem: number of nodes not exponential in rank.



Fibonacci heap: decrease key

Intuition for deceasing the key of node x.

- If heap-order is not violated, decrease the key of x.
- Otherwise, cut tree rooted at x and meld into root list.
- Solution: as soon as a node has its second child cut, cut it off also and meld into root list (and unmark it).

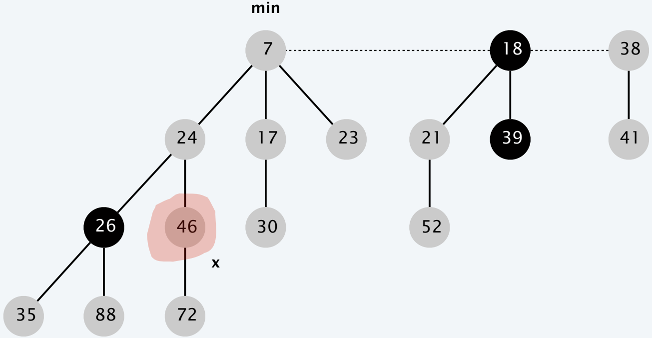


Fibonacci heap: decrease key

Case 1. [heap order not violated]

- Decrease key of x.
- Change heap min pointer (if necessary).

decrease-key of x from 46 to 29

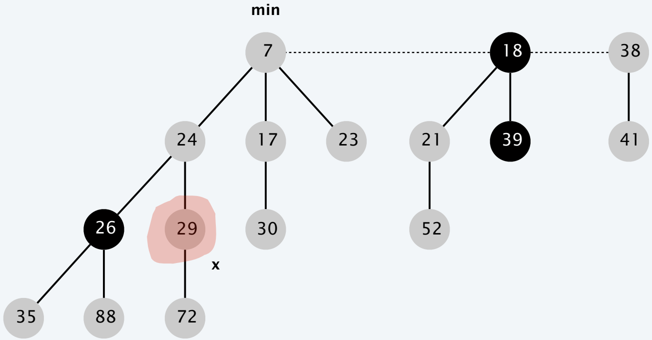


Fibonacci heap: decrease key

Case 1. [heap order not violated]

- Decrease key of x.
- Change heap min pointer (if necessary).

decrease-key of x from 46 to 29

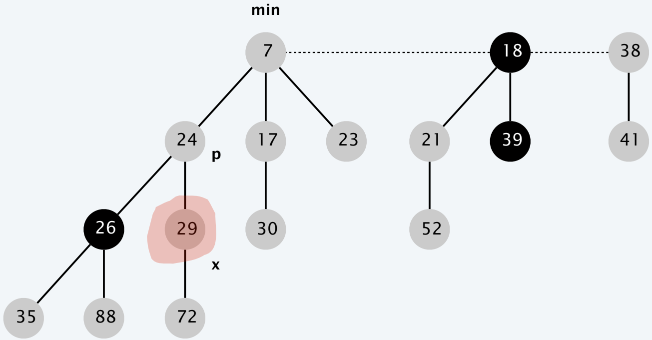


Fibonacci heap: decrease key

Case 2a. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut p, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 29 to 15

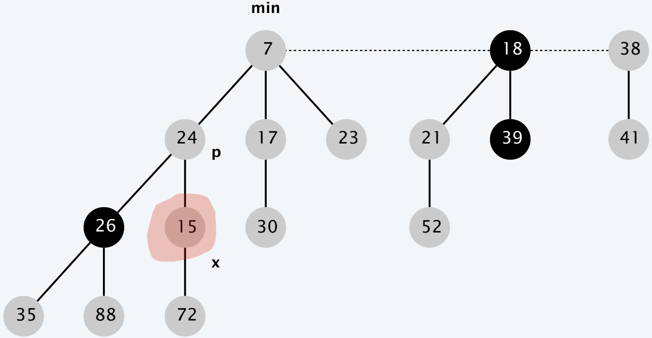


Fibonacci heap: decrease key

Case 2a. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut p, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 29 to 15

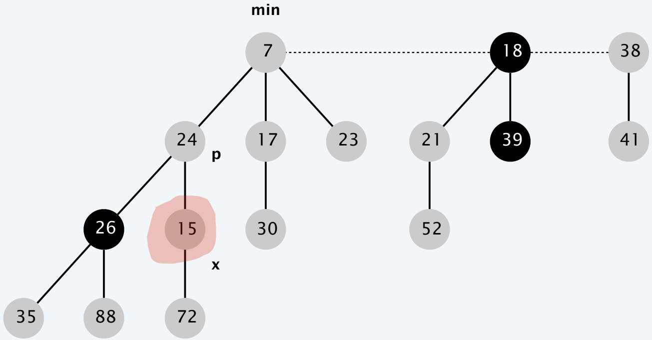


Fibonacci heap: decrease key

Case 2a. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut p, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 29 to 15

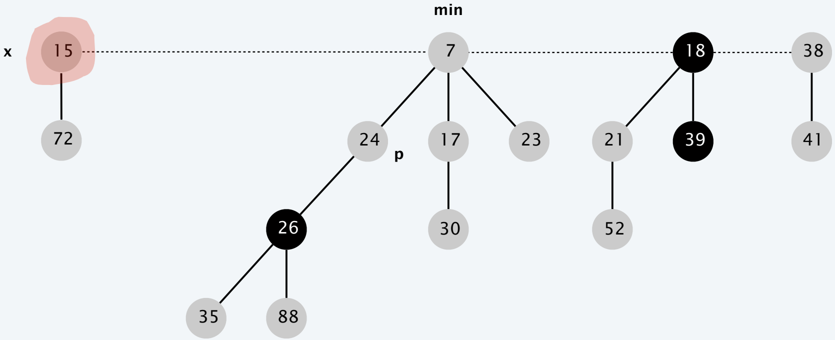


Fibonacci heap: decrease key

Case 2a. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut p, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 29 to 15



Fibonacci heap: decrease key

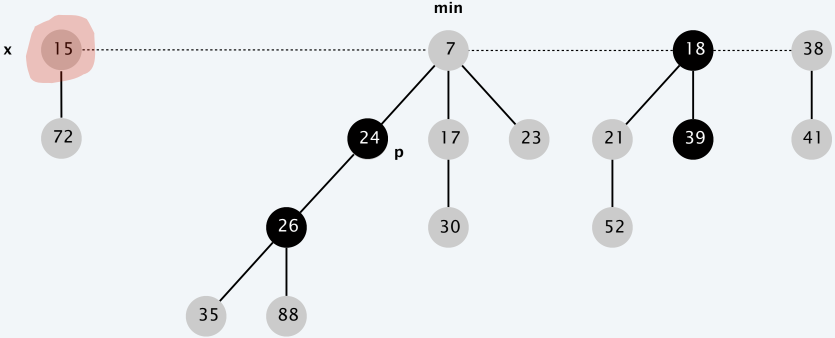
Case 2a. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;

  Otherwise, cut p, meld into root list, and unmark

  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 29 to 15

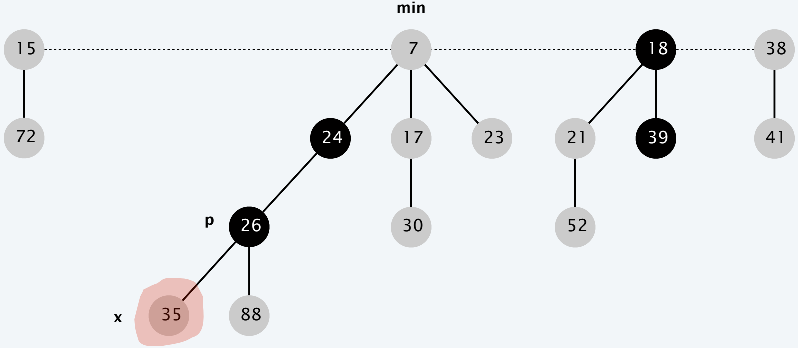


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut p, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5

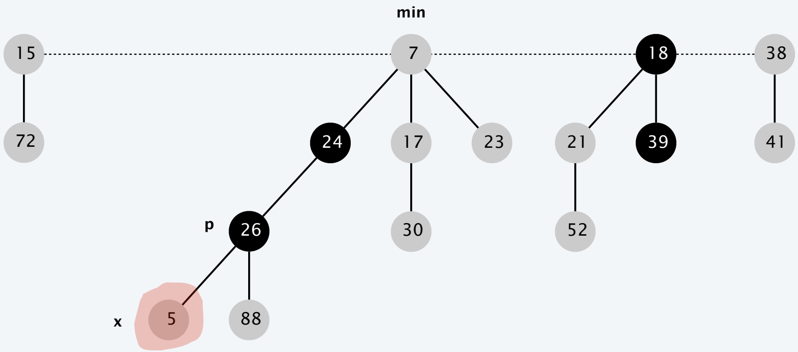


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut p, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5

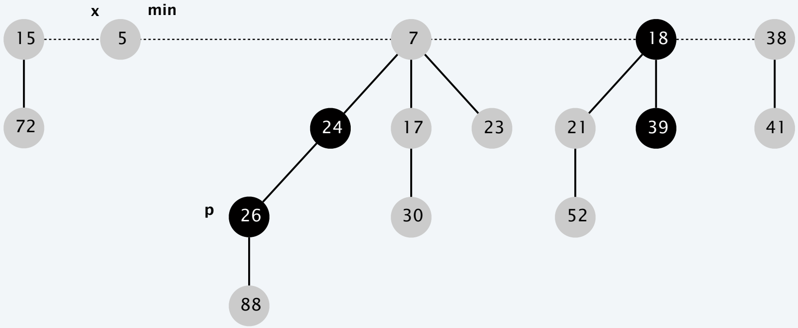


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut p, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5



Fibonacci heap: decrease key

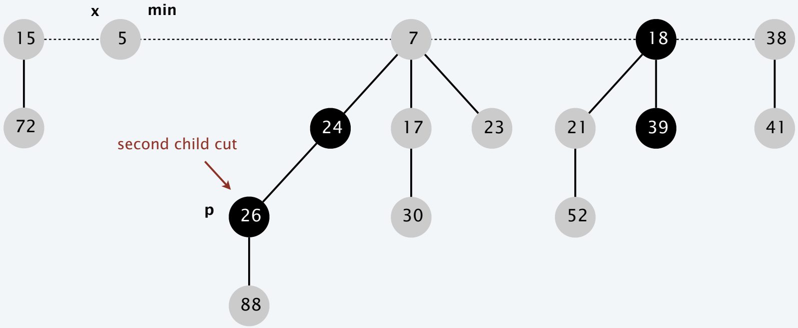
Case 2b. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;

  Otherwise, cut p, meld into root list, and unmark

  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5



Fibonacci heap: decrease key

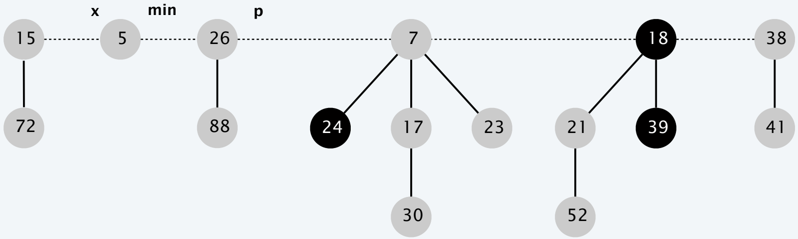
Case 2b. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;

  Otherwise, cut p, meld into root list, and unmark

  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5

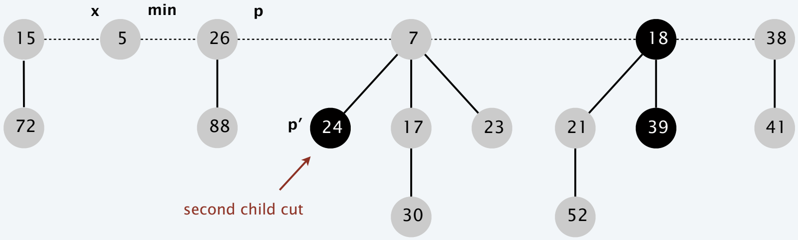


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut p, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5

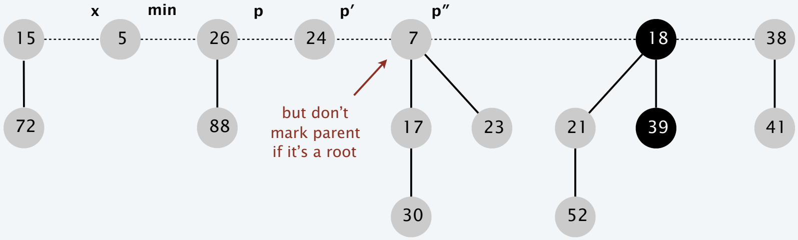


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut p, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5



Fibonacci heap: decrease key analysis

Actual cost. $c_i = O(c)$, where c is the number of cuts.

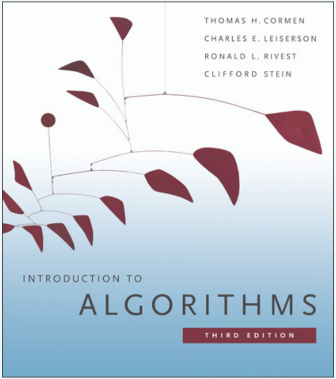
- $O(1)$ time for changing the key.
- $O(1)$ time for each of c cuts, plus melding into root list.

Change in potential. $\Delta\Phi = O(1) - c$.

- $trees(H') = trees(H) + c$.
- $marks(H') \leq marks(H) - c + 2$. ← each cut (except first) unmarks a node; last cut may or may not mark a node
- $\Delta\Phi \leq c + 2 \cdot (-c + 2) = 4 - c$.

Amortized cost. $\hat{c}_i = c_i + \Delta\Phi = O(1)$.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



SECTION 19.4

FIBONACCI HEAPS

- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

Analysis summary

Insert. $O(1)$.

Delete-min. $O(rank(H))$ amortized.

Decrease-key. $O(1)$ amortized.

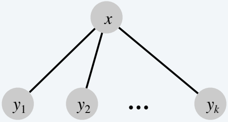
Fibonacci lemma. Let H be a Fibonacci heap with n elements. Then, $rank(H) = O(\log n)$.

number of nodes is
exponential in rank

Bounding the rank

Lemma 1. Fix a point in time. Let x be a node of rank k, and let $y_1, \ldots, y_k$ denote its current children in the order in which they were linked to x. Then:

$$rank(y_i) \;\geq\; \begin{cases} 0 & \text{if } i = 1 \\ i - 2 & \text{if } i \geq 2 \end{cases}$$



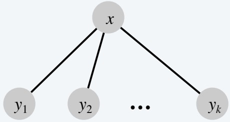
Pf.

- When y_i was linked into x, x had at least $i - 1$ children $y_1, \ldots, y_{i-1}$.
- Since only trees of equal rank are linked, at that time $rank(y_i) = rank(x) \geq i - 1$.
- Since then, y_i has lost at most one child (or y_i would have been cut).
- Thus, right now $rank(y_i) \geq i - 2$. ▪

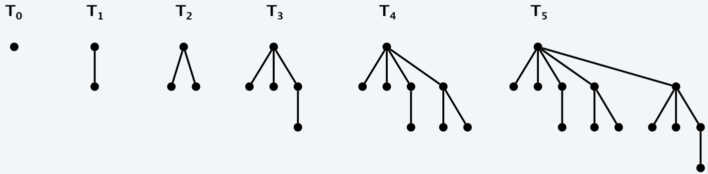
Bounding the rank

Lemma 1. Fix a point in time. Let x be a node of rank k, and let $y_1, \ldots, y_k$ denote its current children in the order in which they were linked to x. Then:

$$rank(y_i) \geq \begin{cases} 0 & \text{if } i = 1 \\ i - 2 & \text{if } i \geq 2 \end{cases}$$



Def. Let T_k be smallest possible tree of rank k satisfying property.

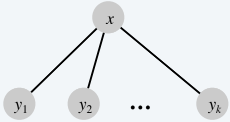


$F_2 = 1$ $F_3 = 2$ $F_4 = 3$ $F_5 = 5$ $F_6 = 8$ $F_7 = 13$

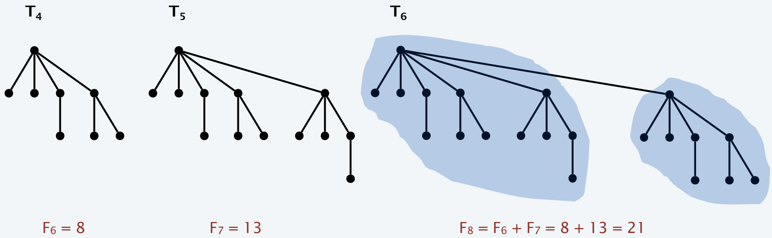
Bounding the rank

Lemma 1. Fix a point in time. Let x be a node of rank k, and let $y_1, \ldots, y_k$ denote its current children in the order in which they were linked to x. Then:

$$rank(y_i) \geq \begin{cases} 0 & \text{if } i = 1 \\ i - 2 & \text{if } i \geq 2 \end{cases}$$



Def. Let T_k be smallest possible tree of rank k satisfying property.



Bounding the rank

Lemma 2. Let s_k be minimum number of elements in any Fibonacci heap of rank k. Then $s_k \geq F_{k+2}$, where F_k is the k^{th} Fibonacci number.

Pf. [by strong induction on k]

- Base cases: $s_0 = 1$ and $s_1 = 2$.
- Inductive hypothesis: assume $s_i \geq F_{i+2}$ for $i = 0, \ldots, k-1$.
- As in Lemma 1, let let $y_1, \ldots, y_k$ denote its current children in the order in which they were linked to x.

$$\begin{aligned} s_k &\geq 1 + 1 + (s_0 + s_1 + \ldots + s_{k-2}) && \text{(Lemma 1)} \\ &\geq (1 + F_1) + F_2 + F_3 + \ldots + F_k && \text{(inductive hypothesis)} \\ &= F_{k+2}. \quad \blacksquare && \text{(Fibonacci fact 1)} \end{aligned}$$

Bounding the rank

Fibonacci lemma. Let H be a Fibonacci heap with n elements.
Then, $rank(H) \le \log_\phi n$, where ϕ is the golden ratio $= (1 + \sqrt{5}) / 2 \approx 1.618$.

Pf.

- Let H is a Fibonacci heap with n elements and rank k.
- Then $\underbrace{n \ \ge}_{\text{Lemma 2}} \ \underbrace{F_{k+2} \ \ge}_{\text{Fibonacci Fact 2}} \ \phi^k$.
- Taking logs, we obtain $rank(H) = k \le \log_\phi n$. ▪

Fibonacci fact 1

Def. The Fibonacci sequence is: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Fibonacci fact 1. For all integers $k \geq 0$, $F_{k+2} = 1 + F_0 + F_1 + \ldots + F_k$.

Pf. [by induction on k]

- Base case: $F_2 = 1 + F_0 = 2$.
- Inductive hypothesis: assume $F_{k+1} = 1 + F_0 + F_1 + \ldots + F_{k-1}$.

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} && \text{(definition)} \\ &= F_k + (1 + F_0 + F_1 + \ldots + F_{k-1}) && \text{(inductive hypothesis)} \\ &= 1 + F_0 + F_1 + \ldots + F_{k-1} + F_k. \quad \blacksquare && \text{(algebra)} \end{aligned}$$

Fibonacci fact 2

Def. The Fibonacci sequence is: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

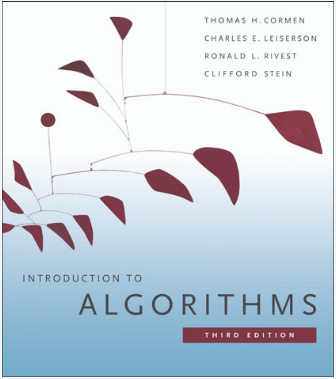
$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Fibonacci fact 2. $F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5}) / 2 \approx 1.618$.

Pf. [by induction on k]

- Base cases: $F_2 = 1 \geq 1$, $F_3 = 2 \geq \phi$.
- Inductive hypotheses: assume $F_k \geq \phi^k$ and $F_{k+1} \geq \phi^{k+1}$

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} && \text{(definition)} \\ &\geq \phi^{k-1} + \phi^{k-2} && \text{(inductive hypothesis)} \\ &= \phi^{k-2}(1 + \phi) && \text{(algebra)} \\ &= \phi^{k-2}\ \phi^2 && (\phi^2 = \phi + 1) \\ &= \phi^k. \ \blacksquare && \text{(algebra)} \end{aligned}$$



SECTION 19.2, 19.3

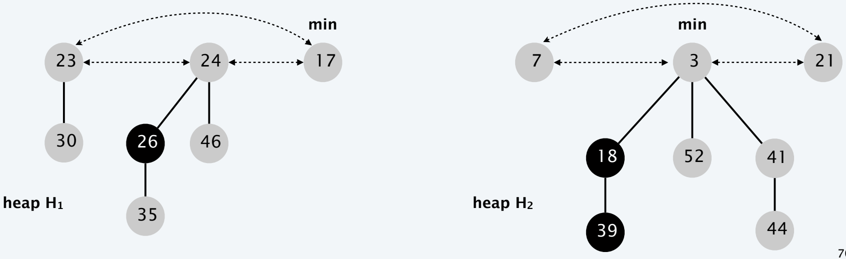
FIBONACCI HEAPS

- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

Fibonacci heap: meld

Meld. Combine two Fibonacci heaps (destroying old heaps).

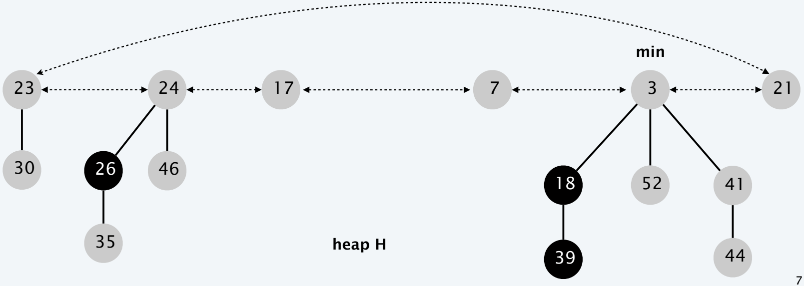
Recall. Root lists are circular, doubly-linked lists.



Fibonacci heap: meld

Meld. Combine two Fibonacci heaps (destroying old heaps).

Recall. Root lists are circular, doubly-linked lists.



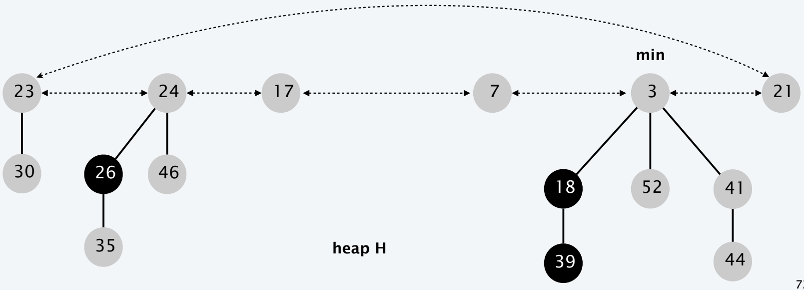
Fibonacci heap: meld analysis

Actual cost. $c_i = O(1)$.

Change in potential. $\Delta\Phi = 0$.

Amortized cost. $\hat{c}_i = c_i + \Delta\Phi = O(1)$.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



Fibonacci heap: delete

Delete. Given a handle to an element x, delete it from heap H.

- DECREASE-KEY($H, x, -\infty$).
- EXTRACT-MIN(H).

Amortized cost. $\hat{c}_i = O(rank(H))$.

- $O(1)$ amortized for DECREASE-KEY.
- $O(rank(H))$ amortized for EXTRACT-MIN.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Priority queues performance cost summary

operation	linked list	binary heap	binomial heap	Fibonacci heap †
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

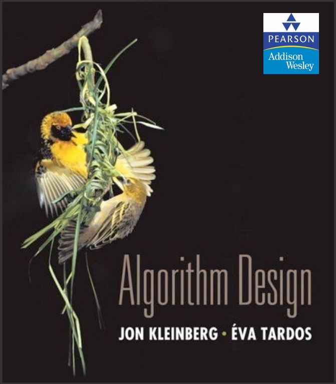
† amortized

Accomplished. $O(1)$ INSERT and DECREASE-KEY, $O(\log n)$ EXTRACT-MIN.

Heaps of heaps

- b-heaps.
- Fat heaps.
- 2–3 heaps.
- Leaf heaps.
- Thin heaps.
- Skew heaps.
- Splay heaps.
- Weak heaps.
- Leftist heaps.
- Quake heaps.
- Pairing heaps.
- Violation heaps.
- Run-relaxed heaps.
- Rank-pairing heaps.
- Skew-pairing heaps.
- Rank-relaxed heaps.
- Lazy Fibonacci heaps.

DISJOINT-SETS DATA STRUCTURES



Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

UNION–FIND

- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- *link-by-rank with path compression*
- *context*

Last updated on 11/22/17 6:04 AM

Disjoint-sets data type

Goal. Support three operations on a collection of disjoint sets.

- MAKE-SET(x): create a new set containing only element x.
- FIND(x): return a canonical element in the set containing x.
- UNION(x, y): replace the sets containing x and y with their union.

Performance parameters.

- m = number of calls to MAKE-SET, FIND, and UNION.
- n = number of elements = number of calls to MAKE-SET.

Dynamic connectivity. Given an initially empty graph G, support three operations. ⟵ disjoint sets = connected components

- ADD-NODE(u): add node u. ⟵ 1 MAKE-SET operation
- ADD-EDGE(u, v): add an edge between nodes u and v. ⟵ 1 UNION operation
- IS-CONNECTED(u, v): is there a path between u and v ? ⟵ 2 FIND operations

Disjoint-sets data type: applications

Original motivation. Compiling EQUIVALENCE, DIMENSION, and COMMON statements in Fortran.

An Improved Equivalence Algorithm

BERNARD A. GALLER AND MICHAEL J. FISHER
University of Michigan, Ann Arbor, Michigan

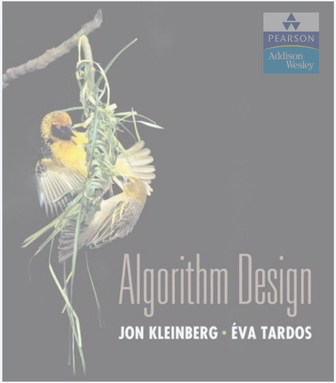
An algorithm for assigning storage on the basis of EQUIVALENCE, DIMENSION and COMMON declarations is presented. The algorithm is based on a tree structure, and has reduced computation time by 40 percent over a previously published algorithm by identifying all equivalence classes with one scan of the EQUIVALENCE declarations. The method is applicable in any problem in which it is necessary to identify equivalence classes, given the element pairs defining the equivalence relation.

Note. This 1964 paper also introduced key data structure for problem.

Disjoint-sets data type: applications

Applications.

- Percolation.
- Kruskal's algorithm.
- Connected components.
- Computing LCAs in trees.
- Computing dominators in digraphs.
- Equivalence of finite state automata.
- Checking flow graphs for reducibility.
- Hoshen–Kopelman algorithm in physics.
- Hinley–Milner polymorphic type inference.
- Morphological attribute openings and closings.
- Matlab's BW-LABEL function for image processing.
- Compiling EQUIVALENCE, DIMENSION and COMMON statements in Fortran.
- ...



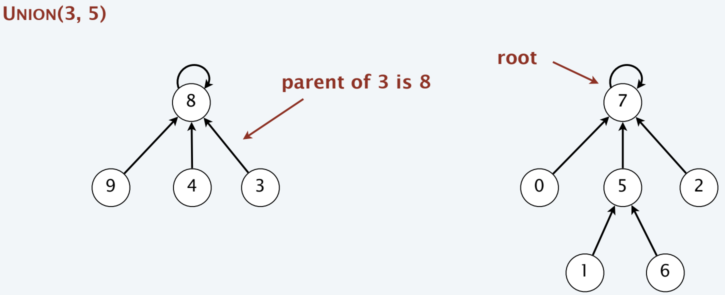
UNION–FIND

- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- *link-by-rank with path compression*
- *context*

Disjoint-sets data structure

Parent-link representation. Represent each set as a tree of elements.

- Each element has an explicit parent pointer in the tree.
- The root serves as the canonical element (and points to itself).
- FIND(x): find the root of the tree containing x.
- UNION(x, y): merge trees containing x and y.

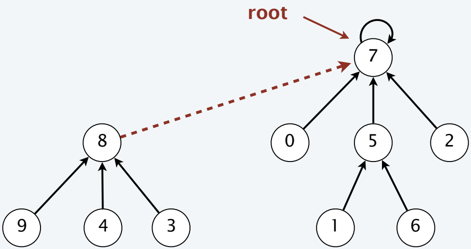


Disjoint-sets data structure

Parent-link representation. Represent each set as a tree of elements.

- Each element has an explicit parent pointer in the tree.
- The root serves as the canonical element (and points to itself).
- FIND(x): find the root of the tree containing x.
- UNION(x, y): merge trees containing x and y.

UNION(3, 5)

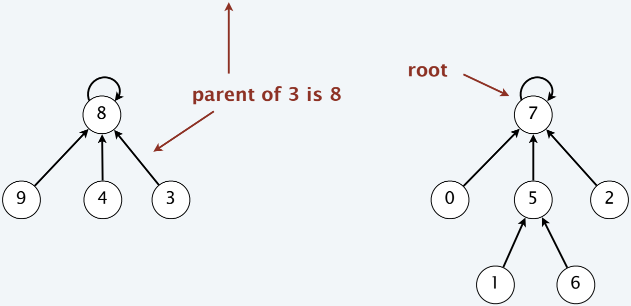


Disjoint-sets data structure

Array representation. Represent each set as a tree of elements.

- Allocate an array $parent[]$ of length n. ⟵ must know number of elements n a priori
- $parent[i] = j$ means parent of element i is element j.

	0	1	2	3	4	5	6	7	8	9
parent[]	7	5	7	8	8	7	5	7	8	8

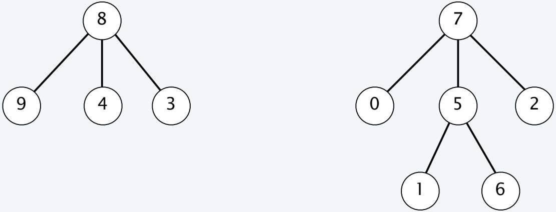


Note. For brevity, we suppress arrows and self loops in figures.

Naïve linking

Naïve linking. Link root of first tree to root of second tree.

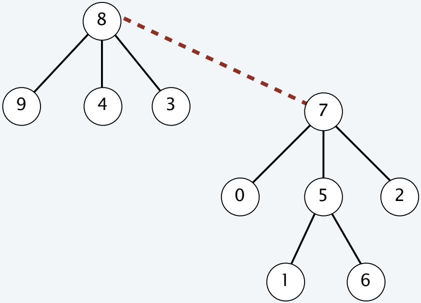
UNION(5, 3)



Naïve linking

Naïve linking. Link root of first tree to root of second tree.

UNION(5, 3)



Naïve linking

Naïve linking. Link root of first tree to root of second tree.

MAKE-SET(x)

$parent[x] \leftarrow x$.

UNION(x, y)

$r \leftarrow$ FIND(x).

$s \leftarrow$ FIND(y).

$parent[r] \leftarrow s$.

FIND(x)

WHILE ($x \neq parent[x]$)

  $x \leftarrow parent[x]$.

RETURN x.

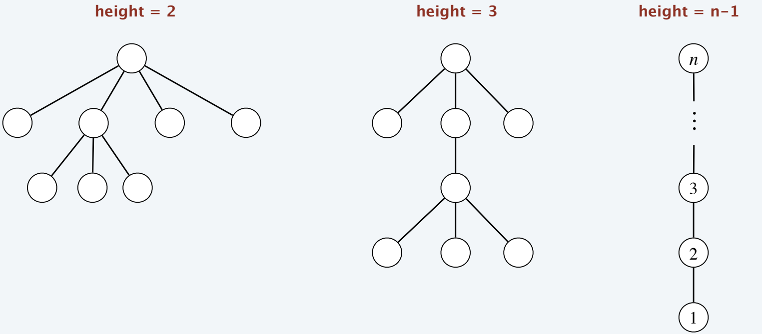
Naïve linking: analysis

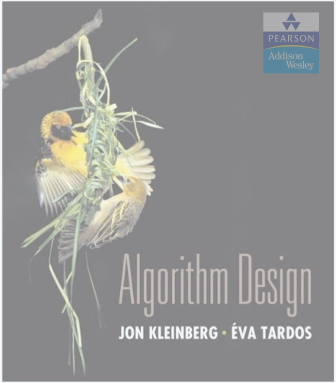
Theorem. Using naïve linking, a UNION or FIND operation can take $\Theta(n)$ time in the worst case, where n is the number of elements.

max number of links on any path from root to leaf node

Pf.

- In the worst case, FIND takes time proportional to the height of the tree.
- Height of the tree is $n - 1$ after the sequence of union operations:
  UNION$(1, 2)$, UNION$(2, 3)$, …, UNION$(n - 1, n)$.





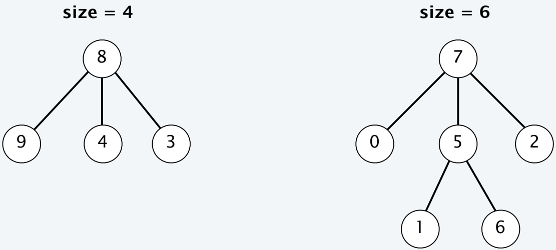
UNION–FIND

- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- *link-by-rank with path compression*
- *context*

Link-by-size

Link-by-size. Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

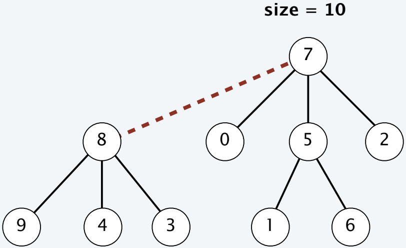
UNION(5, 3)



Link-by-size

Link-by-size. Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

UNION(5, 3)



Link-by-size

Link-by-size. Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

MAKE-SET(x)

$parent[x] \leftarrow x.$

$size[x] \leftarrow 1.$

FIND(x)

WHILE $(x \neq parent[x])$

  $x \leftarrow parent[x].$

RETURN x.

UNION(x, y)

$r \leftarrow$ FIND(x).

$s \leftarrow$ FIND(y).

IF $(r = s)$ RETURN.

ELSE IF $(size[r] > size[s])$

  $parent[s] \leftarrow r.$

  $size[r] \leftarrow size[r] + size[s].$

ELSE

  $parent[r] \leftarrow s.$

  $size[s] \leftarrow size[r] + size[s].$

link-by-size

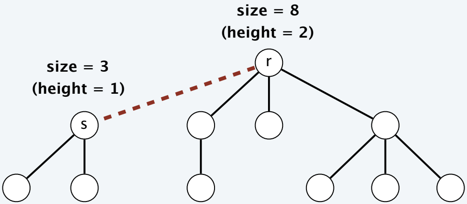
Link-by-size: analysis

Property. Using link-by-size, for every root node r : $size[r] \geq 2^{height(r)}$.

Pf. [by induction on number of links]

- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first i links.
- Tree rooted at r changes only when a smaller (or equal) size tree rooted at s is linked into r.
- Case 1. [$height(r) > height(s)$]

$$\begin{aligned} size'[r] &> size[r] \\ &\geq 2^{height(r)} \quad \longleftarrow \text{inductive hypothesis} \\ &= 2^{height'(r)}. \end{aligned}$$



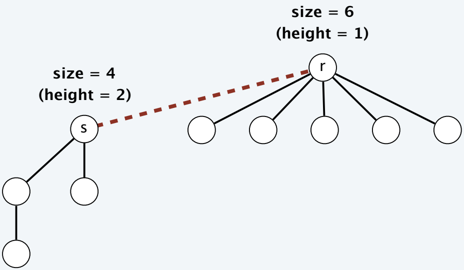
Link-by-size: analysis

Property. Using link-by-size, for every root node r : $size[r] \geq 2^{height(r)}$.

Pf. [by induction on number of links]

- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first i links.
- Tree rooted at r changes only when a smaller (or equal) size tree rooted at s is linked into r.
- Case 2. [$height(r) \leq height(s)$]

$$\begin{aligned} size'[r] &= size[r] + size[s] \\ &\geq 2\, size[s] && \longleftarrow \text{link-by-size} \\ &\geq 2 \cdot 2^{height(s)} && \longleftarrow \text{inductive hypothesis} \\ &= 2^{height(s)+1} \\ &= 2^{height'(r)}. \quad \blacksquare \end{aligned}$$



Link-by-size: analysis

Theorem. Using link-by-size, any UNION or FIND operation takes $O(\log n)$ time in the worst case, where n is the number of elements.

Pf.

- The running time of each operation is bounded by the tree height.
- By the previous property, the height is $\leq \lfloor \lg n \rfloor$. ▪

$\lg n = \log_2 n$

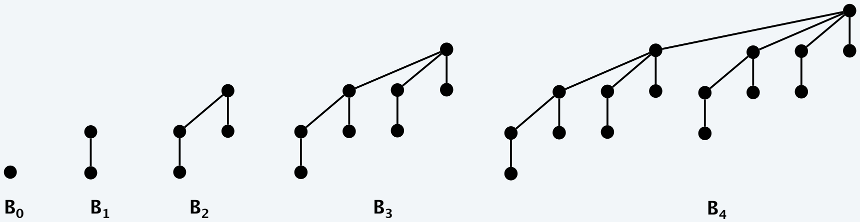
Note. The UNION operation takes $O(1)$ time except for its two calls to FIND.

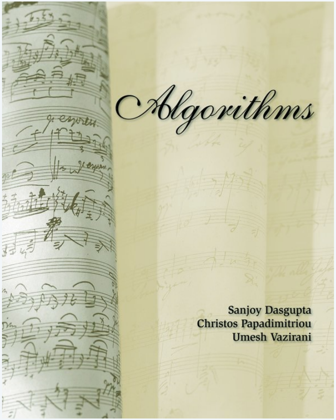
A tight upper bound

Theorem. Using link-by-size, a tree with n nodes can have height $= \lg n$.

Pf.

- Arrange $2^k - 1$ calls to UNION to form a binomial tree of order k.
- An order-k binomial tree has 2^k nodes and height k. ▪





SECTION 5.1.4

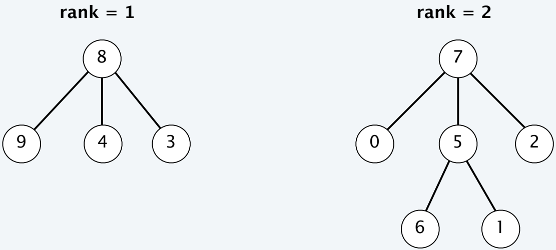
UNION–FIND

- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- *link-by-rank with path compression*
- *context*

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of new root by 1.

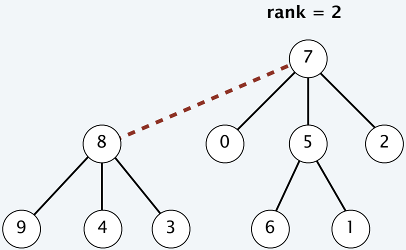
UNION(7, 3)



Note. For now, rank = height.

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of new root by 1.



Note. For now, rank = height.

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of new root by 1.

MAKE-SET(x)

$parent[x] \leftarrow x.$

$rank[x] \leftarrow 0.$

FIND(x)

WHILE ($x \neq parent[x]$)

  $x \leftarrow parent[x].$

RETURN x.

UNION(x, y)

$r \leftarrow$ FIND(x).

$s \leftarrow$ FIND(y).

IF ($r = s$) RETURN.

ELSE IF ($rank[r] > rank[s]$)

  $parent[s] \leftarrow r.$

ELSE IF ($rank[r] < rank[s]$)

  $parent[r] \leftarrow s.$

ELSE

  $parent[r] \leftarrow s.$

  $rank[s] \leftarrow rank[s] + 1.$

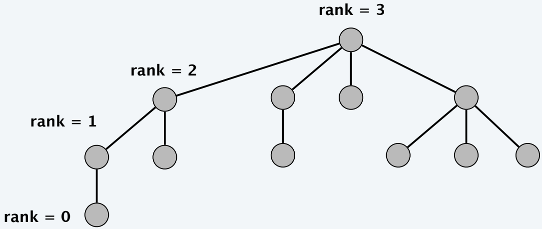
link-by-rank

Link-by-rank: properties

PROPERTY 1. If x is not a root node, then $rank[x] < rank[parent[x]]$.
Pf. A node of rank k is created only by linking two roots of rank $k-1$. ▪

PROPERTY 2. If x is not a root node, then $rank[x]$ will never change again.
Pf. Rank changes only for roots; a nonroot never becomes a root. ▪

PROPERTY 3. If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.
Pf. The parent can change only for a root, so before linking $parent[x] = x$.
After x is linked-by-rank to new root r we have $rank[r] > rank[x]$. ▪



Link-by-rank: properties

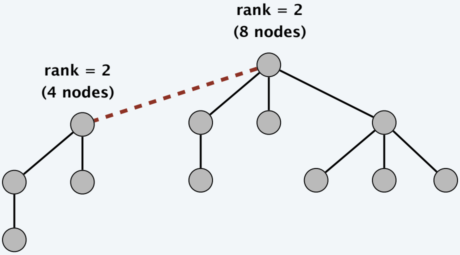
PROPERTY 4. Any root node of rank k has $\geq 2^k$ nodes in its tree.

