Greedy Algorithms and Data Compression.

Curs 2018

Greedy Algorithms

A greedy algorithm, is a technique that always makes a locally optimal choice in the myopic hope that this choice will lead to a globally optimal solution.

Greedy algorithms are mainly applied to optimization problems: Given as input a set S of elements, and a function $f : S \to \mathbb{R}$, called the objective function, S we have to choose a of subset of compatible elements in S such that it maximizes (o minimizes) f.

Example: $S = G(V, E), w : E \to \mathbb{Z}$, for any $u, v \in V$, $f(u, v)$, distance between u and v. The problem consists in given specific $v, u \in V$, find the minimum graph distance between u and v. S paths of edges in G, with $f =$ sum of weights of edges between u and v.

Greedy Algorithms

Greedy algorithms are very easy to design, for most optimization problems, but they do not always yield optimal solutions. Sometimes the greedy techniques yield heuristics, not algorithms.

- At each step we choose the best (myopic) choice at the moment and then solve the subproblem that arise later.
- The choice may depend on previous choices, but not on future choices.
- At each choice, the algorithm reduces the problem into a smaller one.
- A greedy algorithm never backtracks.

Greedy Algorithms

For the greedy strategy to work correctly, it is necessary that the problem under consideration has two characteristics:

- Greedy choice property: We can arrive to the global optimum by selecting a local optimums.
- Optimal substructure: An optimal solution to the problem contains the optimal solutions to subproblems.

Fractional knapsack problem

Fractional Knapsack
INPUT:a set $I = \{i\}_1^n$ of items that can be fractioned, each i with weight w_i and value v_i. A maximum weight W permissible
QUESTION: select a set of items or fractions of item, to maximize the profit, within allowed weight W

Example.

Item	I:	1	2	3
Value	V:	60	100	120
Weight	w:	10	20	30

$W = 28$

Fractional knapsack

Greedy for fractional knapsack (I, V, W)
Sort I in decreasing value of v_i/w_i
Take the maximum amount of the first item
while Total weight taken $\leq W$ **do**
  Take the maximum amount of the next item
end while

If n is the number of items, The algorithm has a cost of $T(n) = O(n + n \log n)$.

Example.

Item	I:	1	2	3
Value	V:	60	100	120
Weight	w:	10	20	30
v/w	:	6	5	4

As $W = 28$ then take 10 of 1 and 18 of 2

Correctness?

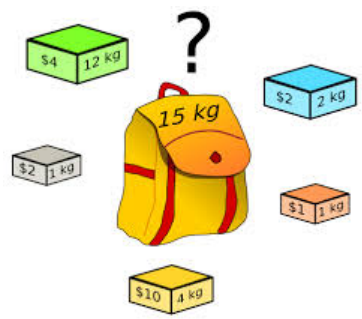
Greedy does not always work

0-1 Knapsack
INPUT:a set I of n items that can NOT be fractioned, each i with weight w_i and value v_i. A maximum weight W permissible
QUESTION: select the items to maximize the profit, within allowed weight W.

For example

Item	I:	1	2	3
Value	V:	60	100	120
Weight	w:	10	20	30
v/w	:	6	5	4

with



$W = 50$.
Then any solution which includes item 1 is not optimal. The optimal solution consists of items 2 and 3.

Activity scheduling problems

A set of activities $S = \{1, 2, \ldots, n\}$ to be processed by a single processor, according to different constrains.

1. Interval scheduling problem: Each $i \in S$ has a start time s_i and a finish time f_i. Maximize the set of mutually compatible activities
2. Weighted interval scheduling problem: Each $i \in S$ has a s_i, a f_i, and a weight w_i. Find the set of mutually compatible such that it maximizes $\sum_{i \in S} w_i$
3. Job scheduling problem (Lateness minimization): Each $i \in S$ has a processing time t_i (could start at any time s_i) but it has a deadline d_i, define lateness L_i of i by $\max_i\{0, (s_i + t_i) - d_i\}$. Find the schedule of the s_i for all the tasks s.t. no two tasks are planned to be processed at the time and the lateness is minimized.

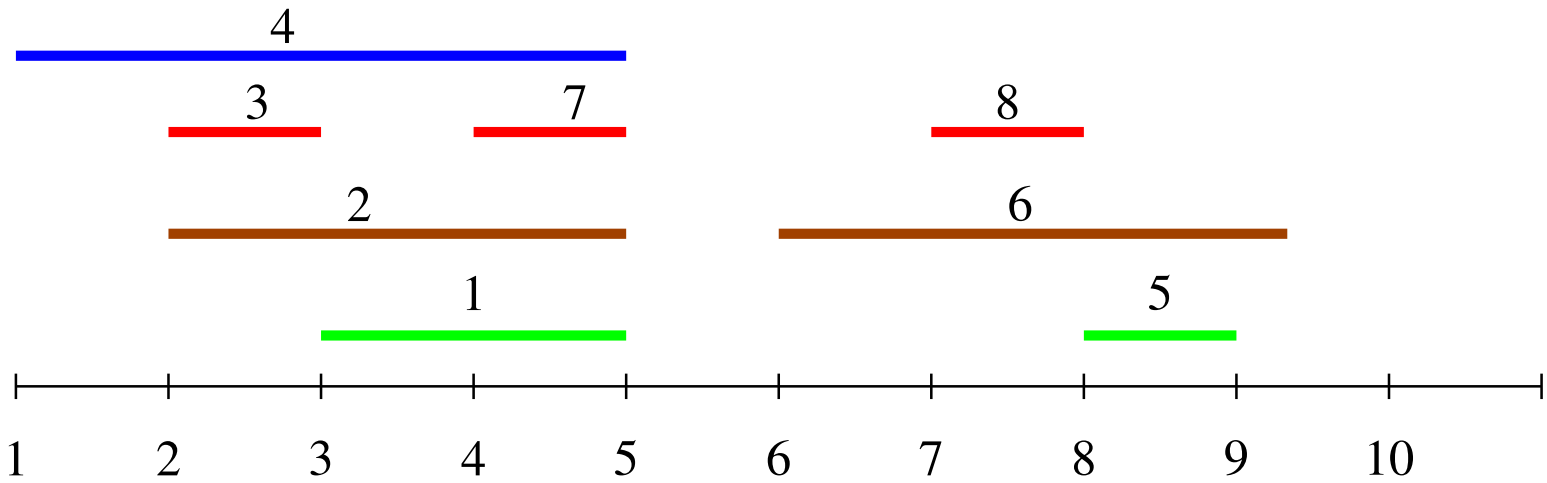
The interval scheduling problem

Activity Selection Problem
INPUT: a set $S = \{1, 2, \ldots, n\}$ of activities to be processed by a single resource. Each activity i has a start time s_i and a finish time f_i, with $f_i > s_i$.
QUESTION: Maximize the set of mutually compatible activities, where activities i and j are compatible if $[s_i f_i) \cap (s_j f_j] = \emptyset$.

Notice, the set of compatible solution is the set of activities with empty intersection, and the objective function to maximize is the cardinality of every compatible set.

Example.

Activity :	1	2	3	4	5	6	7	8
Start (s):	3	2	2	1	8	6	4	7
Finish (f):	5	5	3	5	9	9	5	8



To apply the greedy technique to a problem, we must take into consideration the following,

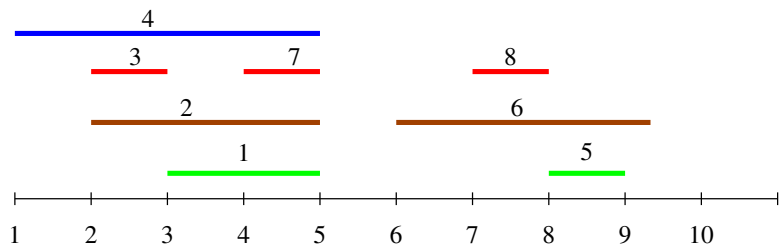
- A local criteria to allow the selection,
- a condition to determine if a partial solution can be completed,
- a procedure to test that we have the optimal solution.

The Activity Selection problem.

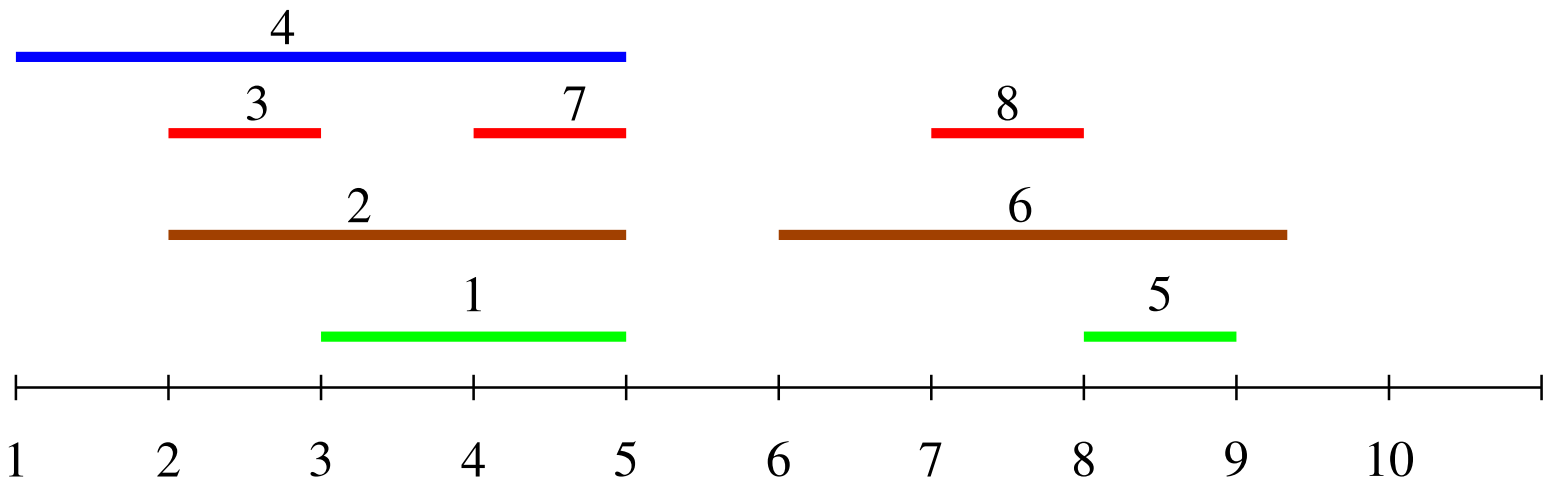
Given a set A of activities, wish to maximize the number of compatible activities.

Activity selection A
Sort A by increasing order of f_i
Let $a_1, a_2, \ldots, a_n$ the resulting sorted list of activities
$S := \{a_1\}$
$j := 1$ {pointer in sorted list}
for $i = 2$ **to** n **do**
  if $s_i \geq f_j$ **then**
    $S := S \cup \{a_i\}$ and $j := i$
  end if
end for
return S.

A : 3 1 2 7 8 5 6; f_i : 3 5 5 5 8 9 9
$\Rightarrow$ SOL: 3 1 8 5



Notice: In the activity problem we are maximizing the number of activities, independently of the occupancy of the resource under consideration. For example in:



solution 3185 is as valid as 3785. If we were asking for maximum occupancy 456 will be a solution.

Problem: How would you modify the previous algorithm to deal with the maximum occupancy problem?

Theorem

The previous algorithm produces an optimal solution to the activity selection problem.

There is an optimal solution that includes the activity with earlier finishing time.

Proof.

Given $A = \{1, \ldots, n\}$ sorted by finishing time, we must show there is an optimal solution that begins with activity 1. Let $S = \{k, \ldots, m\}$ be a solution. If $k = 1$ done. Otherwise, define $B = (S - \{k\}) \cup \{1\}$. As $f_1 \leq f_k$ the activities in B are disjoint. As $|B| = |S|$, B is also an optimal solution. If S is an optimal solution to A, then $S' = A - \{1\}$ is an optimal solution for $A' = \{i \in A | s_i \geq f_1\}$. Therefore, after each greedy choice we are left with an optimization problem of the same form as the original. Induction on the number of choices, the greedy strategy produces an optimal solution $\square$

Notice the optimal substructure of the problem: If an optimal solution S to a problem includes a_k, then the partial solutions excluding a_k from S should also be optimal in their corresponding domains.

Greedy does not always work.

Weighted Activity Selection Problem

INPUT: a set $A = \{1, 2, \ldots, n\}$ of activities to be processed by a single resource. Each activity i has a start time s_i and a finish time f_i, with $f_i > s_i$, and a weight w_i.

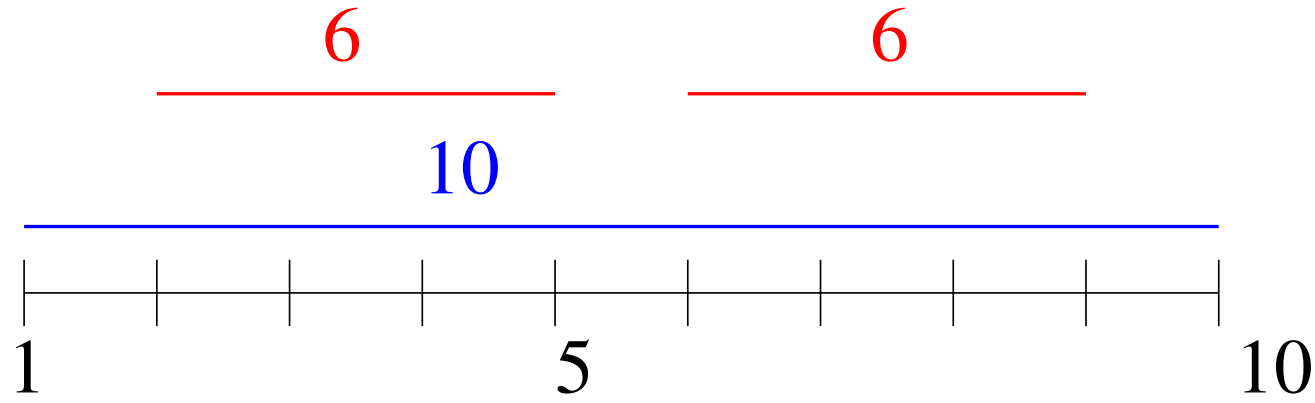
QUESTION: Find the set of mutually compatible activities such that it maximizes $\sum_{i \in S} w_i$

$S := \emptyset$
sort $W = \{w_i\}$ by decreasing value
choose the max. weight w_m
add $m = (s_m, f_m)$ to S
remove all w from W,
which the correspond to activities overlapping with m
while there are $w \in W$ **do**
  repeat the greedy procedure
end while
return S

Correctness?