Pf. [by induction on k]

- Base case: true for $k = 0$.
- Inductive hypothesis: assume true for $k - 1$.
- A node of rank k is created only by linking two roots of rank $k - 1$.
- By inductive hypothesis, each subtree has $\geq 2^{k-1}$ nodes
  $\Rightarrow$ resulting tree has $\geq 2^k$ nodes. ▪

PROPERTY 5. The highest rank of a node is $\leq \lfloor \lg n \rfloor$.

Pf. Immediate from PROPERTY 1 and PROPERTY 4. ▪



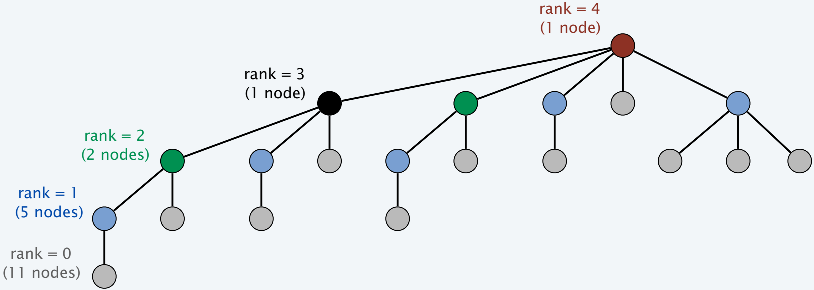
Link-by-rank: properties

PROPERTY 6. For any integer $k \geq 0$, there are $\leq n / 2^k$ nodes with rank k.

Pf.

- Any root node of rank k has $\geq 2^k$ descendants. [PROPERTY 4]
- Any nonroot node of rank k has $\geq 2^k$ descendants because:
  - it had this property just before it became a nonroot [PROPERTY 4]
  - its rank doesn't change once it became a nonroot [PROPERTY 2]
  - its set of descendants doesn't change once it became a nonroot
- Different nodes of rank k can't have common descendants. [PROPERTY 1]

■

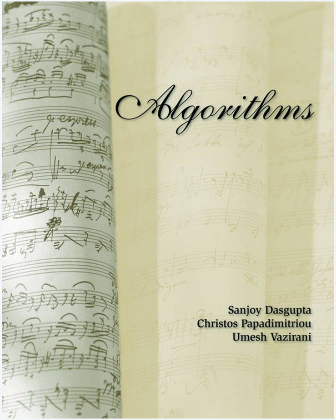


Link-by-rank: analysis

Theorem. Using link-by-rank, any UNION or FIND operation takes $O(\log n)$ time in the worst case, where n is the number of elements.

Pf.

- The running time of UNION and FIND is bounded by the tree height.
- By PROPERTY 5, the height is $\leq \lfloor \lg n \rfloor$. ▪



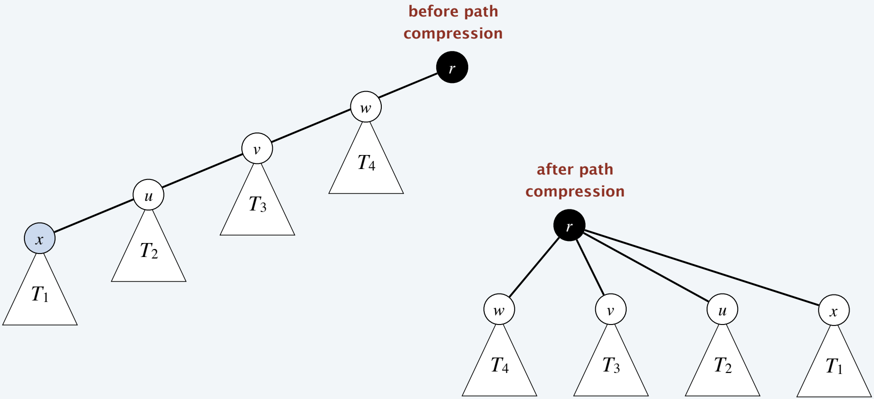
SECTION 5.1.4

UNION–FIND

- naïve linking
- link-by-size
- link-by-rank
- *path compression*
- link-by-rank with path compression
- context

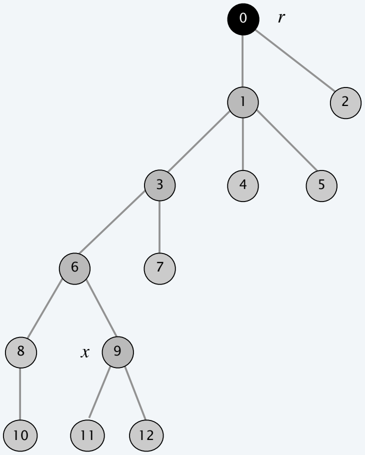
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



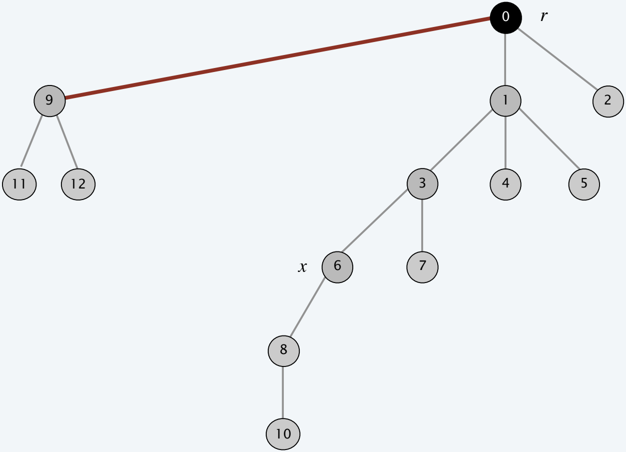
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



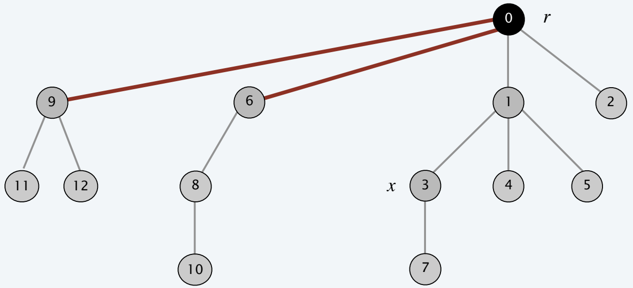
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



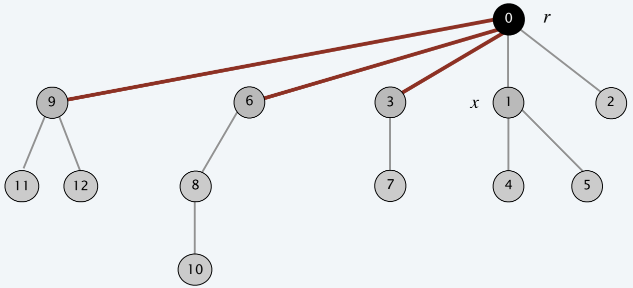
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



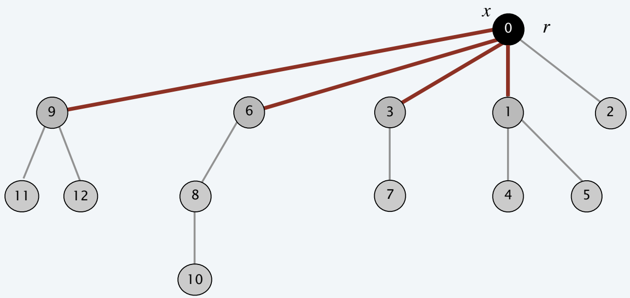
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.

FIND(x)

IF ($x \neq parent[x]$)

  $parent[x] \leftarrow$ FIND($parent[x]$).

RETURN $parent[x]$.

this FIND implementation changes the tree structure (!)

Note. Path compression does not change the rank of a node; so $height(x) \leq rank[x]$ but they are not necessarily equal.

Path compression

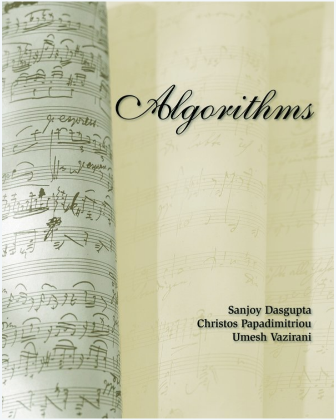
Fact. Path compression with naïve linking can require $\Omega(n)$ time to perform a single UNION or FIND operation, where n is the number of elements.

Pf. The height of the tree is $n - 1$ after the sequence of union operations: UNION$(1, 2)$, UNION$(2, 3)$, $\ldots$, UNION$(n - 1, n)$. ▪

naïve linking: link root of first tree to root of second tree

Theorem. [Tarjan–van Leeuwen 1984] Starting from an empty data structure, path compression with naïve linking performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log n)$ time.

Pf. Nontrivial (but omitted).



SECTION 5.1.4

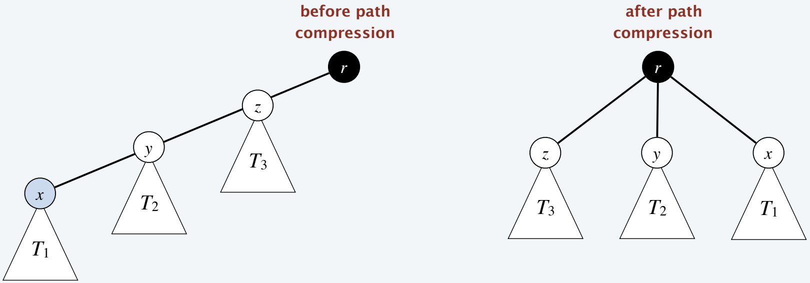
UNION–FIND

- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- ***link-by-rank with path compression***
- *context*

Link-by-rank with path compression: properties

PROPERTY. The tree roots, node ranks, and elements within a tree are the same with or without path compression.

Pf. Path compression does not create new roots, change ranks, or move elements from one tree to another. ▪



Link-by-rank with path compression: properties

PROPERTY. The tree roots, node ranks, and elements within a tree are the same with or without path compression.

COROLLARY. PROPERTY 2, 4–6 hold for link-by-rank with path compression.

PROPERTY 1. If x is not a root node, then $rank[x] < rank[parent[x]]$.
PROPERTY 2. If x is not a root node, then $rank[x]$ will never change again.
PROPERTY 3. If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.
PROPERTY 4. Any root node of rank k has $\geq 2^k$ nodes in its tree.
PROPERTY 5. The highest rank of a node is $\leq \lfloor \lg n \rfloor$.
PROPERTY 6. For any integer $k \geq 0$, there are $\leq n / 2^k$ nodes with rank k.

Bottom line. PROPERTY 1–6 hold for link-by-rank with path compression.
(but we need to recheck PROPERTY 1 and PROPERTY 3)

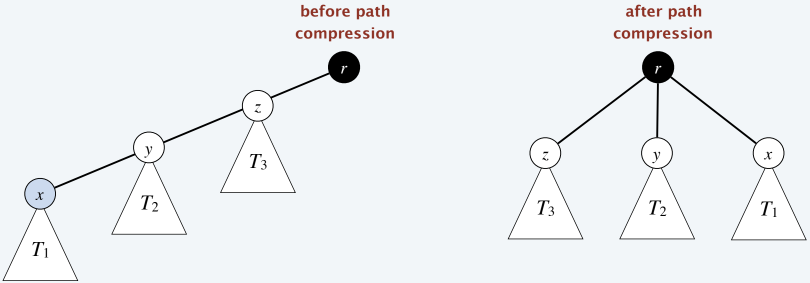
Link-by-rank with path compression: properties

PROPERTY 3. If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.

Pf. Path compression can make x point to only an ancestor of $parent[x]$.

PROPERTY 1. If x is not a root node, then $rank[x] < rank[parent[x]]$.

Pf. Path compression doesn't change any ranks, but it can change parents. If $parent[x]$ doesn't change during a path compression, the inequality continues to hold; if $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.



Iterated logarithm function

Def. The iterated logarithm function is defined by:

$$\lg^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \lg^*(\lg n) & \text{otherwise} \end{cases}$$

n	$\lg^* n$
1	0
2	1
$[3, 4]$	2
$[5, 16]$	3
$[17, 65536]$	4
$[65537, 2^{65536}]$	5

iterated lg function

Note. We have $\lg^* n \leq 5$ unless n exceeds the # atoms in the universe.

Analysis

Divide nonzero ranks into the following groups:

- $\{ 1 \}$
- $\{ 2 \}$
- $\{ 3, 4 \}$
- $\{ 5, 6, \ldots, 16 \}$
- $\{ 17, 18, \ldots, 2^{16} \}$
- $\{ 65537, 65538, \ldots, 2^{65536} \}$
- ...

Property 7. Every nonzero rank falls within one of the first $\lg^* n$ groups.

Pf. The rank is between 0 and $\lfloor \lg n \rfloor$. [PROPERTY 5]

Creative accounting

Credits. A node receives credits as soon as it ceases to be a root. If its rank is in the interval $\underbrace{\{\, k+1, k+2, \ldots, 2^k \,\}}_{\text{group k}}$, we give it 2^k credits.

Proposition. Number of credits disbursed to all nodes is $\le n \lg^* n$.

Pf.

- By PROPERTY 6, the number of nodes with rank $\ge k+1$ is at most

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \;\le\; \frac{n}{2^k}$$

- Thus, nodes in group k need at most n credits in total.
- There are $\le \lg^* n$ groups. [PROPERTY 7] ▪

Running time of FIND

Running time of FIND. Bounded by number of parent pointers followed.

- Recall: the rank strictly increases as you go up a tree. [PROPERTY 1]
- Case 0: *parent*[x] is a root $\Rightarrow$ only happens for one link per FIND.
- Case 1: *rank*[*parent*[x]] is in a higher group than *rank*[x].
- Case 2: *rank*[*parent*[x]] is in the same group as *rank*[x].

Case 1. At most $\lg^* n$ nodes on path can be in a higher group. [PROPERTY 7]

Case 2. These nodes are charged 1 credit to follow parent pointer.

- Each time x pays 1 credit, *rank*[*parent*[x]] strictly increases. [PROPERTY 1]
- Therefore, if *rank*[x] is in the group $\{ k+1, \ldots, 2^k \}$, the rank of its parent will be in a higher group before x pays 2^k credits.
- Once *rank*[*parent*[x]] is in a higher group than *rank*[x], it remains so because:
  - *rank*[x] does not change once it ceases to be a root. [PROPERTY 2]
  - *rank*[*parent*[x]] does not decrease. [PROPERTY 3]
  - thus, x has enough credits to pay until it becomes a Case 1 node. ▪

Link-by-rank with path compression

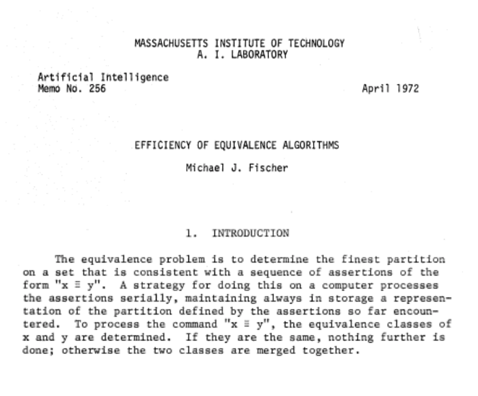
Theorem. Starting from an empty data structure, link-by-rank with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log^* n)$ time.

Union–Find

- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- *link-by-rank with path compression*
- *context*

Link-by-size with path compression

Theorem. [Fischer 1972] Starting from an empty data structure, link-by-size with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log \log n)$ time.



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 256 April 1972

EFFICIENCY OF EQUIVALENCE ALGORITHMS

Michael J. Fischer

1. INTRODUCTION

The equivalence problem is to determine the finest partition on a set that is consistent with a sequence of assertions of the form "x ≡ y". A strategy for doing this on a computer processes the assertions serially, maintaining always in storage a representation of the partition defined by the assertions so far encountered. To process the command "x ≡ y", the equivalence classes of x and y are determined. If they are the same, nothing further is done; otherwise the two classes are merged together.

Link-by-size with path compression

Theorem. [Hopcroft–Ullman 1973] Starting from an empty data structure, link-by-size with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log^* n)$ time.

SIAM J. COMPUT.
Vol. 2, No. 4, December 1973

SET MERGING ALGORITHMS*

J. E. HOPCROFT† AND J. D. ULLMAN‡

Abstract. This paper considers the problem of merging sets formed from a total of n items in such a way that at any time, the name of a set containing a given item can be ascertained. Two algorithms using different data structures are discussed. The execution times of both algorithms are bounded by a constant times $nG(n)$, where $G(n)$ is a function whose asymptotic growth rate is less than that of any finite number of logarithms of n.

Key words. algorithm, algorithmic analysis, computational complexity, data structure, equivalence algorithm, merging, property grammar, set, spanning tree

Link-by-size with path compression

Theorem. [Tarjan 1975] Starting from an empty data structure, link-by-size with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m\, \alpha(m, n))$ time, where $\alpha(m, n)$ is a functional inverse of the Ackermann function.

Efficiency of a Good But Not Linear Set Union Algorithm

ROBERT ENDRE TARJAN

University of California, Berkeley, California

ABSTRACT. Two types of instructions for manipulating a family of disjoint sets which partition a universe of n elements are considered $FIND(x)$ computes the name of the (unique) set containing element x $UNION(A, B, C)$ combines sets A and B into a new set named C. A known algorithm for implementing sequences of these instructions is examined It is shown that, if $t(m, n)$ is the maximum time required by a sequence of $m \geq n$ $FIND$s and $n - 1$ intermixed $UNION$s, then $k_1 m\alpha(m, n) \leq t(m, n) \leq k_2 m\alpha(m, n)$ for some positive constants k_1 and k_2, where $\alpha(m, n)$ is related to a functional inverse of Ackermann's function and is *very* slow-growing.

Ackermann function

Ackermann function. [Ackermann 1928] A computable function that is not primitive recursive.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Zum Hilbertschen Aufbau der reellen Zahlen.

Von

Wilhelm Ackermann in Göttingen.

Um den Beweis für die von Cantor aufgestellte Vermutung zu erbringen, daß sich die Menge der reellen Zahlen, d. h. der zahlentheoretischen Funktionen, mit Hilfe der Zahlen der zweiten Zahlklasse auszählen läßt, benutzt Hilbert einen speziellen Aufbau der zahlentheoretischen Funktionen. Wesentlich bei diesem Aufbau ist der Begriff des Typs einer Funktion. Eine Funktion vom Typ 1 ist eine solche, deren Argumente und Werte ganze Zahlen sind, also eine gewöhnliche zahlentheoretische Funktion. Die Funktionen vom Typ 2 sind die Funktionenfunktionen. Eine derartige Funktion ordnet jeder zahlentheoretischen Funktion eine Zahl zu. Eine Funktion vom Typ 3 ordnet wieder den Funktionenfunktionen Zahlen zu, usw. Die Definition der Typen läßt sich auch ins Transfinite fortsetzen, für den Gegenstand dieser Arbeit ist das aber nicht von Belang[1]).

Note. There are many inequivalent definitions.

Ackermann function

Ackermann function. [Ackermann 1928] A computable function that is not primitive recursive.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Inverse Ackermann function.

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\}$$

" I am not smart enough to understand this easily. "

— Raymond Seidel

Inverse Ackermann function

Definition.

$$\alpha_k(n) = \begin{cases} \lceil n/2 \rceil & \text{if } k = 1 \\ 0 & \text{if } n = 1 \text{ and } k \geq 2 \\ 1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} \end{cases}$$

Ex.

- $\alpha_1(n) = \lceil n / 2 \rceil$.
- $\alpha_2(n) = \lceil \lg n \rceil$ = # of times we divide n by 2, until we reach 1.
- $\alpha_3(n) = \lg^* n$ = # of times we apply the lg function to n, until we reach 1.
- $\alpha_4(n)$ = # of times we apply the iterated lg function to n, until we reach 1.

$$2 \uparrow 65536 = \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{65536 \text{ times}}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…	2^{16}	…	2^{65536}	…	$2 \uparrow 65536$
$\alpha_1(n)$	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	…	2^{15}	…	2^{65535}	…	*huge*
$\alpha_2(n)$	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	…	16	…	65536	…	$2 \uparrow 65535$
$\alpha_3(n)$	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	…	4	…	5	…	65536
$\alpha_4(n)$	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	…	3	…	3	…	4

Inverse Ackermann function

Definition.

$$\alpha_k(n) = \begin{cases} \lceil n/2 \rceil & \text{if } k = 1 \\ 0 & \text{if } n = 1 \text{ and } k \geq 2 \\ 1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} \end{cases}$$

Property. For every $n \geq 5$, the sequence $\alpha_1(n), \alpha_2(n), \alpha_3(n), \ldots$ converges to 3.

Ex. $[n = 9876!]$ $\alpha_1(n) \geq 10^{35163}$, $\alpha_2(n) = 116812$, $\alpha_3(n) = 6$, $\alpha_4(n) = 4$, $\alpha_5(n) = 3$.

One-parameter inverse Ackermann. $\alpha(n) = \min \{ k : \alpha_k(n) \leq 3 \}$.

Ex. $\alpha(9876!) = 5$.

Two-parameter inverse Ackermann. $\alpha(m, n) = \min \{ k : \alpha_k(n) \leq 3 + m / n \}$.

A tight lower bound

Theorem. [Fredman–Saks 1989] In the worst case, any CELL-PROBE$(\log n)$ algorithm requires $\Omega(m\,\alpha(m, n))$ time to perform an intermixed sequence of m MAKE-SET, UNION, and FIND operations on a set of n elements.

Cell-probe model. [Yao 1981] Count only number of words of memory accessed; all other operations are free.

The Cell Probe Complexity of Dynamic Data Structures

Michael L. Fredman [1]
Bellcore and
U.C. San Diego

Michael E. Saks [2]
U.C. San Diego,
Bellcore and
Rutgers University

1. Summary of Results

Dynamic data structure problems involve the representation of data in memory in such a way as to permit certain types of modifications of the data (**updates**) and certain types of questions about the data (**queries**). This paradigm encompasses many fundamental problems in computer science.

The purpose of this paper is to prove new lower and upper bounds on the time per operation to implement solutions to some familiar dynamic data structure problems including list representation, subset ranking, partial sums, and the set union problem . The main features of our lower bounds are:

(1) They hold in the *cell probe* model of computation (A. Yao

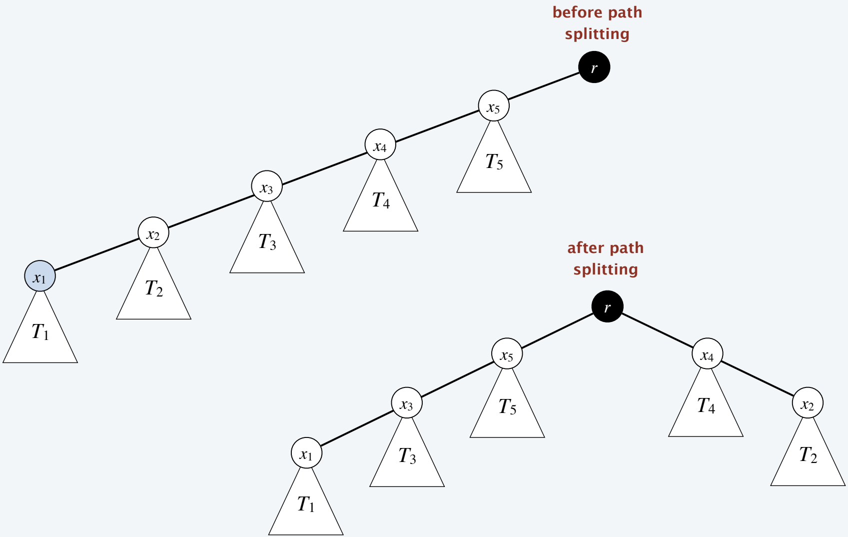
register size from $\log n$ to polylog(n) only reduces the time complexity by a constant factor. On the other hand, decreasing the register size from $\log n$ to 1 increases time complexity by a $\log n$ factor for one of the problems we consider and only a $\log\log n$ factor for some other problems.

The first two specific data structure problems for which we obtain bounds are:

List Representation. This problem concerns the represention of an ordered list of at most n (not necessarily distinct) elements from the universe $U = \{1, 2, ..., n\}$. The operations to be supported are **report**(k), which returns the k^{th} element of the list, **insert**(k, u) which inserts element u into the list between the

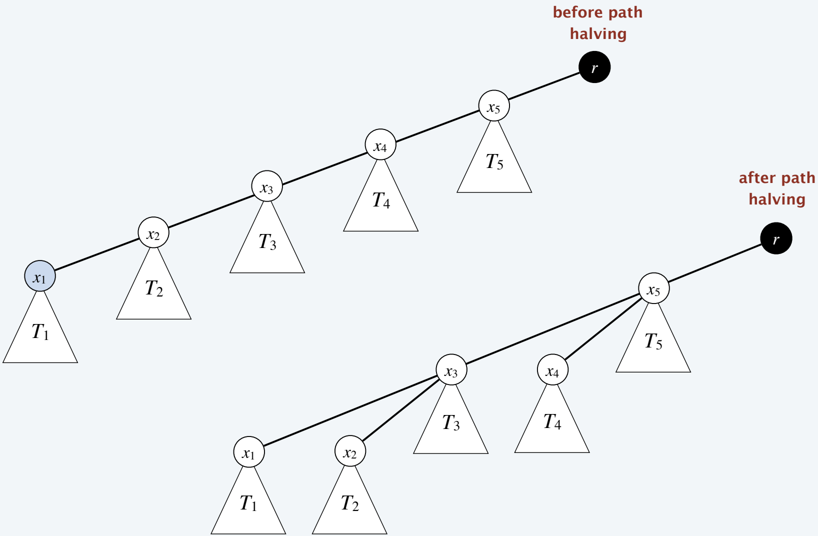
Path compaction variants

Path splitting. Make every node on path point to its grandparent.



Path compaction variants

Path halving. Make every other node on path point to its grandparent.



Linking variants

Link-by-size. Number of nodes in tree.

Link-by-rank. Rank of tree.

Link-by-random. Label each element with a random real number between 0.0 and 1.0. Link root with smaller label into root with larger label.

Disjoint-sets data structures

Theorem. [Tarjan–van Leeuwen 1984] Starting from an empty data structure, link-by- { size, rank } combined with { path compression, path splitting, path halving } performs any intermixed sequence of $m \ge n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m\,\alpha(m, n))$ time.

Worst-Case Analysis of Set Union Algorithms

ROBERT E. TARJAN

AT&T Bell Laboratories, Murray Hill, New Jersey

AND

JAN VAN LEEUWEN

University of Utrecht, Utrecht, The Netherlands

Abstract. This paper analyzes the asymptotic worst-case running time of a number of variants of the well-known method of path compression for maintaining a collection of disjoint sets under union. We show that two one-pass methods proposed by van Leeuwen and van der Weide are asymptotically optimal, whereas several other methods, including one proposed by Rem and advocated by Dijkstra, are slower than the best methods.

Part III

Algorithm Design Techniques

Algorithmic paradigms

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

Divide-and-conquer. Break up a problem into independent subproblems; solve each subproblem; combine solutions to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form solution to large subproblem.

fancy name for
caching intermediate results
in a table for later reuse

DIVIDE AND CONQUER

DIVIDE AND CONQUER

Nothing is particularly hard if you divide it into small jobs.

Henry Ford

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Most common usage.

- Divide problem of size n into two subproblems of size $n/2$. ← $O(n)$ time
- Solve (conquer) two subproblems recursively.
- Combine two solutions into overall solution. ← $O(n)$ time

Consequence.

- Brute force: $\Theta(n^2)$.
- Divide-and-conquer: $O(n \log n)$.

attributed to Julius Caesar

DIVIDE AND CONQUER

MAXIMAL AND MINIMAL ELEMENTS

NAIVE ALGORITHM

complexity: number of comparisons

Algorithm: Iterative MaxMin

Input: sequence $S[1 \dots n]$

Output: maximal and minimal element

```
max ← S[1]; min ← S[1]
for i ← 2 to n do
    if S[i] > max then max ← S[i]
    if S[i] < min then min ← S[i]
return max, min
```

$2(n-1)$ comparisons

DIVIDE AND CONQUER ALGORITHM

divide the sequence into *two equally sizek* subsequences

solve find maximal and minimal elements of both subsequences

combine greater of the maximal elements is the maximal element of the whole sequence(*the same for the minimal element*)

Algorithm: MaxMin

Input: sequence $S[1 \dots n]$, indices x, y

Output: maximal and minimal element of $S[x \dots y]$

if $y = x$ **then return** $(S[x], S[x])$

if $y = x + 1$ **then return** $(\max(S[x], S[y]), \min(S[x], S[y]))$

if $y > x + 1$ **then**

  $(l_1, l_2) \leftarrow \text{MaxMin}(S, x, \lfloor (x+y)/2 \rfloor)$

  $(r_1, r_2) \leftarrow \text{MaxMin}(S, \lfloor (x+y)/2 \rfloor + 1, y)$

return $(\max(l_1, r_1), \min(l_2, r_2))$

correctness induction w.r.t. the length of the sequence

complexity

$$T(n) = \begin{cases} 1 & \text{for } n = 2 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2 & \text{for } n > 2 \end{cases}$$

by induction w.r.t. n we can check that $T(n) < \frac{5}{3}n - 2$

$n = 2$ $T(2) = 1$ and $1 < \frac{5}{3} \cdot 2 - 2$

$n > 2$ assumption: the inequality is true for all i, $2 \leq i < n$

let us prove the inequality for n

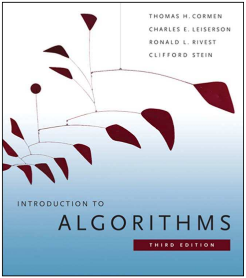
$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2 \\ &< \frac{5}{3}\lfloor n/2 \rfloor - 2 + \frac{5}{3}\lceil n/2 \rceil - 2 + 2 = \frac{5}{3}n - 2 \end{aligned}$$

DIVIDE AND CONQUER

PEAK FINDING

6.006

Introduction to Algorithms



Lecture 2: Peak Finding

Prof. Erik Demaine

1D Peak Finding

- Given an array $A[0..n-1]$:

A:	$-\infty$	1	2	6	5	3	7	4	$-\infty$
		0	1	2	3	4	5	6	

- $A[i]$ is a **peak** if it is not smaller than its neighbor(s):

$$A[i-1] \leq A[i] \geq A[i+1]$$

where we imagine

$$A[-1] = A[n] = -\infty$$

- <u>Goal:</u> Find *any* peak

"Brute Force" Algorithm

- Test all elements for peakyness

for i in range(n):
    if $A[i-1] \le A[i] \ge A[i+1]$: $O(1)$ $O(n)$
        return i

A:

1	2	6	5	3	7	4
0	1	2	3	4	5	6

Algorithm 1½

- max(A)
  - Global maximum is a local maximum

```
m = 0
for i in range(1, n):
        if A[i] > A[m]:
                m = i
return m
```

$\Theta(1)$

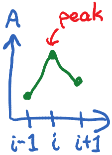
$\Theta(n)$

A:

1	2	6	5	3	7	4
0	1	2	3	4	5	6

Cleverer Idea

- Look at any element $A[i]$ and its neighbors $A[i-1]$ & $A[i+1]$
  - If peak: return i
  - Otherwise: locally rising on some side
    - Must be a peak in that direction
    - So can throw away rest of array, leaving $A[:i]$ or $A[i+1:]$



A:	1	2	6	5	3	7	4
	0	1	2	3	4	5	6

Where to Sample?

- Want to minimize the worst-case remaining elements in array
  - Balance $A[:i]$ of length i with $A[i+1:]$ of length $n-i-1$
  - $i \;=\; n-i-1$
  - $i \;=\; (n-1)/2$: **middle element**
  - Reduce n to $(n-1)/2$

A:	1	2	6	5	3	7	4
	0	1	2	3	4	5	6

Algorithm

peak1d(A, i, j):
    $m = \lfloor (i + j)/2 \rfloor$
    if $A[m-1] \leq A[m] \geq A[m+1]$:
        return m
    elif $A[m-1] > A[m]$:
        return peak1d$(A, i, m-1)$
    elif $A[m] < A[m+1]$:
        return peak1d$(A, m+1, j)$

A:

1	2	6	5	3	7	4
0	1	2	3	4	5	6

Divide & Conquer

- General design technique:

1. **Divide** input into part(s)
2. **Conquer** each part recursively
3. **Combine** result(s) to solve original problem

- 1D peak:

1. One half
2. Recurse
3. Return

Divide & Conquer Analysis

- **Recurrence** for time $T(n)$ taken by problem size n

1. **Divide** input into part(s):
   $n_1, n_2, \dots, n_k$
2. **Conquer** each part recursively
3. **Combine** result(s) to solve original problem

$T(n) =$

divide cost +

$T(n_1) + T(n_2) + \cdots + T(n_k)$

+ combine cost

1D Peak Finding Analysis

- Divide problem into 1 problem of size $\sim\frac{n}{2}$
- Divide cost: $O(1)$
- Combine cost: $O(1)$
- Recurrence:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Solving Recurrence

don't use O(1) notation to keep track of constant

$$T(n) = T\left(\frac{n}{2}\right) + c$$

$$T(n) = T\left(\frac{n}{4}\right) + c + c$$

$$T(n) = T\left(\frac{n}{8}\right) + c + c + c$$

$$T(n) = T\left(\frac{n}{2^k}\right) + c\,k$$

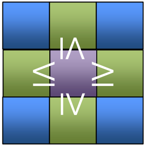
$$T(n) = T\left(\frac{\cancel{n}}{\cancel{2^{\lg n}}}\right) + c \lg n$$

$$T(n) = T(1) + c \lg n$$

$$T(n) = \Theta(\lg n)$$

2D Peak Finding

- Given $n \times n$ matrix of numbers
- Want an entry not smaller than its (up to) 4 neighbors:



9	3	5	2	4	9	8
7	2	5	1	4	0	3
9	8	9	3	2	4	8
7	6	3	1	3	2	3
9	0	6	0	4	6	4
8	9	8	0	5	3	0
2	1	2	1	1	1	1

Divide & Conquer #0

- Looking at center element doesn't split the problem into pieces…

9	3	5	2	4	9	8
7	2	5	1	4	0	3
9	8	9	3	2	4	8
7	6	3	1	3	2	3
9	0	6	0	4	6	4
8	9	8	0	5	3	0
2	1	2	1	1	1	1

Divide & Conquer #½

- Consider max element in each column
- 1D algorithm would solve max array in $O(\lg n)$ time
- But $\Theta(n^2)$ time to compute max array

9	3	5	2	4	9	8
7	2	5	1	4	0	3
9	8	9	3	2	4	8
7	6	3	1	3	2	3
9	0	6	0	4	6	4
8	9	8	0	5	3	0
2	1	2	1	1	1	1

9	9	9	3	5	9	8

Divide & Conquer #1

- Look at center column
- Find global max within
- If peak: return it
- Else:
  - Larger left/right neighbor
  - Larger max in that column
  - Recurse in left/right half
- <u>Base case:</u> 1 column
  - Return global max within

9	3	5	2	4	9	8
7	2	5	1	4	0	3
9	8	9	3	2	4	8
7	6	3	1	3	2	3
9	0	6	0	4	6	4
8	9	8	0	5	3	0
2	1	2	1	1	1	1

9	9	9	3	5	9	8

Analysis #1

- $O(n)$ time to find max in column
- $O(\lg n)$ iterations (like binary search)
- $O(n \lg n)$ time total

9	3	5	2	4	9	8
7	2	5	1	4	0	3
9	8	9	3	2	4	8
7	6	3	1	3	2	3
9	0	6	0	4	6	4
8	9	8	0	5	3	0
2	1	2	1	1	1	1

- Can we do better?

Divide & Conquer #2

- Look at boundary, center row, and center column (**window**)
- Find global max within
- If it's a peak: return it
- Else:
  - Find larger neighbor
  - Can't be in window
  - Recurse in quadrant, including green boundary

0	0	0	0	0	0	0	0	0
0	9	3	5	2	4	9	8	0
0	7	2	5	1	4	0	3	0
0	9	8	9	3	2	4	8	0
0	7	6	3	1	3	2	3	0
0	9	0	6	0	4	6	4	0
0	8	9	8	0	5	3	0	0
0	2	1	2	1	1	1	1	0
0	0	0	0	0	0	0	0	0

Correctness

- Lemma: If you enter a quadrant, it contains a peak of the overall array [climb up]
- Invariant: Maximum element of window never decreases as we descend in recursion
- Theorem: Peak in visited quadrant is also peak in overall array

0	0	0	0	0	0	0	0	0
0	9	3	5	2	4	9	8	0
0	7	2	5	1	4	0	3	0
0	9	8	9	3	2	4	8	0
0	7	6	3	1	3	2	3	0
0	9	0	6	0	4	6	4	0
0	8	9	8	0	5	3	0	0
0	2	1	2	1	1	1	1	0
0	0	0	0	0	0	0	0	0

→ proofs in recitation

Analysis #2

- Reduce $n \times n$ matrix to $\sim \frac{n}{2} \times \frac{n}{2}$ submatrix in $O(n)$ time (|window|)

0	0	0	0	0	0	0	0	0
0	9	3	5	2	4	9	8	0
0	7	2	5	1	4	0	3	0
0	9	8	9	3	2	4	8	0
0	7	6	3	1	3	2	3	0
0	9	0	6	0	4	6	4	0
0	8	9	8	0	5	3	0	0
0	2	1	2	1	1	1	1	0
0	0	0	0	0	0	0	0	0

$$T(n) = T\left(\frac{n}{2}\right) + c\,n$$

$$T(n) = T\left(\frac{n}{4}\right) + c\,\frac{n}{2} + c\,n$$

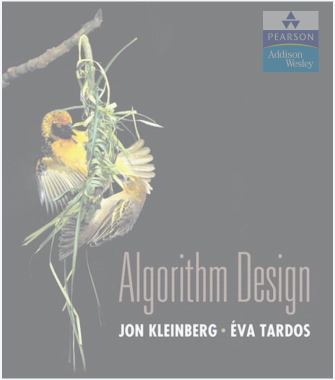
$$T(n) = T\left(\frac{n}{8}\right) + c\,\frac{n}{4} + c\,\frac{n}{2} + c\,n$$

$\Theta(n)$

$$T(n) = T(1) + c\left(1 + 2 + 4 + \cdots + \frac{n}{4} + \frac{n}{2} + n\right)$$

DIVIDE AND CONQUER

CLOSEST PAIR OF POINTS



SECTION 5.4

5. DIVIDE AND CONQUER

- *mergesort*
- *counting inversions*
- *randomized quicksort*
- *median and selection*
- *closest pair of points*

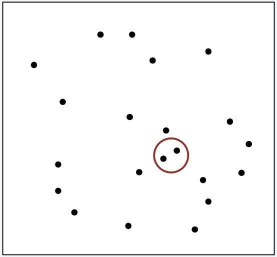
Closest pair of points

Closest pair problem. Given n points in the plane, find a pair of points with the smallest Euclidean distance between them.

Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.

fast closest pair inspired fast algorithms for these problems



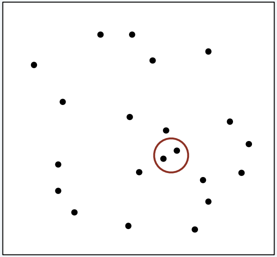
Closest pair of points

Closest pair problem. Given n points in the plane, find a pair of points with the smallest Euclidean distance between them.

Brute force. Check all pairs with $\Theta(n^2)$ distance calculations.

1D version. Easy $O(n \log n)$ algorithm if points are on a line.

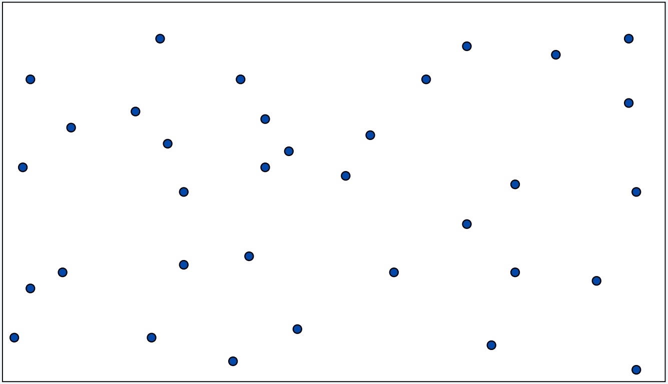
Non-degeneracy assumption. No two points have the same x-coordinate.



Closest pair of points: first attempt

Sorting solution.

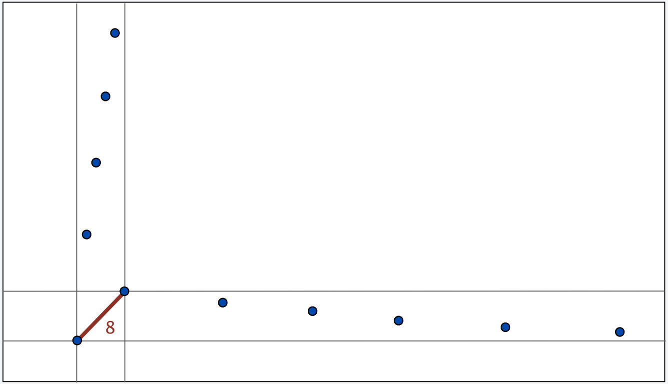
- Sort by x-coordinate and consider nearby points.
- Sort by y-coordinate and consider nearby points.



Closest pair of points: first attempt

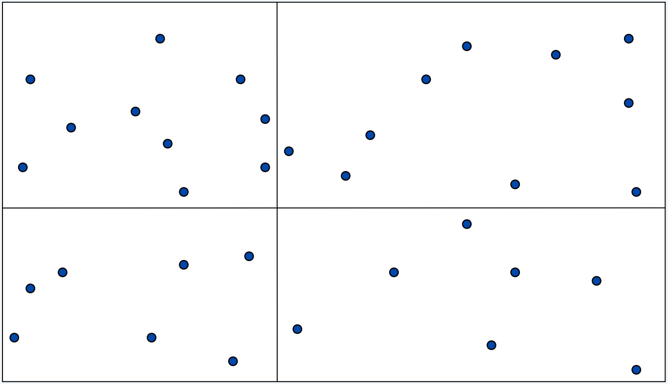
Sorting solution.

- Sort by x-coordinate and consider nearby points.
- Sort by y-coordinate and consider nearby points.



Closest pair of points: second attempt

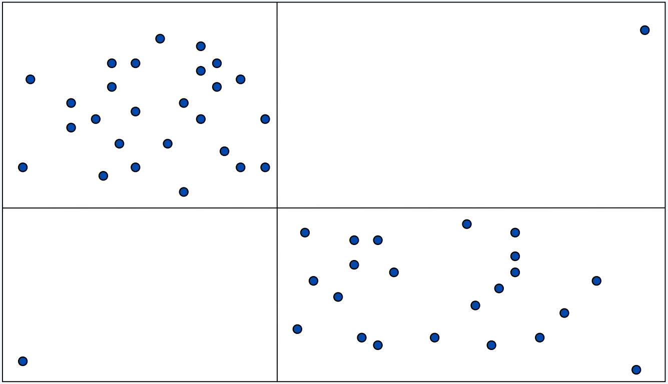
Divide. Subdivide region into 4 quadrants.



Closest pair of points: second attempt

Divide. Subdivide region into 4 quadrants.

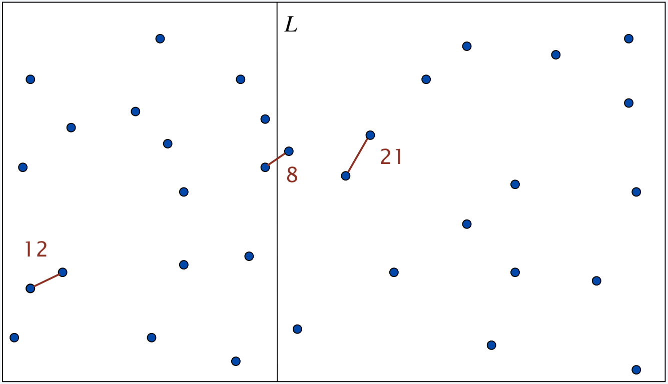
Obstacle. Impossible to ensure $n/4$ points in each piece.



Closest pair of points: divide-and-conquer algorithm

- Divide: draw vertical line L so that $n/2$ points on each side.
- Conquer: find closest pair in each side recursively.
- Combine: find closest pair with one point in each side.
- Return best of 3 solutions.

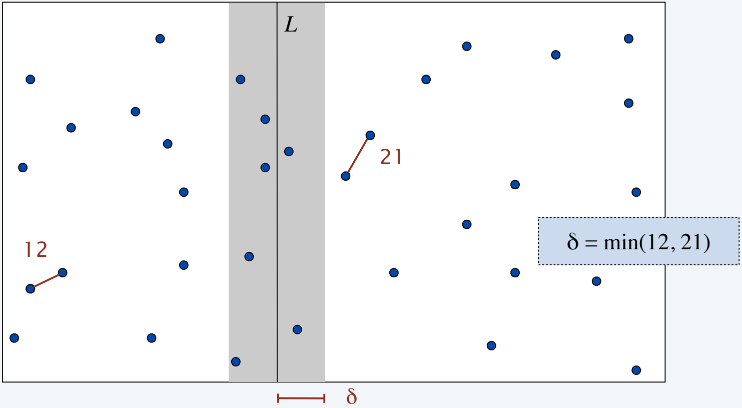
seems like $\Theta(n^2)$



How to find closest pair with one point in each side?

Find closest pair with one point in each side, assuming that distance < δ.

- Observation: suffices to consider only those points within δ of line L.

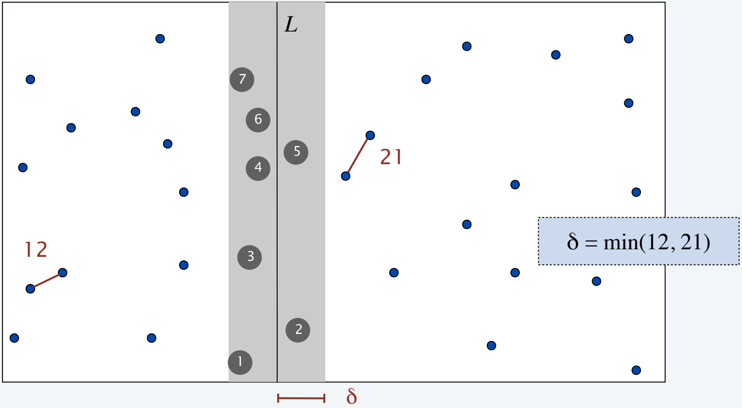


How to find closest pair with one point in each side?

Find closest pair with one point in each side, assuming that distance < δ.

- Observation: suffices to consider only those points within δ of line L.
- Sort points in 2δ-strip by their y-coordinate.
- Check distances of only those points within 7 positions in sorted list!

why?



How to find closest pair with one point in each side?

Def. Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.

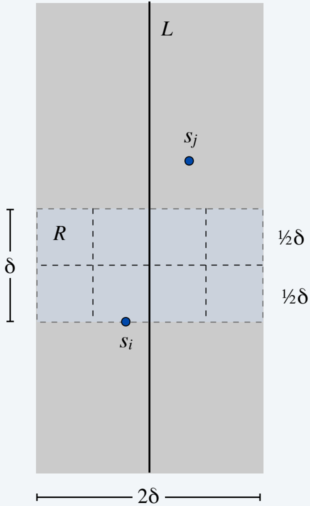
Claim. If $|j - i| > 7$, then the distance between s_i and s_j is at least δ.

Pf.

- Consider the 2δ-by-δ rectangle R in strip whose min y-coordinate is y-coordinate of s_i.
- Distance between s_i and any point s_j above R is $\geq \delta$.
- Subdivide R into 8 squares.
- At most 1 point per square.
- At most 7 other points can be in R. ▪

diameter is $\delta / \sqrt{2} < \delta$

constant can be improved with more refined geometric packing argument



Closest pair of points: divide-and-conquer algorithm

CLOSEST-PAIR$(p_1, p_2, \ldots, p_n)$

Compute vertical line L such that half the points are on each side of the line. ← $O(n)$

$\delta_1 \leftarrow$ CLOSEST-PAIR(points in left half). ← $T(n / 2)$

$\delta_2 \leftarrow$ CLOSEST-PAIR(points in right half). ← $T(n / 2)$

$\delta \leftarrow \min \{ \delta_1 , \delta_2 \}$.

Delete all points further than δ from line L. ← $O(n)$

Sort remaining points by y-coordinate. ← $O(n \log n)$

Scan points in y-order and compare distance between each point and next 7 neighbors. If any of these distances is less than δ, update δ. ← $O(n)$

RETURN δ.

Divide-and-conquer: quiz 6

What is the solution to the following recurrence?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) \; + \; T(\lceil n/2 \rceil) \; + \; \Theta(n \log n) & \text{if } n > 1 \end{cases}$$

A. $T(n) = \Theta(n)$.

B. $T(n) = \Theta(n \log n)$.

C. $T(n) = \Theta(n \log^2 n)$.

D. $T(n) = \Theta(n^2)$.

Refined version of closest-pair algorithm

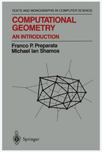
Q. How to improve to $O(n \log n)$?

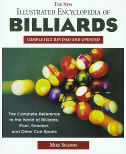
A. Don't sort points in strip from scratch each time.

- Each recursive call returns two lists: all points sorted by x-coordinate, and all points sorted by y-coordinate.
- Sort by merging two pre-sorted lists.

Theorem. [Shamos 1975] The divide-and-conquer algorithm for finding a closest pair of points in the plane can be implemented in $O(n \log n)$ time.

Pf.
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) \;+\; T(\lceil n/2 \rceil) \;+\; \Theta(n) & \text{if } n > 1 \end{cases}$$





What is the complexity of the 2D closest pair problem?

A. $\Theta(n)$.

B. $\Theta(n \log^* n)$.

C. $\Theta(n \log \log n)$.

D. $\Theta(n \log n)$.

E. Not even Tarjan knows.

Computational complexity of closest-pair problem

Theorem. [Ben-Or 1983, Yao 1989] In quadratic decision tree model, any algorithm for closest pair (even in 1D) requires $\Omega(n \log n)$ quadratic tests.

$(x_1 - x_2)^2 + (y_1 - y_2)^2$

Lower Bounds for Algebraic Computation Trees

with Integer Inputs*

Andrew Chi-Chih Yao
Department of Computer Science
Princeton University
Princeton, New Jersey 08544

Theorem. [Rabin 1976] There exists an algorithm to find the closest pair of points in the plane whose expected running time is $O(n)$.

not subject to $\Omega(n \log n)$ lower bound
because it uses the floor function

A NOTE ON RABIN'S NEAREST-NEIGHBOR ALGORITHM*

Steve FORTUNE and John HOPCROFT
Department of Computer Science, Cornell University, Ithaca, NY, U.S.A.

Received 20 July 1978, revised version received 21 August 1978

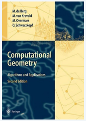
Probabilistic algorithms, nearest neighbor, hashing

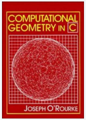
Digression: computational geometry

Ingenious divide-and-conquer algorithms for core geometric problems.

problem	brute	clever
closest pair	$O(n^2)$	$O(n \log n)$
farthest pair	$O(n^2)$	$O(n \log n)$
convex hull	$O(n^2)$	$O(n \log n)$
Delaunay/Voronoi	$O(n^4)$	$O(n \log n)$
Euclidean MST	$O(n^2)$	$O(n \log n)$

running time to solve a 2D problem with n points

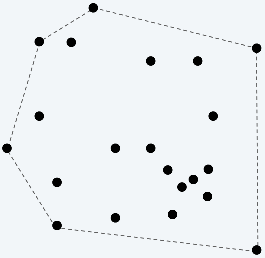




Note. 3D and higher dimensions test limits of our ingenuity.

Convex hull

The convex hull of a set of n points is the smallest perimeter fence enclosing the points.

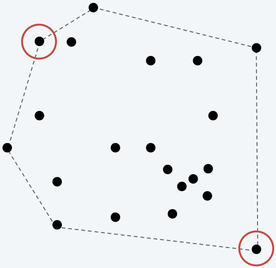


Equivalent definitions.

- Smallest area convex polygon enclosing the points.
- Intersection of all convex set containing all the points.

Farthest pair

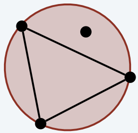
Given n points in the plane, find a pair of points with the largest Euclidean distance between them.



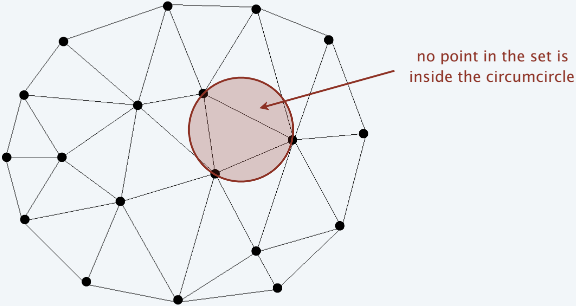
Fact. Points in farthest pair are extreme points on convex hull.

Delaunay triangulation

The Delaunay triangulation is a triangulation of n points in the plane such that no point is inside the circumcircle of any triangle.



point inside circumcircle of 3 points



Delaunay triangulation of 19 points

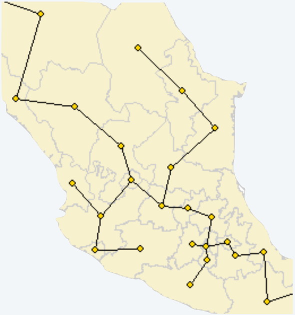
Some useful properties.

- No edges cross.
- Among all triangulations, it maximizes the minimum angle.
- Contains an edge between each point and its nearest neighbor.

Euclidean MST

Given n points in the plane, find MST connecting them.

[distances between point pairs are Euclidean distances]



Fact. Euclidean MST is subgraph of Delaunay triangulation.

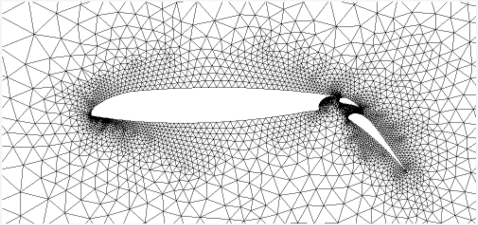
Implication. Can compute Euclidean MST in $O(n \log n)$ time.

- Compute Delaunay triangulation.
- Compute MST of Delaunay triangulation. ← it's planar ($\leq 3n$ edges)

Computational geometry applications

Applications.

- Robotics.
- VLSI design.
- Data mining.
- Medical imaging.
- Computer vision.
- Scientific computing.
- Finite-element meshing.
- Astronomical simulation.
- Models of physical world.
- Geographic information systems.
- Computer graphics (movies, games, virtual reality).



airflow around an aircraft wing

http://www.ics.uci.edu/~eppstein/geom.html

DYNAMIC PROGRAMMING

DYNAMIC PROGRAMMING - ETYMOLOGY

Jeff Erickson: Algorithms

The dynamic programming paradigm was developed by Richard Bellman in the mid-1950s, while working at the RAND Corporation. Bellman deliberately chose the name dynamic programming to hide the mathematical character of his work from his military bosses, who were actively hostile toward anything resembling mathematical research. Here, the word programming does not refer to writing code, but rather to the older sense of planning *or* scheduling, *typically by filling in a table. For example, sports programs and theater programs are schedules of important events; television programming involves filling each available time slot with a show; degree programs are schedules of classes to be taken. The Air Force funded Bellman an other to develop methods for constructing training and logistics schedules, or as they called them, programs. The word dynamic is meant to suggest that the table is filled in over time, rather than all at once (as in linear programming).*

Dynamic programming history

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

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense had pathological fear of mathematical research.
- Bellman sought a "dynamic" adjective to avoid conflict.

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

Dynamic programming applications

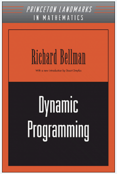
Application areas.

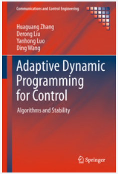
- Computer science: AI, compilers, systems, graphics, theory, ...
- Operations research.
- Information theory.
- Control theory.
- Bioinformatics.

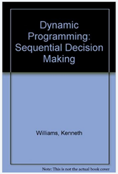
Some famous dynamic programming algorithms.

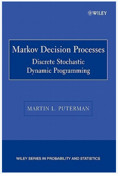
- Avidan–Shamir for seam carving.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Bellman–Ford–Moore for shortest path.
- Knuth–Plass for word wrapping text in $\TeX$.
- Cocke–Kasami–Younger for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman for sequence alignment.

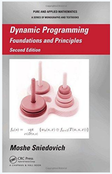
Dynamic programming books

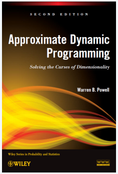


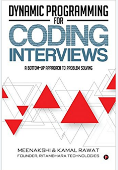


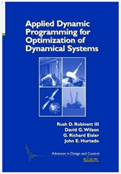


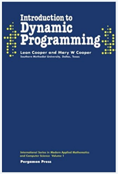


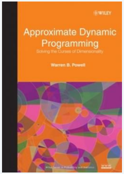


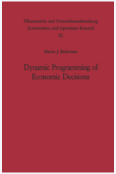


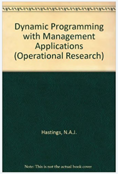


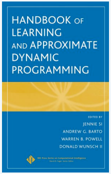


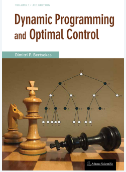


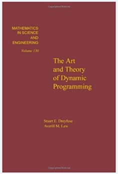


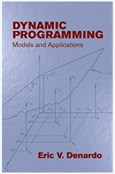


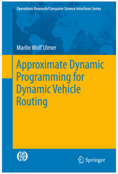


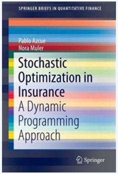


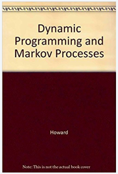


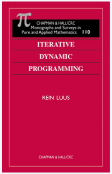


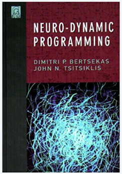






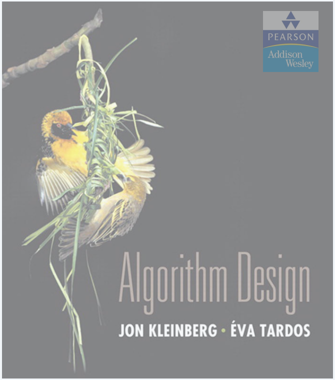






DYNAMIC PROGRAMMING

INTERVAL SCHEDULING



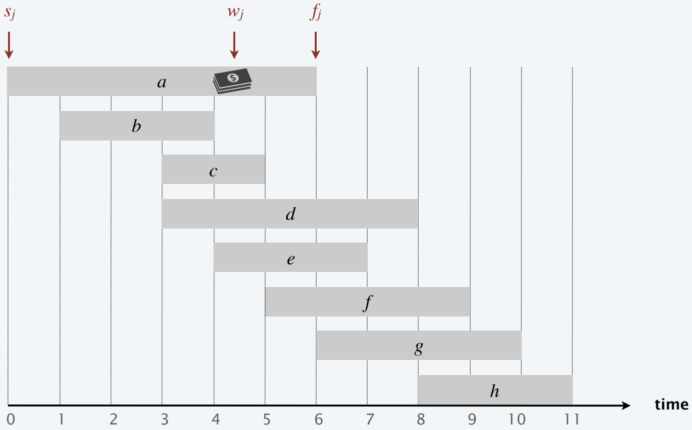
SECTIONS 6.1–6.2

6. DYNAMIC PROGRAMMING I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Weighted interval scheduling

- Job j starts at s_j, finishes at f_j, and has weight $w_j > 0$.
- Two jobs are compatible if they don't overlap.
- Goal: find max-weight subset of mutually compatible jobs.



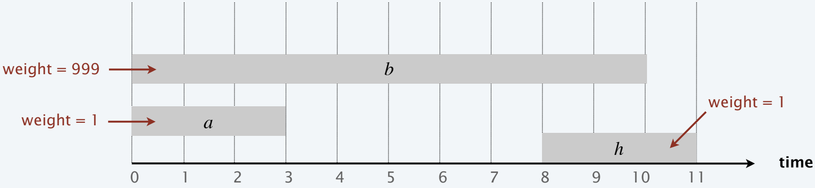
Earliest-finish-time first algorithm

Earliest finish-time first.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Recall. Greedy algorithm is correct if all weights are 1.

Observation. Greedy algorithm fails spectacularly for weighted version.



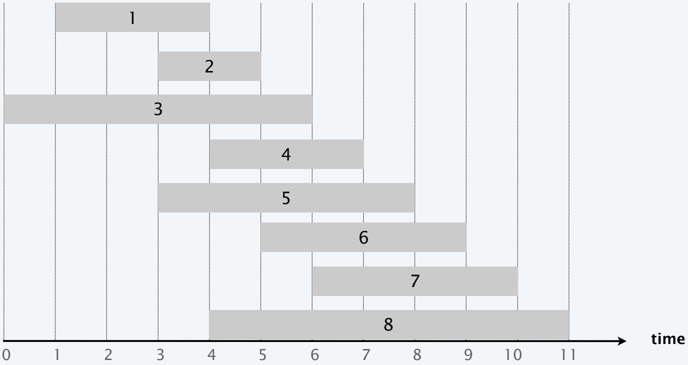
Weighted interval scheduling

Convention. Jobs are in ascending order of finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

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

Ex. $p(8) = 1, p(7) = 3, p(2) = 0.$

i is leftmost interval that ends before j begins



Dynamic programming: binary choice

Def. $OPT(j)$ = max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs $1, 2, ..., j$.

Goal. $OPT(n)$ = max weight of any subset of mutually compatible jobs.

Case 1. $OPT(j)$ does not select job j.

- Must be an optimal solution to problem consisting of remaining jobs $1, 2, ..., j-1$.

optimal substructure property
(proof via exchange argument)

Case 2. $OPT(j)$ selects job j.

- Collect profit w_j.
- Can't use incompatible jobs $\{ p(j)+1, p(j)+2, ..., j-1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., p(j)$.

Bellman equation. $$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ OPT(j-1),\ w_j + OPT(p(j)) \} & \text{if } j > 0 \end{cases}$$

Weighted interval scheduling: brute force

BRUTE-FORCE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF $(j = 0)$

  RETURN 0.

ELSE

  RETURN $\max\{$COMPUTE-OPT($j-1$), w_j + COMPUTE-OPT($p[j]$) $\}$.

Dynamic programming: quiz 1

What is running time of COMPUTE-OPT(n) in the worst case?

A. $\Theta(n \log n)$

B. $\Theta(n^2)$

C. $\Theta(1.618^n)$

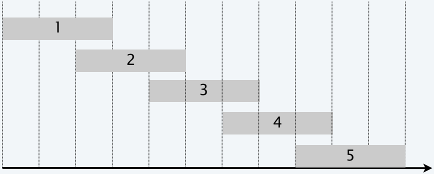
D. $\Theta(2^n)$

```
COMPUTE-OPT(j)
_______________________________
IF (j = 0)
   RETURN 0.
ELSE
   RETURN max {COMPUTE-OPT(j – 1), w_j + COMPUTE-OPT(p[j]) }.
```

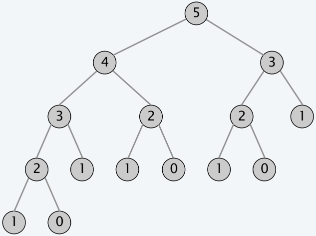
Weighted interval scheduling: brute force

Observation. Recursive algorithm is spectacularly slow because of overlapping subproblems $\Rightarrow$ exponential-time algorithm.

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



p(1) = 0, p(j) = j–2



recursion tree

Weighted interval scheduling: memoization

Top-down dynamic programming (memoization).

- Cache result of subproblem j in $M[j]$.
- Use $M[j]$ to avoid solving subproblem j more than once.

TOP-DOWN$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

$M[0] \leftarrow 0.$ ← global array

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

  $M[j] \leftarrow \max \{$ M-COMPUTE-OPT $(j-1)$, w_j + M-COMPUTE-OPT$(p[j])$ $\}$.

RETURN $M[j]$.

Weighted interval scheduling: running time

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

Pf.

- Sort by finish time: $O(n \log n)$ via mergesort.
- Compute $p[j]$ for each j : $O(n \log n)$ via binary search.

- $\text{M-C}\textsc{ompute}\text{-O}\textsc{pt}(j)$: each invocation takes $O(1)$ time and either
  - (1) returns an initialized value $M[j]$
  - (2) initializes $M[j]$ and makes two recursive calls

- Progress measure $\Phi = \#$ initialized entries among $M[1..n]$.
  - initially $\Phi = 0$; throughout $\Phi \le n$.
  - (2) increases Φ by $1 \Rightarrow \le 2n$ recursive calls.

- Overall running time of $\text{M-C}\textsc{ompute}\text{-O}\textsc{pt}(n)$ is $O(n)$. ▪

Those who cannot remember the past are condemned to repeat it.

- Dynamic Programming

Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find optimal solution?

A. Make a second pass by calling FIND-SOLUTION(n).

```
FIND-SOLUTION(j)
_____________________________________________
IF (j = 0)
   RETURN ∅.
ELSE IF (w_j + M[p[j]] > M[j–1])
   RETURN { j } ∪ FIND-SOLUTION(p[j]).
ELSE
   RETURN FIND-SOLUTION(j–1).
```

$$M[j] = \max \{ M[j-1],\ w_j + M[p[j]] \}.$$

Analysis. # of recursive calls $\leq n \Rightarrow O(n)$.

Weighted interval scheduling: bottom-up dynamic programming

Bottom-up dynamic programming. Unwind recursion.

BOTTOM-UP$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$.

$M[0] \leftarrow 0$.

previously computed values

FOR $j = 1$ TO n

$\quad M[j] \leftarrow \max \{ M[j-1], \; w_j + M[p[j]] \}$.

Running time. The bottom-up version takes $O(n \log n)$ time.

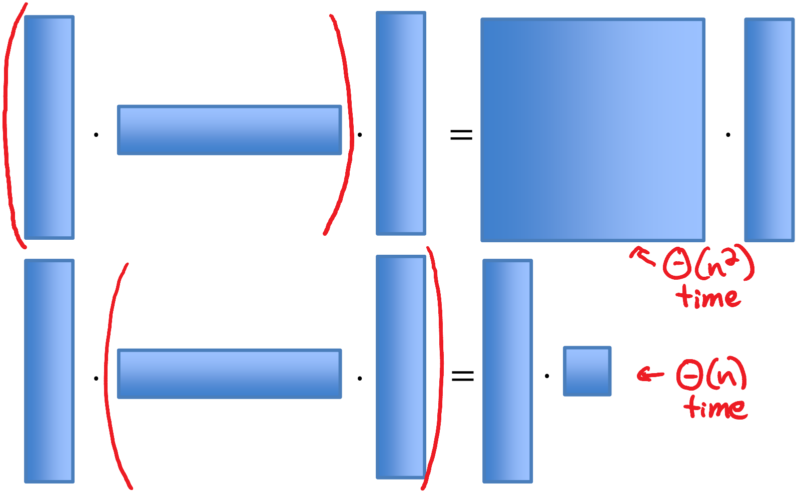
DYNAMIC PROGRAMMING

PARENTHESIZATION PROBLEM

PARENTHESIZATION PROBLEM

- given sequence of matrices $\langle A_1, \ldots, A_n \rangle$ of dimension $p_0 \times p_1$,$p_1 \times p_2, \ldots, p_{n-1} \times p_n$
- compute associative product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ using sequence of normal matrix multiplies in the order that minimizes cost
- cost to multiply $i \times j$ with $j \times k$ is ijk

Parenthesization Example



NUMBER OF PARENTHESIZATIONS

- denote the number of alternative parenthesizations if a sequence of n matrices by $P(n)$
- $$P(n) = \begin{cases} 1 & \text{pro } n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & \text{pro } n > 1 \end{cases}$$
- the solution to the recurrence is $\Omega(2^n)$
- brute force algorithm is exponential

STRUCTURE OF AN OPTIMAL PARENTHESIZATION

to compute the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ we have first for an index k compute products $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$

Q: which index k ?

A: we have to examine all possibilities

Q: how to compute products $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$?

A: in an optimal way $\Longrightarrow$ *subproblems of the original problem*

COST OF AN OPTIMAL SOLUTION

- given matrices $\langle A_1, \ldots, A_n \rangle$ of dimension $p_0 \times p_1, p_1 \times p_2, \ldots, p_{n-1} \times p_n$

- let us define a function $m : \{1, \ldots, n\} \times \{1, \ldots, n\} \to \mathbb{N}$ where $m(i, j)$ is the minimal cost to multiply $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$

- we can define $m(i, j)$ recursively as follows

$$m(i,j) \stackrel{def}{=} \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{ m(i,k) + m(k+1,j) + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

- the optimal cost to multiply the sequence $\langle A_1, \ldots, A_n \rangle$ is $m(1, n)$

COMPUTING THE COST FUNCTION RECURSIVELY

Function $\mathrm{M}(i, j)$

Input: i, j

Output: value $m(i, j)$

if $i = j$ **then return** 0 **else**

  return $\min_{i \leq k < j}\{M(i, k) + M(k+1, j) + p_{i-1}p_kp_j\}$

- let $T(n)$ denote the time complexity of the computation of $m(i, j)$ for $n = j - i + 1$
- for $n > 0$ and a constant d

$$T(n) = \sum_{k=1}^{n-1}(T(k) + T(n-k)) + dn = 2\sum_{k=1}^{n-1} T(k) + dn$$

- $T(n) = \Theta(3^n)$

COMPUTING THE COST FUNCTION BOTTOM UP

- make use of dependencies
- the order is given by the number of matrices

- $m(1,1), m(2,2), \ldots, m(n,n)$
  $m(1,2), m(2,3) \ldots, m(n-1,n)$
  $m(1,3), m(2,4) \ldots, m(n-2,n)$
  $\ldots\ldots$
  $m(1,n-1), m(2,n)$
  $m(1,n)$

COMPUTING THE COST FUNCTION BOTTOM UP

Algorithm: Matrix Multiplication

Input: dimensions $p_0, p_1, p_2, \ldots, p_n$ of matrices

Output: value $m(1, n)$

for $i = 1$ **to** n **do** $(M(i,i) \leftarrow 0$
for $r = 2$ **to** n **do**
    for $i = 1$ **to** $n - r + 1$ **do**
        $j \leftarrow i + r - 1$
        $M(i,j) \leftarrow \infty$
        for $k = i$ **to** $j - 1$ **do**
            $q \leftarrow M(i,k) + M(k+1,j) + p_{i-1}p_kp_j$
            if $q < M(i,j)$ **then** $M(i,j) \leftarrow q$
return $M(1, n)$

complexity $\quad T(n) = \mathcal{O}(n^3)$

COMPUTING THE OPTIMAL SOLUTION BOTTOM UP

modify line 8 to

if $q < M(i,j)$ **then** $M(i,j) \leftarrow q, S(i,j) \leftarrow k$

Function $\textsc{Parenthesis}(S, i, j)$

Input: function S, indices i, j

Output: parenthetization of the sequence $A_i, \ldots, A_j$

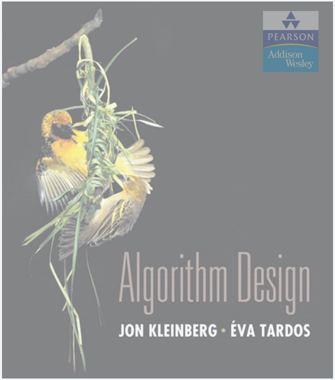
```
if i = j then print A_i   else
    print '('
    ; PARENTHESIS((S, i, S(i, j)))
    PARENTHESIS((S, S(i, j) + 1, j))
    print ')'
```

ALTERNATIVE SOLUTIONS

- $m(1,1), m(2,2), \ldots, m(n,n)$
  $m(1,2), m(2,3) \ldots, m(n-1,n)$
  $m(1,3), m(2,4) \ldots, m(n-2,n)$
  $\cdots\cdots$
  $m(1,n)$
- $m(n,n), m(n-1,n-1), \ldots, m(1,1)$
  $m(n-1,n), m(n-2,n-1) \ldots, m(1,2)$
  $m(n-2,n), m(n-3,n-1) \ldots, m(1,3)$
  $\cdots\cdots$
  $m(1,n)$
- $m(n,n)$
  $m(n-1,n-1), m(n-1,n)$
  $m(n-2,n-2), m(n-2,n-1), m(n-2,n)$
  $\cdots\cdots$
  $m(1,1), m(1,2), \ldots, m(1,n)$
- $m(1,1)$
  $m(2,2), m(1,2)$
  $m(3,3), m(2,3), m(1,3)$
  $\cdots\cdots$
  $m(n,n), m(n-1,n), \ldots, m(1,n)$

DYNAMIC PROGRAMMING

KNAPSACK PROBLEM



SECTION 6.4

6. DYNAMIC PROGRAMMING I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

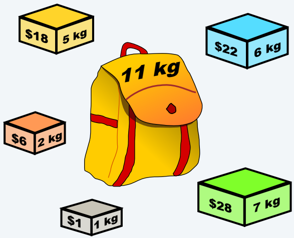
Knapsack problem

Goal. Pack knapsack so as to maximize total value.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Knapsack has weight capacity of W.

Assumption. All input values are integral.

Ex. $\{1, 2, 5\}$ has value \$35 (and weight 10).
Ex. $\{3, 4\}$ has value \$40 (and weight 11).



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance
(weight limit W = 11)

Dynamic programming: adding a new variable

Def. $OPT(i, w)$ = max-profit subset of items $1, \ldots, i$ with weight limit w.
Goal. $OPT(n, W)$.

possibly because $w_i > w$

Case 1. $OPT(i, w)$ does not select item i.

- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i-1 \}$ using weight limit w.

optimal substructure property
(proof via exchange argument)

Case 2. $OPT(i, w)$ selects item i.

- Collect value v_i.
- New weight limit $= w - w_i$.
- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i-1 \}$ using this new weight limit.

Bellman equation.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max \{ OPT(i-1, w),\ \ v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$$

Knapsack problem: bottom-up dynamic programming

KNAPSACK($n, W, w_1, \ldots, w_n, v_1, \ldots, v_n$)

FOR $w = 0$ TO W

  $M[0, w] \leftarrow 0.$

FOR $i = 1$ TO n

  FOR $w = 0$ TO W

    IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w].$

    ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], \; v_i + M[i-1, w-w_i] \}.$

RETURN $M[n, W].$

---

previously computed values

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max \{ OPT(i-1, w), \; v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

Knapsack problem: bottom-up dynamic programming demo

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, …, i

	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = max-profit subset of items 1, …, i with weight limit w.

Knapsack problem: running time

Theorem. The DP algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(n\,W)$ time and $\Theta(n\,W)$ space.

weights are integers between 1 and W

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\,W)$ table entries.
- After computing optimal values, can trace back to find solution: $OPT(i, w)$ takes item i iff $M[i, w] > M[i-1, w]$. ▪

Dynamic programming: quiz 4

Does there exist a poly-time algorithm for the knapsack problem?

A. Yes, because the DP algorithm takes $\Theta(n\ W)$ time.

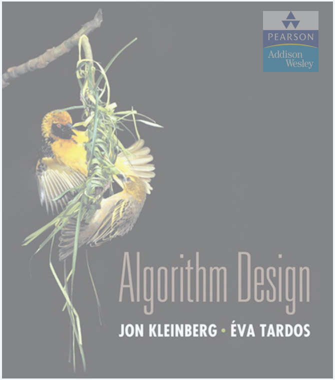
B. No, because $\Theta(n\ W)$ is not a polynomial function of the input size.

C. No, because the problem is **NP**-hard.

D. Unknown.

DYNAMIC PROGRAMMING

SEQUENCE ALIGNEMENT



SECTION 6.6

6. DYNAMIC PROGRAMMING II

- *sequence alignment*
- *Hirschberg's algorithm*
- *Bellman–Ford–Moore algorithm*
- *distance-vector protocols*
- *negative cycles*

String similarity

Q. How similar are two strings?

Ex. ocurrance and occurrence.

o	c	u	r	r	a	n	c	e	–
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	–	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	–	u	r	r	–	a	n	c	e
o	c	c	u	r	r	e	–	n	c	e

0 mismatches, 3 gaps

Edit distance

Edit distance. [Levenshtein 1966, Needleman–Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.

C	T	–	G	A	C	C	T	A	C	G
C	T	G	G	A	C	G	A	A	C	G

$$\textbf{cost} = \delta + \alpha_{CG} + \alpha_{TA}$$

assuming $\alpha_{AA} = \alpha_{CC} = \alpha_{GG} = \alpha_{TT} = 0$

Applications. Bioinformatics, spell correction, machine translation, speech recognition, information extraction, ...

```
Spokesperson confirms     senior government adviser was found
Spokesperson said     the senior            adviser was found
```

BLOSUM matrix for proteins

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	7	-3	-3	-3	-1	-2	-2	0	-3	-3	-3	-1	-2	-4	-1	2	0	-5	-4	-1
R	-3	9	-1	-3	-6	1	-1	-4	0	-5	-4	3	-3	-5	-3	-2	-2	-5	-4	-4
N	-3	-1	9	2	-5	0	-1	-1	1	-6	-6	0	-4	-6	-4	1	0	-7	-4	-5
D	-3	-3	2	10	-7	-1	2	-3	-2	-7	-7	-2	-6	-6	-3	-1	-2	-8	-6	-6
C	-1	-6	-5	-7	13	-5	-7	-6	-7	-2	-3	-6	-3	-4	-6	-2	-2	-5	-5	-2
Q	-2	1	0	-1	-5	9	3	-4	1	-5	-4	2	-1	-5	-3	-1	-1	-4	-3	-4
E	-2	-1	-1	2	-7	3	8	-4	0	-6	-6	1	-4	-6	-2	-1	-2	-6	-5	-4
G	0	-4	-1	-3	-6	-4	-4	9	-4	-7	-7	-3	-5	-6	-5	-1	-3	-6	-6	-6
H	-3	0	1	-2	-7	1	0	-4	12	-6	-5	-1	-4	-2	-4	-2	-3	-4	3	-5
I	-3	-5	-6	-7	-2	-5	-6	-7	-6	7	2	-5	2	-1	-5	-4	-2	-5	-3	4
L	-3	-4	-6	-7	-3	-4	-6	-7	-5	2	6	-4	3	0	-5	-4	-3	-4	-2	1
K	-1	3	0	-2	-6	2	1	-3	-1	-5	-4	8	-3	-5	-2	-1	-1	-6	-4	-4
M	-2	-3	-4	-6	-3	-1	-4	-5	-4	2	3	-3	9	0	-4	-3	-1	-3	-3	1
F	-4	-5	-6	-6	-4	-5	-6	-6	-2	-1	0	-5	0	10	-6	-4	-4	0	4	-2
P	-1	-3	-4	-3	-6	-3	-2	-5	-4	-5	-5	-2	-4	-6	12	-2	-3	-7	-6	-4
S	2	-2	1	-1	-2	-1	-1	-1	-2	-4	-4	-1	-3	-4	-2	7	2	-6	-3	-3
T	0	-2	0	-2	-2	-1	-2	-3	-3	-2	-3	-1	-1	-4	-3	2	8	-5	-3	0
W	-5	-5	-7	-8	-5	-4	-6	-6	-4	-5	-4	-6	-3	0	-7	-6	-5	16	3	-5
Y	-4	-4	-4	-6	-5	-3	-5	-6	3	-3	-2	-4	-3	4	-6	-3	-3	3	11	-3
V	-1	-4	-5	-6	-2	-4	-4	-6	-5	4	1	-4	1	-2	-4	-3	0	-5	-3	7

Dynamic programming: quiz 1

What is edit distance between these two strings?

P A L E T T E   P A L A T E

Assume gap penalty = 2 and mismatch penalty = 1.

- **A.** 1
- **B.** 2
- **C.** 3
- **D.** 4
- **E.** 5

Sequence alignment

Goal. Given two strings $x_1\, x_2 \ldots x_m$ and $y_1\, y_2 \ldots y_n$, find a min-cost alignment.

Def. An alignment M is a set of ordered pairs $x_i - y_j$ such that each character appears in at most one pair and no crossings.

$x_i - y_j$ and $x_{i'} - y_{j'}$ cross if $i < i'$, but $j > j'$

Def. The cost of an alignment M is:

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i\,:\,x_i \text{ unmatched}} \delta + \sum_{j\,:\,y_j \text{ unmatched}} \delta}_{\text{gap}}$$

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	–	G
–	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

an alignment of CTACCG and TACATG

$$M = \{\, x_2\text{–}y_1,\, x_3\text{–}y_2,\, x_4\text{–}y_3,\, x_5\text{–}y_4,\, x_6\text{–}y_6 \,\}$$

Sequence alignment: problem structure

Def. $OPT(i, j)$ = min cost of aligning prefix strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$.
Goal. $OPT(m, n)$.