Greedy does not always work.

The previous greedy does not always solve the problem!



The algorithm chooses the interval $(1, 10)$ with weight 10, and the solution is the intervals $(2, 5)$ and $(5, 9)$ with total weight of 12

Job scheduling problem

Also known as the Lateness minimisation problem.
We have a single resource and n requests to use the resource, each request i taking a time t_i.

In contrast to the previous problem, each request instead of having an starting and finishing time, it has a deadline d_i. The goal is to schedule the resources (processors) as to minimize over all the requests, the maximal amount of time that a request exceeds the deadline.

Minimize Lateness

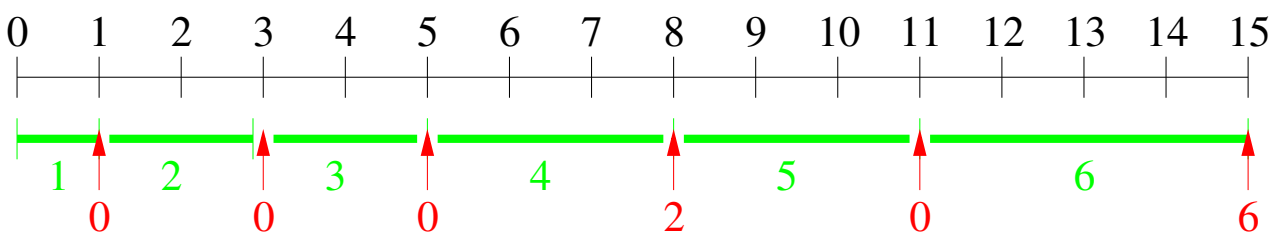
- We have a single processor
- We have n jobs such that job i:
  - requires t_i units of processing time,
  - it has to be finished by time d_i,
- Lateness of i:

$$L_i = \begin{cases} 0 & \text{if } f_i \leq d_i, \\ f_i - d_i & \text{otherwise.} \end{cases}$$

i	t_i	d_i
1	1	9
2	2	8
3	2	15
4	3	6
5	3	14
6	4	9

Goal: schedule the jobs to minimize the maximal lateness, over all the jobs

i.e. We must assign starting time s_i to each i, as to $\min_i \max L_i$.



Minimize Lateness

Schedule jobs according to some ordering

(1.-) Sort in increasing order of t_i:
Process jobs with short time first

i	t_i	d_i
1	1	6
2	5	5

(2.-) Sort in increasing order of $d_i - t_i$:
Process first jobs with less slack time

i	t_i	d_i	$d_1 - t_i$
1	1	2	1
2	10	10	0

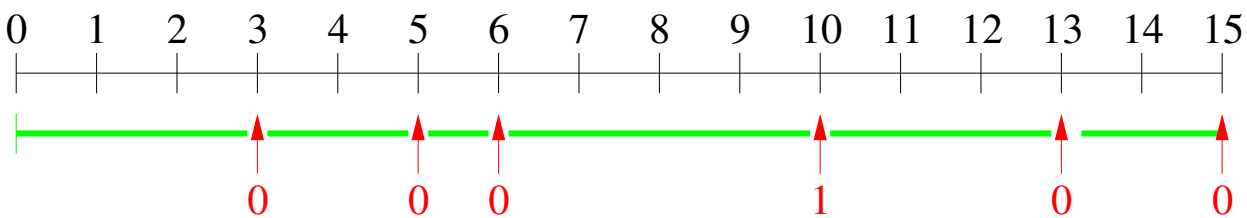
In this case, job 2 should be processed first, which doesn't minimise lateness.

Process urgent jobs first

(3.-) Sort in increasing order of d_i.

LatenessA $\{i, t_i, d_i\}$
SORT by increasing order of d_i:
$\{d_1, d_2, \ldots, d_n\}$
Rearrange the jobs $i : 1, 2, \ldots, n$
$t = 0$
for $i = 2$ **to** n **do**
  Assign job i to $[t, t + t_i]$
  $t = t + t_i$
  $s_i = t; f_i = t + t_i$
end for
return $S = \{[s_1, f_1], \ldots [s_n, f_n]\}$.

i	t_i	d_i	sorted i
1	1	9	3
2	2	8	2
3	2	15	6
4	3	6	1
5	3	14	5
6	4	9	4



d: 6 8 9 9 14 15
i: 1 2 3 4 5 6

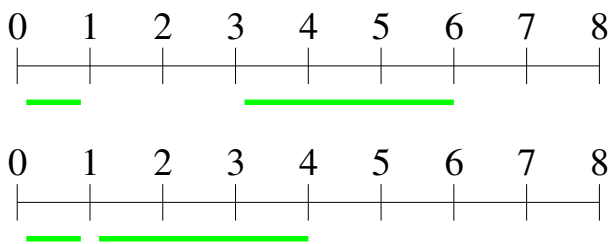
Complexity and idle time

Time complexity

Running-time of the algorithm without comparison sorting: $O(n)$
Total running-time: $O(n \lg n)$

Idle steps

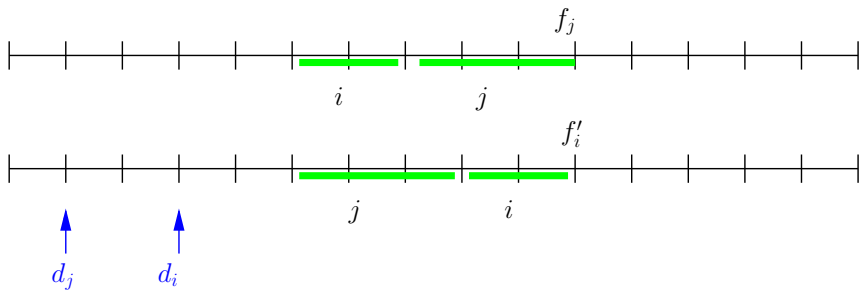
From an optimal schedule with idle steps, we always can eliminate gaps to obtain another optimal schedule:



There exists an optimal schedule with no idle steps.

Inversions and exchange argument

An schedule has an inversion if i is scheduled before j with $d_j < d_i$.



Lemma

Exchanging two inverted jobs reduces the number of inversions by 1 and does not increase the max lateness.

Proof Let L = lateness before exchange and let L' be the lateness after the exchange, let L_i, L_j, L'_i, L'_j, the corresponding quantities for i, j.

Notice that $f_j > f_i$, $f'_j < f'_j$ and $f'_j < f_i$, using the fact that $d_j < d_i$
$\Rightarrow L'_i = f'_i - d_i < f_j - d_i < f_j - d_j = L_j$

Therefore the swapping does not increase the maximum lateness of the schedule. □

Correctness of LatenessA

Notice the output S produced by LatenessA has no inversions and no idle steps.

Theorem

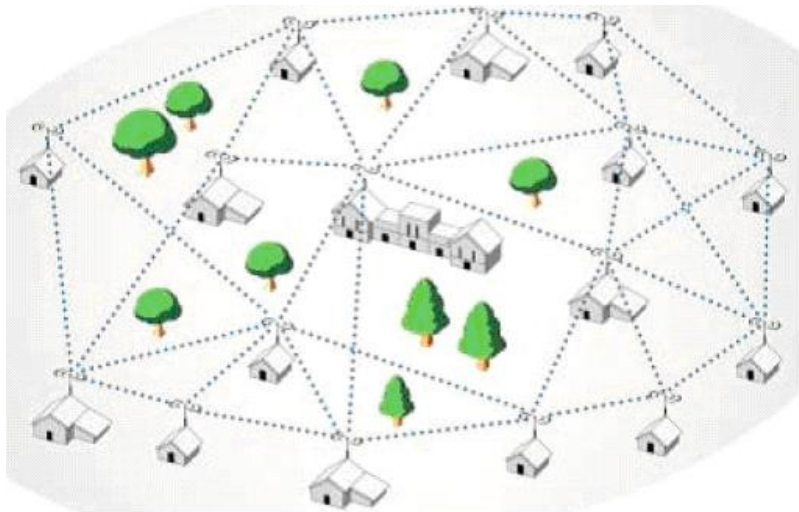
Algorithm LatenessA returns an optimal schedule S.

Proof

Assume $\hat{S}$ is an optimal schedule with the minimal number of inversions (and no idle steps).
If $\hat{S}$ has 0 inversions then $\hat{S} = S$.
If number inversions in $\hat{S}$ is > 0, let $i - j$ be an adjacent inversion.
Exchanging i and j does not increase lateness and decrease the number of inversions.
Therefore, max lateness $S \leq$ max lateness $\hat{S}$. □

Network construction: Minimum Spanning Tree

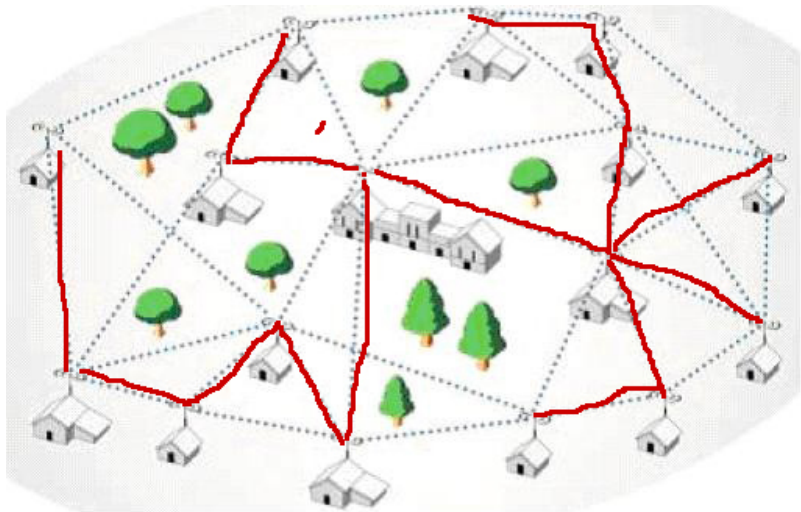
- We have a set of locations $V = \{v_1, \ldots, v_n\}$,
- we want to build a communication network on top of them
- we want that any v_i can communicate with any v_j,
- for any pair (v_i, v_j) there is a cost $w(v_i, v_j)$ of building a direct link,
- if E is the set of all possible edges ($|E| \leq n(n-1)/2$), we want to find a subset $T(E) \subseteq E$ s.t. $(V, T(E))$ is connected and minimizes $\sum_{e \in T(E)} w(e)$.



Network construction: Minimum Spanning Tree

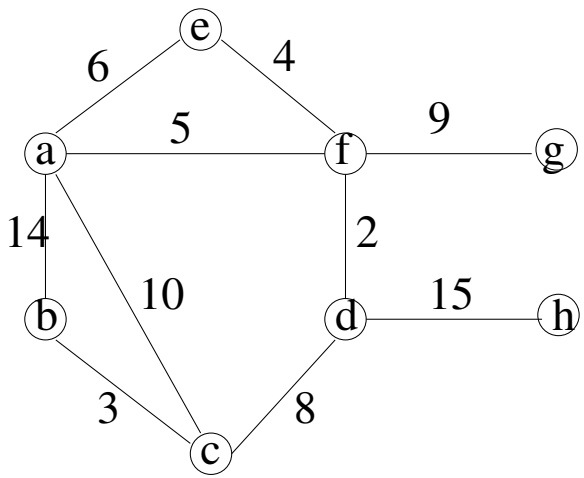
- We have a set of locations $V = \{v_1, \ldots, v_n\}$,
- we want to build a communication network on top of them
- we want that any v_i can communicate with any v_j,
- for any pair (v_i, v_j) there is a cost $w(v_i, v_j)$ of building a direct link,
- if E is the set of all possible edges ($|E| \leq n(n-1)/2$), we want to find a subset $T(E) \subseteq E$ s.t. $(V, T(E))$ is connected and minimizes $\sum_{e \in T(E)} w(e)$.

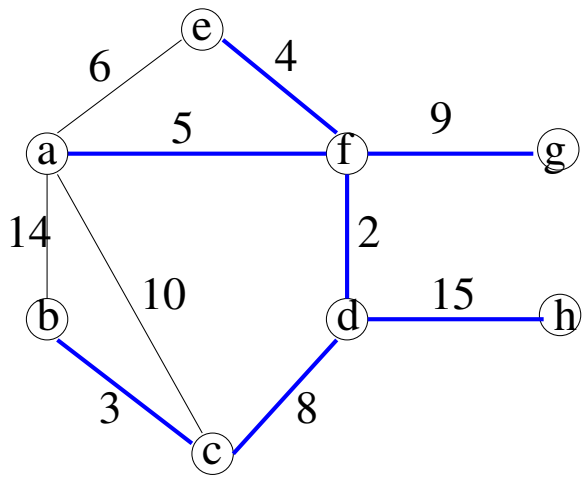
Construct the MST



Minimum Spanning Tree (MST).

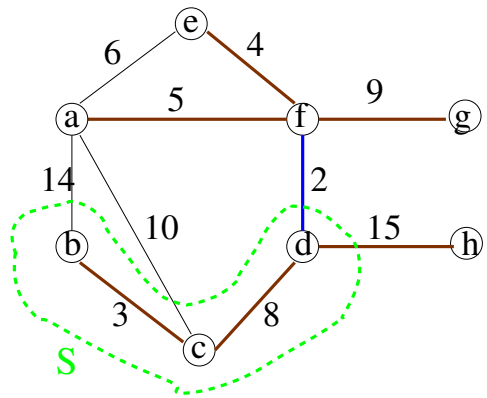
INPUT: An edge weighted graph $G = (V, E)$,
$|V| = n, \forall e \in E, w(e) \in \mathbb{R}$,
QUESTION: Find a tree T with $V(T) = V$ and $E(T) \subseteq E$, such that it minimizes $w(T) = \sum_{e \in E(T)} w(e)$.





Some definitions

Given $G = (V, E)$:
A path is a sequence of consecutive edges. A cyle is a path with no repeated vertices other that the one that it starts and ends.
A cut is a partition of V into S and $V - S$.
The cut-set of a cut is the set of edges with one end in S and the other in $V - S$.