Case 1. $OPT(i, j)$ matches $x_i - y_j$.
Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1\, x_2 \ldots x_{i-1}$ and $y_1\, y_2 \ldots y_{j-1}$.

Case 2a. $OPT(i, j)$ leaves x_i unmatched.
Pay gap for x_i + min cost of aligning $x_1\, x_2 \ldots x_{i-1}$ and $y_1\, y_2 \ldots y_j$.

optimal substructure property
(proof via exchange argument)

Case 2b. $OPT(i, j)$ leaves y_j unmatched.
Pay gap for y_j + min cost of aligning $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_{j-1}$.

Bellman equation.

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ i\delta & \text{if } j = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \end{cases}$$

Sequence alignment: analysis

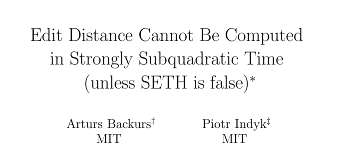
Theorem. The DP algorithm computes the edit distance (and an optimal alignment) of two strings of lengths m and n in $\Theta(mn)$ time and space.

Pf.

- Algorithm computes edit distance.
- Can trace back to extract optimal alignment itself. ▪

Theorem. [Backurs–Indyk 2015] If can compute edit distance of two strings of length n in $O(n^{2-\varepsilon})$ time for some constant $\varepsilon > 0$, then can solve SAT with n variables and m clauses in $poly(m)\ 2^{(1-\delta)\,n}$ time for some constant $\delta > 0$.

which would disprove SETH
(strong exponential time hypothesis)



Edit Distance Cannot Be Computed
in Strongly Subquadratic Time
(unless SETH is false)*

Arturs Backurs† MIT

Piotr Indyk‡ MIT

Sequence alignment: traceback

		S	I	M	I	L	A	R	I	T	Y
	0	2	4	6	8	10	12	14	16	18	20
I	2	4	1	3	2	4	6	8	7	9	11
D	4	6	3	3	4	4	6	8	9	9	11
E	6	8	5	5	6	6	6	8	10	11	11
N	8	10	7	7	8	8	8	8	10	12	13
T	10	12	9	9	9	10	10	10	10	9	11
I	12	14	8	10	8	10	12	12	9	11	11
T	14	16	10	10	10	10	12	14	11	8	11
Y	16	18	12	12	12	12	12	14	13	10	7

Sequence alignment: analysis

Theorem. The DP algorithm computes the edit distance (and an optimal alignment) of two strings of lengths m and n in $\Theta(mn)$ time and space.

Pf.

- Algorithm computes edit distance.
- Can trace back to extract optimal alignment itself. ▪

Theorem. [Backurs–Indyk 2015] If can compute edit distance of two strings of length n in $O(n^{2-\varepsilon})$ time for some constant $\varepsilon > 0$, then can solve SAT with n variables and m clauses in $poly(m)\ 2^{(1-\delta)\,n}$ time for some constant $\delta > 0$.

which would disprove SETH
(strong exponential time hypothesis)

Edit Distance Cannot Be Computed
in Strongly Subquadratic Time
(unless SETH is false)*

Arturs Backurs† MIT
Piotr Indyk‡ MIT

It is easy to modify the DP algorithm for edit distance to…

A. Compute edit distance in $O(mn)$ time and $O(m + n)$ space.

B. Compute an optimal alignment in $O(mn)$ time and $O(m + n)$ space.

C. Both A and B.

D. Neither A nor B.

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ i\delta & \text{if } j = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \end{cases}$$

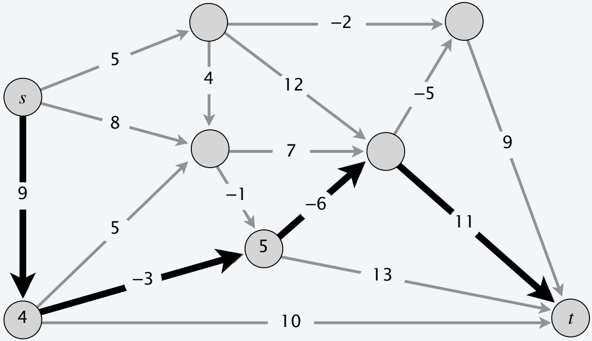
DYNAMIC PROGRAMMING

SHORTEST PATHS — BELLMAN-FORD ALGORITHM

Shortest paths with negative weights

Shortest-path problem. Given a digraph $G = (V, E)$, with arbitrary edge lengths ℓ_{vw}, find shortest path from source node s to destination node t.

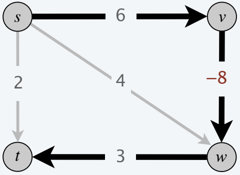
assume there exists a path from every node to t



length of shortest path from s to t = 9 − 3 − 6 + 11 = 11

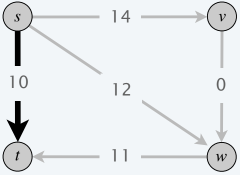
Shortest paths with negative weights: failed attempts

Dijkstra. May not produce shortest paths when edge lengths are negative.



Dijkstra selects the vertices in the order s, t, w, v
But shortest path from s to t is $s \to v \to w \to t$.

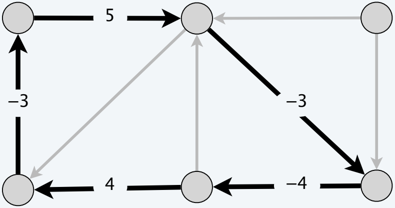
Reweighting. Adding a constant to every edge length does not necessarily make Dijkstra's algorithm produce shortest paths.



Adding 8 to each edge weight changes the shortest path from $s \to v \to w \to t$ to $s \to t$.

Negative cycles

Def. A negative cycle is a directed cycle for which the sum of its edge lengths is negative.

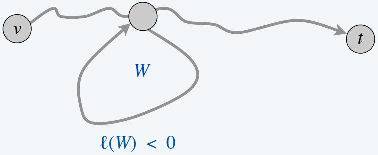


a negative cycle W : $\ell(W) \ = \ \sum_{e \in W} \ell_e \ < \ 0$

Shortest paths and negative cycles

Lemma 1. If some $v \rightsquigarrow t$ path contains a negative cycle, then there does not exist a shortest $v \rightsquigarrow t$ path.

Pf. If there exists such a cycle W, then can build a $v \rightsquigarrow t$ path of arbitrarily negative length by detouring around W as many times as desired. ▪

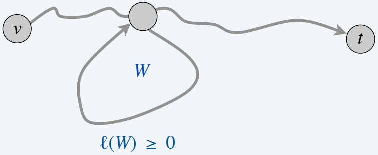


Shortest paths and negative cycles

Lemma 2. If G has no negative cycles, then there exists a shortest $v{\leadsto}t$ path that is simple (and has $\le\ n - 1$ edges).

Pf.

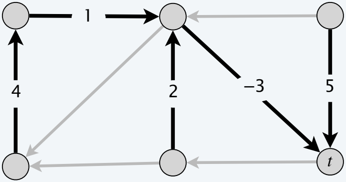
- Among all shortest $v{\leadsto}t$ paths, consider one that uses the fewest edges.
- If that path P contains a directed cycle W, can remove the portion of P corresponding to W without increasing its length. ▪



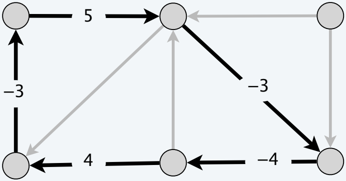
Shortest-paths and negative-cycle problems

Single-destination shortest-paths problem. Given a digraph $G = (V, E)$ with edge lengths ℓ_{vw} (but no negative cycles) and a distinguished node t, find a shortest $v \rightarrow t$ path for every node v.

Negative-cycle problem. Given a digraph $G = (V, E)$ with edge lengths ℓ_{vw}, find a negative cycle (if one exists).



shortest-paths tree



negative cycle

Dynamic programming: quiz 5

Which subproblems to find shortest $v \rightsquigarrow t$ paths for every node v?

A. $OPT(i, v)$ = length of shortest $v \rightsquigarrow t$ path that uses exactly i edges.

B. $OPT(i, v)$ = length of shortest $v \rightsquigarrow t$ path that uses at most edges.

C. Neither A nor B.

Shortest paths with negative weights: dynamic programming

Def. $OPT(i, v)$ = length of shortest $v \to t$ path that uses $\leq i$ edges.

Goal. $OPT(n-1, v)$ for each v.

by Lemma 2, if no negative cycles,
there exists a shortest $v \to t$ path that is simple

Case 1. Shortest $v \to t$ path uses $\leq i-1$ edges.

- $OPT(i, v) = \text{OPT}(i-1, v)$.

optimal substructure property
(proof via exchange argument)

Case 2. Shortest $v \to t$ path uses exactly i edges.

- if (v, w) is first edge in shortest such $v \to t$ path, incur a cost of ℓ_{vw}.
- Then, select best $w \to t$ path using $\leq i-1$ edges.

Bellman equation.

$$OPT(i,v) \;=\; \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ \infty & \text{if } i = 0 \text{ and } v \neq t \\ \min \left\{ OPT(i-1,v), \; \min\limits_{(v,w) \in E} \{ OPT(i-1,w) + \ell_{vw} \} \right\} & \text{if } i > 0 \end{cases}$$

Shortest paths with negative weights: implementation

SHORTEST-PATHS(V, E, ℓ, t)

FOREACH node $v \in V$:

  $M[0, v] \leftarrow \infty$.

$M[0, t] \leftarrow 0$.

FOR i = 1 TO $n - 1$

  FOREACH node $v \in V$:

    $M[i, v] \leftarrow M[i-1, v]$.

    FOREACH edge $(v, w) \in E$:

      $M[i, v] \leftarrow \min \{ M[i, v],\ M[i-1, w] + \ell_{vw} \}$.

Shortest paths with negative weights: implementation

Theorem 1. Given a digraph $G = (V, E)$ with no negative cycles, the DP algorithm computes the length of a shortest $v \to t$ path for every node v in $\Theta(mn)$ time and $\Theta(n^2)$ space.

Pf.

- Table requires $\Theta(n^2)$ space.
- Each iteration i takes $\Theta(m)$ time since we examine each edge once. ▪

Finding the shortest paths.

- Approach 1: Maintain $successor[i, v]$ that points to next node on a shortest $v \to t$ path using $\leq i$ edges.
- Approach 2: Compute optimal lengths $M[i, v]$ and consider only edges with $M[i, v] = M[i-1, w] + \ell_{vw}$. Any directed path in this subgraph is a shortest path.

Dynamic programming: quiz 6

It is easy to modify the DP algorithm for shortest paths to…

A. Compute lengths of shortest paths in $O(mn)$ time and $O(m + n)$ space.

B. Compute shortest paths in $O(mn)$ time and $O(m + n)$ space.

C. Both A and B.

D. Neither A nor B.

Shortest paths with negative weights: practical improvements

Space optimization. Maintain two 1D arrays (instead of 2D array).

- $d[v]$ = length of a shortest $v \rightarrow t$ path that we have found so far.
- $successor[v]$ = next node on a $v \rightarrow t$ path.

Performance optimization. If $d[w]$ was not updated in iteration $i - 1$, then no reason to consider edges entering w in iteration i.

Bellman–Ford–Moore: efficient implementation

BELLMAN–FORD–MOORE(V, E, c, t)

FOREACH node $v \in V$:

  $d[v] \leftarrow \infty$.

  $successor[v] \leftarrow null$.

$d[t] \leftarrow 0$.

FOR $i = 1$ TO $n - 1$

  FOREACH node $w \in V$:

    IF ($d[w]$ was updated in previous pass)

      FOREACH edge $(v, w) \in E$:

        IF ($d[v] > d[w] + \ell_{vw}$)

          $d[v] \leftarrow d[w] + \ell_{vw}$.

          $successor[v] \leftarrow w$.

  IF (no $d[\cdot]$ value changed in pass i) STOP.

pass i
$O(m)$ time

Dynamic programming: quiz 7

Which properties must hold after pass i of Bellman–Ford–Moore?

A. $d[v]$ = length of a shortest $v \rightsquigarrow t$ path using $\leq i$ edges.

B. $d[v]$ = length of a shortest $v \rightsquigarrow t$ path using exactly i edges.

C. Both A and B.

D. Neither A nor B.

Bellman–Ford–Moore: analysis

Lemma 3. For each node v : $d[v]$ is the length of some $v \rightsquigarrow t$ path.

Lemma 4. For each node v : $d[v]$ is monotone non-increasing.

Lemma 5. After pass i, $d[v] \le$ length of a shortest $v \rightsquigarrow t$ path using $\le i$ edges.

Pf. [by induction on i]

- Base case: $i = 0$.
- Assume true after pass i.
- Let P be any $v \rightsquigarrow t$ path with $\le i + 1$ edges.
- Let (v, w) be first edge in P and let P' be subpath from w to t.
- By inductive hypothesis, at the end of pass i, $d[w] \le c(P')$ because P' is a $w \rightsquigarrow t$ path with $\le i$ edges.
- After considering edge (v, w) in pass $i + 1$:

and by Lemma 4, $d[w]$ does not increase

$$
\begin{aligned}
d[v] &\le \ell_{vw} + d[w] \\
&\le \ell_{vw} + c(P') \\
&= \ell(P) \quad \blacksquare
\end{aligned}
$$

and by Lemma 4, $d[v]$ does not increase

Bellman–Ford–Moore: analysis

Theorem 2. Assuming no negative cycles, Bellman–Ford–Moore computes the lengths of the shortest $v \rightarrow t$ paths in $O(mn)$ time and $\Theta(n)$ extra space.

Pf. Lemma 2 + Lemma 5. ▪

shortest path exists and has at most $n-1$ edges

after i passes, $d[v] \leq$ length of shortest path that uses $\leq i$ edges

Remark. Bellman–Ford–Moore is typically faster in practice.

- Edge (v, w) considered in pass $i + 1$ only if $d[w]$ updated in pass i.
- If shortest path has k edges, then algorithm finds it after $\leq k$ passes.

Assuming no negative cycles, which properties must hold throughout Bellman–Ford–Moore?

A. Following $successor[v]$ pointers gives a directed $v \rightarrow t$ path.

B. If following $successor[v]$ pointers gives a directed $v \rightarrow t$ path, then the length of that $v \rightarrow t$ path is $d[v]$.

C. Both A and B.

D. Neither A nor B.

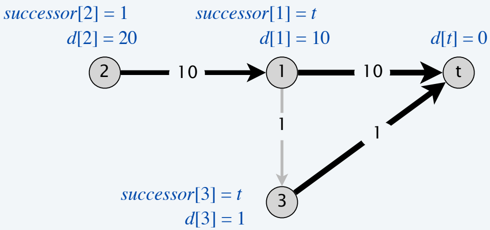
Bellman–Ford–Moore: analysis

Claim. ~~Throughout Bellman–Ford–Moore, following the $successor[v]$ pointers gives a directed path from v to t of length $d[v]$.~~

Counterexample. Claim is false!

- Length of successor $v \rightarrow t$ path may be strictly shorter than $d[v]$.

consider nodes in order: t, 1, 2, 3



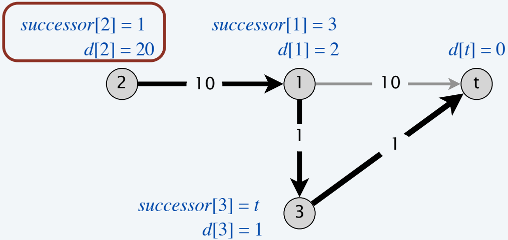
Bellman–Ford–Moore: analysis

Claim. ~~Throughout Bellman–Ford–Moore, following the *successor*[*v*] pointers gives a directed path from *v* to *t* of length *d*[*v*].~~

Counterexample. Claim is false!

- Length of successor $v \rightarrow t$ path may be strictly shorter than $d[v]$.

consider nodes in order: t, 1, 2, 3



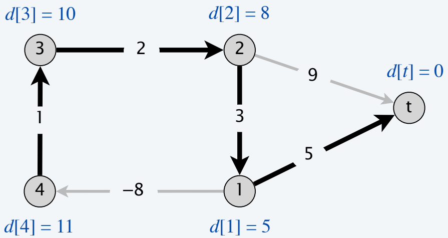
Bellman–Ford–Moore: analysis

Claim. ~~Throughout Bellman–Ford–Moore, following the $successor[v]$ pointers gives a directed path from v to t of length $d[v]$.~~

Counterexample. Claim is false!

- Length of successor $v \rightsquigarrow t$ path may be strictly shorter than $d[v]$.
- If negative cycle, successor graph may have directed cycles.

consider nodes in order: t, 1, 2, 3, 4



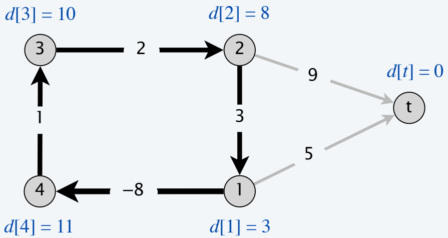
Bellman–Ford–Moore: analysis

Claim. ~~Throughout Bellman–Ford–Moore, following the *successor*[v] pointers gives a directed path from v to t of length $d[v]$.~~

Counterexample. Claim is false!

- Length of successor $v \rightsquigarrow t$ path may be strictly shorter than $d[v]$.
- If negative cycle, successor graph may have directed cycles.

consider nodes in order: t, 1, 2, 3, 4



Bellman–Ford–Moore: finding the shortest paths

Lemma 6. Any directed cycle W in the successor graph is a negative cycle.

Pf.

- If $successor[v] = w$, we must have $d[v] \geq d[w] + \ell_{vw}$.
  (LHS and RHS are equal when $successor[v]$ is set; $d[w]$ can only decrease; $d[v]$ decreases only when $successor[v]$ is reset)
- Let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$ be the sequence of nodes in a directed cycle W.
- Assume that (v_k, v_1) is the last edge in W added to the successor graph.
- Just prior to that:

$$\begin{array}{llll} d[v_1] & \geq & d[v_2] & + \ \ell(v_1, v_2) \\ d[v_2] & \geq & d[v_3] & + \ \ell(v_2, v_3) \\ \vdots & \vdots & & \vdots \\ d[v_{k-1}] & \geq & d[v_k] & + \ \ell(v_{k-1}, v_k) \\ d[v_k] & > & d[v_1] & + \ \ell(v_k, v_1) \end{array}$$

holds with strict inequality since we are updating $d[v_k]$

- Adding inequalities yields $\underbrace{\ell(v_1, v_2) + \ell(v_2, v_3) + \ldots + \ell(v_{k-1}, v_k) + \ell(v_k, v_1) < 0}_{W \text{ is a negative cycle}}$. ▪

Bellman–Ford–Moore: finding the shortest paths

Theorem 3. Assuming no negative cycles, Bellman–Ford–Moore finds shortest $v \to t$ paths for every node v in $O(mn)$ time and $\Theta(n)$ extra space.

Pf.

- The successor graph cannot have a directed cycle. [Lemma 6]
- Thus, following the successor pointers from v yields a directed path to t.
- Let $v = v_1 \to v_2 \to \ldots \to v_k = t$ be the nodes along this path P.
- Upon termination, if $successor[v] = w$, we must have $d[v] = d[w] + \ell_{vw}$.
  (LHS and RHS are equal when $successor[v]$ is set; $d[\cdot]$ did not change)

since algorithm terminated

- Thus,

$$\begin{aligned} d[v_1] &= d[v_2] + \ell(v_1, v_2) \\ d[v_2] &= d[v_3] + \ell(v_2, v_3) \\ &\vdots \\ d[v_{k-1}] &= d[v_k] + \ell(v_{k-1}, v_k) \end{aligned}$$

- Adding equations yields $d[v] = d[t] + \underbrace{\ell(v_1, v_2) + \ell(v_2, v_3) + \ldots + \ell(v_{k-1}, v_k)}_{\text{length of path } P}$. ▪

min length of any $v \to t$ path (Theorem 2)

0

Single-source shortest paths with negative weights

year	worst case	discovered by
1955	$O(n^4)$	Shimbel
1956	$O(m\ n^2\ W)$	Ford
1958	$O(m\ n)$	Bellman, Moore
1983	$O(n^{3/4}\ m \log W)$	Gabow
1989	$O(m\ n^{1/2} \log(nW))$	Gabow–Tarjan
1993	$O(m\ n^{1/2} \log W)$	Goldberg
2005	$O(n^{2.38}\ W)$	Sankowsi, Yuster–Zwick
2016	$\tilde{O}(n^{10/7} \log W)$	Cohen–Mądry–Sankowski–Vladu
20xx	???	

single-source shortest paths with weights between –W and W

DYNAMIC PROGRAMMING SUMMARY

conditions

- number of diffferent subproblems is polynomial
- problem solution can be easily deduced from solutions of subproblems
- subproblems can be naturally ordered from *smallest* to *largest*

memoization

- simple to understand
- no need to dictate the ordering of subproblems

dynamic programming

- no recursion overhead
- lower space complexity
- simple complexity analysis

GREEDY ALGORITHMS

Algorithmic paradigms

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

Divide-and-conquer. Break up a problem into independent subproblems; solve each subproblem; combine solutions to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form solution to large subproblem.

fancy name for
caching intermediate results
in a table for later reuse

GREEDY ALGORITHMS

COIN CHANGING

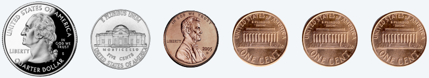
4. Greedy Algorithms I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Coin changing

Goal. Given U. S. currency denominations { 1, 5, 10, 25, 100 }, devise a method to pay amount to customer using fewest coins.

Ex. 34¢.



Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex. $2.89.

Cashier's algorithm

At each iteration, add coin of the largest value that does not take us past the amount to be paid.

CASHIERS-ALGORITHM $(x, c_1, c_2, \ldots, c_n)$

SORT n coin denominations so that $0 < c_1 < c_2 < \ldots < c_n$.

$S \leftarrow \varnothing$. ⟵ multiset of coins selected

WHILE $(x > 0)$

  $k \leftarrow$ largest coin denomination c_k such that $c_k \leq x$.

  IF no such k, RETURN *"no solution."*

  ELSE

    $x \leftarrow x - c_k$.

    $S \leftarrow S \cup \{ k \}$.

RETURN S.

Is the cashier's algorithm optimal?

A. Yes, greedy algorithms are always optimal.

B. Yes, for any set of coin denominations $c_1 < c_2 < \ldots < c_n$ provided $c_1 = 1$.

C. Yes, because of special properties of U.S. coin denominations.

D. No.

Cashier's algorithm (for arbitrary coin denominations)

Q. Is cashier's algorithm optimal for any set of denominations?

A. No. Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

- Cashier's algorithm: 140¢ = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1.
- Optimal: 140¢ = 70 + 70.

A. No. It may not even lead to a feasible solution if $c_1 > 1$: 7, 8, 9.

- Cashier's algorithm: 15¢ = 9 + ?.
- Optimal: 15¢ = 7 + 8.

Properties of any optimal solution (for U.S. coin denominations)

Property. Number of pennies ≤ 4.

Pf. Replace 5 pennies with 1 nickel.

Property. Number of nickels ≤ 1.

Property. Number of quarters ≤ 3.

Property. Number of nickels + number of dimes ≤ 2.

Pf.

- Recall: ≤ 1 nickel.
- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.

dollars (100¢) **quarters (25¢)** **dimes (10¢)** **nickels (5¢)** **pennies (1¢)**

Optimality of cashier's algorithm (for U.S. coin denominations)

Theorem. Cashier's algorithm is optimal for U.S. coins { 1, 5, 10, 25, 100 }.

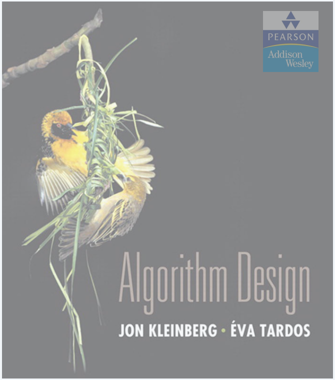
Pf. [by induction on amount to be paid x]

- Consider optimal way to change $c_k \le x < c_{k+1}$: greedy takes coin k.
- We claim that any optimal solution must take coin k.
  - if not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x
  - table below indicates no optimal solution can do this
- Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by cashier's algorithm. ▪

k	c_k	all optimal solutions must satisfy	max value of coin denominations $c_1, c_2, \ldots, c_{k-1}$ in any optimal solution
1	1	$P \le 4$	–
2	5	$N \le 1$	4
3	10	$N + D \le 2$	4 + 5 = 9
4	25	$Q \le 3$	20 + 4 = 24
5	100	*no limit*	75 + 24 = 99

GREEDY ALGORITHMS

INTERVAL SCHEDULING



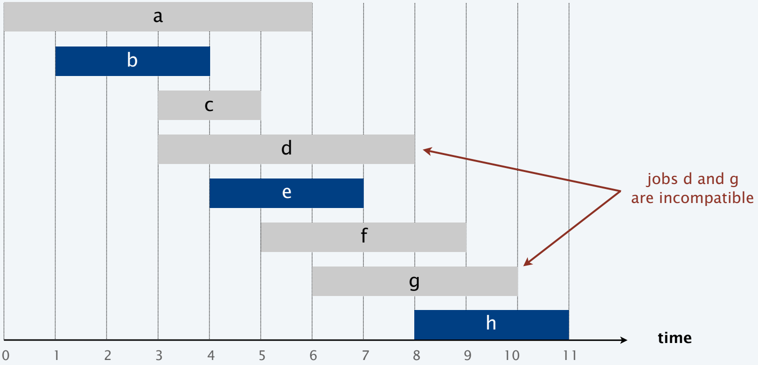
SECTION 4.1

4. GREEDY ALGORITHMS I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Interval scheduling

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Consider jobs in some order, taking each job provided it's compatible with the ones already taken. Which rule is optimal?

A. [Earliest start time] Consider jobs in ascending order of s_j.

B. [Earliest finish time] Consider jobs in ascending order of f_j.

C. [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.

D. None of the above.

Interval scheduling: earliest-finish-time-first algorithm

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

  IF job j is compatible with S

    $S \leftarrow S \cup \{ j \}$.

RETURN S.

Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

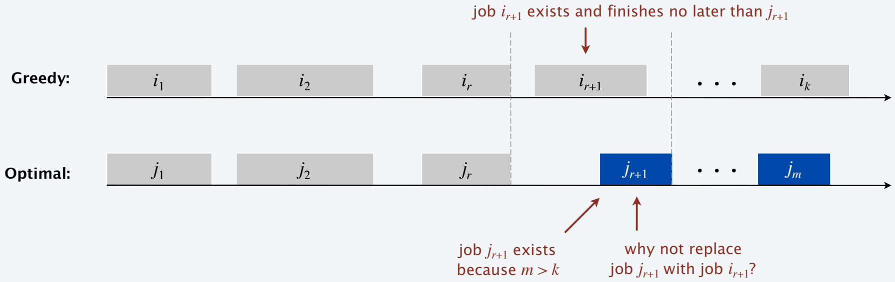
- Keep track of job j^* that was added last to S.
- Job j is compatible with S iff $s_j \ge f_{j^*}$.
- Sorting by finish times takes $O(n \log n)$ time.

Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let i_1, i_2, ... i_k denote set of jobs selected by greedy.
- Let j_1, j_2, ... j_m denote set of jobs in an optimal solution with $i_1 = j_1$, $i_2 = j_2$, ..., $i_r = j_r$ for the largest possible value of r.

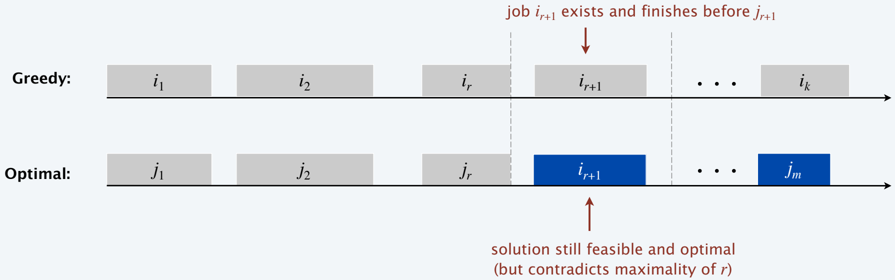


Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

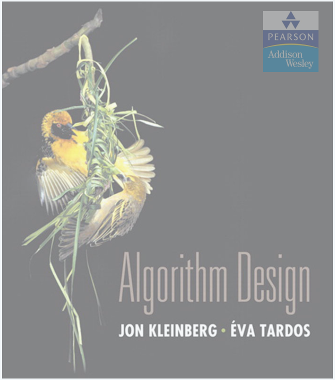


Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals. Is the earliest-finish-time-first algorithm still optimal?

- **A.** Yes, because greedy algorithms are always optimal.
- **B.** Yes, because the same proof of correctness is valid.
- **C.** No, because the same proof of correctness is no longer valid.
- **D.** No, because you could assign a huge weight to a job that overlaps the job with the earliest finish time.

GREEDY ALGORITHMS

INTERVAL PARTITIONING



SECTION 4.1

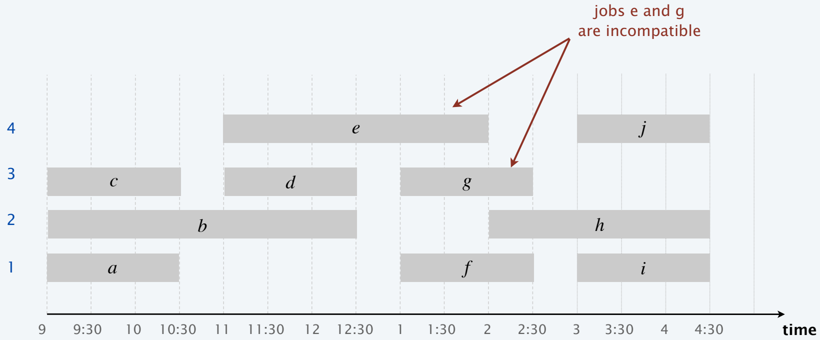
4. GREEDY ALGORITHMS I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Interval partitioning

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

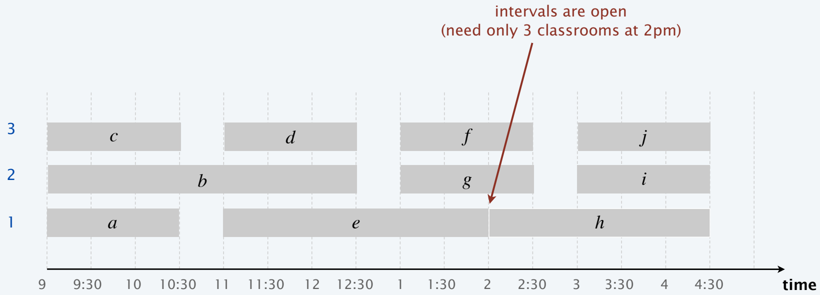
Ex. This schedule uses 4 classrooms to schedule 10 lectures.



Interval partitioning

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

Ex. This schedule uses 3 classrooms to schedule 10 lectures.



Consider lectures in some order, assigning each lecture to first available classroom (opening a new classroom if none is available). Which rule is optimal?

A. [Earliest start time] Consider lectures in ascending order of s_j.

B. [Earliest finish time] Consider lectures in ascending order of f_j.

C. [Shortest interval] Consider lectures in ascending order of $f_j - s_j$.

D. None of the above.

Interval partitioning: earliest-start-time-first algorithm

EARLIEST-START-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT lectures by start times and renumber so that $s_1 \leq s_2 \leq \ldots \leq s_n$.

$d \leftarrow 0.$ ⟵ number of allocated classrooms

FOR $j = 1$ TO n

  IF lecture j is compatible with some classroom

    Schedule lecture j in any such classroom k.

  ELSE

    Allocate a new classroom $d + 1$.

    Schedule lecture j in classroom $d + 1$.

    $d \leftarrow d + 1.$

RETURN schedule.

Interval partitioning: earliest-start-time-first algorithm

Proposition. The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.

Pf. Store classrooms in a priority queue (key = finish time of its last lecture).

- To determine whether lecture j is compatible with some classroom, compare s_j to key of min classroom k in priority queue.
- To add lecture j to classroom k, increase key of classroom k to f_j.
- Total number of priority queue operations is $O(n)$.
- Sorting by start times takes $O(n \log n)$ time. ▪

Remark. This implementation chooses a classroom k whose finish time of its last lecture is the earliest.

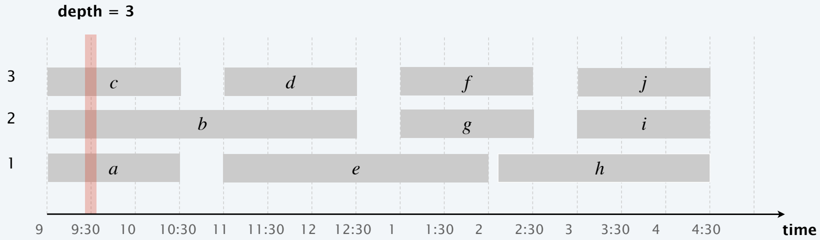
Interval partitioning: lower bound on optimal solution

Def. The depth of a set of open intervals is the maximum number of intervals that contain any given point.

Key observation. Number of classrooms needed $\geq$ depth.

Q. Does minimum number of classrooms needed always equal depth?

A. Yes! Moreover, earliest-start-time-first algorithm finds a schedule whose number of classrooms equals the depth.



Interval partitioning: analysis of earliest-start-time-first algorithm

Observation. The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

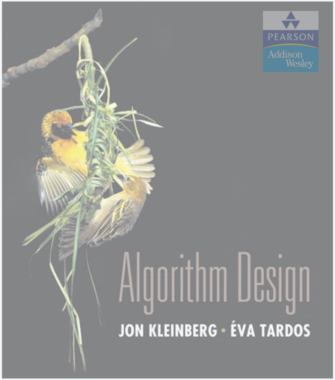
Theorem. Earliest-start-time-first algorithm is optimal.

Pf.

- Let d = number of classrooms that the algorithm allocates.
- Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with a lecture in each of $d - 1$ other classrooms.
- Thus, these d lectures each end after s_j.
- Since we sorted by start time, each of these incompatible lectures start no later than s_j.
- Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq d$ classrooms. ▪

GREEDY ALGORITHMS

SCHEDULING TO MINIMIZE LATENESS



SECTION 4.2

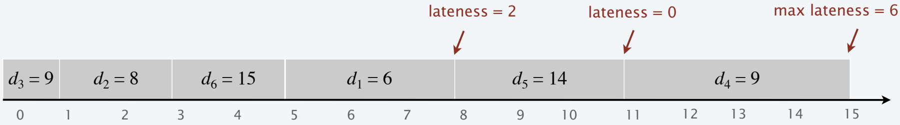
4. GREEDY ALGORITHMS I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Scheduling to minimizing lateness

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max_j \ell_j$.

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Greedy algorithms I: quiz 5

Schedule jobs according to some natural order. Which order minimizes the maximum lateness?

A. [shortest processing time] Ascending order of processing time t_j.

B. [earliest deadline first] Ascending order of deadline d_j.

C. [smallest slack] Ascending order of slack: $d_j - t_j$.

D. None of the above.

Minimizing lateness: earliest deadline first

EARLIEST-DEADLINE-FIRST $(n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n)$

SORT jobs by due times and renumber so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

$t \leftarrow 0$.

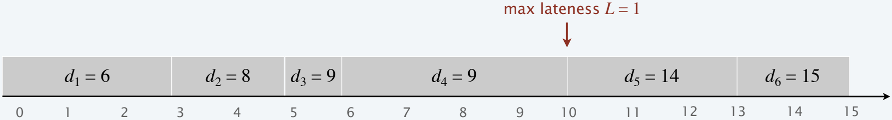
FOR $j = 1$ TO n

  Assign job j to interval $[t, t + t_j]$.

  $s_j \leftarrow t$; $f_j \leftarrow t + t_j$.

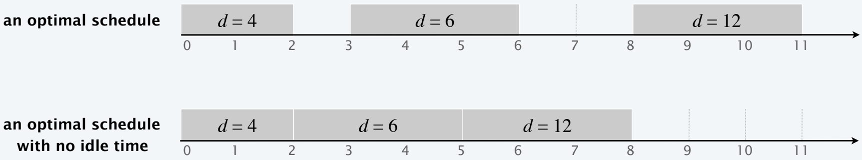
  $t \leftarrow t + t_j$.

RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.



Minimizing lateness: no idle time

Observation 1. There exists an optimal schedule with no idle time.

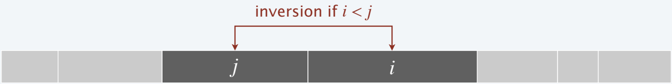


Observation 2. The earliest-deadline-first schedule has no idle time.

Minimizing lateness: inversions

Def. Given a schedule S, an inversion is a pair of jobs i and j such that: $i < j$ but j is scheduled before i.

a schedule with an inversion



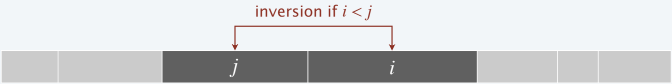
recall: we assume the jobs are numbered so that $d_1 \le d_2 \le \ldots \le d_n$

Observation 3. The earliest-deadline-first schedule is the unique idle-free schedule with no inversions.

Minimizing lateness: inversions

Def. Given a schedule S, an inversion is a pair of jobs i and j such that: $i < j$ but j is scheduled before i.

a schedule with an inversion



recall: we assume the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$

Observation 4. If an idle-free schedule has an inversion, then it has an adjacent inversion.

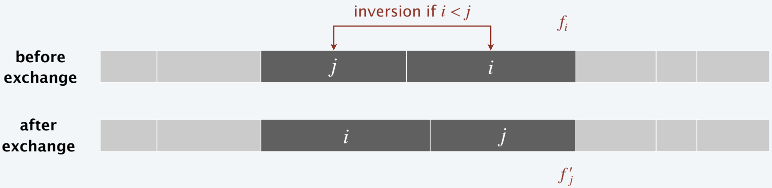
two inverted jobs scheduled consecutively

Pf.

- Let $i–j$ be a closest inversion.
- Let k be element immediately to the right of j.
- Case 1. $[\, j > k \,]$ Then $j–k$ is an adjacent inversion.
- Case 2. $[\, j < k \,]$ Then $i–k$ is a closer inversion since $i < j < k$. ※

Minimizing lateness: inversions

Def. Given a schedule S, an **inversion** is a pair of jobs i and j such that: $i < j$ but j is scheduled before i.



Key claim. Exchanging two adjacent, inverted jobs i and j reduces the number of inversions by 1 and does not increase the max lateness.

Pf. Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$.
- $\ell'_i \leq \ell_i$.
- If job j is late, ℓ'_j

$$
\begin{aligned}
\ell'_j &= f'_j - d_j && \longleftarrow \text{definition} \\
&= f_i - d_j && \longleftarrow j \text{ now finishes at time } f_i \\
&\leq f_i - d_i && \longleftarrow i < j \Rightarrow d_i \leq d_j \\
&\leq \ell_i \,. && \longleftarrow \text{definition}
\end{aligned}
$$

Minimizing lateness: analysis of earliest-deadline-first algorithm

Theorem. The earliest-deadline-first schedule S is optimal.

optimal schedule can have inversions

Pf. [by contradiction]

Define S^* to be an optimal schedule with the fewest inversions.

- Can assume S^* has no idle time. ⟵ Observation 1
- Case 1. [S^* has no inversions] Then $S = S^*$. ⟵ Observation 3
- Case 2. [S^* has an inversion]
  - let i–j be an adjacent inversion ⟵ Observation 4
  - exchanging jobs i and j decreases the number of inversions by 1 without increasing the max lateness ⟵ key claim
  - contradicts "fewest inversions" part of the definition of S^* ※

Greedy analysis strategies

Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

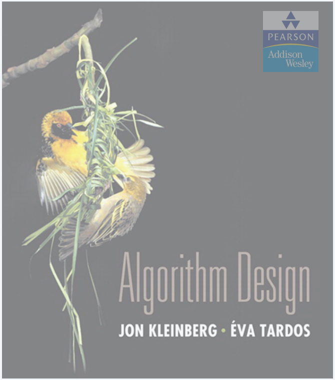
Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Other greedy algorithms. Gale–Shapley, Kruskal, Prim, Dijkstra, Huffman, …

GREEDY ALGORITHMS

OPTIMAL CACHING



SECTION 4.3

4. GREEDY ALGORITHMS I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Optimal offline caching

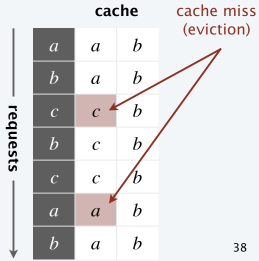
Caching.

- Cache with capacity to store k items.
- Sequence of m item requests $d_1, d_2, \ldots, d_m$.
- Cache hit: item in cache when requested.
- Cache miss: item not in cache when requested.
  (must evict some item from cache and bring requested item into cache)

Applications. CPU, RAM, hard drive, web, browser, ….

Goal. Eviction schedule that minimizes the number of evictions.

Ex. $k = 2$, initial cache $= ab$, requests: a, b, c, b, c, a, b.
Optimal eviction schedule. 2 evictions.

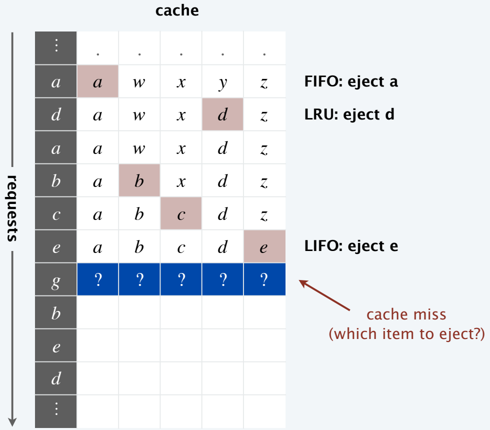


Optimal offline caching: greedy algorithms

LIFO/FIFO. Evict item brought in least (most) recently.

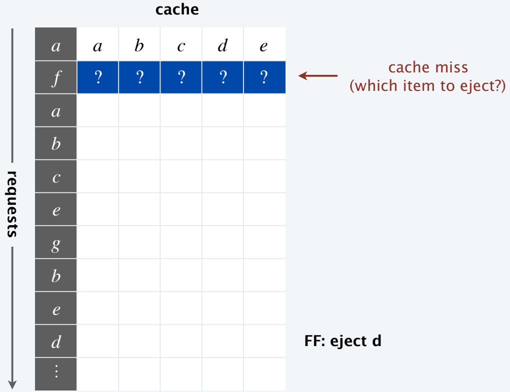
LRU. Evict item whose most recent access was earliest.

LFU. Evict item that was least frequently requested.



Optimal offline caching: farthest-in-future (clairvoyant algorithm)

Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.



Theorem. [Bélády 1966] FF is optimal eviction schedule.

Pf. Algorithm and theorem are intuitive; proof is subtle.

Greedy algorithms I: quiz 6

Which item will be evicted next using farthest-in-future schedule?

A.

B.

C.

D.

E.

cache

requests				
⋮	.	.	.	.
B	D	B	Y	A
C	D	B	C	A
E	D	E	C	A
F	?	?	?	?
C				
D				
A				
E				
A				
C				
⋮				

← cache miss (which item to eject?)

Reduced eviction schedules

Def. A reduced schedule is a schedule that brings an item d into the cache in step j only if there is a request for d in step j and d is not already in the cache.

a	a	b	c
a	a	b	c
c	a	d	c
d	a	d	c
a	a	c	b
b	a	c	b
c	a	c	b
d	d	c	b
d	d	c	d

d enters cache without a request

d enters cache even though already in cache

an unreduced schedule

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
d	d	c	b
d	d	c	b

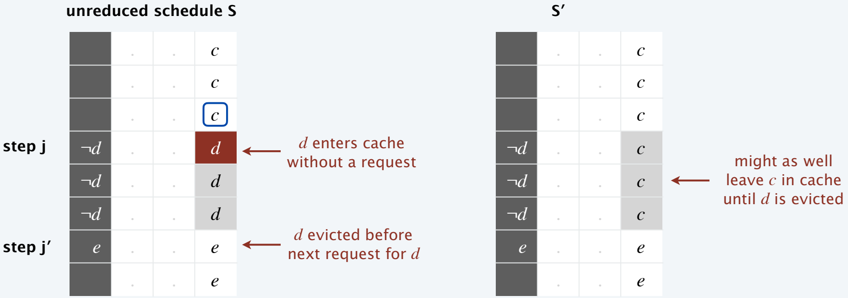
a reduced schedule

Reduced eviction schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

Pf. [by induction on number of steps j]

- Suppose S brings d into the cache in step j without a request.
- Let c be the item S evicts when it brings d into the cache.
- Case 1a: d evicted before next request for d.

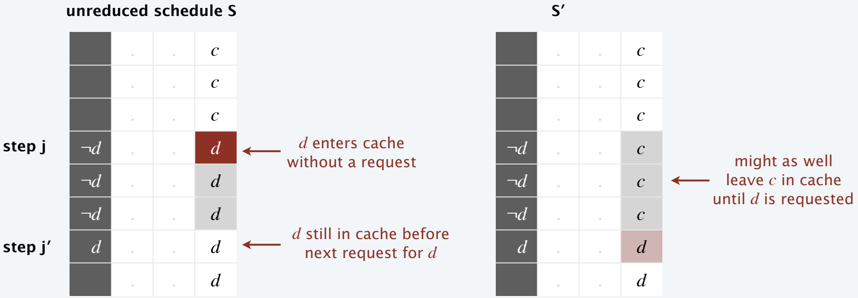


Reduced eviction schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

Pf. [by induction on number of steps j]

- Suppose S brings d into the cache in step j without a request.
- Let c be the item S evicts when it brings d into the cache.
- Case 1a: d evicted before next request for d.
- Case 1b: next request for d occurs before d is evicted.



Reduced eviction schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

Pf. [by induction on number of steps j]

- Suppose S brings d into the cache in step j even though d is in cache.
- Let c be the item S evicts when it brings d into the cache.
- Case 2a: d evicted before it is needed.

unreduced schedule S

		d_1	a	c	
		d_1	a	c	
		d_1	a	$\boxed{c}$	
step j	d	d_1	a	d_3	← d_3 enters cache even though d_1 is already in cache
	d	d_1	a	d_3	← d_3 not needed
	c	c	a	d_3	
step j'	b	c	a	b	← d_3 evicted
	d	c	a	d_3	← d_3 needed

S'

	d_1	a	c	
	d_1	a	c	
	d_1	a	c	
d	d_1	a	c	← might as well leave c in cache until d_3 in evicted
d	d_1	a	c	
c	c	a	c	
b	c	a	b	
d	c	a	d_3	

Reduced eviction schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

Pf. [by induction on number of steps j]

- Suppose S brings d into the cache in step j even though d is in cache.
- Let c be the item S evicts when it brings d into the cache.
- Case 2a: d evicted before it is needed.
- Case 2b: d needed before it is evicted.

unreduced schedule S

		d_1	a	c	
		d_1	a	c	
		d_1	a	$\boxed{c}$	
step j	d	d_1	a	d_3	← d_3 enters cache even though d_1 is already in cache
	d	d_1	a	d_3	← d_3 not needed
	c	c	a	d_3	
	a	c	a	d_3	
step j′	d	c	a	d_3	← d_3 needed

S′

	d_1	a	c	
	d_1	a	c	
	d_1	a	c	
d	d_1	a	c	← might as well leave c in cache until d_3 in needed
d	d_1	a	c	
c	c	a	c	
a	c	a	c	
d	c	a	d_3	

Reduced eviction schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

Pf. [by induction on number of steps j]

- Case 1: S brings d into the cache in step j without a request. ✔
- Case 2: S brings d into the cache in step j even though d is in cache. ✔
- If multiple unreduced items in step j, apply each one in turn, dealing with Case 1 before Case 2. ▪

resolving Case 1 might trigger Case 2

Farthest-in-future: analysis

Theorem. FF is optimal eviction algorithm.
Pf. Follows directly from the following invariant.

Invariant. There exists an optimal reduced schedule S that has the same eviction schedule as S_{FF} through the first j steps.
Pf. [by induction on number of steps j]
Base case: $j = 0$.
Let S be reduced schedule that satisfies invariant through j steps.
We produce S' that satisfies invariant after $j + 1$ steps.

- Let d denote the item requested in step $j + 1$.
- Since S and S_{FF} have agreed up until now, they have the same cache contents before step $j + 1$.
- Case 1: d is already in the cache.
  $S' = S$ satisfies invariant.
- Case 2: d is not in the cache and S and S_{FF} evict the same item.
  $S' = S$ satisfies invariant.

Farthest-in-future: analysis

Pf. [continued]

- Case 3: d is not in the cache; S_{FF} evicts e; S evicts $f \neq e$.
  - begin construction of S' from S by evicting e instead of f

  - now S' agrees with S_{FF} for first $j+1$ steps; we show that having item f in cache is no worse than having item e in cache
  - let S' behave the same as S until S' is forced to take a different action (because either S evicts e; or because either e or f is requested)

Farthest-in-future: analysis

Let j' be the first step after $j+1$ that S' must take a different action from S; let g denote the item requested in step j'.

involves either e or f (or both)

same	e	**step j′**	*same*	f

S **S′**

- Case 3a: $g = e$.
  Can't happen with FF since there must be a request for f before e.

  S' agrees with S_{FF} through first $j+1$ steps

- Case 3b: $g = f$.
  Element f can't be in cache of S; let e' be the item that S evicts.
  - if $e' = e$, S' accesses f from cache; now S and S' have same cache
  - if $e' \neq e$, we make S' evict e' and bring e into the cache; now S and S' have the same cache

  We let S' behave exactly like S for remaining requests.

S' is no longer reduced, but can be transformed into a reduced schedule that agrees with FF through first $j+1$ steps

Farthest-in-future: analysis

Let j' be the first step after $j+1$ that S' must take a different action from S;
let g denote the item requested in step j'.

involves wither e or f (or both)

same	e	**step j′**	*same*	f

S **S′**

otherwise S' could have taken the same action

- Case 3c: $g \neq e, f$. S evicts e.
  - make S' evict f.

same	g	**step j′**	*same*	g

S **S′**

  - now S and S' have the same cache
  - let S' behave exactly like S for the remaining requests ▪

Caching perspective

Online vs. offline algorithms.

- Offline: full sequence of requests is known a priori.
- Online (reality): requests are not known in advance.
- Caching is among most fundamental online problems in CS.

LIFO. Evict item brought in most recently.

LRU. Evict item whose most recent access was earliest.

FF with direction of time reversed!

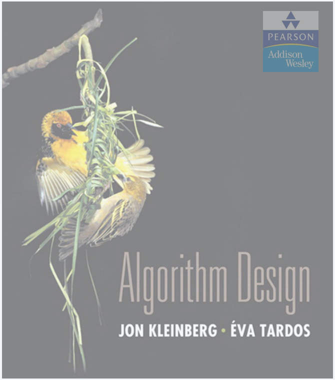
Theorem. FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LIFO can be arbitrarily bad.
- LRU is k-competitive: for any sequence of requests σ, $LRU(\sigma) \leq k\, FF(\sigma) + k$.
- Randomized marking is $O(\log k)$-competitive.

see SECTION 13.8

GREEDY ALGORITHMS

SHORTEST PATHS — DIJKSTRA'S ALGORITHM



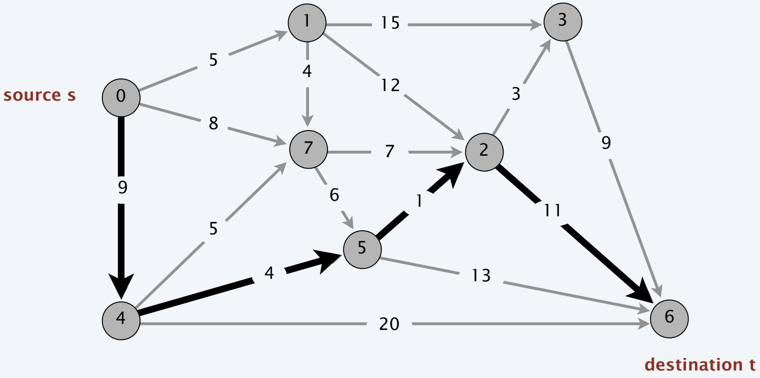
SECTION 4.4

4. GREEDY ALGORITHMS II

- *Dijkstra's algorithm*
- *minimum spanning trees*
- *Prim, Kruskal, Boruvka*
- *single-link clustering*
- *min-cost arborescences*

Single-pair shortest path problem

Problem. Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, and destination $t \in V$, find a shortest directed path from s to t.

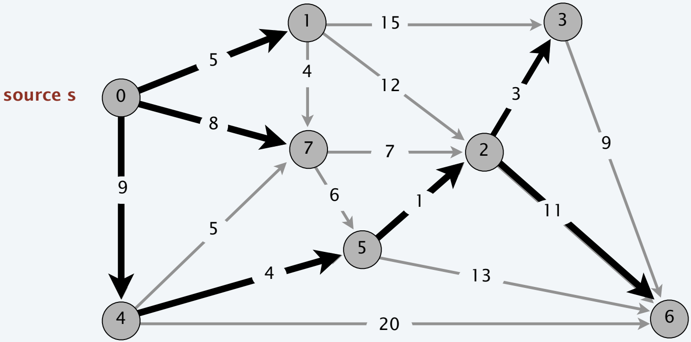


length of path = 9 + 4 + 1 + 11 = 25

Single-source shortest paths problem

Problem. Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, find a shortest directed path from s to every node.

Assumption. There exists a path from s to every node.



shortest-paths tree

Shortest paths: quiz 1

Suppose that you change the length of every edge of G as follows. For which is every shortest path in G a shortest path in G′?

- **A.** Add 17.
- **B.** Multiply by 17.
- **C.** Either A or B.
- **D.** Neither A nor B.

Shortest paths: quiz 2

Which variant in car GPS?

A. Single source: from one node s to every other node.

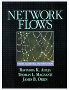
B. Single sink: from every node to one node t.

C. Source–sink: from one node s to another node t.

D. All pairs: between all pairs of nodes.

Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in LaTeX.
- Urban traffic planning.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.



Network Flows: Theory, Algorithms, and Applications,
by Ahuja, Magnanti, and Orlin, Prentice Hall, 1993.

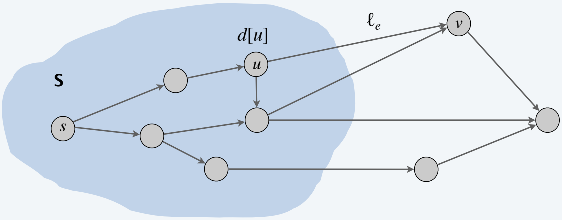
Dijkstra's algorithm (for single-source shortest paths problem)

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \rightarrow u$ path.

- Initialize $S \leftarrow \{ s \}$, $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) \;=\; \min_{e\,=\,(u,v)\,:\,u \in S} \boxed{d[u] + \ell_e}$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$



Dijkstra's algorithm (for single-source shortest paths problem)

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \rightsquigarrow u$ path.

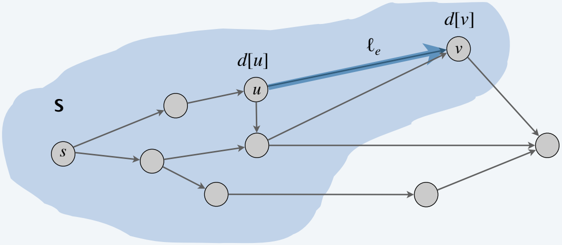
- Initialize $S \leftarrow \{ s \}$, $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) \;=\; \min_{e\,=\,(u,v)\,:\,u \in S} \boxed{d[u] + \ell_e}$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$

add v to S, and set $d[v] \leftarrow \pi(v)$.

- To recover path, set $pred[v] \leftarrow e$ that achieves min.



Dijkstra's algorithm: proof of correctness

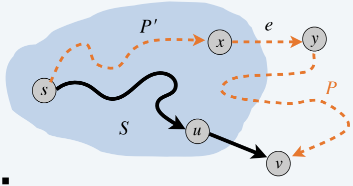
Invariant. For each node $u \in S$: $d[u]$ = length of a shortest $s \leadsto u$ path.

Pf. [by induction on $|S|$]

Base case: $|S| = 1$ is easy since $S = \{ s \}$ and $d[s] = 0$.

Inductive hypothesis: Assume true for $|S| \geq 1$.

- Let v be next node added to S, and let (u, v) be the final edge.
- A shortest $s \leadsto u$ path plus (u, v) is an $s \leadsto v$ path of length $\pi(v)$.
- Consider any other $s \leadsto v$ path P. We show that it is no shorter than $\pi(v)$.
- Let $e = (x, y)$ be the first edge in P that leaves S, and let P' be the subpath from s to x.
- The length of P is already $\geq \pi(v)$ as soon as it reaches y:



$$\underbrace{\ell(P) \geq \ell(P') + \ell_e}_{\text{non-negative lengths}} \quad \underbrace{\geq d[x] + \ell_e}_{\text{inductive hypothesis}} \quad \underbrace{\geq \pi(y)}_{\text{definition of } \pi(y)} \quad \underbrace{\geq \pi(v)}_{\text{Dijkstra chose } v \text{ instead of } y} \quad \blacksquare$$

Dijkstra's algorithm: efficient implementation

Critical optimization 1. For each unexplored node $v \notin S$:
explicitly maintain $\pi[v]$ instead of computing directly from definition

$$\pi(v) \;=\; \min_{e\,=\,(u,v)\,:\,u\in S} \; d[u] + \ell_e$$

- For each $v \notin S$: $\pi(v)$ can only decrease (because set S increases).

- More specifically, suppose u is added to S and there is an edge $e = (u, v)$ leaving u. Then, it suffices to update:

$$\pi[v] \leftarrow \min \{ \pi[v], \; \pi[u] + \ell_e) \}$$

recall: for each $u \in S$,
$\pi[u] = d[u] =$ length of shortest $s \rightarrow u$ path

Critical optimization 2. Use a min-oriented priority queue (PQ)
to choose an unexplored node that minimizes $\pi[v]$.

Dijkstra's algorithm: efficient implementation

Implementation.

- Algorithm maintains $\pi[v]$ for each node v.
- Priority queue stores unexplored nodes, using $\pi[\cdot]$ as priorities.
- Once u is deleted from the PQ, $\pi[u]$ = length of a shortest $s \rightsquigarrow u$ path.

DIJKSTRA (V, E, ℓ, s)

FOREACH $v \neq s$: $\pi[v] \leftarrow \infty$, $pred[v] \leftarrow null$; $\pi[s] \leftarrow 0$.

Create an empty priority queue pq.

FOREACH $v \in V$: INSERT$(pq, v, \pi[v])$.

WHILE (IS-NOT-EMPTY(pq))

    $u \leftarrow$ DEL-MIN(pq).

    FOREACH edge $e = (u, v) \in E$ leaving u:

        IF $(\pi[v] > \pi[u] + \ell_e)$

            DECREASE-KEY$(pq, v, \pi[u] + \ell_e)$.

            $\pi[v] \leftarrow \pi[u] + \ell_e$; $pred[v] \leftarrow e$.

Dijkstra's algorithm: which priority queue?

Performance. Depends on PQ: n INSERT, n DELETE-MIN, $\leq m$ DECREASE-KEY.

- Array implementation optimal for dense graphs. ⟵ $\Theta(n^2)$ edges
- Binary heap much faster for sparse graphs. ⟵ $\Theta(n)$ edges
- 4-way heap worth the trouble in performance-critical situations.

priority queue	INSERT	DELETE-MIN	DECREASE-KEY	total
unordered array	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
d-way heap (Johnson 1975)	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(m \log_{m/n} n)$
Fibonacci heap (Fredman–Tarjan 1984)	$O(1)$	$O(\log n)$ †	$O(1)$ †	$O(m + n \log n)$
integer priority queue (Thorup 2004)	$O(1)$	$O(\log \log n)$	$O(1)$	$O(m + n \log \log n)$

† amortized

Shortest paths: quiz 3

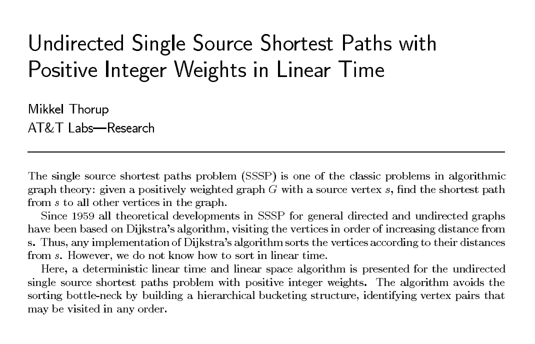
How to solve the the single-source shortest paths problem in undirected graphs with positive edge lengths?

- **A.** Replace each undirected edge with two antiparallel edges of same length. Run Dijkstra's algorithm in the resulting digraph.
- **B.** Modify Dijkstra's algorithms so that when it processes node u, it consider all edges incident to u (instead of edges leaving u).
- **C.** Either A or B.
- **D.** Neither A nor B.

Shortest paths: quiz 3

Theorem. [Thorup 1999] Can solve single-source shortest paths problem in undirected graphs with positive integer edge lengths in $O(m)$ time.

Remark. Does not explore nodes in increasing order of distance from s.



Undirected Single Source Shortest Paths with Positive Integer Weights in Linear Time

Mikkel Thorup
AT&T Labs—Research

The single source shortest paths problem (SSSP) is one of the classic problems in algorithmic graph theory: given a positively weighted graph G with a source vertex s, find the shortest path from s to all other vertices in the graph.

Since 1959 all theoretical developments in SSSP for general directed and undirected graphs have been based on Dijkstra's algorithm, visiting the vertices in order of increasing distance from s. Thus, any implementation of Dijkstra's algorithm sorts the vertices according to their distances from s. However, we do not know how to sort in linear time.

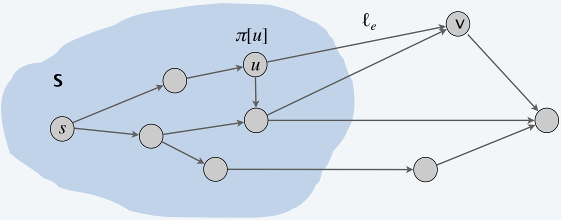
Here, a deterministic linear time and linear space algorithm is presented for the undirected single source shortest paths problem with positive integer weights. The algorithm avoids the sorting bottle-neck by building a hierarchical bucketing structure, identifying vertex pairs that may be visited in any order.

Extensions of Dijkstra's algorithm

Dijkstra's algorithm and proof extend to several related problems:

- Shortest paths in undirected graphs: $\pi[v] \leq \pi[u] + \ell(u, v)$.
- Maximum capacity paths: $\pi[v] \geq \min \{ \pi[u],\ c(u, v) \}$.
- Maximum reliability paths: $\pi[v] \geq \pi[u] \times \gamma(u, v)$.
- …

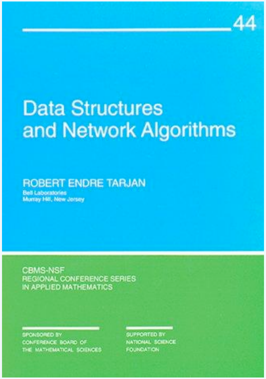
Key algebraic structure. Closed semiring (min-plus, bottleneck, Viterbi, …).



$$
\begin{aligned}
a + b &= b + a \\
a + (b + c) &= (a + b) + c \\
a + 0 &= a \\
a \cdot (b \cdot c) &= (a \cdot b) \cdot c \\
a \cdot 0 &= 0 \cdot a = 0 \\
a \cdot 1 &= 1 \cdot a = a \\
a \cdot (b + c) &= a \cdot b + a \cdot c \\
(a + b) \cdot c &= a \cdot c + b \cdot c \\
a^* &= 1 + a \cdot a^* = 1 + a^* \cdot a
\end{aligned}
$$

GREEDY ALGORITHMS

MINIMUM SPANNING TREES — PRIM, KRUSKAL AND BORŮVKA



SECTION 6.1

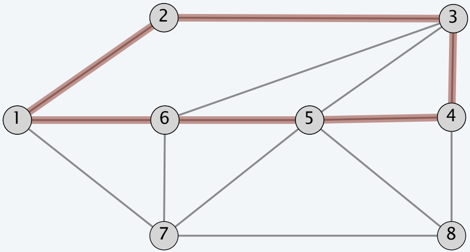
4. GREEDY ALGORITHMS II

- *Dijkstra's algorithm*
- ***minimum spanning trees***
- *Prim, Kruskal, Boruvka*
- *single-link clustering*
- *min-cost arborescences*

Cycles

Def. A path is a sequence of edges which connects a sequence of nodes.

Def. A cycle is a path with no repeated nodes or edges other than the starting and ending nodes.



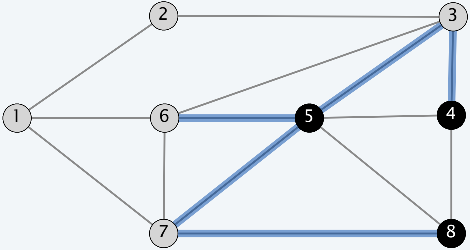
path P = { (1, 2), (2, 3), (3, 4), (4, 5), (5, 6) }

cycle C = { (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1) }

Cuts

Def. A cut is a partition of the nodes into two nonempty subsets S and $V - S$.

Def. The cutset of a cut S is the set of edges with exactly one endpoint in S.



cut S = { 4, 5, 8 }

cutset D = { (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) }

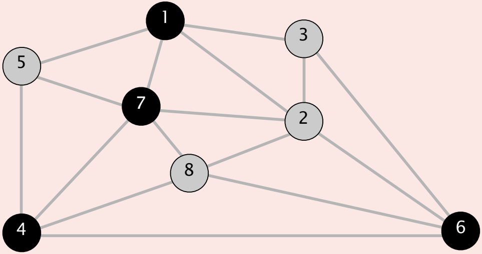
Consider the cut S = { 1, 4, 6, 7 }. Which edge is in the cutset of S?

A. S is not a cut (not connected)

B. 1–7

C. 5–7

D. 2–3

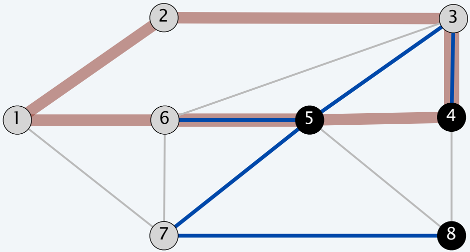


Let C be a cycle and let D be a cutset. How many edges do C and D have in common? Choose the best answer.

- **A.** 0
- **B.** 2
- **C.** not 1
- **D.** an even number

Cycle–cut intersection

Proposition. A cycle and a cutset intersect in an even number of edges.



cycle C = { (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1) }

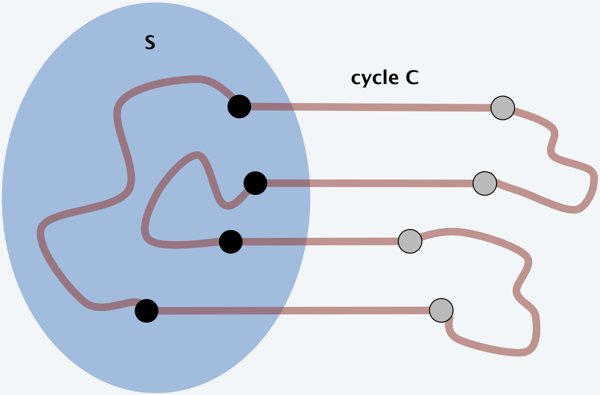
cutset D = { (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) }

intersection C ∩ D = { (3, 4), (5, 6) }

Cycle–cut intersection

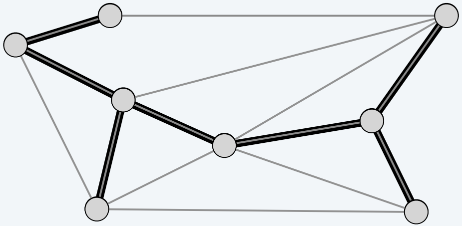
Proposition. A cycle and a cutset intersect in an even number of edges.

Pf. [by picture]



Spanning tree definition

Def. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$.
H is a spanning tree of G if H is both acyclic and connected.



graph G = (V, E)
spanning tree H = (V, T)

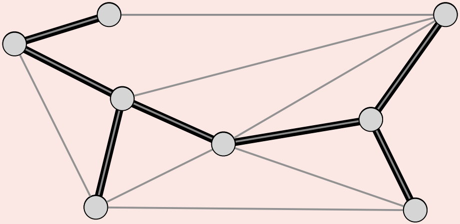
Which of the following properties are true for all spanning trees H?

A. Contains exactly $|V| - 1$ edges.

B. The removal of any edge disconnects it.

C. The addition of any edge creates a cycle.

D. All of the above.



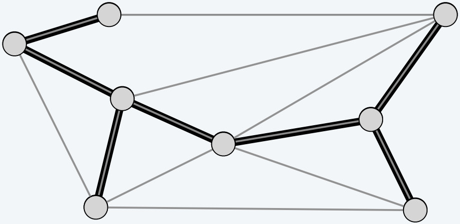
graph G = (V, E)

spanning tree H = (V, T)

Spanning tree properties

Proposition. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$.
Then, the following are equivalent:

- H is a spanning tree of G.
- H is acyclic and connected.
- H is connected and has $|V| - 1$ edges.
- H is acyclic and has $|V| - 1$ edges.
- H is minimally connected: removal of any edge disconnects it.
- H is maximally acyclic: addition of any edge creates a cycle.



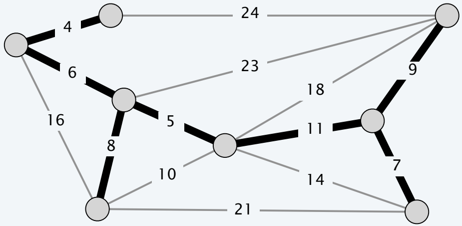
graph G = (V, E)

spanning tree H = (V, T)


A TREE WITH A CYCLE

Minimum spanning tree (MST)

Def. Given a connected, undirected graph $G = (V, E)$ with edge costs c_e, a minimum spanning tree (V, T) is a spanning tree of G such that the sum of the edge costs in T is minimized.



MST cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

Cayley's theorem. The complete graph on n nodes has n^{n-2} spanning trees.

can't solve by brute force

Suppose that you change the cost of every edge in G as follows.
For which is every MST in G an MST in G′ (and vice versa)?
Assume c(e) > 0 for each e.

A. $c'(e) = c(e) + 17$.

B. $c'(e) = 17 \times c(e)$.

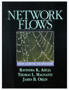
C. $c'(e) = \log_{17} c(e)$.

D. All of the above.

Applications

MST is fundamental problem with diverse applications.

- Dithering.
- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Model locality of particle interactions in turbulent fluid flows.
- Reducing data storage in sequencing amino acids in a protein.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
- Network design (communication, electrical, hydraulic, computer, road).

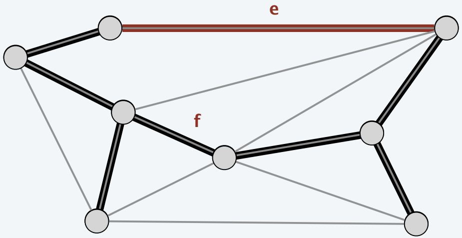


Network Flows: Theory, Algorithms, and Applications,
by Ahuja, Magnanti, and Orlin, Prentice Hall, 1993.

Fundamental cycle

Fundamental cycle. Let $H = (V, T)$ be a spanning tree of $G = (V, E)$.

- For any non tree-edge $e \in E$: $T \cup \{ e \}$ contains a unique cycle, say C.
- For any edge $f \in C$: $T \cup \{ e \} - \{ f \}$ is a spanning tree.



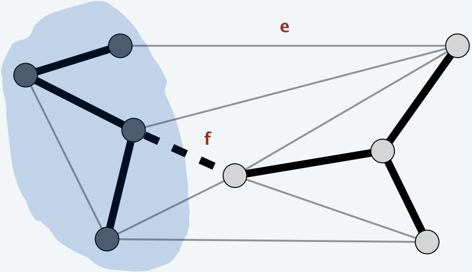
graph G = (V, E)
spanning tree H = (V, T)

Observation. If $c_e < c_f$, then (V, T) is not an MST.

Fundamental cutset

Fundamental cutset. Let $H = (V, T)$ be a spanning tree of $G = (V, E)$.

- For any tree edge $f \in T$: $T - \{ f \}$ contains two connected components. Let D denote corresponding cutset.
- For any edge $e \in D$: $T - \{ f \} \cup \{ e \}$ is a spanning tree.



graph G = (V, E)

spanning tree H = (V, T)

Observation. If $c_e < c_f$, then (V, T) is not an MST.

The greedy algorithm

Red rule.

- Let C be a cycle with no red edges.
- Select an uncolored edge of C of max cost and color it red.

Blue rule.

- Let D be a cutset with no blue edges.
- Select an uncolored edge in D of min cost and color it blue.

Greedy algorithm.

- Apply the red and blue rules (nondeterministically!) until all edges are colored. The blue edges form an MST.
- Note: can stop once $n - 1$ edges colored blue.

Greedy algorithm: proof of correctness

Color invariant. There exists an MST (V, T^*) containing every blue edge and no red edge.
Pf. [by induction on number of iterations]

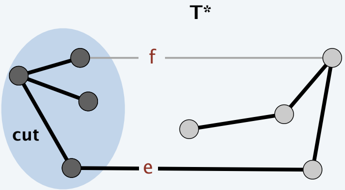
Base case. No edges colored $\Rightarrow$ every MST satisfies invariant.

Greedy algorithm: proof of correctness

Color invariant. There exists an MST (V, T^*) containing every blue edge and no red edge.
Pf. [by induction on number of iterations]

Induction step (blue rule). Suppose color invariant true before blue rule.

- let D be chosen cutset, and let f be edge colored blue.
- if $f \in T^*$, then T^* still satisfies invariant.
- Otherwise, consider fundamental cycle C by adding f to T^*.
- let $e \in C$ be another edge in D.
- e is uncolored and $c_e \geq c_f$ since
  - $e \in T^* \Rightarrow e$ not red
  - blue rule $\Rightarrow e$ not blue and $c_e \geq c_f$
- Thus, $T^* \cup \{ f \} - \{ e \}$ satisfies invariant.

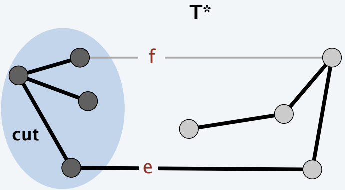


Greedy algorithm: proof of correctness

Color invariant. There exists an MST (V, T^*) containing every blue edge and no red edge.
Pf. [by induction on number of iterations]

Induction step (red rule). Suppose color invariant true before red rule.

- let C be chosen cycle, and let e be edge colored red.
- if $e \notin T^*$, then T^* still satisfies invariant.
- Otherwise, consider fundamental cutset D by deleting e from T^*.
- let $f \in D$ be another edge in C.
- f is uncolored and $c_e \geq c_f$ since
  - $f \notin T^* \Rightarrow f$ not blue
  - red rule $\Rightarrow f$ not red and $c_e \geq c_f$
- Thus, $T^* \cup \{ f \} - \{ e \}$ satisfies invariant. ▪



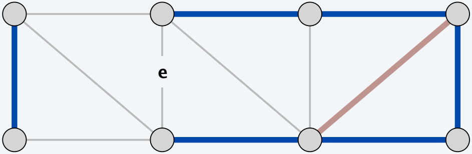
Greedy algorithm: proof of correctness

Theorem. The greedy algorithm terminates. Blue edges form an MST.

Pf. We need to show that either the red or blue rule (or both) applies.

- Suppose edge e is left uncolored.
- Blue edges form a forest.
- Case 1: both endpoints of e are in same blue tree.

  $\Rightarrow$ apply red rule to cycle formed by adding e to blue forest.



Case 1

Greedy algorithm: proof of correctness

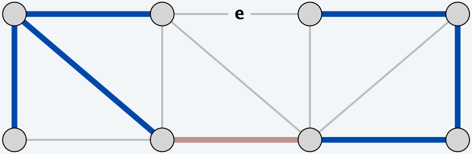
Theorem. The greedy algorithm terminates. Blue edges form an MST.

Pf. We need to show that either the red or blue rule (or both) applies.

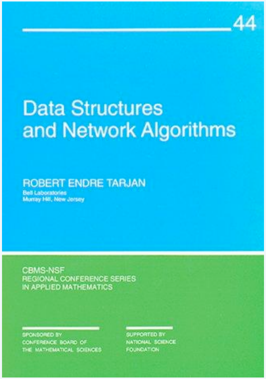
- Suppose edge e is left uncolored.
- Blue edges form a forest.
- Case 1: both endpoints of e are in same blue tree.

  $\Rightarrow$ apply red rule to cycle formed by adding e to blue forest.
- Case 2: both endpoints of e are in different blue trees.

  $\Rightarrow$ apply blue rule to cutset induced by either of two blue trees. ▪



Case 2



SECTION 6.2

4. GREEDY ALGORITHMS II

- *Dijkstra's algorithm*
- *minimum spanning trees*
- *Prim, Kruskal, Boruvka*
- *single-link clustering*
- *min-cost arborescences*

Prim's algorithm

Initialize S = any node, $T = \varnothing$.

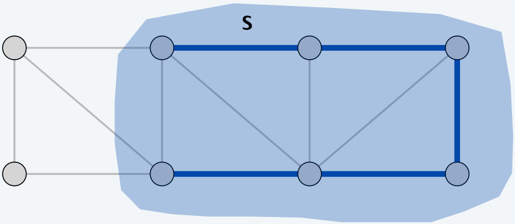
Repeat $n - 1$ times:

- Add to T a min-cost edge with one endpoint in S.
- Add new node to S.

by construction, edges in cutset are uncolored

Theorem. Prim's algorithm computes an MST.

Pf. Special case of greedy algorithm (blue rule repeatedly applied to S). ▪



Kruskal's algorithm

Consider edges in ascending order of cost:

- Add to tree unless it would create a cycle.

Theorem. Kruskal's algorithm computes an MST.

Pf. Special case of greedy algorithm.

- Case 1: both endpoints of e in same blue tree.

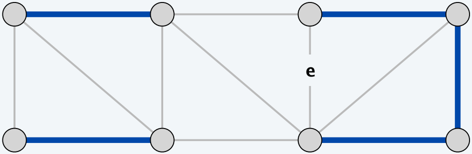
  $\Rightarrow$ color e red by applying red rule to unique cycle.

  all other edges in cycle are blue

- Case 2: both endpoints of e in different blue trees.

  $\Rightarrow$ color e blue by applying blue rule to cutset defined by either tree. ▪

  no edge in cutset has smaller cost (since Kruskal chose it first)



Kruskal's algorithm: implementation

Theorem. Kruskal's algorithm can be implemented to run in $O(m \log m)$ time.

- Sort edges by cost.
- Use union–find data structure to dynamically maintain connected components.

KRUSKAL (V, E, c)

SORT m edges by cost and renumber so that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$.

$T \leftarrow \varnothing$.

FOREACH $v \in V$: MAKE-SET(v).

FOR $i = 1$ TO m

  $(u, v) \leftarrow e_i$.

  IF (FIND-SET$(u) \neq$ FIND-SET(v)) ← are u and v in same component?

    $T \leftarrow T \cup \{ e_i \}$.

    UNION(u, v). ← make u and v in same component

RETURN T.

Reverse-delete algorithm

Start with all edges in T and consider them in descending order of cost:

- Delete edge from T unless it would disconnect T.

Theorem. The reverse-delete algorithm computes an MST.

Pf. Special case of greedy algorithm.

- Case 1. [deleting edge e does not disconnect T]

  $\Rightarrow$ apply red rule to cycle C formed by adding e to another path in T between its two endpoints

  no edge in C is more expensive
  (it would have already been considered and deleted)

- Case 2. [deleting edge e disconnects T]

  $\Rightarrow$ apply blue rule to cutset D induced by either component ▪

  e is the only remaining edge in the cutset
  (all other edges in D must have been colored red / deleted)

Fact. [Thorup 2000] Can be implemented to run in $O(m \log n (\log \log n)^3)$ time.

Review: the greedy MST algorithm

Red rule.

- Let C be a cycle with no red edges.
- Select an uncolored edge of C of max cost and color it red.

Blue rule.

- Let D be a cutset with no blue edges.
- Select an uncolored edge in D of min cost and color it blue.

Greedy algorithm.

- Apply the red and blue rules (nondeterministically!) until all edges are colored. The blue edges form an MST.
- Note: can stop once $n - 1$ edges colored blue.

Theorem. The greedy algorithm is correct.

Special cases. Prim, Kruskal, reverse-delete, ...

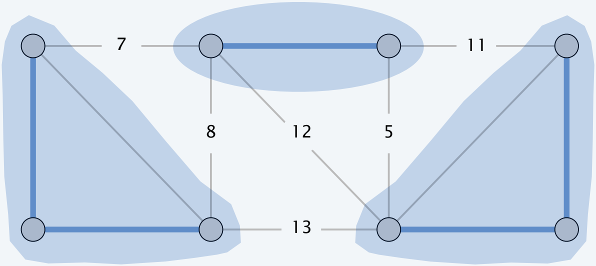
Borůvka's algorithm

Repeat until only one tree.

- Apply blue rule to cutset corresponding to each blue tree.
- Color all selected edges blue.