Overall strategy

Given a MST T in G, with different edge weights, T has the following properties:

- Cut property
  $e \in T \Leftrightarrow e$ is the lighted edge across some cut in G.
- Cycle property
  $e \notin T \Leftrightarrow e$ is the heaviest edge on some cycle in G.

The MST algorithms are methods for ruling edges in or out of T.

The $\Leftarrow$ implication of the cut property will yield the blue (include) rule, which allow us to include a min weight edge in T for $\exists$ cut.

The $\Rightarrow$ implication of the cycle property will yield the red (exclude) rule which allow us to exclude a max weight edge from T for $\exists$ cycles.

The cut rule (Blue rule)

Given an optimal MST T, removing an edge e yields T_1 and T_2 which are optimal for each subgraph.

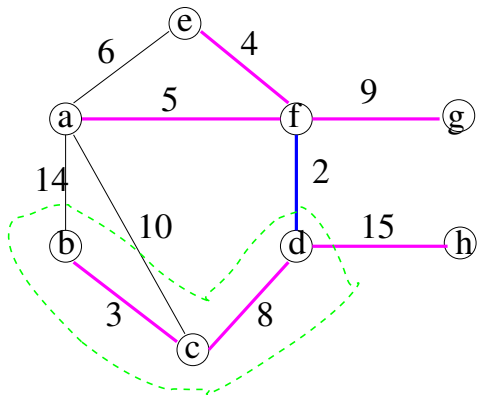
Theorem (The cut rule)

Given $G = (V, E), w : E \to \mathbb{R}$, let T be a MST of G and $S \subseteq T$. Let $e = (u, v)$ an min-weight edge in G connecting S to $V - S$. Then $e \in T$.

The edges incorporated to the solution by this rule, are said to be blue.

Proof.

Assume $e \notin T$. Consider a path from u to v in T. Replacing the first edge in the path, which is not in S, by e, must give a spanning tree of equal or less weight. $\square$



The cycle rule (Red rule)

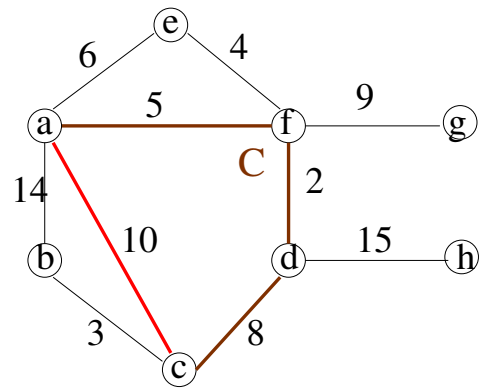
Theorem (The cycle rule)

Given $G = (V, E), w : E \to \mathbb{R}$, let C be a cycle in G, the edge $e \in C$ with greater weight can not be part of the optimal MST T.

The edges processed by this rule, are said to be red.

Proof.

Let C be a cycle spanning through vertices $\{v_i, \ldots, v_l\}$, then removing the max weighted edge gives a a better solution. □



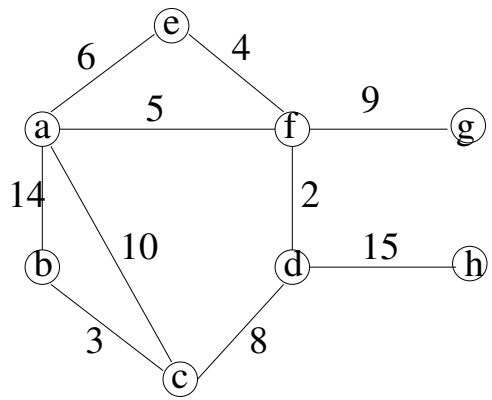
C=cycle spanning {a,c,d,f}

Greedy for MST

The Min. Spanning Tree problem has the optimal substructure problem we can apply greedy.
Robert Tarjan: Data Structures and Network Algorithms, SIAM , 1984

Blue rule: Given a cut-set between S and $V - S$ with no blue edges, select from the cut-set a non-colored edge with min weight and paint it blue

Red rule: Given a cycle C with no red edges, selected an non-colored edge in C with max weight and paint it red.



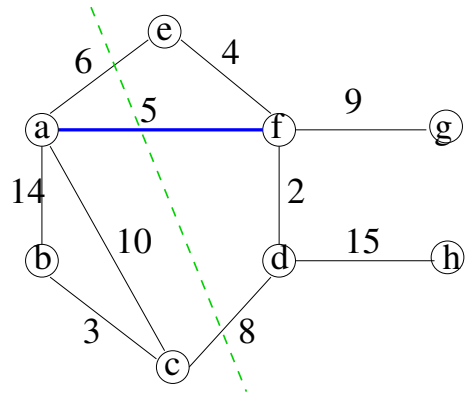
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

Greedy for MST

The Min. Spanning Tree problem has the optimal substructure problem we can apply greedy.
Robert Tarjan: Data Structures and Network Algorithms, SIAM , 1984

Blue rule: Given a cut-set between S and $V - S$ with no blue edges, select a non-colored edge with min weight and paint it blue

Red rule: Given cycle C with no red edges, selected a non-colored edge in C with max weight and paint it red.



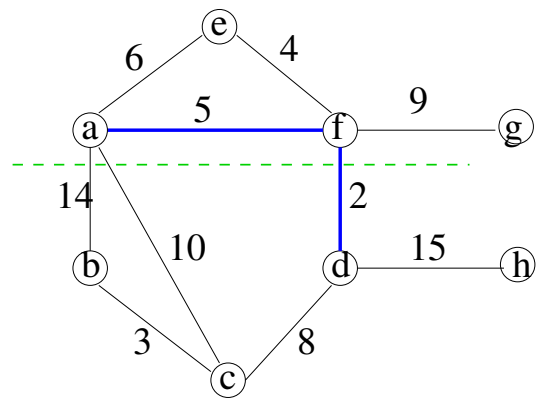
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

Greedy for MST

The Min. Spanning Tree problem has the optimal substructure problem we can apply greedy.
Robert Tarjan: Data Structures and Network Algorithms, SIAM , 1984

Blue rule: Given a cut-set between S and $V - S$ with no blue edges, select a non-colored edge with min weight and paint it blue

Red rule: Given cycle C with no red edges, selected a non-colored edge in C with max weight and paint it red.



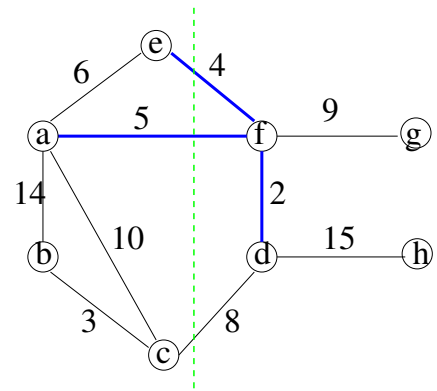
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

Greedy for MST

The Min. Spanning Tree problem has the optimal substructure problem we can apply greedy.
Robert Tarjan: Data Structures and Network Algorithms, SIAM , 1984

Blue rule: Given a cut-set between S and $V - S$ with no blue edges, select a non-colored edge with min weight and paint it blue

Red rule: Given cycle C with no red edges, selected a non-colored edge in C with max weight and paint it red.



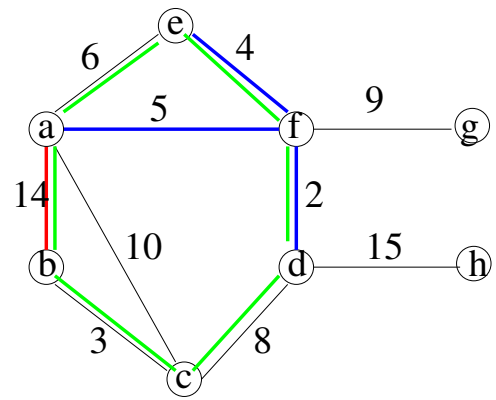
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

Greedy for MST

The Min. Spanning Tree problem has the optimal substructure problem we can apply greedy.
Robert Tarjan: Data Structures and Network Algorithms, SIAM , 1984

Blue rule: Given a cut-set between S and $V - S$ with no blue edges, select a non-colored edge with min weight and paint it blue

Red rule: Given cycle C with no red edges, selected a non-colored edge in C with max weight and remove it.



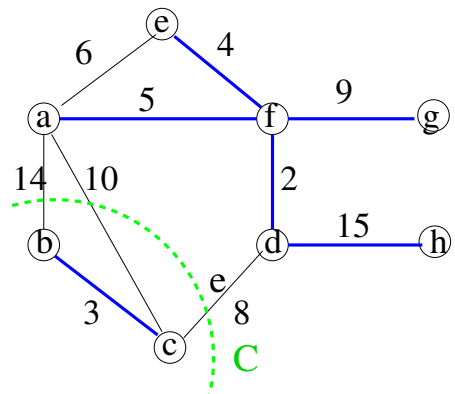
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

Greedy for MST : Correctness

Theorem

There exists a MST T containing only all blue edges. Moreover the algorithm finishes and finds a MST

Sketch of proof Induction on number of iterations for blue and red rules. The base case (no edges colored) is trivial. The induction step is the same that in the proofs of the cut and cycle rules.
Moreover if we have an e not colored, if ends are in different blue tree, apply blue rule, otherwise color red e. □



We need implementations for the algorithm! The ones we present use only the blue rule

A short history of MST implementation

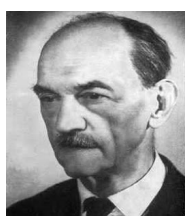
There has been extensive work to obtain the most efficient algorithm to find a MST in a given graph:

- O. Borůvka gave the first greedy algorithm for the MST in 1926. V. Jarnik gave a different greedy for MST in 1930, which was re-discovered by R. Prim in 1957. In 1956 J. Kruskal gave a different greedy algorithms for the MST. All those algorithms run in $O(m \lg n)$.
- Fredman and Tarjan (1984) gave a $O(m \log^* n)$ algorithm, introducing a new data structure for priority queues, the Fibbonacci heap. Recall $\log^* n$ is the number of operations to go from $n = 1$ to $\log^* 1000 = 4$.
- Gabow, Galil, Spencer and Tarjan (1986) improved Fredman-Tarjan to $O(m \log(\log^* n))$.
- Karger, Klein and Tarjan (1995) $O(m)$ randomized algorithm.
- In 1997 B. Chazelle gave an $O(m\alpha(n))$ algorithm, where $\alpha(n)$ is a very slowly growing function, the inverse of the Ackermann function.

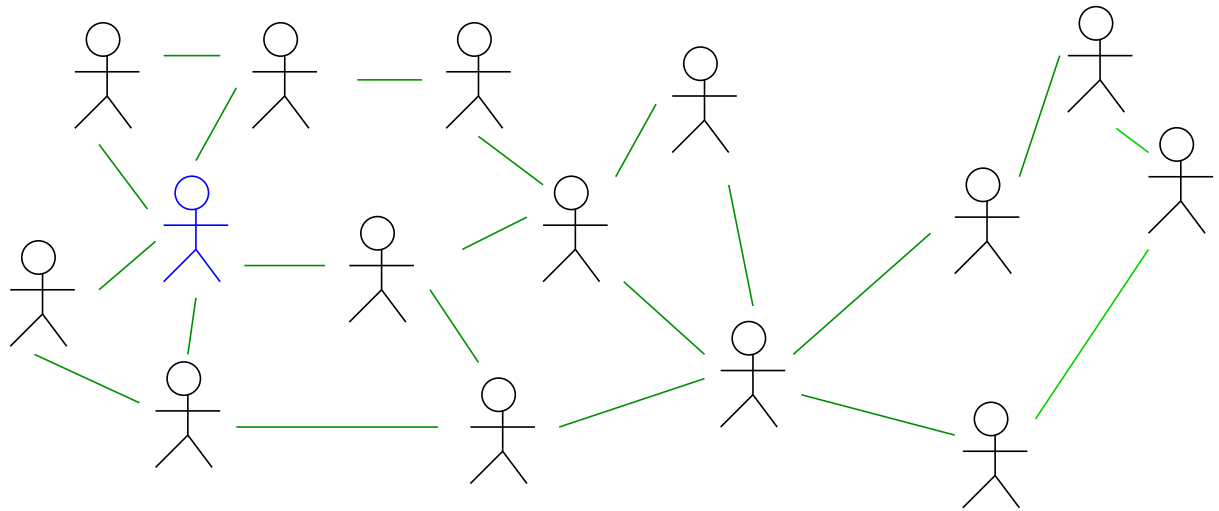
Basic algorithms

Use the greedy

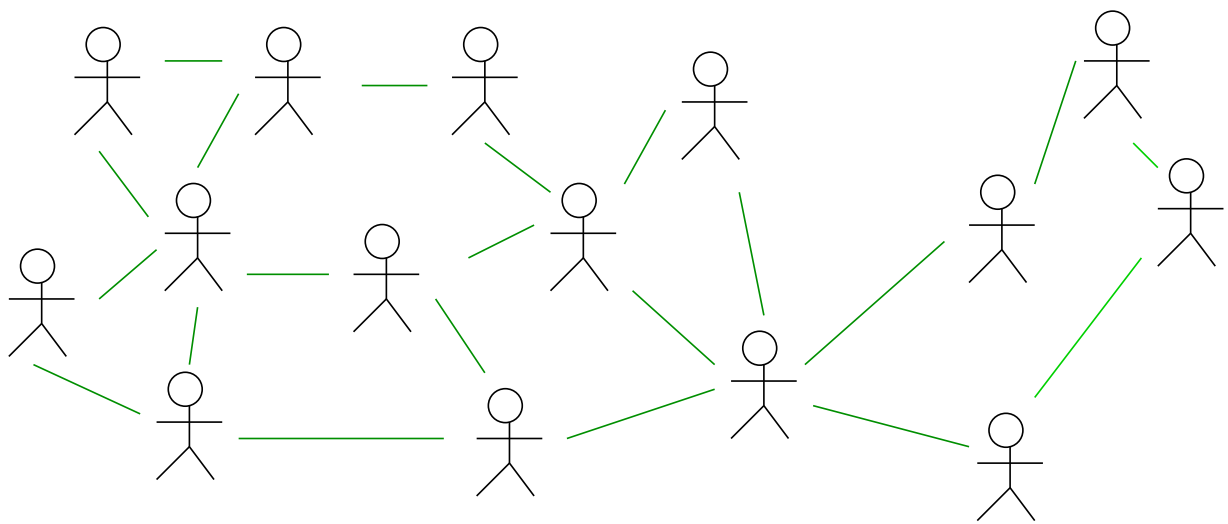
- Jarnik-Prim (Serial centralized) Starting from a vertex v, grows T adding each time the lighter edge already connected to a vertex in T, using the blue's rule. Uses a priority queue (usually a heap) to store the edges to be added and retrieve the lighter one.
- Kruskal (Serial distributed) Considers every edge and grows a forest F by using the blue and red rules to include or discard e. The insight of the algorithm is to consider the edges in order of increasing weight. This makes the complexity of Kruskal's to be dominated by $\Omega(m \lg m)$. At the end F becomes T. The efficient implementation of the algorithm uses Union-find data structure.



Jarnik-Prim vs. Kruskal

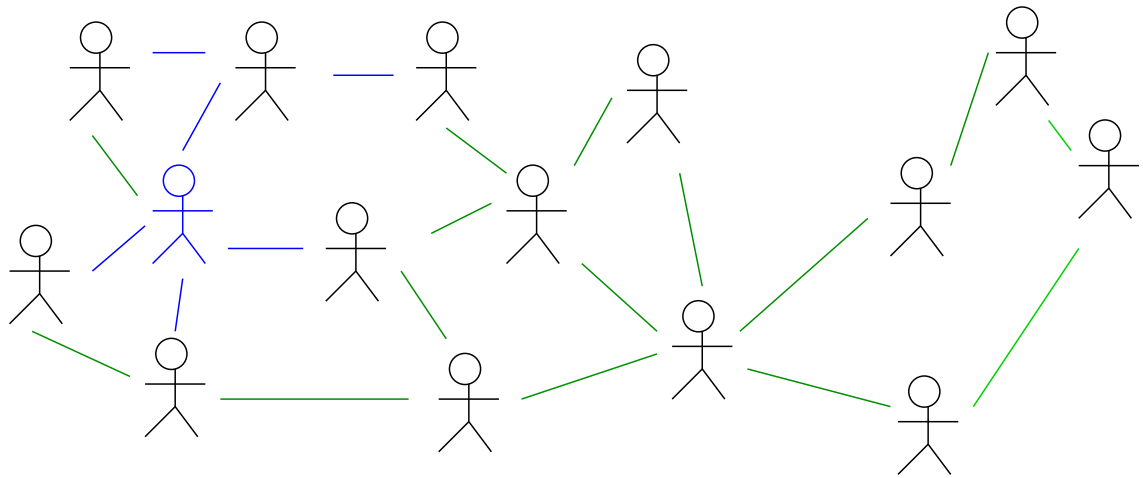


Jarnik–Prim: How blue man can spread his message to everybody

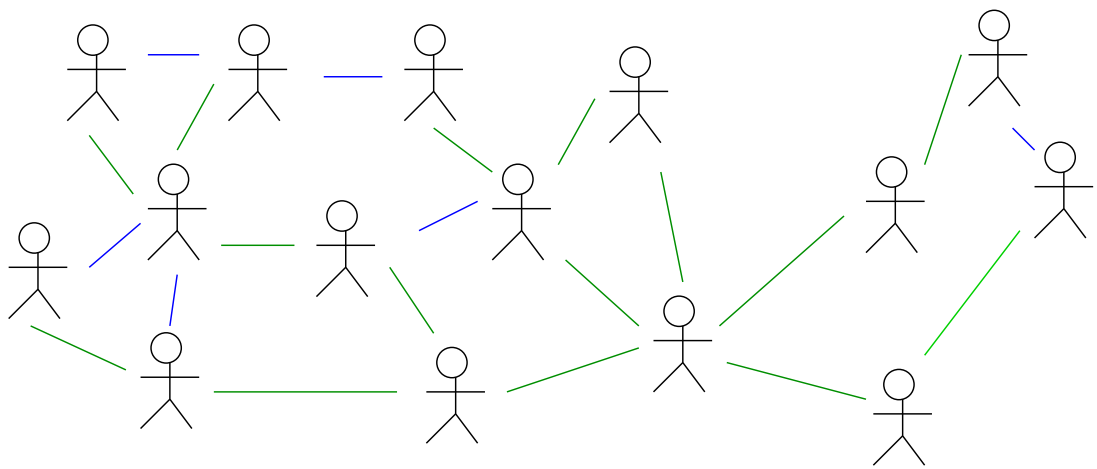


Kruskal: How to stablish a min distance cost network about all men

Jarník-Prim vs. Kruskal



Jarnik–Prim: How blue man can spread his message to everybody
(first 6 steps)



Kruskal: How to stablish a min distance cost network about all men

(6 first edges)

Jarník - Prim greedy algorithm.

V. Jarník, 1936, R. Prim, 1957

Greedy on vertices with Priority queue

Starting with an arbitrary node r, at each step build the MST by incrementing the tree with an edge of minimum weight, which does not form a cycle.

MST (G, w, r)
$T := \emptyset$
for $i = 1$ **to** $|V|$ **do**
  Let $e \in E$: e touches T, it has min weight, and do not form a cycle
  $T := T \cup \{e\}$
end for

Use a priority queue to choose min e connected to the tree already formed.

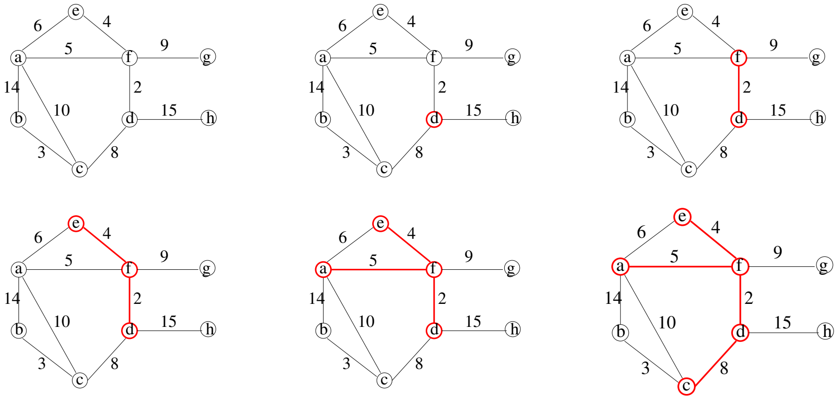
For every $v \in V - T$, let $k[v] =$ minimum weight of any edge connecting v to any vertex in T.

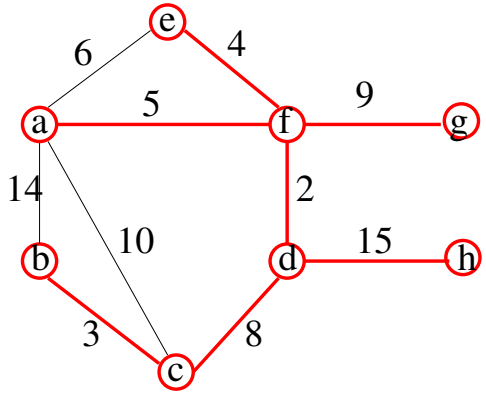
Start with $k[v] = \infty$ for all v.

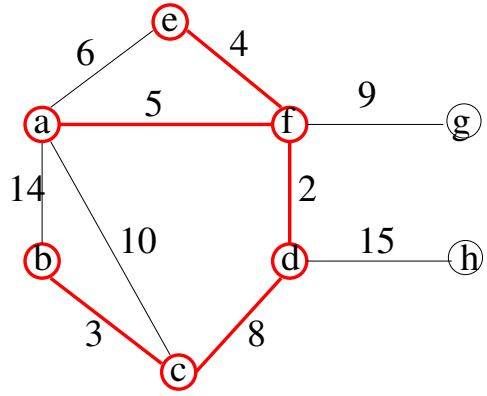
For $v \in T$, let $\pi[v]$ be the parent of v. During the algorithm $T = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$

where r is the arbitrary starting vertex and Q is a min priority queue storing $k[v]$. The algorithm finishes when $Q = \emptyset$.

Example.







$w(T) = 52$

Time: depends on the implementation of Q.

Q an unsorted array: $T(n) = O(|V|^2)$;
Q a heap: $T(n) = O(|E| \lg |V|)$.
Q a Fibonacci heap: $T(n) = O(|E| + |V| \lg |V|)$

Kruskal's greedy algorithm.

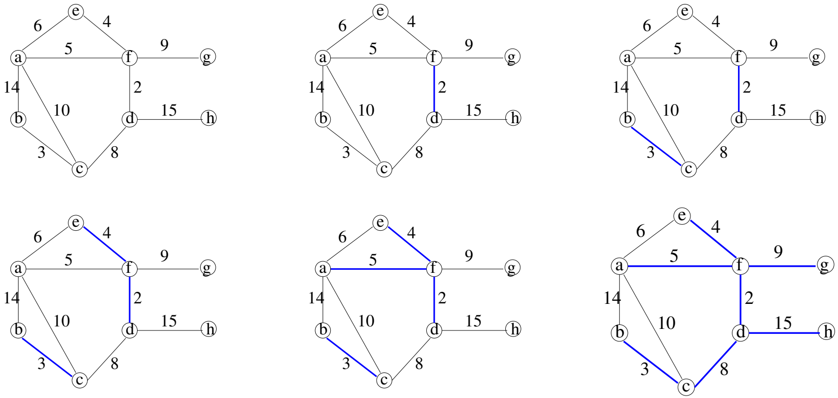
J. Kruskal, 1956

Similar to Jarník - Prim, but chooses minimum weight edges, without keeping the graph connected.

MST (G, w, r)
Sort E by increasing weight
$T := \emptyset$
for $i = 1$ **to** $|V|$ **do**
  Let $e \in E$: with minimum weight and do not form a cycle with T
  $T := T \cup \{e\}$
end for

We have an $O(m \lg m)$ from the sorting the edges.
But as $m \leq n^2$ then $O(m \lg m) = O(m \lg n)$.

Example.


e a f g b d h c
6 4 5 9 14 2 10 15 3 8
e a f g b d h c
6 4 5 9 14 2 10 15 3 8
e a f g b d h c
6 4 5 9 14 2 10 15 3 8
e a f g b d h c
6 4 5 9 14 2 10 15 3 8
e a f g b d h c
6 4 5 9 14 2 10 15 3 8
e a f g b d h c
6 4 5 9 14 2 10 15 3 8

MST induces am equivalence classes partition of V of G

Notice that Kruskal evolves by building clumps of trees and merging the clumps into larger clumps, taking care there are not cycles.

In trees, the connectivity relation is an equivalence relation, two nodes are connected if there is only one path between them.

Given an undirected $G = (V, E)$, the MST T on $G = (V, E)$, induces an equivalence relation between the vertices of V: $u \mathcal{R} v$ iff there a T-path between u and v:

$\mathcal{R}$ partition the elements of V in equivalence classes, which are connected components without cycles

Disjoint Set Union-Find

B. Galler, M. Fisher: An improved equivalence algorithm. ACM Comm., 1964; R.Tarjan 1979-1985

Nice data structure to implement Kruskal, but also useful to maintain any collection of dynamic disjoint sets

Basic idea: Give a disjoint-set, data structure that it maintains a collection $\{S_1, \ldots, S_k\}$ of disjoint dynamic sets, each set identified by a *representative*.

Union-Find supports three operations on partitions of a set:

MAKESET (x): creates a new set containing the single element x.

UNION (x, y): Merge the sets containing x and y, by using their union. (x and y do not have to be the representatives of their sets)

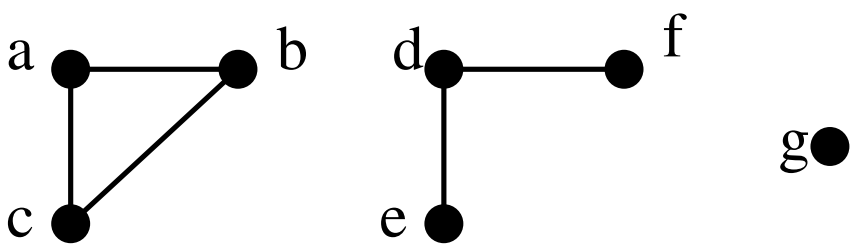
FIND (x): Return the representative of the set containing x.

Graph connectivity

Given a $G = (V, E)$, by the vertices and edges, we want to know if any two $v, u \in V$ belong to the same connected component.

CONNECTED-COMP. G
for each $v \in V(G)$ **do**
  MAKESET (v)
end for
for each $(u, v) \in E(G)$ **do**
  if FIND $(u) \neq$ FIND (v)
  then
    UNION (u, v)
  end if
end for

SAME-COMPO (u, v)
if FIND $u =$ FIND (v) **then**
  return true
else
  return false
end if



{ a b c } { e d f } { g }

Union-Find Data Structure: The problem

We have n initial elements, we start by applying n times MAKESET to have n single element sets.
We want to implement a sequence of m UNION and FIND operations on the initial sets, using the minimum number of steps.

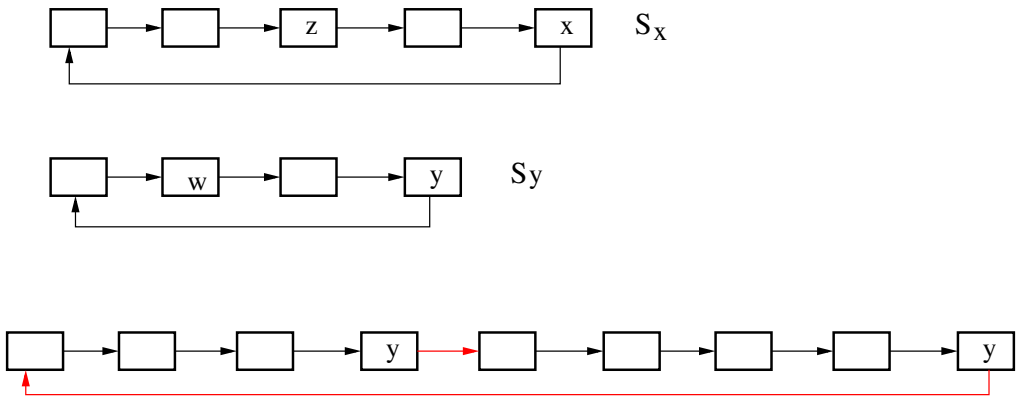
Union-find implementation: First idea

Given $\{S_1, \ldots, S_k\}$, use linked lists to represent

- Each set S_i represented by a linked list.
- The *representative* of S_i is defined to be the element at the head of the list representing the set.