Theorem. Borůvka's algorithm computes the MST. ← assume edge costs are distinct

Pf. Special case of greedy algorithm (repeatedly apply blue rule). ▪

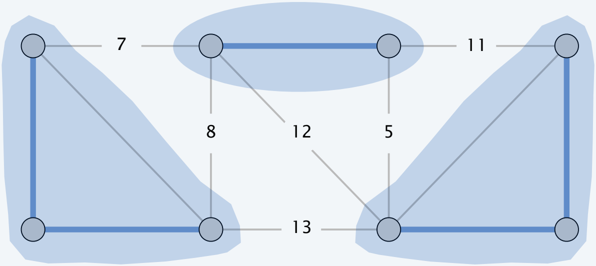


Borůvka's algorithm: implementation

Theorem. Borůvka's algorithm can be implemented to run in $O(m \log n)$ time.

Pf.

- To implement a phase in $O(m)$ time:
  - compute connected components of blue edges
  - for each edge $(u, v) \in E$, check if u and v are in different components; if so, update each component's best edge in cutset
- $\leq \log_2 n$ phases since each phase (at least) halves total # components. ▪



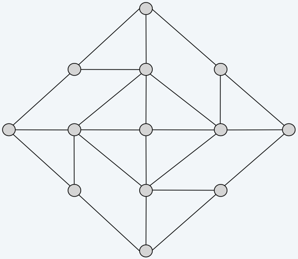
```
Function Boruvka(V, E, c)
K ← ∅
count ← CountAndLabel(K)
while count > 1 do
    for i = 1 to count do  S[i] ← Nil
    forall the (u, v) ∈ E do
        if label(u) ≠ label(v) then
            if c(u, v) < w(S[label(u)]) then  S[label(u)] ← (u, v)
            if c(u, v) < w(S[label(v)]) then  S[label(v)] ← (u, v)
    for i = 1 to count do  if S[i] ≠ nil then add S[i] to K
    count ← CountAndLabel(K)
return K
```

Borůvka's algorithm on planar graphs

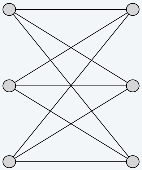
Theorem. Borůvka's algorithm (contraction version) can be implemented to run in $O(n)$ time on planar graphs.

Pf.

- Each Borůvka phase takes $O(n)$ time:
  - Fact 1: $m \leq 3n$ for simple planar graphs.
  - Fact 2: planar graphs remains planar after edge contractions/deletions.
- Number of nodes (at least) halves in each phase.
- Thus, overall running time $\leq cn + cn/2 + cn/4 + cn/8 + \ldots = O(n)$. ▪



planar



$K_{3,3}$ not planar

A hybrid algorithm

Borůvka–Prim algorithm.

- Run Borůvka (contraction version) for $\log_2 \log_2 n$ phases.
- Run Prim on resulting, contracted graph.

Theorem. Borůvka–Prim computes an MST.

Pf. Special case of the greedy algorithm.

Theorem. Borůvka–Prim can be implemented to run in $O(m \log \log n)$ time.

Pf.

- The $\log_2 \log_2 n$ phases of Borůvka's algorithm take $O(m \log \log n)$ time; resulting graph has $\leq n / \log_2 n$ nodes and $\leq m$ edges.
- Prim's algorithm (using Fibonacci heaps) takes $O(m + n)$ time on a graph with $n / \log_2 n$ nodes and m edges. ▪

$$O\left(m + \frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right)$$

Does a linear-time compare-based MST algorithm exist?

year	worst case	discovered by
1975	$O(m \log \log n)$	Yao
1976	$O(m \log \log n)$	Cheriton–Tarjan
1984	$O(m \log^* n)$, $O(m + n \log n)$	Fredman–Tarjan
1986	$O(m \log (\log^* n))$	Gabow–Galil–Spencer–Tarjan
1997	$O(m\ \alpha(n) \log \alpha(n))$	Chazelle
2000	$O(m\ \alpha(n))$	Chazelle
2002	*asymptotically optimal*	Pettie–Ramachandran
20xx	$O(m)$	???

deterministic compare-based MST algorithms

iterated logarithm function

$$\lg^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \lg^*(\lg n) & \text{if } n > 1 \end{cases}$$

n	$\lg^* n$
$(-\infty, 1]$	0
$(1, 2]$	1
$(2, 4]$	2
$(4, 16]$	3
$(16, 2^{16}]$	4
$(2^{16}, 2^{65536}]$	5

Theorem. [Fredman–Willard 1990] $O(m)$ in word RAM model.

Theorem. [Dixon–Rauch–Tarjan 1992] $O(m)$ MST verification algorithm.

Theorem. [Karger–Klein–Tarjan 1995] $O(m)$ randomized MST algorithm.

Part IV

Network Flows

The Ford-Fulkerson Method

- Ford-Fulkerson Algorithm
- Capacity-Scaling Algorithm
- Shortest Augmenting Path
- Dinitz' Algorithm

The Push-Relabel Method

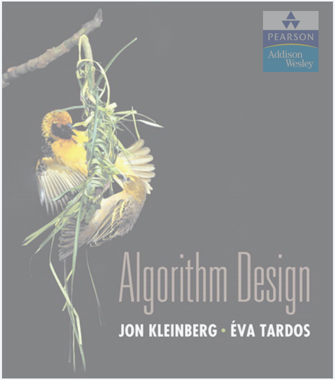
Network Flows — Applications

- Bipartite Matching
- Disjoint Paths
- Multiple Sources and Sinks
- Circulations with Supplies and Demands

THE FORD-FULKERSON METHOD

THE FORD-FULKERSON METHOD

PROBLEM FORMULATION



SECTION 7.1

7. NETWORK FLOW I

- *max-flow and min-cut problems*
- *Ford–Fulkerson algorithm*
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *Dinitz' algorithm*
- *simple unit-capacity networks*

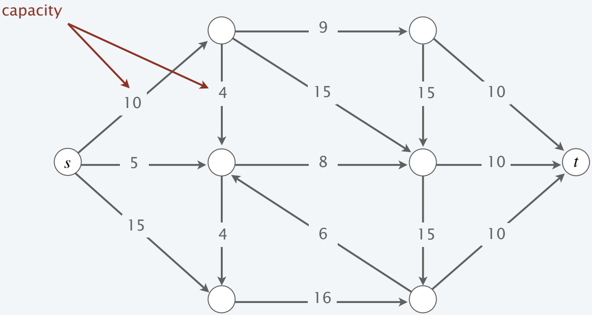
Flow network

A flow network is a tuple $G = (V, E, s, t, c)$.

- Digraph (V, E) with source $s \in V$ and sink $t \in V$.
- Capacity $c(e) > 0$ for each $e \in E$.

assume all nodes are reachable from s

Intuition. Material flowing through a transportation network; material originates at source and is sent to sink.

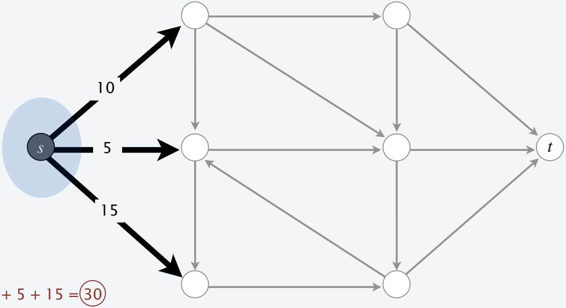


Minimum-cut problem

Def. An *st*-cut (cut) is a partition (A, B) of the nodes with $s \in A$ and $t \in B$.

Def. Its capacity is the sum of the capacities of the edges from A to B.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$



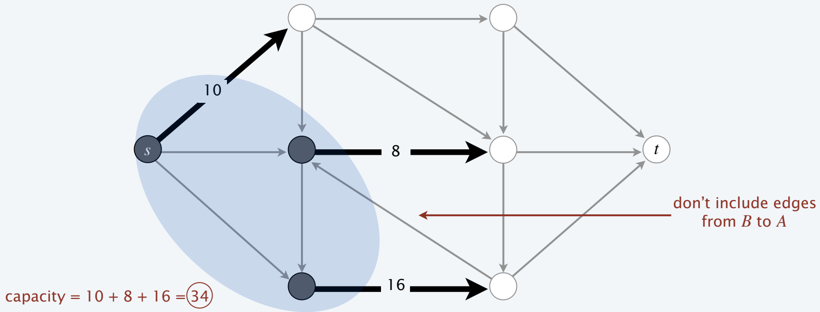
capacity = 10 + 5 + 15 = 30

Minimum-cut problem

Def. An *st*-cut (cut) is a partition (A, B) of the nodes with $s \in A$ and $t \in B$.

Def. Its capacity is the sum of the capacities of the edges from A to B.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$



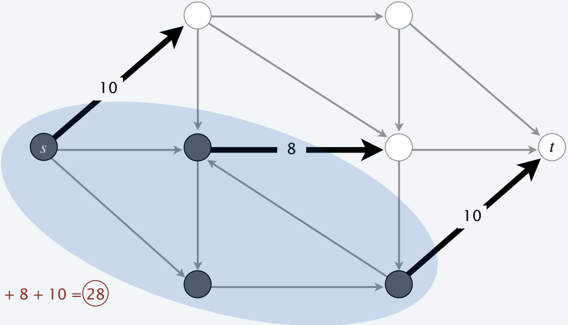
Minimum-cut problem

Def. An *st*-cut (cut) is a partition (A, B) of the nodes with $s \in A$ and $t \in B$.

Def. Its capacity is the sum of the capacities of the edges from A to B.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$

Min-cut problem. Find a cut of minimum capacity.



capacity = 10 + 8 + 10 = 28

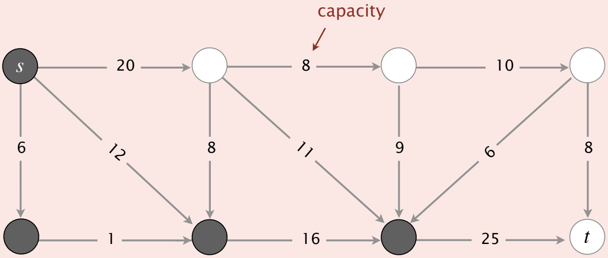
Which is the capacity of the given st-cut?

A. 11 (20 + 25 – 8 – 11 – 9 – 6)

B. 34 (8 + 11 + 9 + 6)

C. 45 (20 + 25)

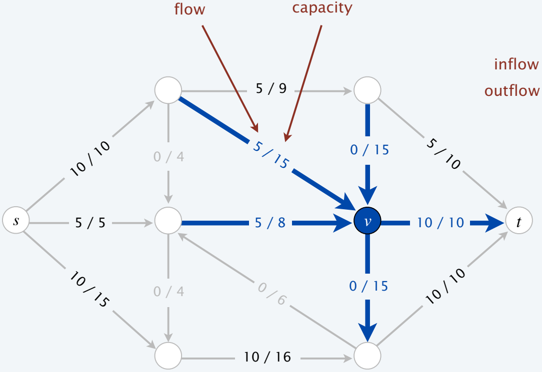
D. 79 (20 + 25 + 8 + 11 + 9 + 6)



Maximum-flow problem

Def. An *st*-flow (flow) f is a function that satisfies:

- For each $e \in E$: $0 \le f(e) \le c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\displaystyle\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]



inflow at v = $5 + 5 + 0 = 10$

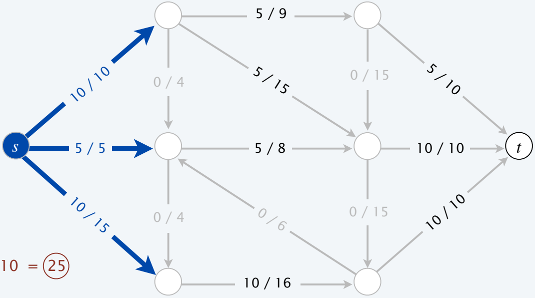
outflow at v = $10 + 0 = 10$

Maximum-flow problem

Def. An *st*-flow (flow) f is a function that satisfies:

- For each $e \in E$: $\quad 0 \le f(e) \le c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\quad \sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Def. The value of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e)$



value = 5 + 10 + 10 = 25

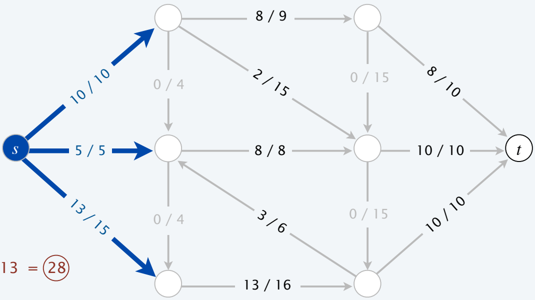
Maximum-flow problem

Def. An *st*-flow (flow) f is a function that satisfies:

- For each $e \in E$: $\quad 0 \le f(e) \le c(e) \quad$ [capacity]
- For each $v \in V - \{s, t\}$: $\quad \sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e) \quad$ [flow conservation]

Def. The value of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e)$

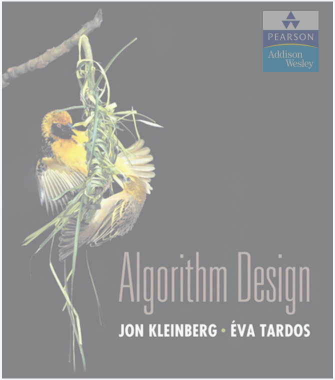
Max-flow problem. Find a flow of maximum value.



value = 10 + 5 + 13 = 28

THE FORD-FULKERSON METHOD

FORD-FULKERSON ALGORITHM



SECTION 7.1

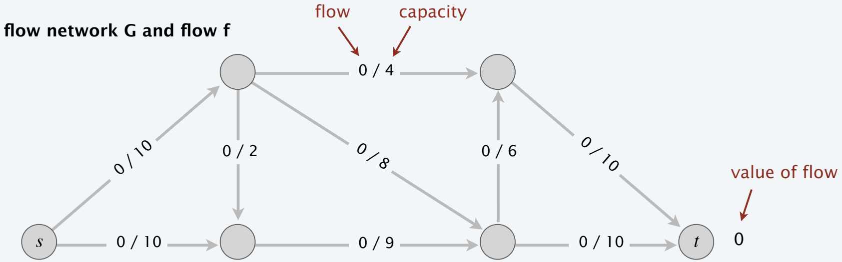
7. NETWORK FLOW I

- max-flow and min-cut problems
- *Ford–Fulkerson algorithm*
- max-flow min-cut theorem
- capacity-scaling algorithm
- shortest augmenting paths
- Dinitz' algorithm
- simple unit-capacity networks

Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.

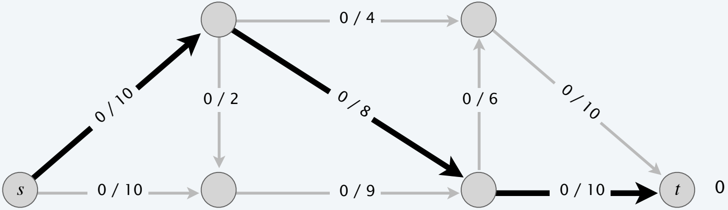


Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.

flow network G and flow f

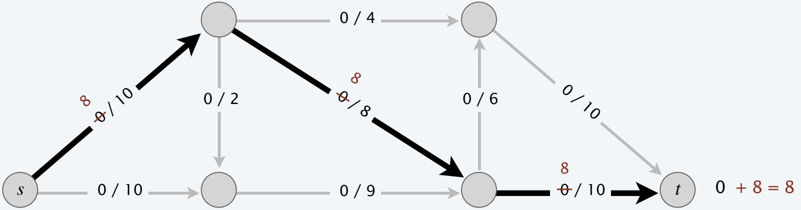


Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.

flow network G and flow f

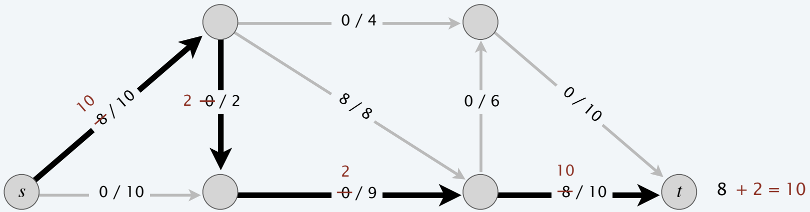


Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.

flow network G and flow f

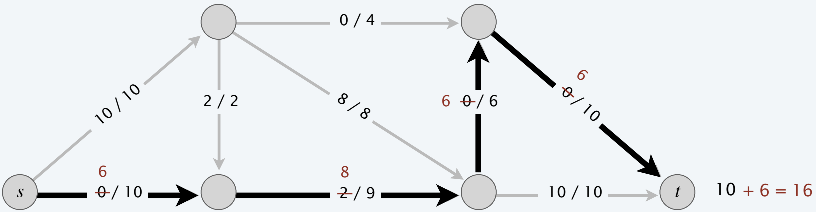


Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightsquigarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.

flow network G and flow f



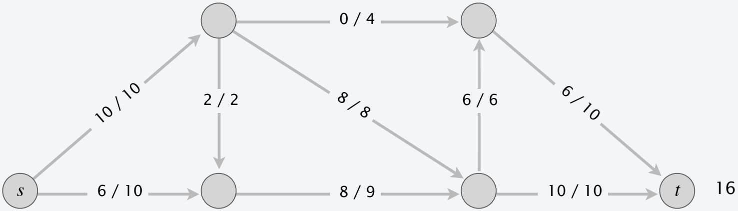
Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.

ending flow value = 16

flow network G and flow f



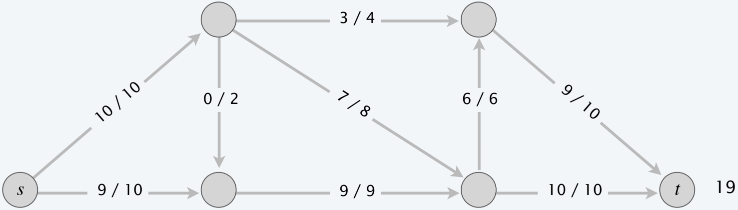
Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.

but max-flow value = 19

flow network G and flow f



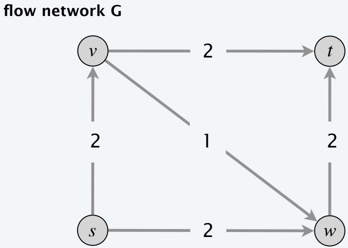
Why the greedy algorithm fails

Q. Why does the greedy algorithm fail?

A. Once greedy algorithm increases flow on an edge, it never decreases it.

Ex. Consider flow network G.

- The unique max flow has $f^*(v, w) = 0$.
- Greedy algorithm could choose $s \rightarrow v \rightarrow w \rightarrow t$ as first augmenting path.

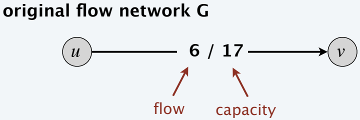


Bottom line. Need some mechanism to "undo" a bad decision.

Residual network

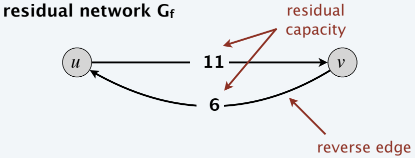
Original edge. $e = (u, v) \in E$.

- Flow $f(e)$.
- Capacity $c(e)$.



Reverse edge. $e^{\text{reverse}} = (v, u)$.

- "Undo" flow sent.



Residual capacity.

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^{\text{reverse}} \in E \end{cases}$$

edges with positive
residual capacity

where flow on a reverse edge
negates flow on
corresponding forward edge

Residual network. $G_f = (V, E_f, s, t, c_f)$.

- $E_f = \{e : f(e) < c(e)\} \cup \{e^{\text{reverse}} : f(e) > 0\}$.
- Key property: f' is a flow in G_f iff $f + f'$ is a flow in G.

Augmenting path

Def. An **augmenting path** is a simple $s \rightsquigarrow t$ path in the residual network G_f.

Def. The **bottleneck capacity** of an augmenting path P is the minimum residual capacity of any edge in P.

Key property. Let f be a flow and let P be an augmenting path in G_f. Then, after calling $f' \leftarrow$ AUGMENT(f, c, P), the resulting f' is a flow and $val(f') = val(f) + bottleneck(G_f, P)$.

```
AUGMENT(f, c, P)
────────────────────────────────────────
δ ← bottleneck capacity of augmenting path P.
FOREACH edge e ∈ P :
    IF (e ∈ E) f(e) ← f(e) + δ.
    ELSE       f(e^reverse) ← f(e^reverse) – δ.
RETURN f.
────────────────────────────────────────
```

Network flow: quiz 2

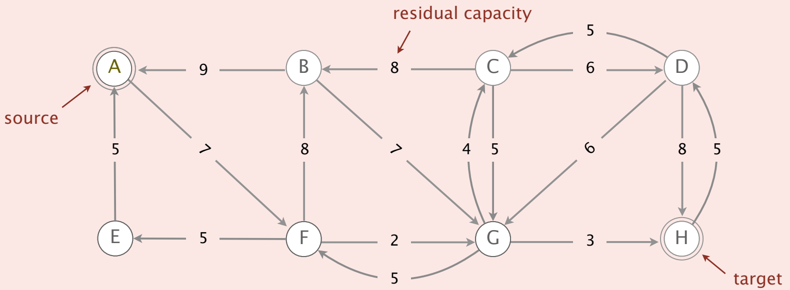
Which is the augmenting path of highest bottleneck capacity?

A. $A \rightarrow F \rightarrow G \rightarrow H$

B. $A \rightarrow B \rightarrow C \rightarrow D \rightarrow H$

C. $A \rightarrow F \rightarrow B \rightarrow G \rightarrow H$

D. $A \rightarrow F \rightarrow B \rightarrow G \rightarrow C \rightarrow D \rightarrow H$



Ford–Fulkerson algorithm

Ford–Fulkerson augmenting path algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightsquigarrow t$ path P in the residual network G_f.
- Augment flow along path P.
- Repeat until you get stuck.

FORD–FULKERSON(G)

FOREACH edge $e \in E$: $f(e) \leftarrow 0$.

$G_f \leftarrow$ residual network of G with respect to flow f.

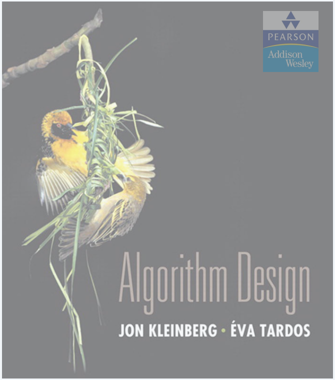
WHILE (there exists an $s \rightsquigarrow t$ path P in G_f)

  $f \leftarrow$ AUGMENT(f, c, P).

  Update G_f.

RETURN f.

augmenting path



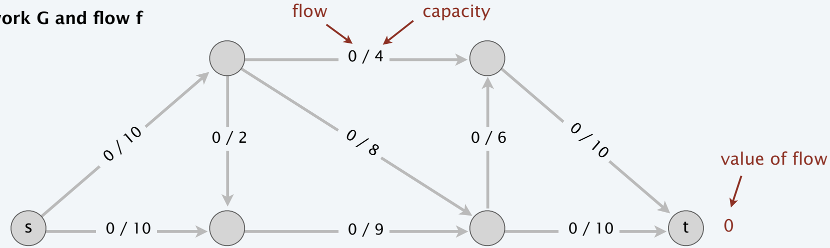
SECTION 7.1

7. NETWORK FLOW I

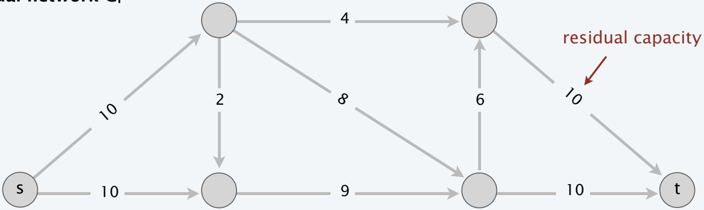
- *Ford–Fulkerson demo*
- *exponential-time example*
- *pathological example*

Ford–Fulkerson algorithm demo

network G and flow f

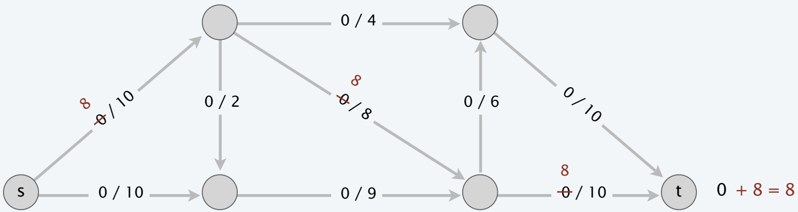


residual network G_f

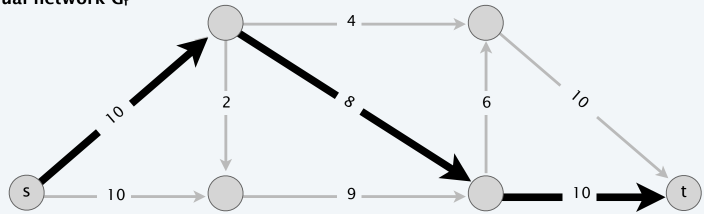


Ford–Fulkerson algorithm demo

network G and flow f

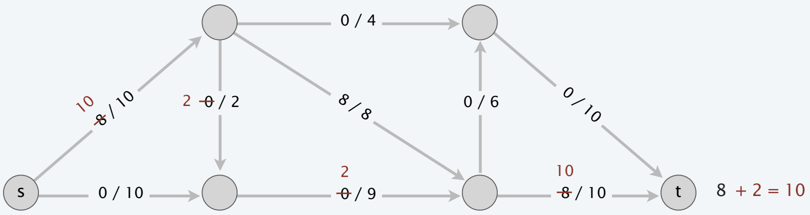


residual network G_f

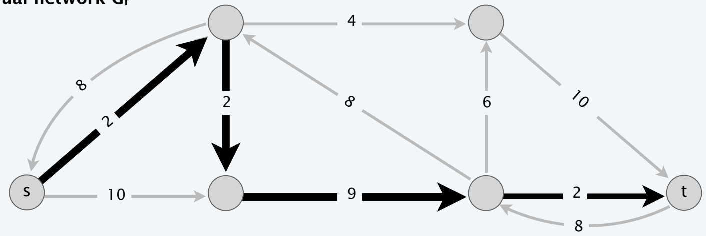


Ford–Fulkerson algorithm demo

network G and flow f

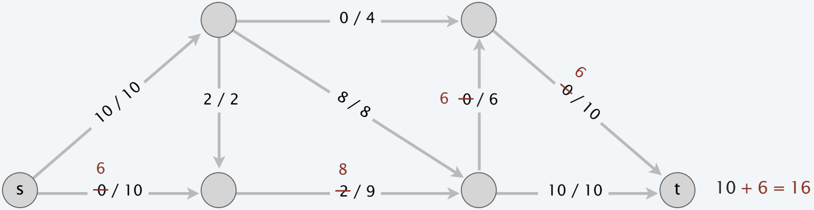


residual network G_f

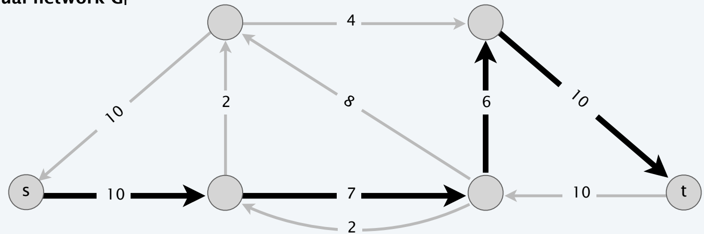


Ford–Fulkerson algorithm demo

network G and flow f

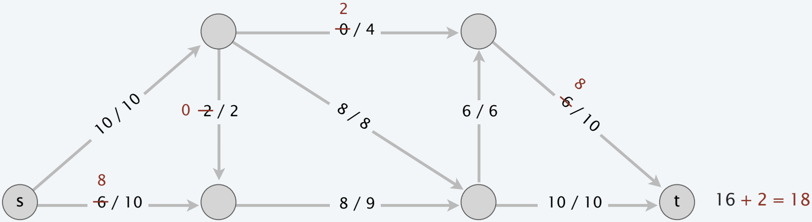


residual network G_f

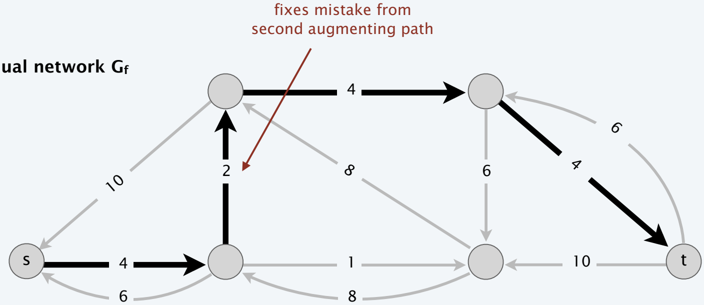


Ford–Fulkerson algorithm demo

network G and flow f

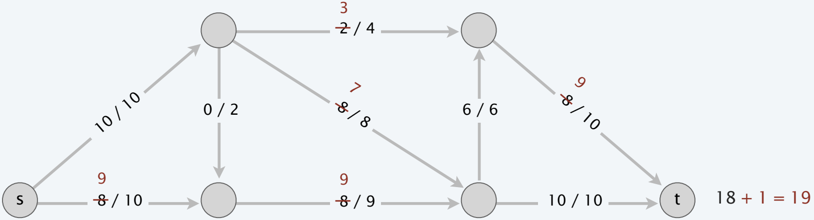


residual network G_f

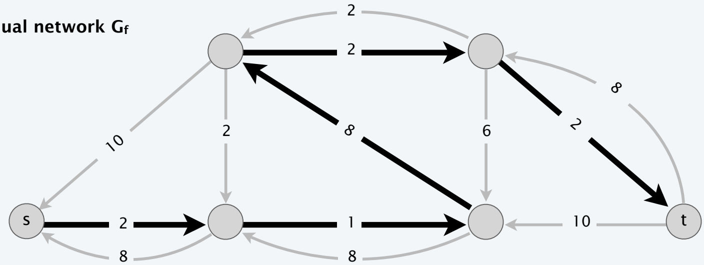


Ford–Fulkerson algorithm demo

network G and flow f

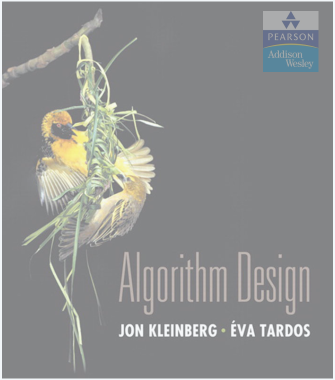


residual network G_f



THE FORD-FULKERSON METHOD

MAX-FLOW MIN-CUT THEOREM



SECTION 7.2

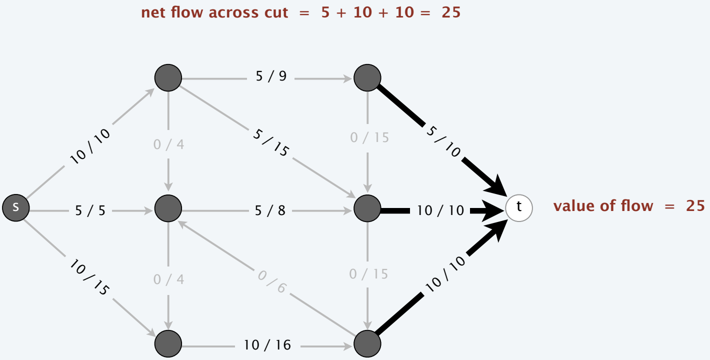
7. NETWORK FLOW I

- *max-flow and min-cut problems*
- *Ford–Fulkerson algorithm*
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *Dinitz' algorithm*
- *simple unit-capacity networks*

Relationship between flows and cuts

Flow value lemma. Let f be any flow and let (A, B) be any cut. Then, the value of the flow f equals the net flow across the cut (A, B).

$$val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$

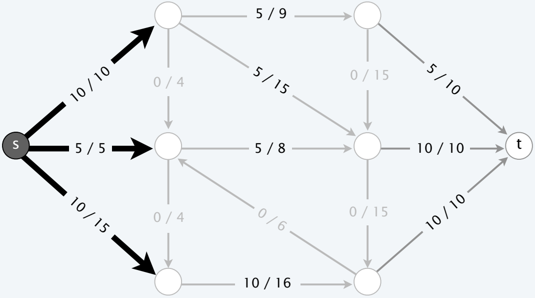


Relationship between flows and cuts

Flow value lemma. Let f be any flow and let (A, B) be any cut. Then, the value of the flow f equals the net flow across the cut (A, B).

$$val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$

net flow across cut = 10 + 5 + 10 = 25



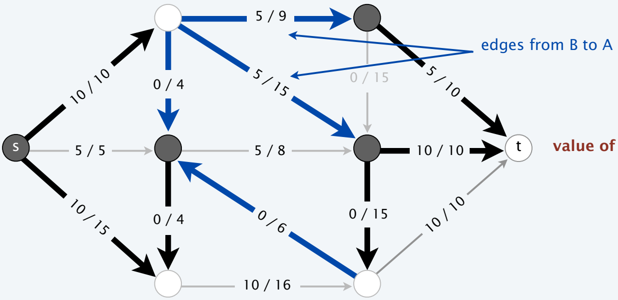
value of flow = 25

Relationship between flows and cuts

Flow value lemma. Let f be any flow and let (A, B) be any cut. Then, the value of the flow f equals the net flow across the cut (A, B).

$$val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$

net flow across cut = (10 + 10 + 5 + 10 + 0 + 0) – (5 + 5 + 0 + 0) = 25



value of flow = 25

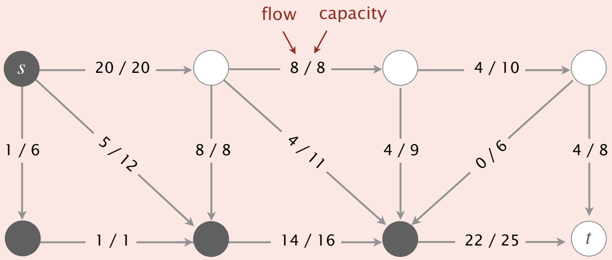
Which is the net flow across the given cut?

A. 11 (20 + 25 – 8 – 11 – 9 – 6)

B. 26 (20 + 22 – 8 – 4 – 4)

C. 42 (20 + 22)

D. 45 (20 + 25)



Relationship between flows and cuts

Flow value lemma. Let f be any flow and let (A, B) be any cut. Then, the value of the flow f equals the net flow across the cut (A, B).

$$val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$

Pf.

$$val(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e)$$

by flow conservation, all terms except for $v = s$ are 0 →

$$= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right)$$

$$= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \quad \blacksquare$$

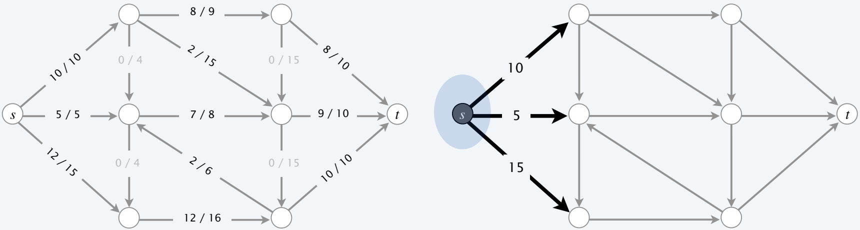
Relationship between flows and cuts

Weak duality. Let f be any flow and (A, B) be any cut. Then, $val(f) \le cap(A, B)$.

Pf.

$$
\begin{aligned}
val(f) &= \sum_{e \text{ out of } A} f(e) \;-\; \sum_{e \text{ in to } A} f(e) \\
&\le \sum_{e \text{ out of } A} f(e) \\
&\le \sum_{e \text{ out of } A} c(e) \\
&= cap(A, B) \quad \blacksquare
\end{aligned}
$$

flow value lemma



value of flow = 27 ≤ **capacity of cut = 30**

Certificate of optimality

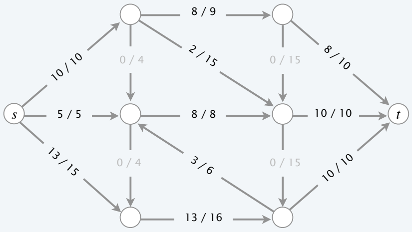
Corollary. Let f be a flow and let (A, B) be any cut.
If $val(f) = cap(A, B)$, then f is a max flow and (A, B) is a min cut.

Pf.

weak duality

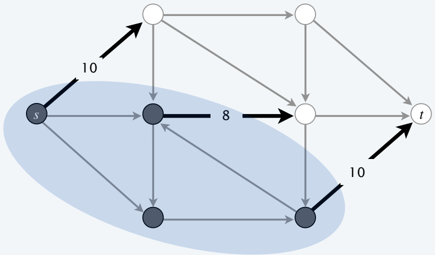
- For any flow f': $val(f') \leq cap(A, B) = val(f)$.
- For any cut (A', B'): $cap(A', B') \geq val(f) = cap(A, B)$. ▪

weak duality



value of flow = 28

=



capacity of cut = 28

Max-flow min-cut theorem

Max-flow min-cut theorem. Value of a max flow = capacity of a min cut.

strong duality

MAXIMAL FLOW THROUGH A NETWORK

L. R. FORD, JR. AND D. R. FULKERSON

Introduction. The problem discussed in this paper was formulated by T. Harris as follows:

"Consider a rail network connecting two cities by way of a number of intermediate cities, where each link of the network has a number assigned to it representing its capacity. Assuming a steady state condition, find a maximal flow from one given city to the other."

ON THE MAX FLOW MIN CUT THEOREM OF NETWORKS

G. B. Dantzig
D. R. Fulkerson

P-826

April 15, 1955

A Note on the Maximum Flow Through a Network*

P. ELIAS†, A. FEINSTEIN‡, AND C. E. SHANNON§

Summary—This note discusses the problem of maximizing the rate of flow from one terminal to another, through a network which consists of a number of branches, each of which has a limited capacity. The main result is a theorem: The maximum possible flow from left to right through a network is equal to the minimum value among all simple cut-sets. This theorem is applied to solve a more general problem, in which a number of input nodes and a number of output nodes are used.

from one terminal to the other in the original network passes through at least one branch in the cut-set. In the network above, some examples of cut-sets are (d, e, f), and (b, c, e, g, h), (d, g, h, i). By a *simple cut-set* we will mean a cut-set such that if any branch is omitted it is no longer a cut-set. Thus (d, e, f) and (b, c, e, g, h) are simple cut-sets while (d, g, h, i) is not. When a simple cut-set is

Max-flow min-cut theorem

Max-flow min-cut theorem. Value of a max flow = capacity of a min cut.

Augmenting path theorem. A flow f is a max flow iff no augmenting paths.

Pf. The following three conditions are equivalent for any flow f:

i. There exists a cut (A, B) such that $cap(A, B) = val(f)$.

ii. f is a max flow.

iii. There is no augmenting path with respect to f. ← if Ford–Fulkerson terminates, then f is max flow

[i ⇒ ii]

- This is the weak duality corollary. ▪

Max-flow min-cut theorem

Max-flow min-cut theorem. Value of a max flow = capacity of a min cut.

Augmenting path theorem. A flow f is a max flow iff no augmenting paths.

Pf. The following three conditions are equivalent for any flow f:

i. There exists a cut (A, B) such that $cap(A, B) = val(f)$.

ii. f is a max flow.

iii. There is no augmenting path with respect to f.

[ii ⇒ iii] We prove contrapositive: ¬ iii ⇒ ¬ ii.

- Suppose that there is an augmenting path with respect to f.
- Can improve flow f by sending flow along this path.
- Thus, f is not a max flow. ▪

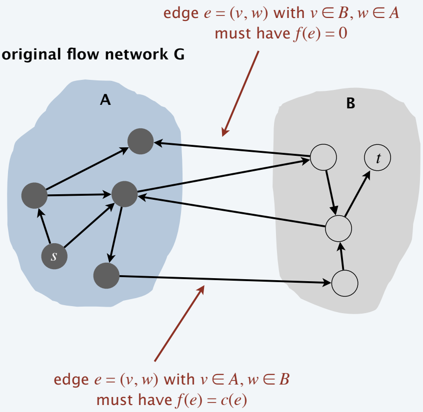
Max-flow min-cut theorem

[iii ⇒ i]

- Let f be a flow with no augmenting paths.
- Let A be set of nodes reachable from s in residual network G_f.
- By definition of A: $s \in A$.
- By definition of flow f: $t \notin A$.

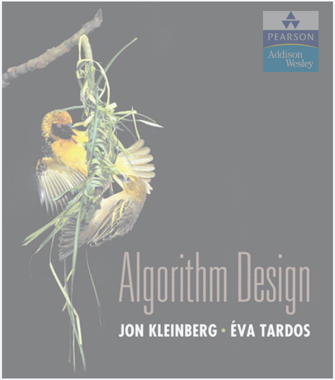
$$\begin{aligned} val(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &= \sum_{e \text{ out of } A} c(e) - 0 \\ &= cap(A, B) \quad \blacksquare \end{aligned}$$

flow value lemma



THE FORD-FULKERSON METHOD

CAPACITY-SCALING ALGORITHM



SECTION 7.3

7. NETWORK FLOW I

- max-flow and min-cut problems
- Ford–Fulkerson algorithm
- max-flow min-cut theorem
- capacity-scaling algorithm
- shortest augmenting paths
- Dinitz' algorithm
- simple unit-capacity networks

Analysis of Ford–Fulkerson algorithm (when capacities are integral)

Assumption. Every edge capacity $c(e)$ is an integer between 1 and C.