  - MAKESET (x): Initializes x as a lone list. Worst time $\Theta(1)$.
  - UNION (z, w): Find the representative y, and point to the tail of the list S_x implementing $S_x \cup S_y$. Worst time $\Theta(|S_x| + |S_y|)$.
  - FIND (z): Goes left from z to the head of S_x. Worst case $\Theta(|S_x|)$.

Union-find list implementation: Example

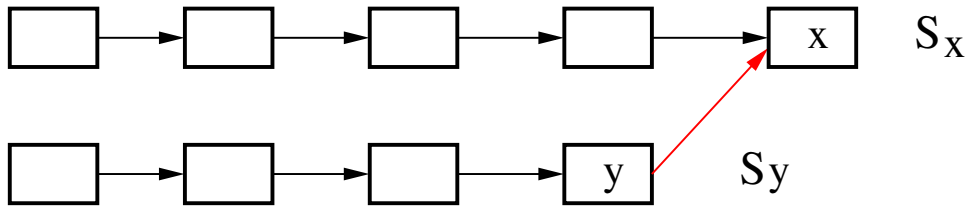


We start with $x_1, x_2, \ldots, x_n$, do n MAKESET, followed by UNION(x_1, x_2), UNION(x_2, x_3), $\ldots$, UNION(x_{n-1}, x_n), we are doing $2n - 1$ operations, with a total cost of $\Theta(n^2)$ why?

Notice we have that the on *average* cost of each operation is $\Theta(n^2)/(2n - 1) \sim \Theta(n)$.

That is called the amortized time analysis.

Union-Find implementation: Tree like structure



- MAKESET (x): Initializes x as a lone list. Worst time $\Theta(1)$.
- FIND (z): Goes left from z to the head of S_x. Worst case $\Theta(\max\{|S_x|, |S_y|\})$.
- UNION (z, w): FIND(z), FIND(w), and y points to the tail of S_x implementing $S_x \cup S_y$. Worst time $\Theta(|S_y| + |S_x|)$.

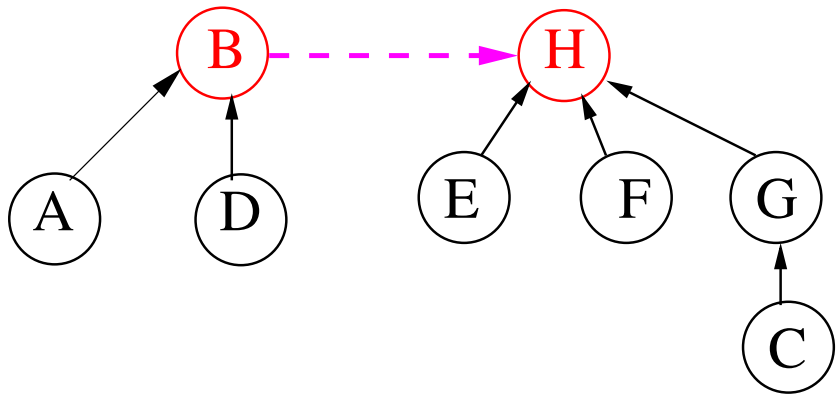
Recall: We have n initial sets and we want see the number of steps to implement a sequence of m UNION and FIND operations on the initial sets ($m > n$).

Union-Find implementation: Link by size of forest of Trees

Represent each set as a tree of elements:

Notice that the root contains the representative.

- MAKESET (x): $\Theta(1)$
- FIND (z): find the root of the tree containing z. $\Theta(\text{height})$
- UNION (x, y): make the root of the tree with less elements point to the root of the tree with more elements. Complexity?



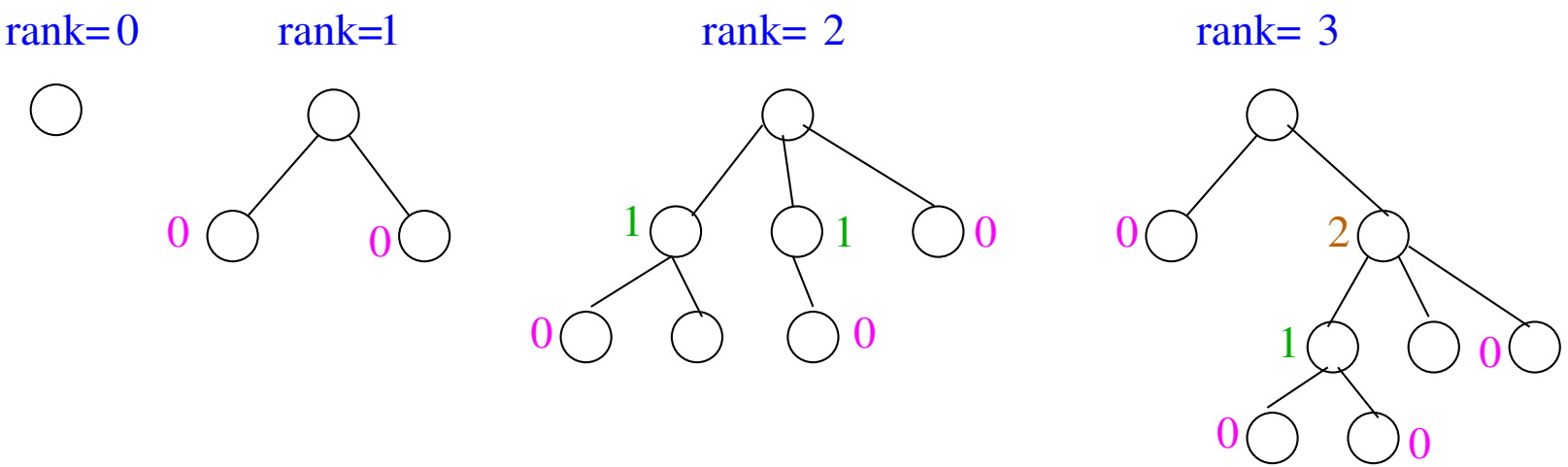
It is not difficult to see the cost of making m UNION operations on n singleton sets, is the same that in the list case.

A better implementation: Link by rank

Let $r(x)$ = height of subtree rooted at x.

Any singleton element has rank=0

Inductively as we join trees, increase the rank of the root.

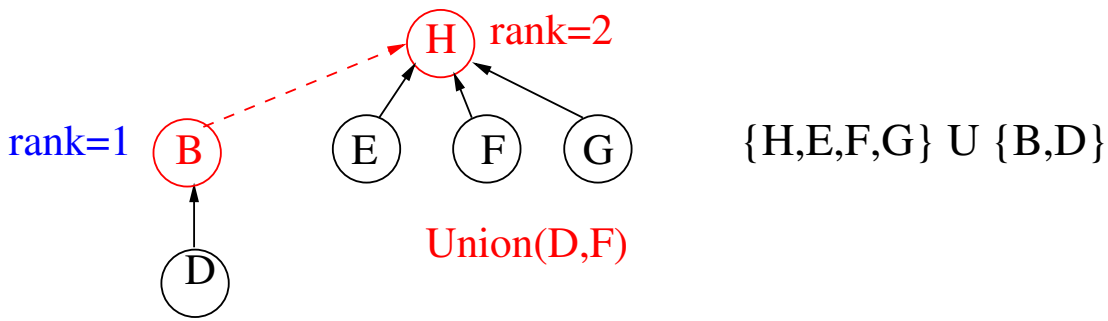


Link by rank

Union rule: Link the root of smaller rank tree to the root of larger rank tree.

In case the roots of both trees have the same rank, choose arbitrarily and increase $+1$ the rank of the winer

except for the root, a node does not change rank during the process



• UNION (x, y): climbs to the roots of the tree containing x and y and merges sets by making the tree with less rank a subtree of the other tree. This takes Θ(height) steps.

Link by rank

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

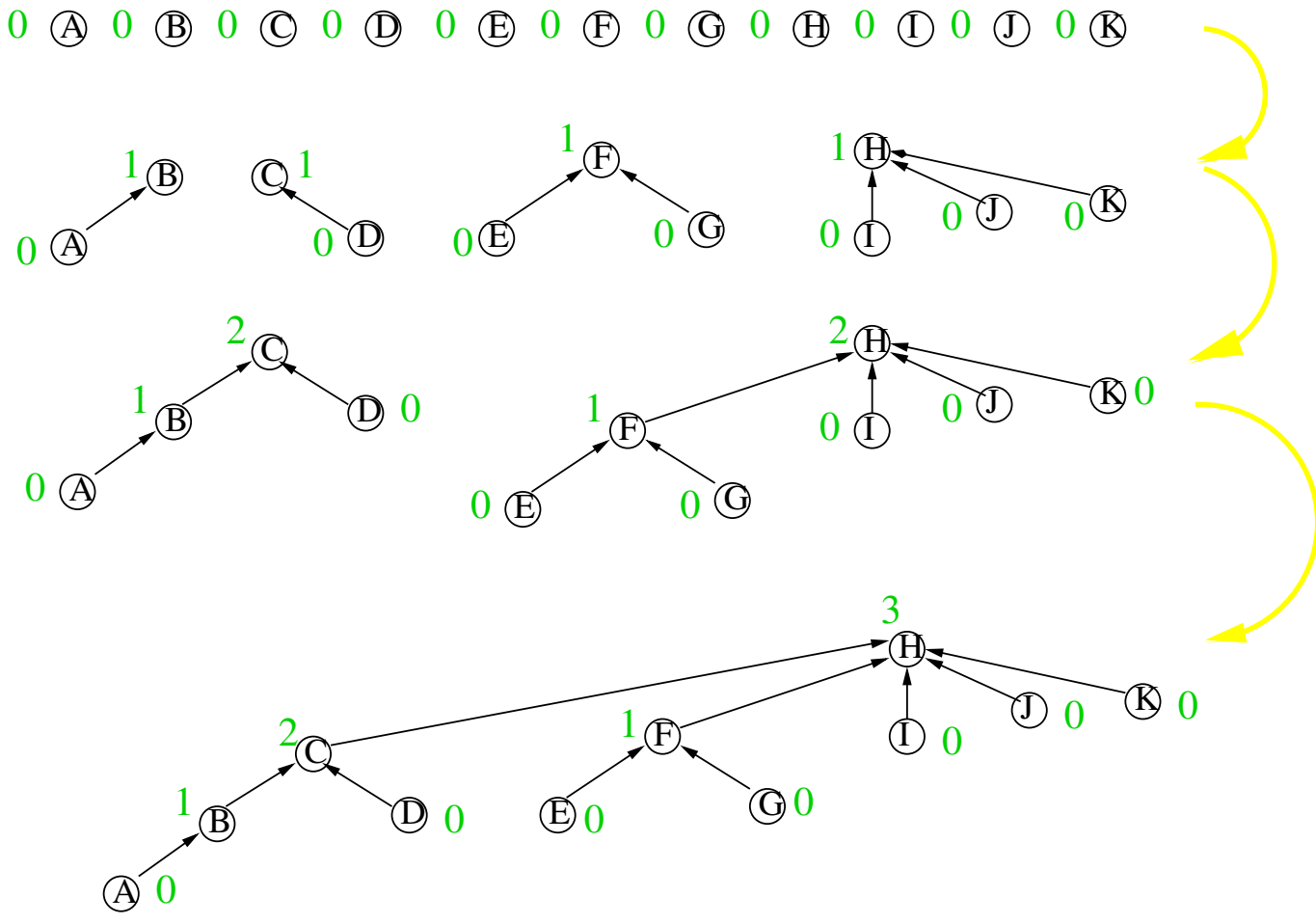
Let $p(z)$ be the parent of (z) in the forest, and let $r(z)$ be the rank of (z).

MAKESET (x)
$p(x) = x$
$r(x) = 0$

FIND (z)
while $(z \neq p(z)$ **do**
  $z = p(z)$
end while

UNION (x, y)
$r_x =$ **FIND**(x)
$r_y =$ **FIND**(y)
if $r_x = r_y$ **then**
  STOP
else if $r(r_x) > r(r_y)$ **then**
  $p(r_y) = r_x$
else if $r(r_x) < r(r_y)$ **then**
  $p(r_x) = r_y$
else
  $p(r_x) = r_y$
  $r(r_y) = r(r_y) + 1$
end if

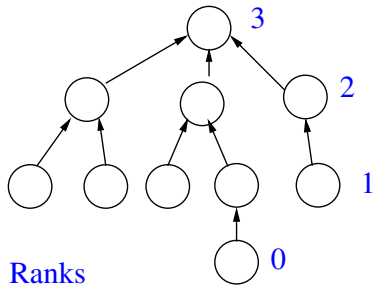
Example of construction by Union-Find



Properties of Link by rank

P1.- If x is not a root then $r(x) < r(p(x))$

P2.- If $p(x)$ changes then $r(p(x))$ increases



P3.- Any root of rank k has $\geq 2^k$ descendants.

Proof (Induction on k) True for $k = 0$, if true for $k - 1$ then a node of rank k results from the merging of 2 nodes with rank $k - 1$. $\square$

Properties of Link by rank

P4.- The highest rank of a root is $\leq \lfloor \lg n \rfloor$

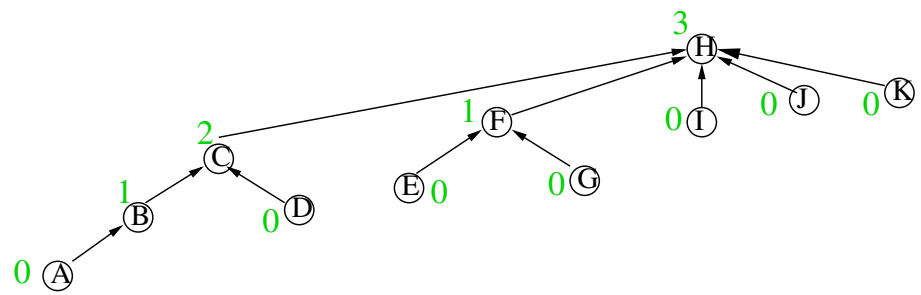
Proof Follows P1 and P3. $\square$

P5.- For any $r \geq 0$, there are $\leq n/2^r$ nodes with rank r.