Integrality invariant. Throughout Ford–Fulkerson, every edge flow $f(e)$ and residual capacity $c_f(e)$ is an integer.

Pf. By induction on the number of augmenting paths. ▪

consider cut $A = \{ s \}$
(assumes no parallel edges)

Theorem. Ford–Fulkerson terminates after at most $val(f^*) \le nC$ augmenting paths, where f^* is a max flow.

Pf. Each augmentation increases the value of the flow by at least 1. ▪

Corollary. The running time of Ford–Fulkerson is $O(m n C)$.

Pf. Can use either BFS or DFS to find an augmenting path in $O(m)$ time. ▪

$f(e)$ is an integer for every e

Integrality theorem. There exists an integral max flow f^*.

Pf. Since Ford–Fulkerson terminates, theorem follows from integrality invariant (and augmenting path theorem). ▪

Ford–Fulkerson: exponential example

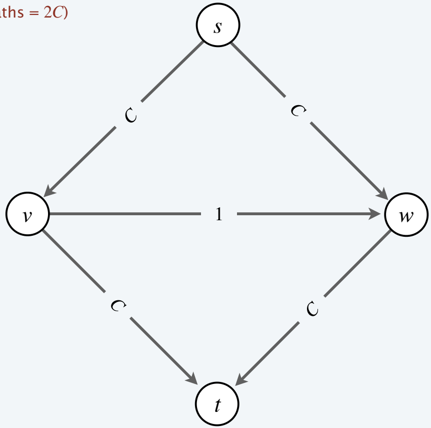
Q. Is generic Ford–Fulkerson algorithm poly-time in input size?

m, n, and $\log C$

A. No. If max capacity is C, then algorithm can take $\geq C$ iterations.

- $s \to v \to w \to t$
- $s \to w \to v \to t$
- $s \to v \to w \to t$
- $s \to w \to v \to t$
- …
- $s \to v \to w \to t$
- $s \to w \to v \to t$

each augmenting path sends only 1 unit of flow (# augmenting paths = $2C$)



The Ford–Fulkerson algorithm is guaranteed to terminate if the edge capacities are …

A. Rational numbers.

B. Real numbers.

C. Both A and B.

D. Neither A nor B.

Choosing good augmenting paths

Use care when selecting augmenting paths.

- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.

Pathology. When edge capacities can be irrational, no guarantee that Ford–Fulkerson terminates (or converges to a maximum flow)!

Goal. Choose augmenting paths so that:

- Can find augmenting paths efficiently.
- Few iterations.

Choosing good augmenting paths

Choose augmenting paths with:

- Max bottleneck capacity ("fattest"). ⟵ how to find?
- Sufficiently large bottleneck capacity. ⟵ next
- Fewest edges. ⟵ ahead

Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems

JACK EDMONDS

University of Waterloo, Waterloo, Ontario, Canada

AND

RICHARD M. KARP

University of California, Berkeley, California

ABSTRACT. This paper presents new algorithms for the maximum flow problem, the Hitchcock transportation problem, and the general minimum-cost flow problem. Upper bounds on the numbers of steps in these algorithms are derived, and are shown to compare favorably with upper bounds on the numbers of steps required by earlier algorithms.

Edmonds–Karp 1972 (USA)

Dokl. Akad. Nauk SSSR
Tom 194 (1970), No. 4

Soviet Math. Dokl.
Vol. 11 (1970), No. 5

ALGORITHM FOR SOLUTION OF A PROBLEM OF MAXIMUM FLOW IN A NETWORK WITH POWER ESTIMATION

UDC 518.5

E. A. DINIC

Different variants of the formulation of the problem of maximal stationary flow in a network and its many applications are given in [1]. There also is given an algorithm solving the problem in the case where the initial data are integers (or, what is equivalent, commensurable). In the general case this algorithm requires preliminary rounding off of the initial data, i.e. only an approximate solution of the problem is possible. In this connection the rapidity of convergence of the algorithm is inversely proportional to the relative precision.

Dinitz 1970 (Soviet Union)

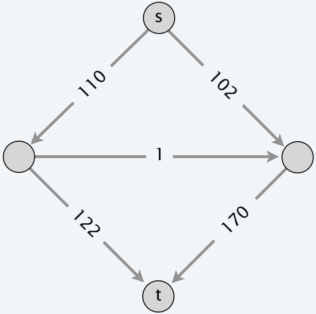
invented in response to a class exercises by Adel'son-Vel'skiĭ

Capacity-scaling algorithm

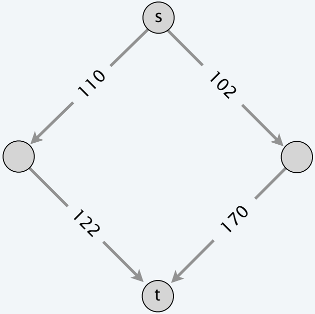
Overview. Choosing augmenting paths with "large" bottleneck capacity.

- Maintain scaling parameter Δ.
- Let $G_f(\Delta)$ be the part of the residual network containing only those edges with capacity $\geq \Delta$.
- Any augmenting path in $G_f(\Delta)$ has bottleneck capacity $\geq \Delta$.

though not necessarily largest



G_f



$G_f(\Delta)$, $\Delta = 100$

Capacity-scaling algorithm

CAPACITY-SCALING(G)

FOREACH edge $e \in E$: $f(e) \leftarrow 0$.

$\Delta \leftarrow$ largest power of $2 \le C$.

WHILE $(\Delta \ge 1)$

  $G_f(\Delta) \leftarrow \Delta$-residual network of G with respect to flow f.

  WHILE (there exists an $s \rightsquigarrow t$ path P in $G_f(\Delta)$)

    $f \leftarrow$ AUGMENT(f, c, P).

    Update $G_f(\Delta)$.

  $\Delta \leftarrow \Delta / 2$.

Δ-scaling phase

RETURN f.

Capacity-scaling algorithm: proof of correctness

Assumption. All edge capacities are integers between 1 and C.

Invariant. The scaling parameter Δ is a power of 2.
Pf. Initially a power of 2; each phase divides Δ by exactly 2. ▪

Integrality invariant. Throughout the algorithm, every edge flow $f(e)$ and residual capacity $c_f(e)$ is an integer.
Pf. Same as for generic Ford–Fulkerson. ▪

Theorem. If capacity-scaling algorithm terminates, then f is a max flow.
Pf.

- By integrality invariant, when $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$.
- Upon termination of $\Delta = 1$ phase, there are no augmenting paths.
- Result follows augmenting path theorem ▪

Capacity-scaling algorithm: analysis of running time

Lemma 1. There are $1 + \lfloor \log_2 C \rfloor$ scaling phases.

Pf. Initially $C/2 < \Delta \le C$; Δ decreases by a factor of 2 in each iteration. ▪

Lemma 2. Let f be the flow at the end of a Δ-scaling phase.
Then, the max-flow value $\le val(f) + m\,\Delta$.

Pf. Next slide.

Lemma 3. There are $\le 2m$ augmentations per scaling phase.

Pf.

- Let f be the flow at the beginning of a Δ-scaling phase.
- Lemma 2 $\Rightarrow$ max-flow value $\le val(f) + m\,(2\,\Delta)$.
- Each augmentation in a Δ-phase increases $val(f)$ by at least Δ. ▪

or equivalently, at the end of a 2Δ-scaling phase

Theorem. The capacity-scaling algorithm takes $O(m^2 \log C)$ time.

Pf.

- Lemma 1 + Lemma 3 $\Rightarrow$ $O(m \log C)$ augmentations.
- Finding an augmenting path takes $O(m)$ time. ▪

Capacity-scaling algorithm: analysis of running time

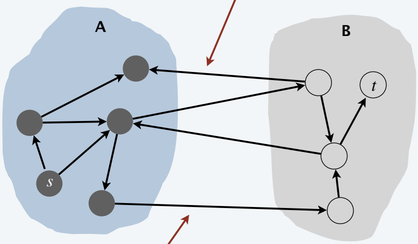
Lemma 2. Let f be the flow at the end of a Δ-scaling phase.
Then, the max-flow value $\le val(f) + m\,\Delta$.

Pf.

- We show there exists a cut (A, B) such that $cap(A, B) \le val(f) + m\,\Delta$.
- Choose A to be the set of nodes reachable from s in $G_f(\Delta)$.
- By definition of A: $s \in A$.
- By definition of flow f: $t \notin A$.

edge $e = (v, w)$ with $v \in B, w \in A$
must have $f(e) < \Delta$

original flow network



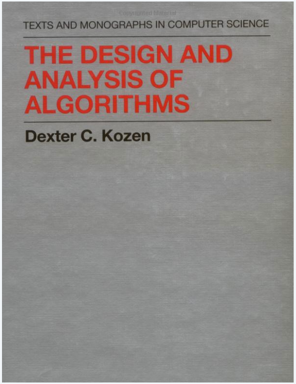
edge $e = (v, w)$ with $v \in A, w \in B$
must have $f(e) > c(e) - \Delta$

$$
\begin{aligned}
val(f) &= \sum_{e \text{ out of } A} f(e) \;-\; \sum_{e \text{ in to } A} f(e) \\
&\ge \sum_{e \text{ out of } A} (c(e) - \Delta) \;-\; \sum_{e \text{ in to } A} \Delta \\
&\ge \sum_{e \text{ out of } A} c(e) \;-\; \sum_{e \text{ out of } A} \Delta \;-\; \sum_{e \text{ in to } A} \Delta \\
&\ge cap(A, B) \;-\; m\Delta \quad \blacksquare
\end{aligned}
$$

flow value lemma

THE FORD-FULKERSON METHOD

SHORTEST AUGMENTING PATH



SECTION 17.2

7. NETWORK FLOW I

- *max-flow and min-cut problems*
- *Ford–Fulkerson algorithm*
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *Dinitz' algorithm*
- *simple unit-capacity networks*

Shortest augmenting path

Q. How to choose next augmenting path in Ford–Fulkerson?

A. Pick one that uses the fewest edges.

can find via BFS

SHORTEST-AUGMENTING-PATH(G)

FOREACH $e \in E$: $f(e) \leftarrow 0$.

$G_f \leftarrow$ residual network of G with respect to flow f.

WHILE (there exists an $s \rightarrow t$ path in G_f)

$P \leftarrow$ BREADTH-FIRST-SEARCH(G_f).

$f \leftarrow$ AUGMENT(f, c, P).

Update G_f.

RETURN f.

Shortest augmenting path: overview of analysis

Lemma 1. The length of a shortest augmenting path never decreases.
Pf. Ahead.

number of edges

Lemma 2. After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.
Pf. Ahead.

Theorem. The shortest-augmenting-path algorithm takes $O(m^2 n)$ time.
Pf.

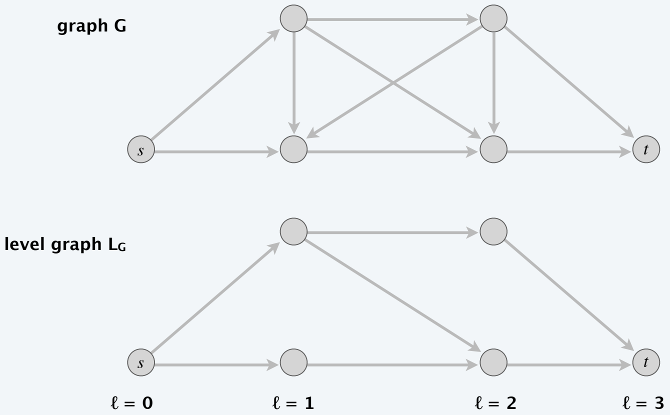
- $O(m)$ time to find a shortest augmenting path via BFS.
- There are $\le m n$ augmentations.
  - at most m augmenting paths of length k ← Lemma 1 + Lemma 2
  - at most $n-1$ different lengths ▪

augmenting paths are simple paths

Shortest augmenting path: analysis

Def. Given a digraph $G = (V, E)$ with source s, its level graph is defined by:

- $\ell(v)$ = number of edges in shortest $s \rightsquigarrow v$ path.
- $L_G = (V, E_G)$ is the subgraph of G that contains only those edges $(v, w) \in E$ with $\ell(w) = \ell(v) + 1$.



Network flow: quiz 5

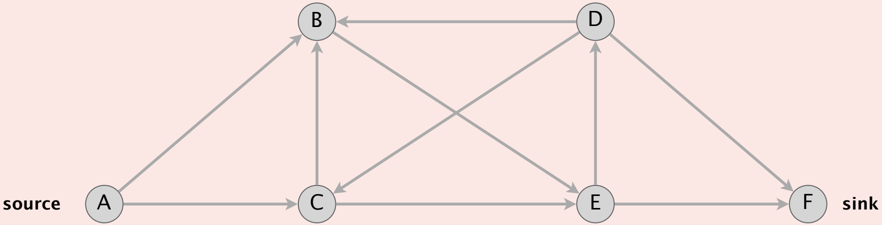
Which edges are in the level graph of the following digraph?

A. D→F.

B. E→F.

C. Both A and B.

D. Neither A nor B.

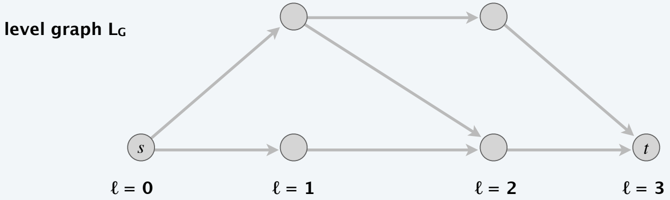


Shortest augmenting path: analysis

Def. Given a digraph $G = (V, E)$ with source s, its level graph is defined by:

- $\ell(v)$ = number of edges in shortest $s \rightsquigarrow v$ path.
- $L_G = (V, E_G)$ is the subgraph of G that contains only those edges $(v, w) \in E$ with $\ell(w) = \ell(v) + 1$.

Key property. P is a shortest $s \rightsquigarrow v$ path in G iff P is an $s \rightsquigarrow v$ path in L_G.

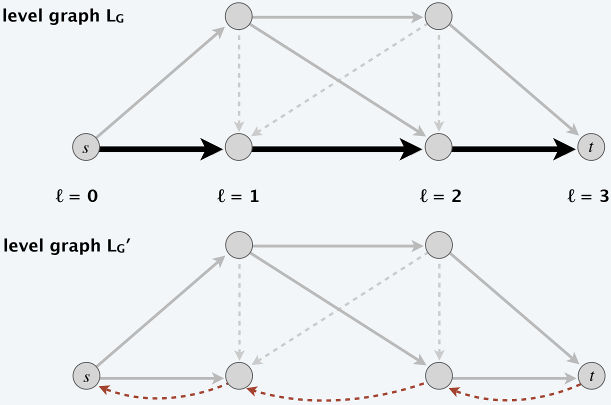


Shortest augmenting path: analysis

Lemma 1. The length of a shortest augmenting path never decreases.

- Let f and f' be flow before and after a shortest-path augmentation.
- Let L_G and $L_{G'}$ be level graphs of G_f and $G_{f'}$.
- Only back edges added to $G_{f'}$

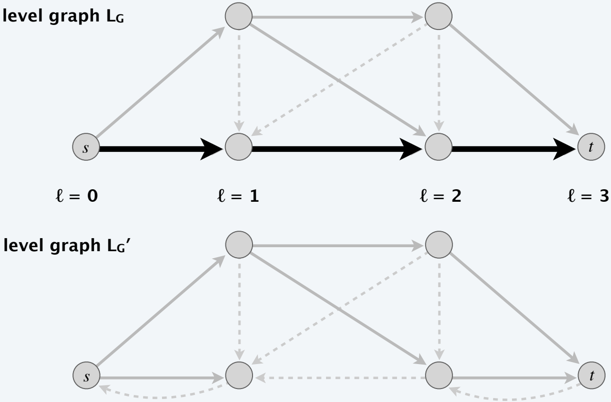
  (any $s \rightarrow t$ path that uses a back edge is longer than previous length) ▪



Shortest augmenting path: analysis

Lemma 2. After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

- At least one (bottleneck) edge is deleted from L_G per augmentation.
- No new edge added to L_G until shortest path length strictly increases. ▪



Shortest augmenting path: review of analysis

Lemma 1. Throughout the algorithm, the length of a shortest augmenting path never decreases.

Lemma 2. After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Theorem. The shortest-augmenting-path algorithm takes $O(m^2 n)$ time.

Shortest augmenting path: improving the running time

Note. $\Theta(m\,n)$ augmentations necessary for some flow networks.

- Try to decrease time per augmentation instead.
- Simple idea $\Rightarrow$ $O(mn^2)$ [Dinitz 1970] ⟵ ahead
- Dynamic trees $\Rightarrow$ $O(m\,n \log n)$ [Sleator–Tarjan 1983]

A Data Structure for Dynamic Trees

DANIEL D. SLEATOR AND ROBERT ENDRE TARJAN

Bell Laboratories, Murray Hill, New Jersey 07974

Received May 8, 1982; revised October 18, 1982

A data structure is proposed to maintain a collection of vertex-disjoint trees under a sequence of two kinds of operations: a *link* operation that combines two trees into one by adding an edge, and a *cut* operation that divides one tree into two by deleting an edge. Each operation requires $O(\log n)$ time. Using this data structure, new fast algorithms are obtained for the following problems:

(1) Computing nearest common ancestors.

(2) Solving various network flow problems including finding maximum flows, blocking flows, and acyclic flows.

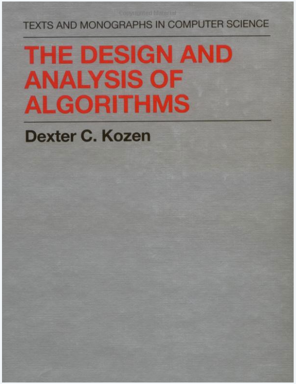
(3) Computing certain kinds of constrained minimum spanning trees.

(4) Implementing the network simplex algorithm for minimum-cost flows.

The most significant application is (2); an $O(mn \log n)$-time algorithm is obtained to find a maximum flow in a network of n vertices and m edges, beating by a factor of $\log n$ the fastest algorithm previously known for sparse graphs.

THE FORD-FULKERSON METHOD

DINITZ' ALGORITHM



SECTION 18.1

7. NETWORK FLOW I

- *max-flow and min-cut problems*
- *Ford–Fulkerson algorithm*
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *Dinitz′ algorithm*
- *simple unit-capacity networks*

Dinitz' algorithm

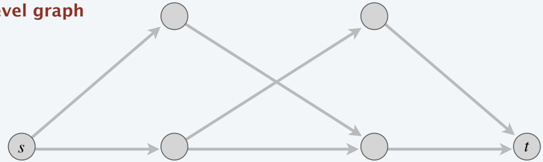
Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations. ← within a phase, length of shortest augmenting path does not change

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

construct level graph



level graph L_G

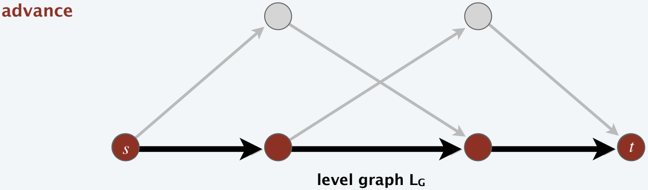
Dinitz' algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



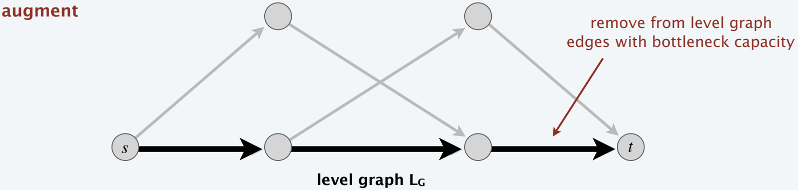
Dinitz' algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



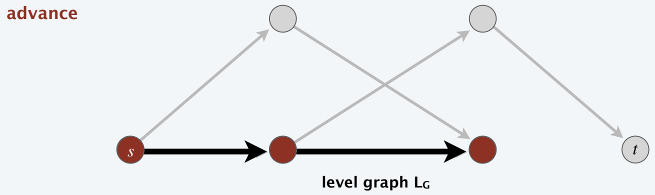
Dinitz' algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



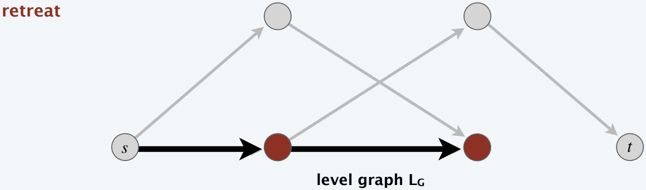
Dinitz' algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



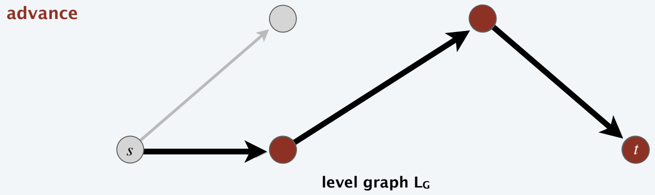
Dinitz' algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



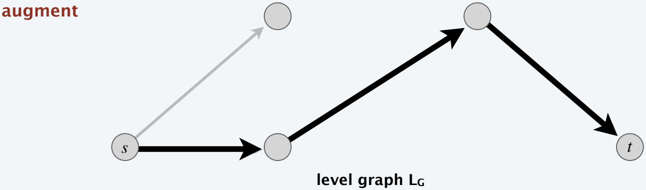
Dinitz' algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



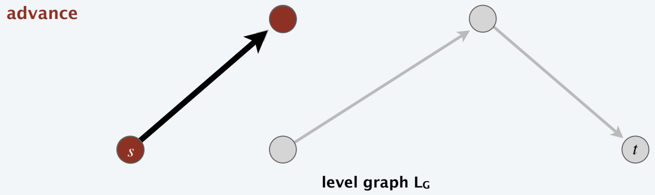
Dinitz' algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



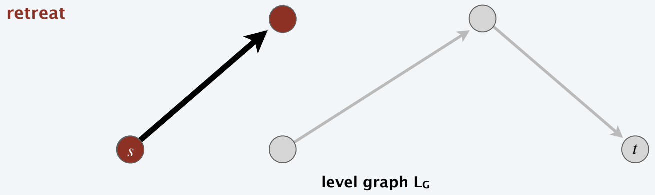
Dinitz' algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



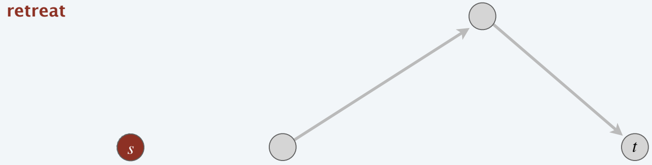
Dinitz' algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



level graph L_G

Dinitz' algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

end of phase

level graph L_G

Dinitz' algorithm (as refined by Even and Itai)

INITIALIZE(G, f)

$L_G \leftarrow$ level-graph of G_f.

$P \leftarrow \varnothing$.

GOTO ADVANCE(s).

RETREAT(v)

IF $(v = s)$

  STOP.

ELSE

  Delete v (and all incident edges) from L_G.

  Remove last edge (u, v) from P.

  GOTO ADVANCE(u).

ADVANCE(v)

IF $(v = t)$

  AUGMENT(P).

  Remove saturated edges from L_G.

  $P \leftarrow \varnothing$.

  GOTO ADVANCE(s).

IF (there exists edge $(v, w) \in L_G$)

  Add edge (v, w) to P.

  GOTO ADVANCE(w).

ELSE

  GOTO RETREAT(v).

Network flow: quiz 6

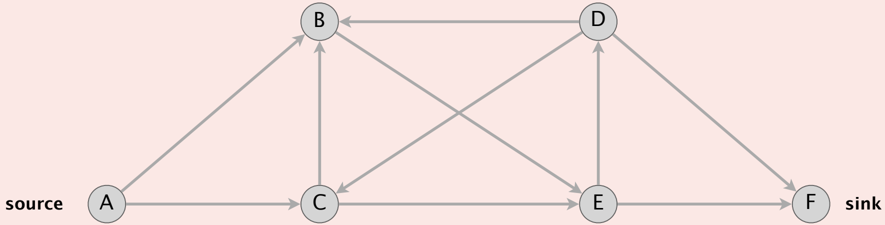
How to compute the level graph L_G efficiently?

A. Depth-first search.

B. Breadth-first search.

C. Both A and B.

D. Neither A nor B.



Dinitz' algorithm: analysis

Lemma. A phase can be implemented to run in $O(mn)$ time.

Pf.

- Initialization happens once per phase. ⟵ $O(m)$ using BFS
- At most m augmentations per phase. ⟵ $O(mn)$ per phase
  (because an augmentation deletes at least one edge from L_G)
- At most n retreats per phase. ⟵ $O(m + n)$ per phase
  (because a retreat deletes one node from L_G)
- At most mn advances per phase. ⟵ $O(mn)$ per phase
  (because at most n advances before retreat or augmentation) ▪

Theorem. [Dinitz 1970] Dinitz' algorithm runs in $O(mn^2)$ time.

Pf.

- By Lemma, $O(mn)$ time per phase.
- At most $n-1$ phases (as in shortest-augmenting-path analysis). ▪

THE FORD-FULKERSON METHOD

SUMMARY

Augmenting-path algorithms: summary

year	method	# augmentations	running time	
1955	**augmenting path**	$n\,C$	$O(m\,n\,C)$	
1972	**fattest path**	$m \log(mC)$	$O(m^2 \log n \log(mC))$	fat paths
1972	**capacity scaling**	$m \log C$	$O(m^2 \log C)$	fat paths
1985	**improved capacity scaling**	$m \log C$	$O(m\,n \log C)$	fat paths
1970	**shortest augmenting path**	$m\,n$	$O(m^2\,n)$	shortest paths
1970	**level graph**	$m\,n$	$O(m\,n^2)$	shortest paths
1983	**dynamic trees**	$m\,n$	$O(m\,n \log n)$	shortest paths

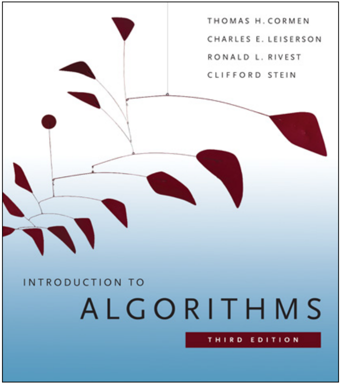
augmenting-path algorithms with m edges, n nodes, and integer capacities between 1 and C

Maximum-flow algorithms: theory highlights

year	method	worst case	discovered by
1951	**simplex**	$O(m\, n^2\, C)$	Dantzig
1955	**augmenting paths**	$O(m\, n\, C)$	Ford–Fulkerson
1970	**shortest augmenting paths**	$O(m\, n^2)$	Edmonds–Karp, Dinitz
1974	**blocking flows**	$O(n^3)$	Karzanov
1983	**dynamic trees**	$O(m\, n \log n)$	Sleator–Tarjan
1985	**improved capacity scaling**	$O(m\, n \log C)$	Gabow
1988	**push-relabel**	$O(m\, n \log (n^2 / m))$	Goldberg–Tarjan
1998	**binary blocking flows**	$O(m^{3/2} \log (n^2 / m) \log C)$	Goldberg–Rao
2013	**compact networks**	$O(m\, n)$	Orlin
2014	**interior-point methods**	$\tilde{O}(m\, m^{1/2} \log C)$	Lee–Sidford
2016	**electrical flows**	$\tilde{O}(m^{10/7}\, C^{1/7})$	Mądry
20xx		???	

max-flow algorithms with m edges, n nodes, and integer capacities between 1 and C

THE PUSH-RELABEL METHOD



kapitola 26.4.

Ford-Fulkerson method vs **Goldberg method**

aka augmenting path method vs push relabel method

- global vs local character
- update flow along an augmenting path vs update flow on edges
- flow conservation vs preflow

pre-flow is a function f with

capacity condition for $e \in E$: $0 \leq f(e) \leq c(e)$

relaxed flow conservation for $v \in V \setminus \{s,t\}$:

$$\sum_{e \text{ into } v} f(e) \geq \sum_{e \text{ out of } v} f(e)$$

overflowing vertex

vertex $v \in V \setminus \{s,t\}$ with $\displaystyle\sum_{e \text{ into } v} f(e) > \sum_{e \text{ out of } v} f(e)$

excess flow into vertex v

the quantity $\displaystyle e_f(v) = \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e)$

a pre-flow becomes a flow if no intermediate node has an excess height function is a function $h : V \to \mathbb{N}_0$

height function h is compatible with preflow f iff

source $h(s) = |V| = n$

sink $h(t) = 0$

height difference $h(v) \leq h(w) + 1$ for every edge (v, w) of the residual network G_f

if $h(v) > h(w) + 1$ then (v, w) is not an edge in the residual network G_f

Lema

If f is a preflow and h is an height function compatible with f then there is no path from the source s to the sink t in the residual network G_f.

- assume that G_f contains a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ with $v_0 = s$ and $v_k = t$
- w.l.o.g. p is simple and thus $k < n$
- because h is a height function $h(v_i) \leq h(v_{i+1}) + 1$ for $i = 0, 1, \ldots, k-1$
- combining inequalities over p yields $h(s) \leq h(t) + k$
- because $h(t) = 0$, we have $h(s) \leq k < n$, which contradits the requirement $h(s) = n$

Lema

If f is a preflow and h is an height function compatible with f then there is no path from the source s to the sink t in the residual network G_f.

- assume that G_f contains a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ with $v_0 = s$ and $v_k = t$
- w.l.o.g. p is simple and thus $k < n$
- because h is a height function $h(v_i) \leq h(v_{i+1}) + 1$ for $i = 0, 1, \ldots, k-1$
- combining inequalities over p yields $h(s) \leq h(t) + k$
- because $h(t) = 0$, we have $h(s) \leq k < n$, which contradits the requirement $h(s) = n$

Lema

If f is a **flow** and h is an height function compatible with f then f is a maximal flow.

initialization — height function

- $h(v) = 0$ for alle $v \in V$, $v \neq s$
- $h(s) = n$

initialization — preflow

- $f(s, v) = c(s, v)$ for each $(s, v) \in E$
- $f(u, v) = 0$ for all other edges

initial preflow and height function are compatible

Algorithm: Generic-Push-Relabel

Input: flow network $G = (V, E, s, t, c)$

Output: maximal flow f

Initialize-PreFlow
while *true* **do**
    if no node is overflowing **then return** f
    select an overflowing vertex v
    if v has a neigbor w in G_f such that $h(v) > h(w)$ **then**
        Push(f, h, v, w)
    else
        Relabel(f, h, v)

Algorithm: Initialize-PreFlow

for $v \in V$ **do** $h(v) \leftarrow 0; \quad e_f(v) \leftarrow 0$

$h(s) \leftarrow n$

for $e \in E$ **do** $f(e) \leftarrow 0$

for $(s, v) \in E$ **do**

  $f(s,v) \leftarrow c(s,v); \quad e_f(v) \leftarrow c(s,v); \quad e_f(s) \leftarrow e_f(s) - c(s,v)$

Push applies when v si overflowing, $c_f(v, w) > 0$, and $h(w) < h(v)$

Function $\textsc{Push}(f, h, v, w)$

$\Delta_f(v, w) \leftarrow \min(e_f(v), c_f(v, w))$
if $(v, w) \in E$ **then**
  $f(v, w) \leftarrow f(v, w) + \Delta_f(v, w)$
else
  $f(w, v) \leftarrow f(w, v) - \Delta_f(v, w)$
$e_f(v) \leftarrow e_f(v) - \Delta_f(v, w)$
$e_f(w) \leftarrow e_f(w) + \Delta_f(v, w)$
return f, h

---

we can change (i.e. increase or decrease) flow from v to w by $\Delta_f(v, w)$ without causing $e_f(v)$ to become negative or the capacity $c(v, w)$ to be exceeded

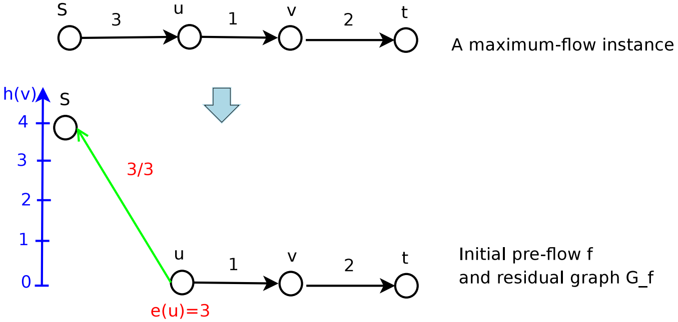
Relabel applies when v si overflowing and for all $w \in V$ such that $(v, w) \in E_f$ we have $h(v) \leq h(w)$

Function $\text{Relabel}(f, h, v)$

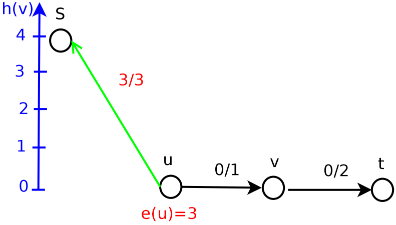
1 $h(v) \leftarrow 1 + \min\{h(w) | (v, w) \in E_f\}$ **return** h

when v is relabeled, E_f must contain at least one edge that leaves v, so that the minimization in the code is over a nonempty set

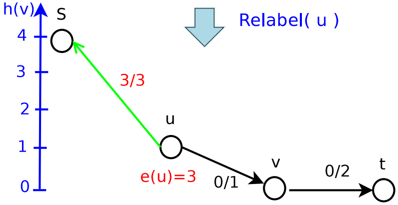
A demo of push-relabel algo: initialization



A demo of push-relabel algo: Step 1

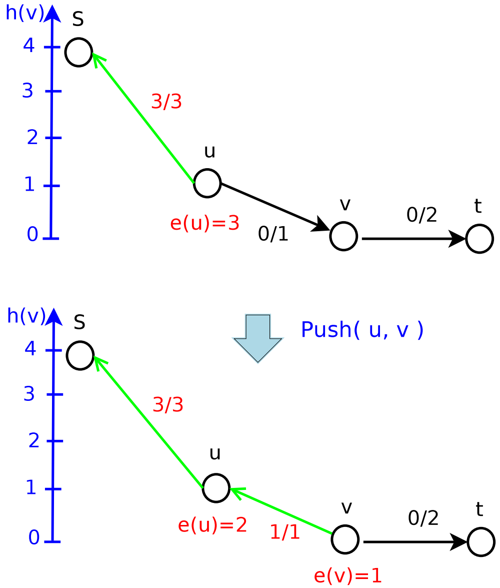


Initial pre-flow f
and residual graph G_f

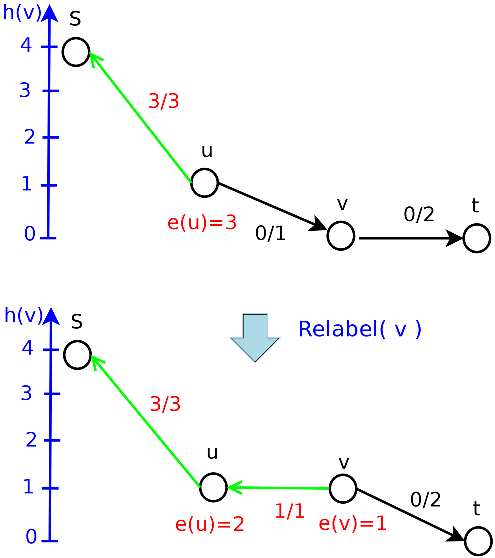


Initial pre-flow f
and residual graph G_f

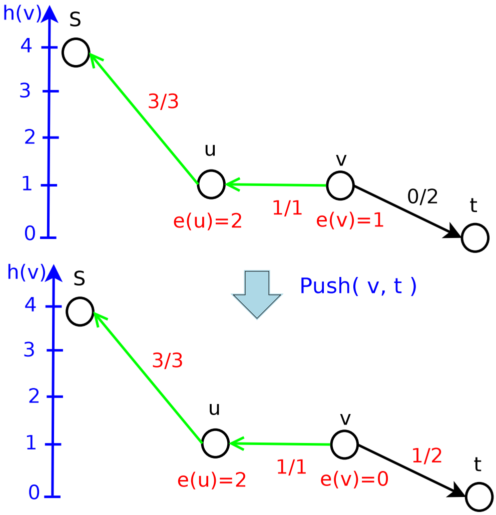
A demo of push-relabel algo: Step 2



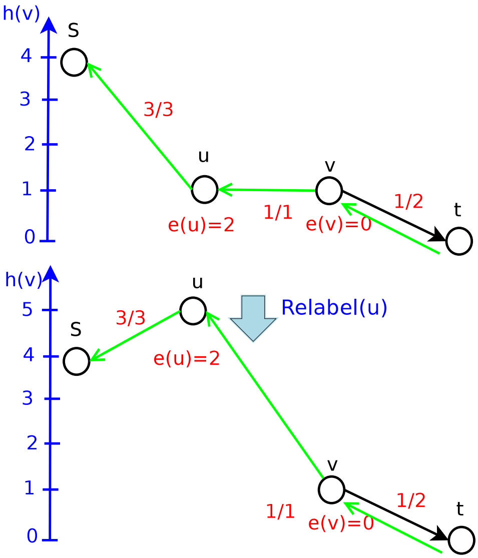
A demo of push-relabel algo: Step 3



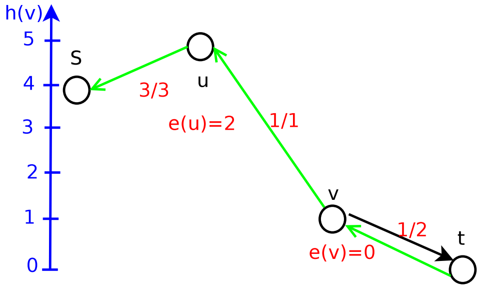
A demo of push-relabel algo: Step 4

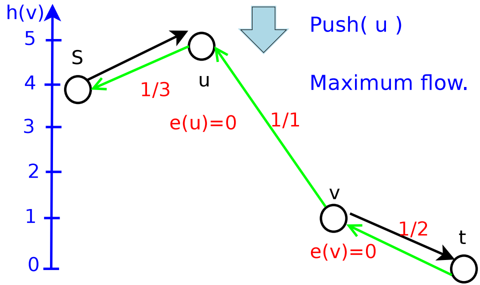


A demo of push-relabel algo: Step 5



A demo of push-relabel algo: Step 6





Push(u)

Maximum flow.

correctness

loop invariant

1. f is a preflow
2. the height function h is compatible with f

- Initialize-PreFlow makes f a preflow and h compatible with f
- Push complies with capacities of edges and if a new edge appears in the residual network, then this edge fulfills the height difference
- Relabel operation affects only height attributes and preserves compatibility

at termination, each inner vertex must have an excess 0, f is flow and is a maximum flow

complexity — bound on Relabel operations

- the initial height of all vertices (except the source s) is 0
- everyRelabel operation increases the height by 1
- we need a bound on the maximal height

Let f be a preflow. Then for every overflowing vertex v, there is a simple path from v to s in the residual network G_f.

- let $B = \{v \mid \text{there is no path from } v \text{ to } s \text{ in } G_f\}$
- let us sum up the excesses of all vertices in B,

$$\sum_{v \in B} e_f(v) = \sum_{v \in B} \Big(\sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) \Big) \geq 0$$

- edge (x, y) with $x, y \in B$ contributes to the sum $\sum_{v \in B} e_f(v)$ with zero value
- for edge (x, y) with $x \notin B$ and $y \in B$, the flow $f((x, y))$ is zero (*otherwise there would be a path from y to s in G_f*)
- edge (x, y) with $x \in B$, $y \notin B$ contributes to the sum $\sum_{v \in B} e_f(v)$ with the value $-f((x, y))$
- $\sum_{v \in B} e_f(v) = -\sum_{e \text{ out of } B} f(e) \geq 0$
- flows are nonnegative and thus $\sum_{e \text{ out of } B} f(e) = 0$ and all vertices in B have zero excess

At any time during the execution of Generic-Push-Relabel we have $h(v) \leq 2n - 1$ for all $v \in V$.