Proof By (P4) any root node x with $r(x) = k$ has $\geq 2^k$ descendants.

Any non-root node y with $r(x) = k$ has $\geq 2^k$ descendants.

By (P1) different nodes with rank $= k$ can't have common descendants. $\square$



Complexity of Link by rank

Theorem

Using link-by-rank, each application of UNION(x, y) or FIND(x) takes $O(\lg n)$ steps.

Proof The number of steps for each operation is bounded by the height of the tree, which is $O(\lg n)$, (P4). □

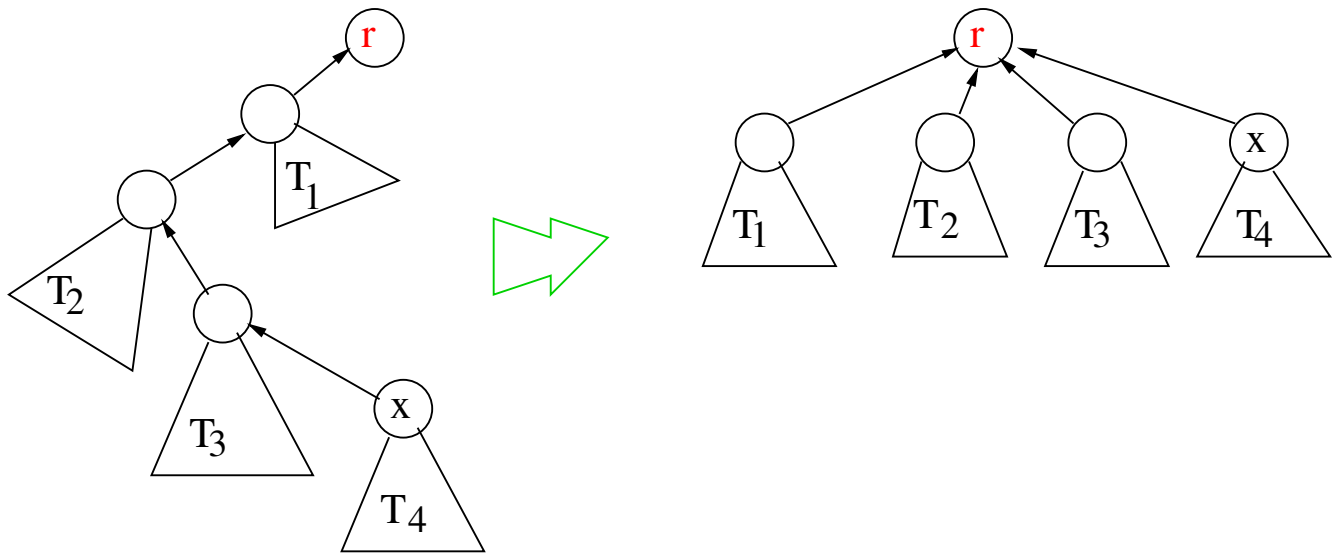
Theorem

Starting from an empty data structure with n elements, link-by-rank performs any intermixed sequence of m **FIND** *and* **UNION** *operations in $O(n + m \lg n)$ steps.*

An improvement: Path compression

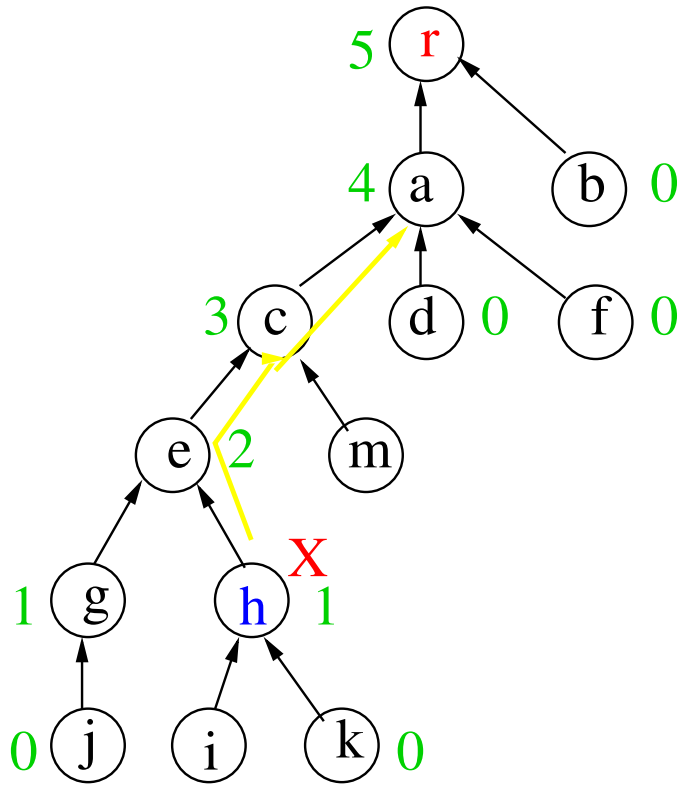
To improve the $O(\log n)$ bound per operation in union-bound, we keep the trees as flat as possible.

We use the path compression: At each use of FIND(x) we follow all the path $\{y_i\}$ of nodes from x to the root r change the pointers of all the $\{y_i\}$ to point to r.



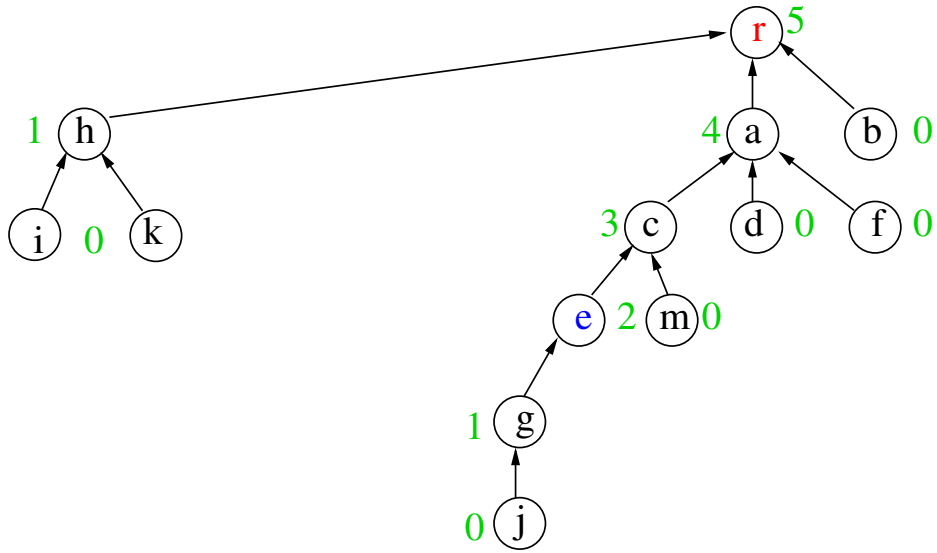
Path compression: Function

FIND (x)
if $(x \neq p(x)$ **then**
    $p(x) =$ **FIND** $p(x)$
    return $p(x)$
end if



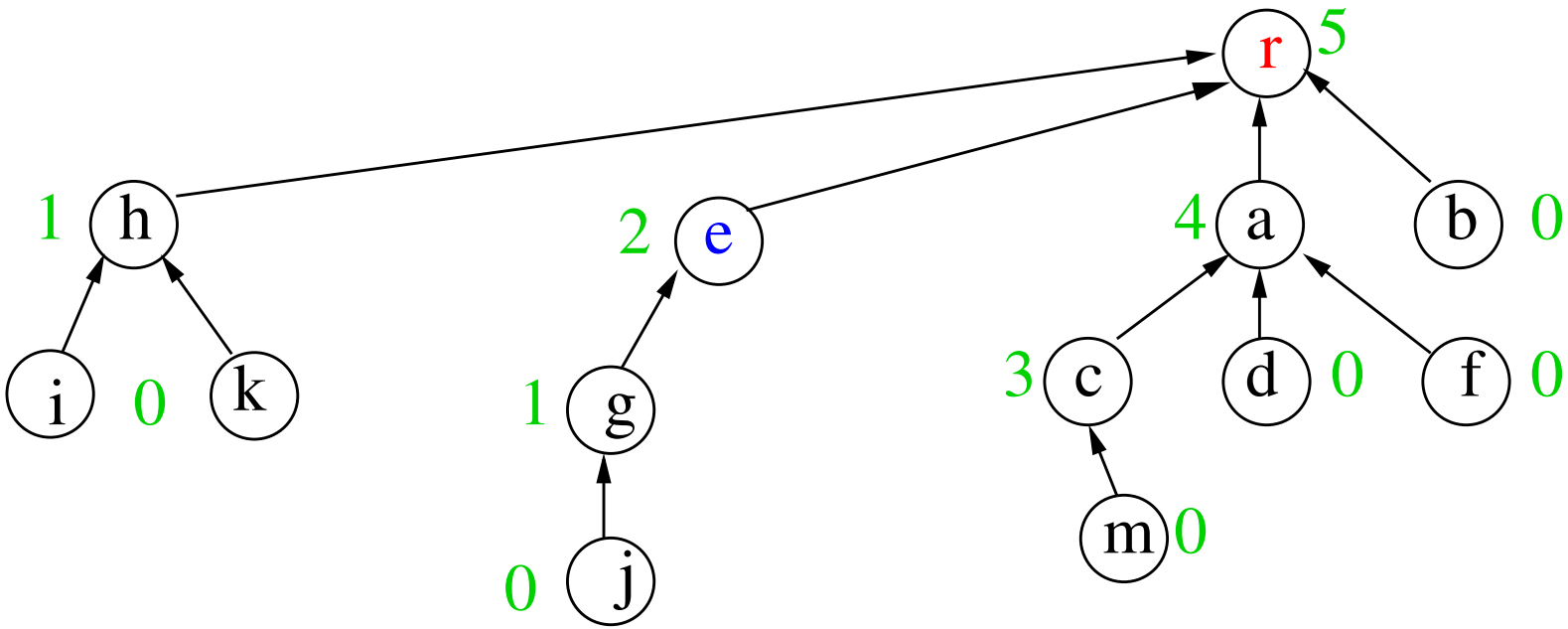
Path compression: Function

FIND (x)
if $(x \neq p(x)$ **then**
    $p(x) =$ **FIND** $p(x)$
    return $p(x)$
end if



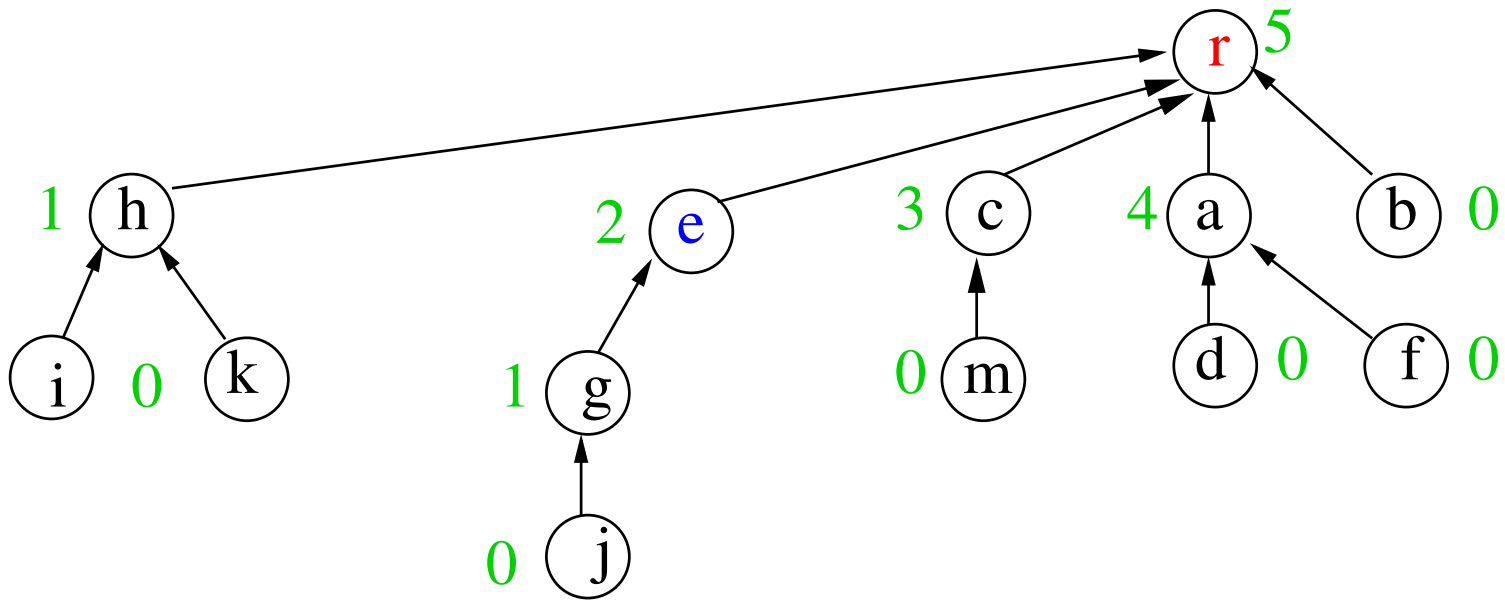
Path compression: Function

FIND (x)
if $(x \neq p(x)$ **then**
  $p(x) =$ **FIND** $p(x)$
  return $p(x)$
end if



Path compression: Function

FIND (x)
if $(x \neq p(x)$ **then**
  $p(x) =$ **FIND** $p(x)$
  return $p(x)$
end if



Union-Find: Link by rank with path compression

This implementation of the data structure is the one that reduces the complexity of making a sequence of m UNION and FIND operations.

Key Observation: Path compression does not create new roots, change ranks, or move elements from one tree to the another

As a corollary, the properties 1 to 5 of Link by rank also hold for this implementation.

Notice, FIND operations only affect the inside nodes, while Union operations only affect roots.
Thus compression has no effect on UNION operations.

Iterated logarithm

The iterated logarithm is defined:

$$\lg^*(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 & \text{if } n = 2 \\ 1 + \lg^*(\lg(n)) & \text{if } n > 2. \end{cases}$$

n	$\lg^* n$
1	0
2	1
[3,4]	2
[5,16]	3
[17,65536]	4
$[65537, 10^{19728}]$	5

Therefore, $\lg^*(n) \leq 5$ for all $n \leq 2^{2^{16}} \sim 10^{19728}$.