- initially, $h(s) = n$ and $h(t) = 0$ and these values never change
- when v is relabeled, it is overflowing and there is a simple path p from v to s if G_f
- there are at most $n - 1$ edges on p, every edge fulfills the height difference condition (*i.e. every edge decreases the height at most by 1*)
- $h(v) - h(s) \leq n - 1$, i.e. $h(v) \leq 2n - 1$

During the execution of Generic-Push-Relabel, the number of relabel operations is at most $2n - 1$ per vertex and at most $(2n-1)(n-2) < 2n^2$ overall.

complexity — bound on Push operations

- there are two types of Push operations
- the operation Push(f, h, v, w) is **saturating push** iff edge (v, w) in the residual network becomes saturate, i.e. $c_f(v, w) = 0$ afterward
- otherwise the operation is **nonsaturating push**

The number of **saturating** pushes is at most $2nm$.

- for any pair of vertices $v, w \in V$, we will count the saturating pushes from v to w
- if there is such a push, (v, w) is an edge of the residual network and $h(v) = h(w) + 1$
- in order for another push from v to w to occur later, $h(w)$ must increase at leat by 2
- heights start at 0 and never exceed $2n - 1$; the number of times any vertex can have its height increased by 2 is less than n
- for a network graph with m edges there can be up to $2m$ edges in the residual network, which gives the upper bound $2nm$ on the number of saturating pushes

The number of **nonsaturating** pushes is at most $4n^2(n+m)$.

- let us define a potential function as $\Phi = \sum_{v.e_f(v)>0} h(v)$
- initially, $\Phi = 0$
- **nonsaturating push decreases** Φ by at least 1
- **relabeling** a vertex v **increases** Φ by less $2n$, since the set over which the sum is taken is the same and the relabeling cannot increases v's height by more than its maximum possible height $2n-1$
- **saturating push** from v to w **increases** Φ by less than $2n$, since no heights change and only vertex w, whose height is at most $2n-1$, can possibly become overflowing
- the total amount of increase in Φ is less than $2n \cdot 2n^2 + 2n \cdot 2nm = 4n^2(n+m)$
- since $\Phi \geq 0$, the total amount of decrease, and therefore the total number of nonsaturating pushes, is less than $4n^2(n+m)$

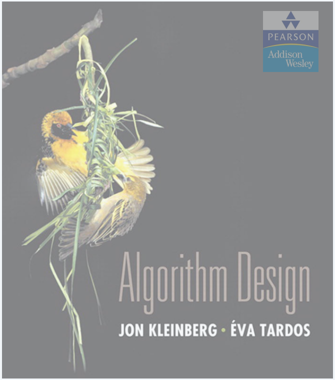
During the execution of Generic-Push-Relabel on any flow network $G = (V, E, c, s, t)$, the number of basic operations is $\mathcal{O}(V^2E)$.

- the push-relabel method allows to apply the basic operations in any order at all
- by choosing the order carefully and managing the network data structure efficiently, we can solve the maximum flow proglem faster than the $\mathcal{O}(V^2E)$ bound
- there is an implementation whose running time is $\mathcal{O}(V^3)$ which is asymptotically at least as good as $\mathcal{O}(V^2E)$, and even better for dense networks

NETWORK FLOWS — APPLICATIONS

NETWORK FLOWS — APPLICATIONS

BIPARTITE MATCHING



SECTION 7.5

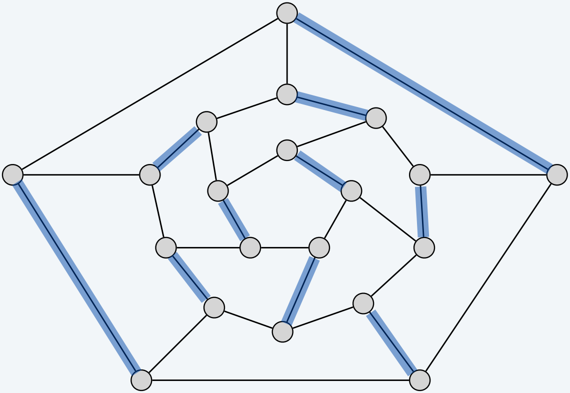
7. NETWORK FLOW II

- *bipartite matching*
- *disjoint paths*
- *extensions to max flow*
- *survey design*
- *airline scheduling*
- *image segmentation*
- *project selection*
- *baseball elimination*

Matching

Def. Given an undirected graph $G = (V, E)$, subset of edges $M \subseteq E$ is a matching if each node appears in at most one edge in M.

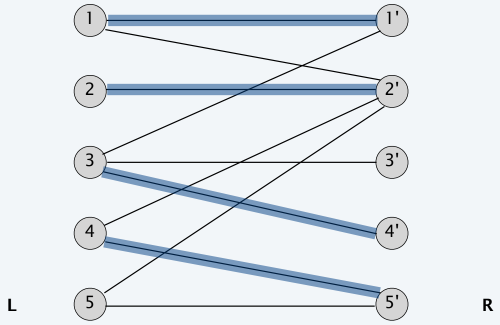
Max matching. Given a graph G, find a max-cardinality matching.



Bipartite matching

Def. A graph G is bipartite if the nodes can be partitioned into two subsets L and R such that every edge connects a node in L with a node in R.

Bipartite matching. Given a bipartite graph $G = (L \cup R, E)$, find a max-cardinality matching.

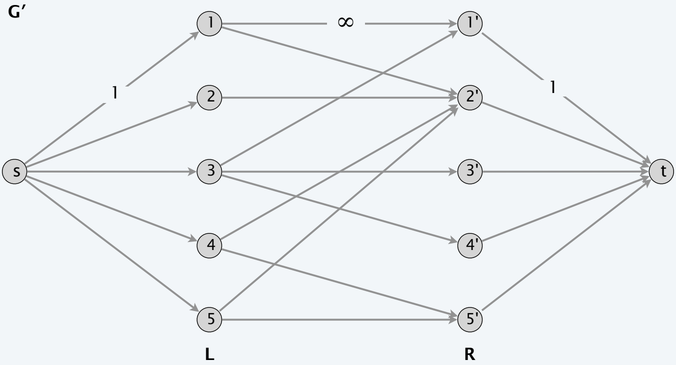


matching: 1–1', 2–2', 3–4', 4–5'

Bipartite matching: max-flow formulation

Formulation.

- Create digraph $G' = (L \cup R \cup \{s, t\},\ E'\ )$.
- Direct all edges from L to R, and assign infinite (or unit) capacity.
- Add unit-capacity edges from s to each node in L.
- Add unit-capacity edges from each node in R to t.



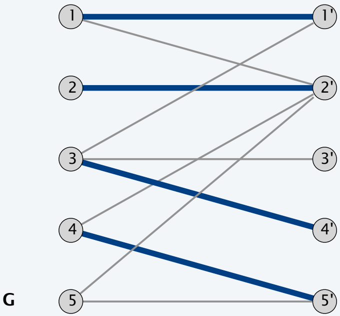
Max-flow formulation: proof of correctness

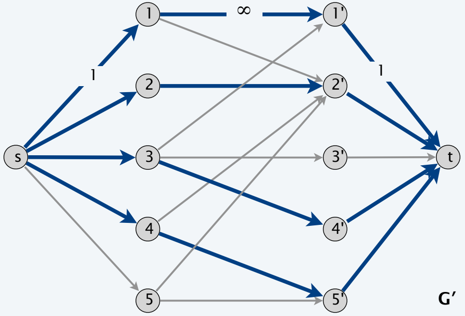
Theorem. 1–1 correspondence between matchings of cardinality k in G and integral flows of value k in G'.

for each edge e: $f(e) \in \{ 0, 1 \}$

Pf. $\Rightarrow$

- Let M be a matching in G of cardinality k.
- Consider flow f that sends 1 unit on each of the k corresponding paths.
- f is a flow of value k. ▪





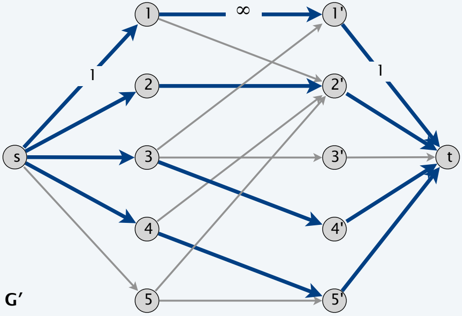
Max-flow formulation: proof of correctness

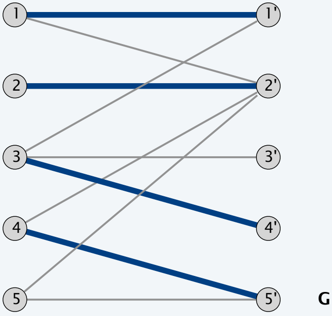
Theorem. 1–1 correspondence between matchings of cardinality k in G and integral flows of value k in G'.

for each edge e: $f(e) \in \{ 0, 1 \}$

Pf. $\Leftarrow$

- Let f be an integral flow in G' of value k.
- Consider M = set of edges from L to R with $f(e) = 1$.
  - each node in L and R participates in at most one edge in M
  - $|M| = k$: apply flow-value lemma to cut $(L \cup \{s\}, R \cup \{t\})$ ▪





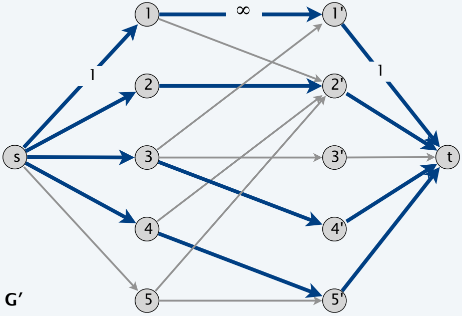
Max-flow formulation: proof of correctness

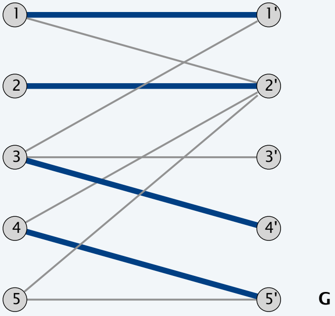
Theorem. 1–1 correspondence between matchings of cardinality k in G and integral flows of value k in G'.

Corollary. Can solve bipartite matching problem via max-flow formulation.

Pf.

- Integrality theorem $\Rightarrow$ there exists a max flow f^* in G' that is integral.
- 1–1 correspondence $\Rightarrow$ f^* corresponds to max-cardinality matching. ▪





What is running time of Ford–Fulkerson algorithms to find a max-cardinality matching in a bipartite graph with $|L| = |R| = n$?

A. $O(m + n)$

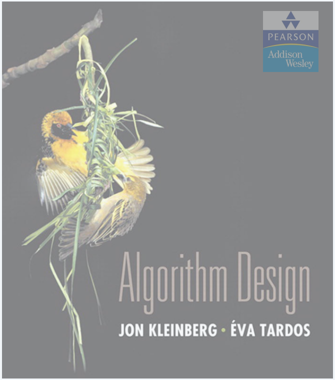
B. $O(mn)$

C. $O(mn^2)$

D. $O(m^2n)$

NETWORK FLOWS — APPLICATIONS

DISJOINT PATHS



SECTION 7.6

7. NETWORK FLOW II

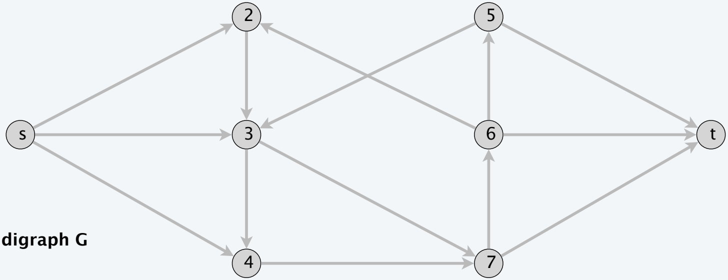
- bipartite matching
- **disjoint paths**
- extensions to max flow
- survey design
- airline scheduling
- image segmentation
- project selection
- baseball elimination

Edge-disjoint paths

Def. Two paths are edge-disjoint if they have no edge in common.

Edge-disjoint paths problem. Given a digraph $G = (V, E)$ and two nodes s and t, find the max number of edge-disjoint $s \to t$ paths.

Ex. Communication networks.



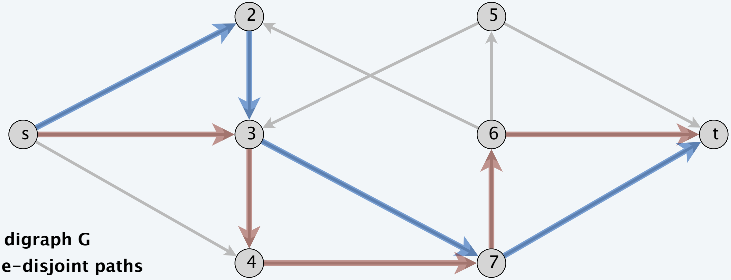
digraph G

Edge-disjoint paths

Def. Two paths are edge-disjoint if they have no edge in common.

Edge-disjoint paths problem. Given a digraph $G = (V, E)$ and two nodes s and t, find the max number of edge-disjoint $s \rightarrow t$ paths.

Ex. Communication networks.



digraph G
2 edge-disjoint paths

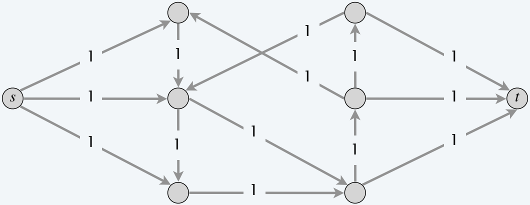
Edge-disjoint paths

Max-flow formulation. Assign unit capacity to every edge.

Theorem. 1–1 correspondence between k edge-disjoint $s \to t$ paths in G and integral flows of value k in G'.

Pf. $\Rightarrow$

- Let $P_1, \ldots, P_k$ be k edge-disjoint $s \to t$ paths in G.
- Set $f(e) = \begin{cases} 1 & \text{edge } e \text{ participates in some path } P_j \\ 0 & \text{otherwise} \end{cases}$
- Since paths are edge-disjoint, f is a flow of value k. ▪



Edge-disjoint paths

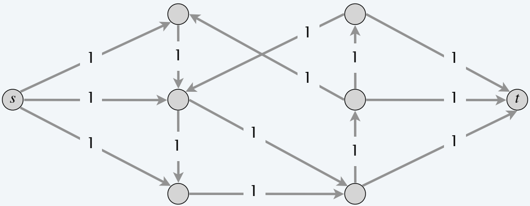
Max-flow formulation. Assign unit capacity to every edge.

Theorem. 1–1 correspondence between k edge-disjoint $s \to t$ paths in G and integral flows of value k in G'.

Pf. $\Leftarrow$

- Let f be an integral flow in G' of value k.
- Consider edge (s, u) with $f(s, u) = 1$.
  - by flow conservation, there exists an edge (u, v) with $f(u, v) = 1$
  - continue until reach t, always choosing a new edge
- Produces k (not necessarily simple) edge-disjoint paths. ▪

can eliminate cycles to get simple paths in $O(mn)$ time if desired (flow decomposition)



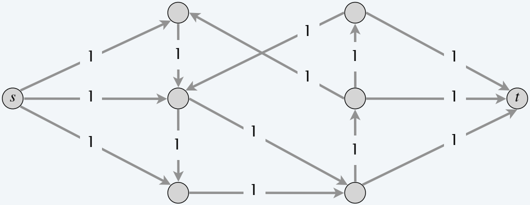
Edge-disjoint paths

Max-flow formulation. Assign unit capacity to every edge.

Theorem. 1–1 correspondence between k edge-disjoint $s{\to}t$ paths in G and integral flows of value k in G'.

Corollary. Can solve edge-disjoint paths problem via max-flow formulation.
Pf.

- Integrality theorem $\Rightarrow$ there exists a max flow f^* in G' that is integral.
- 1–1 correspondence $\Rightarrow$ f^* corresponds to max number of edge-disjoint $s{\to}t$ paths in G. ▪



NETWORK FLOWS — APPLICATIONS

MULTIPLE SOURCES AND SINKS

Which extensions to max flow can be easily modeled?

A. Multiple sources and multiple sinks.

B. Undirected graphs.

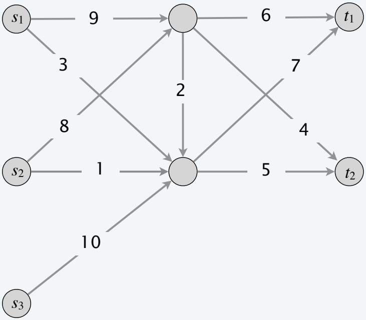
C. Lower bounds on edge flows.

D. All of the above.

Multiple sources and sinks

Def. Given a digraph $G = (V, E)$ with edge capacities $c(e) \geq 0$ and multiple source nodes and multiple sink nodes, find max flow that can be sent from the source nodes to the sink nodes.

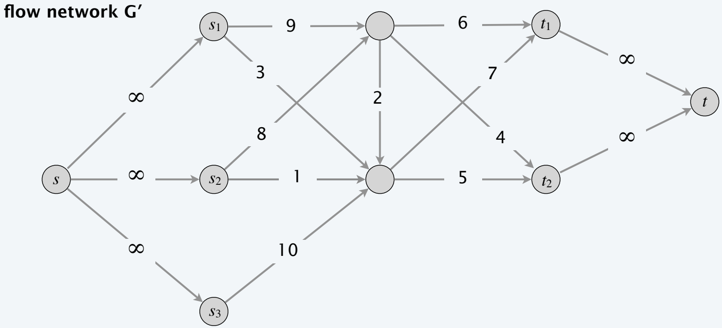
flow network G



Multiple sources and sinks: max-flow formulation

- Add a new source node s and sink node t.
- For each original source node s_i add edge (s, s_i) with capacity ∞.
- For each original sink node t_j, add edge (t_j, t) with capacity ∞.

Claim. 1–1 correspondence betweens flows in G and G'.



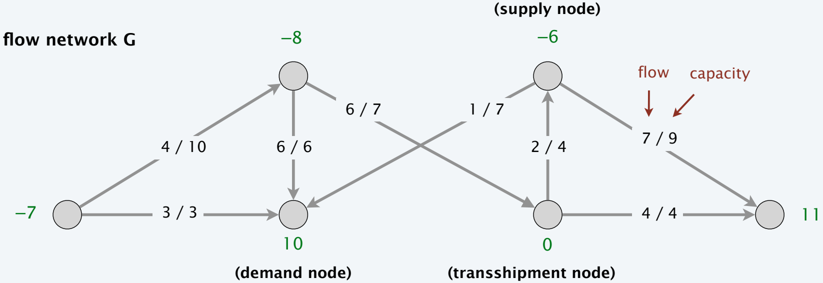
NETWORK FLOWS — APPLICATIONS

CIRCULATIONS WITH SUPPLIES AND DEMANDS

Circulation with supplies and demands

Def. Given a digraph $G = (V, E)$ with edge capacities $c(e) \geq 0$ and node demands $d(v)$, a circulation is a function $f(e)$ that satisfies:

- For each $e \in E$: $\quad 0 \leq f(e) \leq c(e)$ (capacity)
- For each $v \in V$: $\quad \sum_{e \text{ in to } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$ (flow conservation)

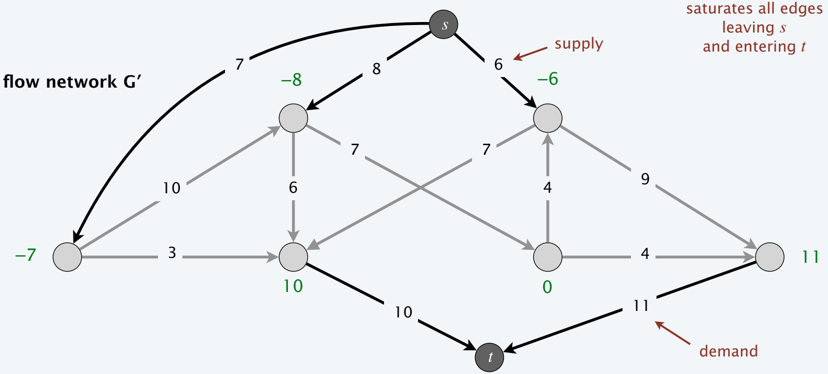


Circulation with supplies and demands: max-flow formulation

- Add new source s and sink t.
- For each v with $d(v) < 0$, add edge (s, v) with capacity $-d(v)$.
- For each v with $d(v) > 0$, add edge (v, t) with capacity $d(v)$.

Claim. G has circulation iff G' has max flow of value $D = \sum_{v\,:\,d(v)>0} d(v) = \sum_{v\,:\,d(v)<0} -d(v)$

saturates all edges leaving s and entering t



Circulation with supplies and demands

Integrality theorem. If all capacities and demands are integers, and there exists a circulation, then there exists one that is integer-valued.

Pf. Follows from max-flow formulation + integrality theorem for max flow.

Theorem. Given (V, E, c, d), there does not exist a circulation iff there exists a node partition (A, B) such that $\Sigma_{v \in B} d(v) > cap(A, B)$.

Pf sketch. Look at min cut in G'.

demand by nodes in B exceeds
supply of nodes in B plus
max capacity of edges going from A to B

Circulation with supplies, demands, and lower bounds

Def. Given a digraph $G = (V, E)$ with edge capacities $c(e) \geq 0$, lower bounds $\ell(e) \geq 0$, and node demands $d(v)$, a circulation $f(e)$ is a function that satisfies:

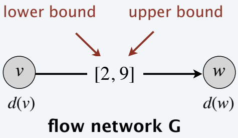
- For each $e \in E$: $\boxed{\ell(e) \leq f(e)} \leq c(e)$ (capacity)
- For each $v \in V$: $\displaystyle\sum_{e \text{ in to } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$ (flow conservation)

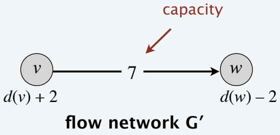
Circulation problem with lower bounds. Given (V, E, ℓ, c, d), does there exist a feasible circulation?

Circulation with supplies, demands, and lower bounds

Max-flow formulation. Model lower bounds as circulation with demands.

- Send $\ell(e)$ units of flow along edge e.
- Update demands of both endpoints.





Theorem. There exists a circulation in G iff there exists a circulation in G'. Moreover, if all demands, capacities, and lower bounds in G are integers, then there exists a circulation in G that is integer-valued.

Pf sketch. $f(e)$ is a circulation in G iff $f'(e) = f(e) - \ell(e)$ is a circulation in G'.

STRING MATCHING

STRING MATCHING

- exact string matching
- edit distance
- local and global alignment
- approximate matching
- indexing

EXACT STRING MATCHING

- strings over a finite alphabet Σ
- given two strings, a text $T[1..n]$ and a pattern $P[1..m]$, find the first substring (all substrings) of the text that is the same as the pattern

more formally

- for any shift s, let T_s denote the substring $T[s..s+m-1]$
- find the smallest shift (all shifts) s such that $T_s = P$ *(or report that there is none)*

ALGORITHMS

algorithm	*preprocessing*	*searching*
brute force	0	$\mathcal{O}((n-m+1)m)$
Karp Rabin	$\Theta(m)$	$\mathcal{O}((n-m+1)m)$
finite automata	$\mathcal{O}(m\vert\Sigma\vert)$	$\Theta(n)$
Knuth Morris Pratt	$\Theta(m)$	$\Theta(n)$
Boyer Moore	$\Theta(m+\vert\Sigma\vert)$	$\mathcal{O}((n-m+1)m)$

average complexity of algorithms Karp Rabin and Boyer Moore is much better than the given worst case complexity

BRUTE FORCE ALGORITHM

Algorithm: AlmostBruteForce($T[1..n], P[1..m]$)

```
for s ← 1 to n − m + 1 do
    equal ← True
    i ← 1
    while equal and i ≤ m do
        if T[s + i − 1] ≠ P[i] then
            equal ← False
        else
            i ← i + 1
        if equal then print s
```

time complexity of the Brute Force Algorithm

- $m - n + 1$ possible shifts
- greatest number of character comparisons possible: $n(m - n + 1)$
  P: aaaa, T: a^n
- least number of character comparisons possible: $m - n + 1$
  P: ab, T: b^n
- breaking out of the inner loop at the first mismatch makes this algorithm quite practical ...... assuming that P and T are both random (the total *expected* number of comparisons is $\mathcal{O}(n)$)

STRING MATCHING

STRINGS AS NUMBERS

STRINGS AS NUMBERS

- let $\Sigma = \{0, 1, \dots, 9\}$ *(can be any other)*
- let p be the numerical value of P , and for any shift s, let t_s be the numerical value of T_s
- $p = \sum_{i=1}^{m} 10^{m-i} P[i], \qquad t_s = \sum_{i=1}^{m} 10^{m-i} T[s+i-1]$
- find shift(s) s such that $p = t_s$

- we can compute p in $\mathcal{O}(n)$ arithmetic operations, without explicitly compute powers of ten, using Horner's scheme

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \dots + 10(P[2] + 10 \cdot P[1]) \dots))$$

- we can compute t_{s+1} in constant time (to make this we need to precompute the constant 10^{m-1})

$$t_{s+1} = 10(t_s - 10^{m-1} \cdot T[s]) + T[s+m]$$

Algorithm: NumberSearch($T[1..n], P[1..m]$)

$S \leftarrow 10^{m-1}$
$p \leftarrow 0$
$t_1 \leftarrow 0$
for $i \leftarrow 1$ **to** m **do**
  $p \leftarrow 10 \cdot p + P[i]$
  $t_1 \leftarrow 10 \cdot t_1 + T[i]$
for $s \leftarrow 1$ **to** $n - m + 1$ **do**
  if $\tilde{p} = \tilde{t}_s$ **then** print s
  $t_{s+1} \leftarrow 10 \cdot (t_s - S \cdot T[s]) + T[s + m]$

complexity: the number of arithmetic operations, acting on numbers with up to m digits, is $\mathcal{O}(n)$

KARP-RABIN FINGERPRINTING

Perform all arithmetic modulo some prime number q.

- choose q so that the value $10q$ fits into a standard integer variable
- values $(p \mod q)$ and $(t_s \mod q)$ are called the fingerprints
- we can compute $(p \mod q)$ and $(t_1 \mod q)$ in $\mathcal{O}(m)$ time

  $p \mod q =$

  $P[m] + 10(P[m-1] + \cdots + 10 \cdot P[1] \mod q) \ldots) \mod q$
- similarly $t_{s+1} \mod q$
- if $(p \mod q) \neq (t_s \mod q)$, then certainly $P \neq T_s$
- if $(p \mod q) = (t_s \mod q)$, we can't tell whether $P = T_s$ or not; we simply do a brute force comparison
- the overall running time is $\mathcal{O}(n + Fm)$, where F is the number of false matches
- the expected number of false matches is $\mathcal{O}(n/m)$

Algorithm: KarpRabin($T[1..n], P[1..m]$)

$q \leftarrow$ a random number between 2 and $\lceil m^2 \log m \rceil$

$S \leftarrow 10^{m-1}$

$\widetilde{p} \leftarrow 0$

$\widetilde{t_1} \leftarrow 0$

for $i \leftarrow 1$ **to** m **do**

- $\widetilde{p} \leftarrow (10 \cdot \widetilde{p} \mod q) + P[i] \mod q$
- $\widetilde{t_1} \leftarrow (10 \cdot \widetilde{t_1} \mod q) + T[i] \mod q$

for $s \leftarrow 1$ **to** $n - m + 1$ **do**

- **if** $\widetilde{p} = \widetilde{t_s}$ **then**
  - **if** $P = T_s$ **then** print s
- $\widetilde{t_{s+1}} \leftarrow (10 \cdot (\widetilde{t_s} - (S \cdot T[s] \mod q)) + T[s+m] \mod q$

STRING MATCHING

FINITE STATE MACHINES AND KNUTH-MORRIS-PRATT ALGORITHM

FINITE STATE MACHINES

- for a given pattern $P[1..m]$ construct a finite automaton $A = (\{0, \ldots, m\}, \Sigma, \delta, \{0\}, \{m\})$
- the transition function δ for a state q and symbol $x \in \Sigma$ is the length of the longest prefix of $P[1..m]$ that is also a suffix of $P[1..q]x$

Function $\textsc{delta}(P, \Sigma)$

for $q \leftarrow 0$ **to** m **do**
  for $x \in \Sigma$ **do**
    $k \leftarrow \min(m+1, q+2)$
    repeat $k \leftarrow k-1$ **until** $P[1 \cdots k]$ is a suffix of $P[1 \cdots q]x$
    $\delta(q, x) \leftarrow k$
return δ

complexity of the preprocessing is in $\Theta(m^3|\Sigma|)$

there is an optimalized version with complexity $\Theta(m|\Sigma|)$

Algorithm: Finite Automaton Matcher(T, A)

$q \leftarrow 0$

for $i \leftarrow 1$ **to** n **do**

  $q \leftarrow \delta(q, T[i])$

  if $q = m$ **then** **print** $i - m$

complexity of string matching is in $\Theta(n)$

can we avoid the expensive preprocessing?

REDUNDANT COMPARISONS

- character-by-character comparison
- once we have found a match for a text character, we never need to do another comparison with that text character again
- the next reasonable shift is the smallest value of s such that $T[s \ldots i-1]$, which is a suffix of the previously-read text, is also a proper prefix of the pattern
- KMP algorithm implements of both of these ideas through a special type of finite-state machines

KNUTH-MORRIS-PRATT ALGORITHM (KMP)

- every state in the string-matching machine is labeled with a character from the pattern, except two special states labeled `S` and `F`
- each state has two outgoing edges, a success edge and a failure edge
- the success edges define a path through the characters of the pattern in order, starting at `S` and ending at `F`
- failure edges always point to earlier characters in the pattern

we use the finite state machine to search for the pattern as follows

- at all times, we have a current text character $T[i]$ and a current state of the machine, which is usually labeled by some pattern character $P[j]$)
- if $T[i] = P[j]$, or the current label is `S`, follow the success edge to the next state and increment i
- if $T[i] \neq P[j]$, follow the failure edge back to an earlier state, but do not change i

KMP — implementation

in a real implementation we need only the failure function encoded in an array $fail[1..m]$

Algorithm: KnuthMorrisPratt($T[1..n], P[1..m]$)

```
ComputeFailure(P[1..m])
j ← 1
for i ← 1 to n do
    while j > 0  and  T[i] ≠ P[j] do
        j ← fail[j]
    if j = m then
        print i − m + 1
        j ← fail[j]
    j ← j + 1
```

KMP — complexity

- assume that a correct failure function is already known
- at each character comparison, either we increase i and j by one, or we decrease j and leave i alone
- we can increment i at most $n-1$ times before we run out of the text, so there are at most $n-1$ succesfull comparisons
- there can be at most $n-1$ failed comparisons, sice the number of times we decrease j cannot exceed the number of times we increment j
- in other words we can amortize character mismatches against earlier character matches
- the total number of character comparisons performed by KMP in the worst case is $\mathcal{O}(n)$

KMP — computing the failure function

$P[1..fail[j]-1]$ is
the longest proper prefix of $P[1..j-1]$ that is also a suffix of $T[1..i-1]$

- if we are comparing $T[i]$ against $P[j]$, then we must have already matched the first $j-1$ characters of the pattern
- we already know that $P[1..j-1]$ is a suffix of $T[1..i-1]$, therefore:

$P[1..fail[j]-1]$ is
the longest proper prefix of $P[1..j-1]$ that is also a suffix of $P[1..j-1]$

Algorithm: ComputeFailure($P[1..m]$)

$j \leftarrow 0$
for $i \leftarrow 1$ **to** m **do**
  $fail[i] \leftarrow j$
  while $j > 0$ and $P[i] \neq P[j]$ **do**
    $j \leftarrow fail[j]$
  $j \leftarrow j + 1$

KMP — complexity of the failure function

- just as we did for KMP, we can analyze ComputeFailure by amortizing character mismatches againgst eralier character matches
- since there are at most m character matches, ComputeFailure runs in $\mathcal{O}(m)$ time

STRING MATCHING

BOYER MOORE ALGORITHM

Can we improve on the naïve algorithm?

```
P: word
T: There would have been a time for such a word
          word
```

u doesn't occur in *P*, so skip next two alignments

```
P: word
T: There would have been a time for such a word
          word
           word   skip!
            word   skip!
             word
```

Boyer-Moore

Learn from character comparisons to skip pointless alignments

1. When we hit a mismatch, move *P* along until the mismatch becomes a match — "Bad character rule"
2. When we move *P* along, make sure characters that matched in the last alignment also match in the next alignment — "Good suffix rule"
3. Try alignments in one direction, but do character comparisons in *opposite* direction — For longer skips

```
P: word
T: There would have been a time for such a word
   ---------word ------------------------------------>
              <---
```

Boyer, RS and Moore, JS. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

Boyer-Moore: Bad character rule

Upon mismatch, skip alignments until (a) mismatch becomes a match, or (b) *P* moves past mismatched character.

(c) If there was no mismatch, don't skip

```
Step 1:  T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
         P: C C T T T T G C                                          Case (a)

Step 2:  T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
         P:       C C T T T T G C                                    Case (b)

Step 3:  T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
         P:                     C C T T T T G C                      Case (c)

Step 4:  T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
         P:                       C C T T T T G C
```

(etc)

Boyer-Moore: Bad character rule

```
Step 1:  T: GCTTCTGCTACCTTTTGCGCGCGCGCGGAA
         P: CCTTTTGC

Step 2:  T: GCTTCTGCTACCTTTTGCGCGCGCGCGGAA
         P:    CCTTTTGC

Step 3:  T: GCTTCTGCTACCTTTTGCGCGCGCGCGGAA
         P:           CCTTTTGC
```

Up to step 3, we skipped 8 alignments

5 characters in *T* were never looked at

Boyer-Moore: Good suffix rule

Let *t* = substring matched by inner loop; skip until (a) there are no mismatches between *P* and *t* *or* (b) *P* moves past *t*

```
                   t
Step 1:  T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
         P: C T T A C T T A C

                         t
Step 2:  T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
         P:         C T T A C T T A C

Step 3:  T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
         P:                 C T T A C T T A C
```

Boyer-Moore: Good suffix rule

Let *t* = substring matched by inner loop; skip until (a) there are no mismatches between *P* and *t* *or* (b) *P* moves past *t*

t

Step 1: *T:* **C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A**

P: **C T T A C T T A C**

t occurs *in its entirety* to the left within *P*

t

Step 2: *T:* **C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A**

P: **C T T A C T T A C**

prefix of *P* matches a *suffix* of *t*

Step 3: *T:* **C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A**

P: **C T T A C T T A C**

Case (a) has two subcases according to whether *t* occurs *in its entirety* to the left within *P* (as in step 1), or a *prefix* of *P* matches a *suffix* of *t* (as in step 2)

Boyer-Moore: Putting it together

How to *combine* bad character and good suffix rules?

```
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P:                     G T A G C G G C G
```

bad char says skip 2, good suffix says skip 7

Take the maximum! (7)

Boyer-Moore: Putting it together

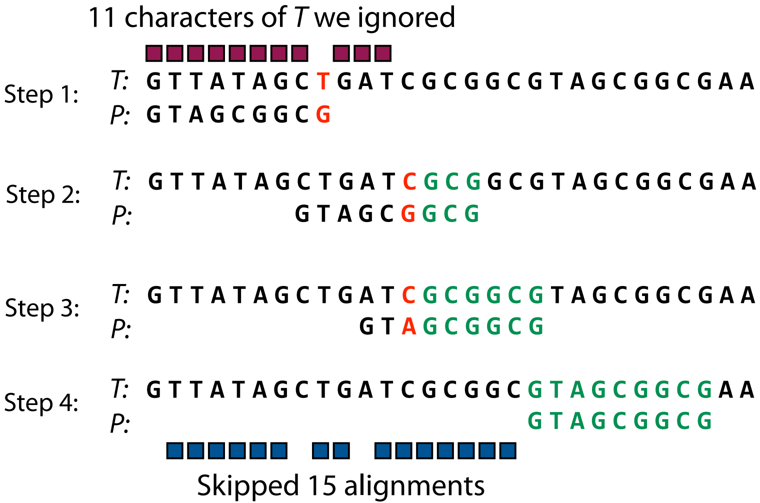
Use bad character or good suffix rule, *whichever skips more*

Step 1:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G   bc: **6**, gs: 0 *bad character*

Step 2:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G   bc: 0, gs: **2** *good suffix*

Step 3:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G   bc: 2, gs: **7** *good suffix*

Step 4:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G


11 characters of T we ignored
Step 1:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G
Step 2:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G
Step 3:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G
Step 4:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G
Skipped 15 alignments

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = TCGC:

Σ \ P	T	C	G	C
A				
C		-		-
G			-	
T	-			

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and *P* = TCGC:

P

Σ

	T	C	G	C
A	0	1	2	3
C	0	-	0	-
G	0	1	-	0
T	-	0	1	2

T: **A A T C A A T A G C**
P: **T C G C**

This can be constructed efficiently. See Gusfield 2.2.2.

Boyer-Moore: Good suffix rule

We learned the *weak* good suffix rule; there is also a *strong* good suffix rule

```
                  t
                |---|
T:  C T T G C C T A C T T A C T T A C T
P:  C T T A C T T A C
      Weak: C T T A C T T A C
            Strong: C T T A C T T A C
guaranteed
mismatch!
```

Strong good suffix rule skips more than weak, at no additional penalty

Strong rule is needed for proof of Boyer-Moore's O(n + m) worst-case time.
Gusfield discusses proof(s) in first several sections of ch. 3

Boyer-Moore: Worst case

Boyer-Moore, with refinements in Gusfield, is O($n + m$) time

Given $n < m$, can simplify to O(m)

Is this better than naïve?

For naïve, worst-case # char comparisons is $n(m - n + 1)$

Boyer-Moore: O(m), naïve: O(nm)

Reminder: $|P| = n \quad |T| = m$

Boyer-Moore: Best case

What's the best case?

P: **bbbb**

T: **aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa**

```
bbbb    bbbb    bbbb    bbbb    bbbb    bbbb
    bbbb    bbbb    bbbb    bbbb    bbbb
```

Every alignment yields immediate mismatch and bad character rule skips *n* alignments

How many character comparisons? *floor(m / n)*

Naive vs Boyer-Moore

As *m* & *n* grow, # characters comparisons grows with...

$|P| = n \quad |T| = m$

	Naïve matching	Boyer-Moore
Worst case	m·n	m
Best case	m	m / n

Performance comparison

Simple Python implementations of naïve and Boyer-Moore:

	Naïve matching		Boyer-Moore		
	# character comparisons	wall clock time	# character comparisons	wall clock time	
P: "tomorrow" **T**: Shakespeare's complete works	5,906,125	2.90 s	785,855	1.54 s	17 matches $\|T\| = 5.59$ M
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	307,013,905	137 s	32,495,111	55 s	336 matches $\|T\| = 249$ M

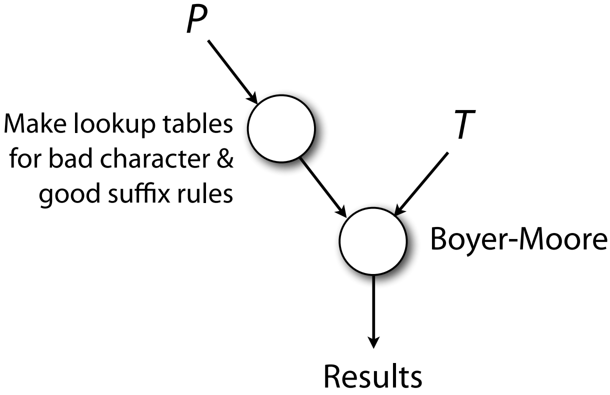
* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

Boyer-Moore implementation

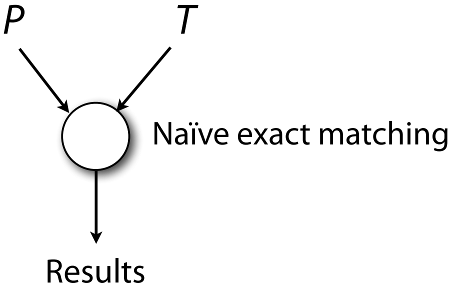
http://j.mp/CG_BoyerMoore

```
def boyer_moore(p, p_bm, t):
    """ Do Boyer-Moore matching """
    i = 0
    occurrences = []
    while i < len(t) - len(p) + 1:  # left to right
        shift = 1
        mismatched = False
        for j in range(len(p)-1, -1, -1):  # right to left
            if p[j] != t[i+j]:
                skip_bc = p_bm.bad_character_rule(j, t[i+j])
                skip_gs = p_bm.good_suffix_rule(j)
                shift = max(shift, skip_bc, skip_gs)
                mismatched = True
                break
        if not mismatched:
            occurrences.append(i)
            skip_gs = p_bm.match_skip()
            shift = max(shift, skip_gs)
        i += shift
    return occurrences
```

Preprocessing: Boyer-Moore



Preprocessing: Naïve algorithm



Preprocessing: Boyer-Moore

Preprocessing: trade one-time cost for reduced work overall via *reuse*

Boyer-Moore preprocesses P into lookup tables that are *reused*

reused for each alignment of P to T_1

If you later give me T_2, I *reuse* the tables to match P to T_2

If you later give me T_3, I *reuse* the tables to match P to T_3

...

Cost of preprocessing is *amortized* over alignments & texts