If we consider that in any computer, each memory bit has size ≥ 1 atoms, using the canonical estimation that the number of atoms in the universe is $\leq 10^{83}$, we can conclude the size of any computer memory is $< 10^{83}$. Therefore it is impossible with today technology, that you can manipulate sets of size 10^{83}

$\Rightarrow$ for all practical purposes we can consider $\lg^*(n) \leq 5$.

Main result

Theorem
Starting from an empty data structure with n disjoint single sets, link-by-rank with path compression performs any intermixed sequence of m **FIND** *and* **UNION** *operations in $O(m \lg^* n)$ steps.*

The proof uses an amortized analysis argument: look at the sequence of FIND and UNION operations from an empty and determine the average time per operation. The amortized costs turns to be $\lg^* n$ (basically constant) instead of $\lg n$.

The detailed argument of the proof is outside the scope of the course. The details could be found in Sect. 21.4 of Cormen, Leiserson, Rivest, Stein or Sec. 5.1.4 of Dasgupta, Papadimitriou, Vazirani.

Back to Kruskal

Kruskal, grows disjoint partial MST and joints them into a final MST solution.

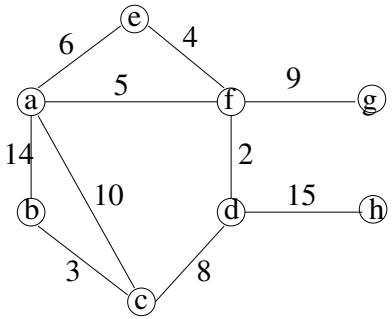
It seems natural apply Union-find to the construction of the MST

MST $(G(V, E), w, r)$, $|V| = n, |E| = m$
Sort E by increasing weight: $\{e_1, \ldots, e_m\}$
$T := \emptyset$
for all $v \in V$ **do**
  MAKESET(v)
end for
for $i = 1$ **to** m **do**
  Chose $e_i = (u, v)$ in order from E
  if FIND(x) $\neq$ Find(y) **then**
    $T := T \cup \{e_i\}$
    UNION(u, v)
  end if
end for

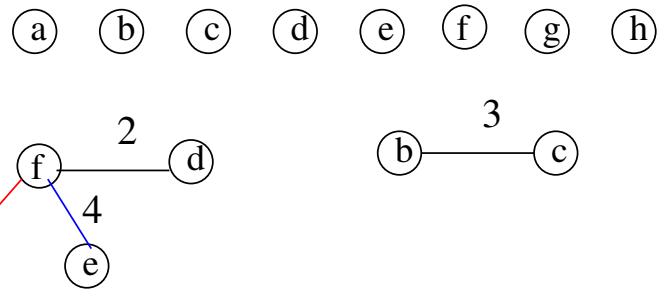
If we use the link-by-rank with path compression implementation, the disjoint set operations take $O(m \lg^* n) = O(m)$.

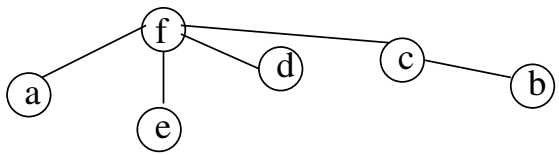
But due to the sorting instruction, the overall complexity of Kuskal still is $O(m \lg n)$, unless we use a range of weight that allow us to use RADIX.

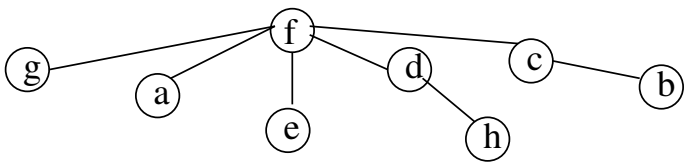
Example of Kruskal using Union-Find



$$E = \{(f,d),(c,b),(e,f),(a,f),(a,e),(c,d),(f,g),(a,c),(a,b),(d,h)\}$$







Greedy and Approximations algorithms

Many times the Greedy strategy yields a local feasible solution with value which is near to the optimum solution.
In many practical cases, when finding the global optimum is hard, it is sufficient to find a good local approximation.

Given an optimization problem (maximization or minimization) an optimal algorithm computes the best output $\mathrm{OPT}(e)$ on any instance e of size n.
An approximation algorithm for the problem computes any valid output.

We want to design approximation algorithms, that are fast and in worst case get an output as close as possible to $\mathrm{OPT}(e)$.

Greedy and Approximations algorithms

Given an optimization problem, an α-approximation algorithm $\mathcal{A}px$ computes a worst case output $\mathcal{A}px(e)$, whose cost is within an $\alpha \geq 1$ factor of OPT (e):

$$\frac{1}{\alpha} \leq \frac{\mathcal{A}px(e)}{\text{OPT}\,(e)} \leq \alpha.$$

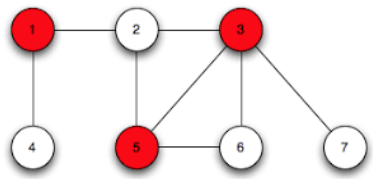
α is denotes as the approximation ratio.

Notice, α measures the factor by which the output of $\mathcal{A}px$ exceeds OPT (e) , on a worst-case input.

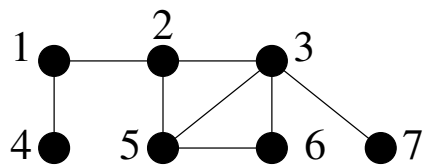
The first $\leq$ works for maximization and the second $\leq$ works for minimization.

An easy example: Vertex cover

Given a graph $G = (V, E)$ with $|V| = n, |E| = m$ find the minimum set of vertices $S \subseteq V$ such that it covers every edge of G.

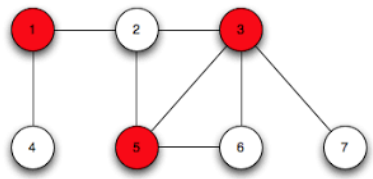


GreedyVC $G = (V, E)$
$E' = E$, $S = \emptyset$,
while $E' \neq \emptyset$ **do**
  Pick $e \in E'$, say $e = (u, v)$
  $S = S \cup \{u, v\}$,
  $E' = E' - \{(u, v) \cup \{\text{edges incident to } u, v\}\}$
end while
return S.

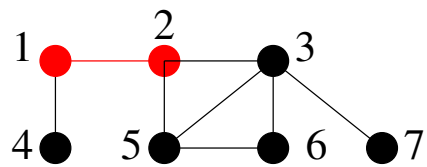


An easy example: Vertex cover

Given a graph $G = (V, E)$ with $|V| = n, |E| = m$ find the minimum set of vertices $S \subseteq V$ such that it covers every edge of G.

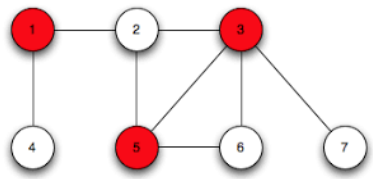


GreedyVC $G = (V, E)$
$E' = E$, $S = \emptyset$,
while $E' \neq \emptyset$ **do**
  Pick $e \in E'$, say $e = (u, v)$
  $S = S \cup \{u, v\}$,
  $E' = E' - \{(u, v) \cup \{\text{edges incident to } u, v\}\}$
end while
return S.

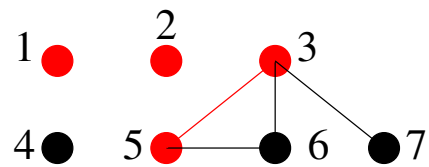


An easy example: Vertex cover

Given a graph $G = (V, E)$ with $|V| = n, |E| = m$ find the minimum set of vertices $S \subseteq V$ such that it covers every edge of G.

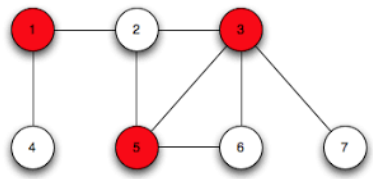


GreedyVC $G = (V, E)$
$E' = E$, $S = \emptyset$,
while $E' \neq \emptyset$ **do**
  Pick $e \in E'$, say $e = (u, v)$
  $S = S \cup \{u, v\}$,
  $E' = E' - \{(u, v) \cup \{\text{edges incident to } u, v\}\}$
end while
return S.

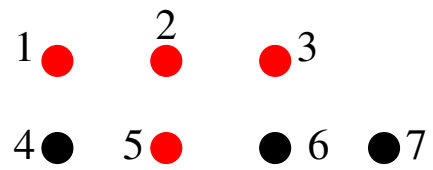


An easy example: Vertex cover

Given a graph $G = (V, E)$ with $|V| = n, |E| = m$ find the minimum set of vertices $S \subseteq V$ such that it covers every edge of G



GreedyVC $G = (V, E)$
$E' = E$, $S = \emptyset$,
while $E' \neq \emptyset$ **do**
  Pick $e \in E'$, say $e = (u, v)$
  $S = S \cup \{u, v\}$,
  $E' = E' - \{(u, v) \cup \{\text{edges incident to } u, v\}\}$
end while
return S.



An easy example: Vertex cover

Theorem

The algorithm $\mathcal{A}px$ runs in $O(m+n)$ steps. Moreover, $|\mathcal{A}px(e)| \leq 2|OPT(e)|$.

Proof.

We use induction to prove $|\mathcal{A}px(e)| \leq 2|\mathrm{OPT}(e)|$. Notice for every $\{u, v\}$ we add to $\mathcal{A}px(e)$, either u or v are in $\mathrm{OPT}(e)$.
Base: If $V = \emptyset$ then $|\mathcal{A}px(e)| = |\mathrm{OPT}(e)| = 0$.
Hipothesis: $|\mathcal{A}px(e) - \{u, v\}| \leq 2|\mathrm{OPT}(e) - \{u, v\}|$. Then,

$$\begin{aligned} |\mathcal{A}px(e)| &= |\mathcal{A}px(e) - \{u, v\}| + 2 \leq 2|\mathrm{OPT}(e) - \{u, v\}| + 2 \\ &\leq 2(|\mathrm{OPT}(e)| - 1) + 2 \leq 2|\mathrm{OPT}(e)|. \end{aligned}$$

□

The decision problem for Vertex Cover is NP-complet. Moreover, unless P=NP, vertex cover can't be approximated within a factor $\alpha \leq 1.36$

Clustering problems

Clustering: process of finding interesting structure in a set of data.
Given a collection of objects, organize them into coherent groups with respect to some (distance function $d(\cdot, \cdot)$).
This not necessarily has to be the physical (Euclidean) distance, it could be similarity distance, time to travel, but it must be a metric, i.e.
Recall if d is a metric: $d(x, x) = 0, d(x, y) > 0$ for $x \neq y$, $d(x, y) = d(y, x)$, $d(x, y) > 0$ and $d(x, y) + d(y, z) \leq d(x, z)$.

k-clustering Problem: Given a set of points $X = \{x_1, x_2, \ldots, x_n\}$ together with a distance function on X and given a $k > 0$, want to partition X into k disjoint subsets, a k-clustering, such as to optimize some function (depending on d).

The k-Center clustering problem

Given as input a set of $X = \{x_1, \ldots, x_n\}$, with distances $D = \{d(x_i, x_j)\}$ and a given integer k:
Find the partition X into k **clusters** $\{C_1, \ldots, C_k\}$ such as to minimize the diameter of the clusters, $\min_j \max_{x,y \in C_j} d(x, y)$.

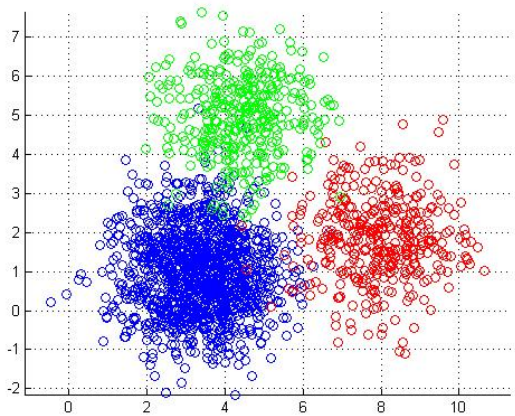
Each ball C_i will be determine by a center c_i and a radius r. Let $C = \{c_1, \ldots, c_k\}$ be the set of centers and $r = r(C)$.

Define $\mathcal{C}$ to be a r-cover for X if $\forall x \in X$, $\exists c_j \in C$ s.t. $d(x, c_j) \leq r$.

The k-Center clustering problem

Equivalent statement of the problem: Given as input (X, D, k), select the centers $C = \{c_1, \ldots, c_k\}$, and $r = r(C)$ such that the resulting $\{C_1, \ldots, C_k\}$ is an r-cover for X, with r as small as possible.

Formal definition of k-center: Given $X \subset \mathbb{Z}^2$ points and $k \in \mathbb{Z}$, compute the set $C = \{c_1, \ldots, c_k\}$ of centers $C \subset X$ such that if $\tilde{X} = X \backslash C$, it maximizes $\min_{x \in \tilde{X}} d(x, C)$.



The k-Center clustering problem: Complexity

For $k > 2$, the decision version of the k-center clustering problem is NP-complete.

There is a deterministic algorithm working in $O(n^k)$. (Can you design one?)

For $k = 1$: Find the smallest radius disk enclosing a point set
The problem can be solved in $O(n \lg n)$ (How?)

The k-Center clustering problem: Greedy algorithm

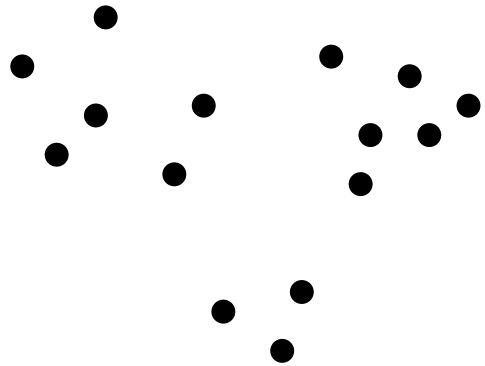
The algorithm iterates k times, at each iteration choose a new center, add a new cluster and it refines the radius r_i of the cluster balls. T. Gonzalez (1985)

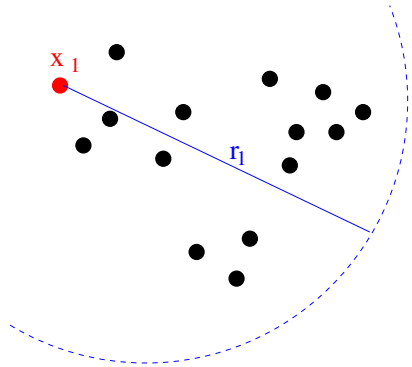
1. Choose arbitrarily x and make $c_1 = x$. Let $C_1 = \{c_1\}$
2. For all $x_i \in X$ let $d_1[x] = d(x_i, c_1)$.
3. Choose $c_2 = x_j$ s.t. $\max\min_{x \in X} d_1[x]$.
4. Let $r_1 = d(c_1, c_2)$ and $C_2 = \{c_1, c_2\}$.
5. For $i = 2$ to k
   - 5.1 At interaction $i + 1$: Let c_{i+1} be the element in $X \backslash C_i$ that maximizes the distances to C_i.
   - 5.2 Let $C_{i+1} = \{c_1, c_2, \ldots, c_{i+1}\}$ and $r_i = \max\min_{j \leq i} d(c_{i+1}, c_j)$,
6. Output the $C = \{c_1, \ldots, c_k\}$ centers and r_k.

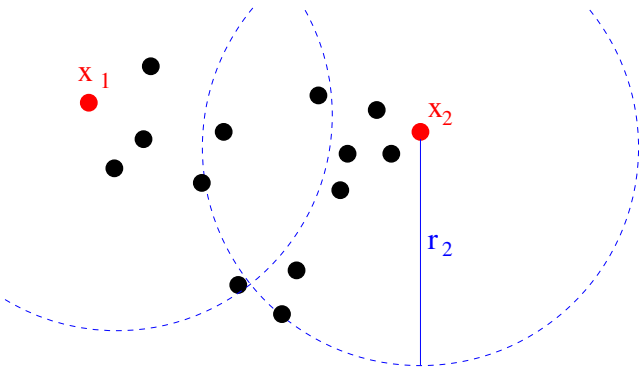
The max min means the max of the P2P distances.

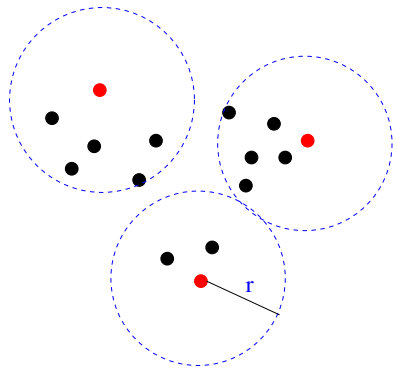
Greedy algorithm: Example

Given X, $k = 3$ and the n^2 distance vector D:









Greedy algorithm: Complexity

We have the set X of points and all their $O(n^2)$ distances. We assume we have a data structure that keeps ordered the set of distances D, so we can and it is quick to retrieve quickly any distance between points in X. How?

- At each step i we have to compute the distance from all $x \in X$ to all current centers $c \in C_{i-1}$, and choose the new c_i and r_i, but
- For each $x \in$ define $d_i[x] = d(x, C_i) = \min\{d_{i-1}[x], \underbrace{d(x, c_i)}_{(*)}\}$
- Therefore at each step, to compute r_i we need to update $(*)$.
- At iteration i, choosing c_i and computing r_i takes $O(n)$ steps, therefore the complexity of the greedy algorithm is $O(kn)$ steps.

Approximation to the k-center problem

Theorem

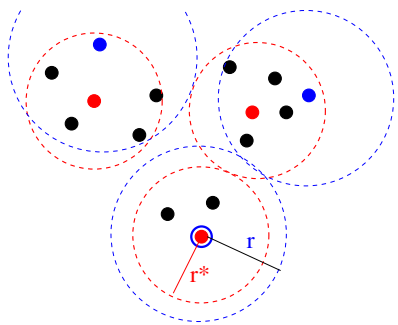
The the resulting diameter in the previous greedy algorithm is an approximation algorithm to the k-center problem, with an approximation ratio of $\alpha = 2$.

(i.e. It returns a set C s.t. $r(C) \leq 2r(C^*)$ where C^* is an optimal set of k-center).

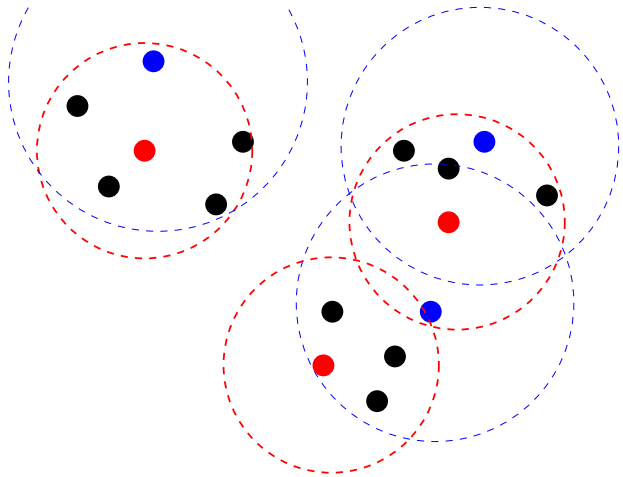
Proof

Let $C^* = \{c_i^*\}_{i=1}^k$ and r^* be the optimal values, and let $C = \{C_i\}_{i=1}^k$ and r the values returned by the algorithm.
Want to prove $r \leq 2r^*$.

Case 1: Every C_j^* covers at least one c_i.
$\Rightarrow$ as $\forall x \in X$, if C_j^* covers x, then
$\exists c_i \in C_j^* \Rightarrow d(x, c_i) \leq 2r^*$.



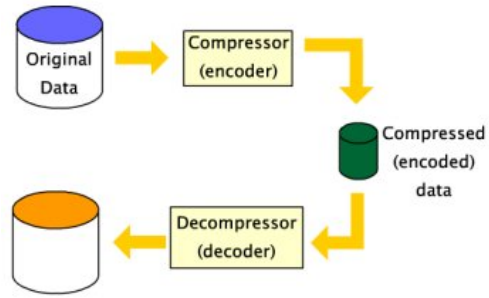
Proof cont.



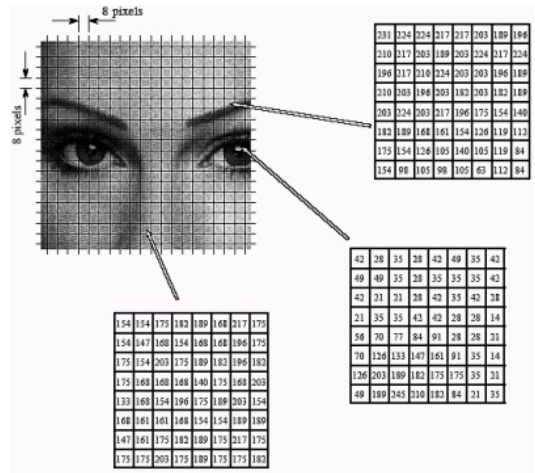
Case 2: At least one C_j^* does not cover any center in C. Then, $\exists C_j^*$ covering at least c_i and $c_j \Rightarrow d(c_i, c_j) \leq 2r^*$.
We need to prove that $d(c_i, c_j) > r$. Wlog assume the algorithm chooses c_j at iteration j and that c_i has been selected as centre in a previous ith. iteration, then $d(c_i, c_j) > r_j$.
Moreover, notice than $r_1 > r_2 > \ldots > r_k = r$,
therefore $d(c_i, c_j) > r_j \geq r$ and $r \leq d(c_i, c_j) \leq 2r^*$ □

Data Compression

INPUT: Given a text $\mathcal{T}$ over an finite alphabet Σ
QUESTION: Represent $\mathcal{T}$ with as few bits as possible.

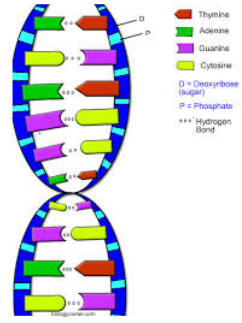


The goal of data compression is to reduce the time to transmit large files, and to reduce the space to store them.
If we are using variable-length encoding we need a system easy to encode and decode.



Example.

$$\underbrace{AAACAGTTGCAT \cdots GGTCCCTAGG}_{130.000.000}$$



- *Fixed-length encoding*: $A = 00$,$C = 01$, $G = 10$ and $T = 11$. Needs 260Mbites to store.
- *Variable-length encoding*: If A appears 7×10^8 times, C appears 3×10^6 times, G 2×10^8 and T 37×10^7, better to assign a shorter string to A and longer to C

Prefix property

Given a set of symbols Σ, a prefix code, is $\phi : \Sigma \to \{0,1\}^+$ (symbols to chain of bits) where for distinct $x, y \in \Sigma$, $\phi(x)$ is not a prefix of $\phi(y)$.

If $\phi(A) = 1$ and $\phi(C) = 101$ then ϕ is no prefix code.

$\phi(A) = 1, \phi(T) = 01, \phi(G) = 000, \phi(C) = 001$ is prefix code.

Prefix codes easy to decode (left-to-right):
00010110011010000101

$$\underbrace{000}_{G}\ \underbrace{1}_{A}\ \underbrace{01}_{T}\ \underbrace{1}_{A}\ \underbrace{001}_{C}\ \underbrace{1}_{A}\ \underbrace{01}_{T}\ \underbrace{000}_{G}\ \underbrace{001}_{C}\ \underbrace{01}_{T}$$

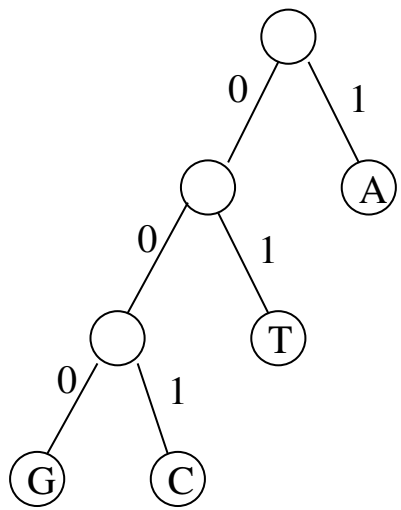
Prefix tree.

Represent encoding with prefix property as a binary tree, the prefix tree:

A prefix tree T is a binary tree with the following properties:

- One leaf for symbol,
- Left edge labeled 0 and right edge labeled 1,
- Labels on the path from the root to a leaf specify the code for that leaf.

For $\Sigma = \{A, T, G, C\}$



Frequency.

To find an efficient code, first given a text S on Σ, with $|S| = n$, first we must find the frequencies of the alphabet symbols.

$\forall x \in \Sigma$, define the frequency

$$f(x) = \frac{\text{number occurrencies of } x \in S}{n}$$

Notice: $\sum_{x \in \Sigma} f(x) = 1$.
Given a prefix code ϕ, which is the total length of the encoding?
The encoding length of S is

$$B(S) = \sum_{x \in \Sigma} nf(x)|\phi(x)| = n \underbrace{\sum_{x \in \Sigma} f(x)|\phi(x)|}_{\alpha}.$$

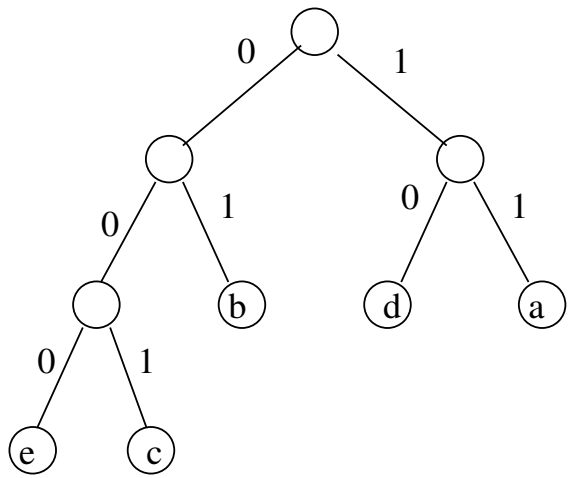
Given ϕ, $\alpha = \sum_{x \in \Sigma} f(x)|\phi(x)|$ is the average number of bits required per symbol.

In terms of prefix tree of ϕ, given x and $f(x)$, the length of the codeword $|\phi(x)|$ is also the depth of x in T, let us denote it by $d_x(T)$.

Let $B(T) = \sum_{x \in \Sigma} f(x) d_x(T)$.

Example.

Let $\Sigma = \{a, b, c, d, e\}$ and let S be a text over Σ.
Let $f(a) = .32, f(b) = .25, f(c) = .20, f(d) = .18, f(e) = .05$
If we use a fixed length code we need $\lceil \lg 5 \rceil = 3$ bits.
Consider the prefix-code ϕ_1:
$\phi_1(a) = 11, \phi_1(b) = 01, \phi_1(c) = 001, \phi_1(d) = 10, \phi_1(e) = 000$

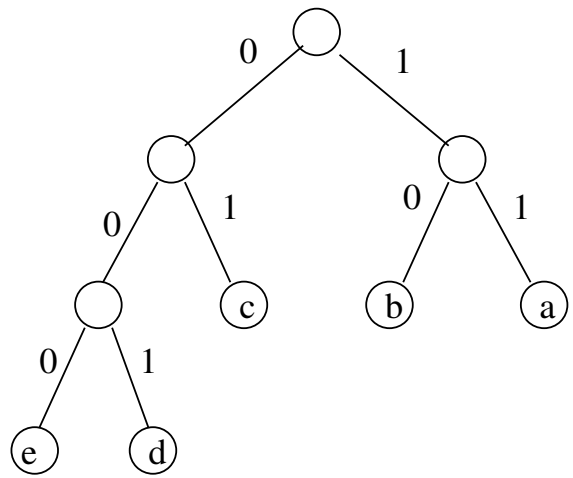


$\alpha = .32 \cdot 2 + .25 \cdot 2 + .20 \cdot 3 + .18 \cdot 2 + .05 \cdot 3 = 2.25$
In average, ϕ_1 reduces the bits per symbol over the fixed-length code from 3 to 2.25, about 25%

Is that the maximum reduction?

Consider the prefix-code ϕ_2:

$\phi_2(a) = 11, \phi_2(b) = 10, \phi_2(c) = 01, \phi_2(d) = 001, \phi_2(e) = 000$



$\alpha = .32 \cdot 2 + .25 \cdot 2 + .20 \cdot 2 + .18 \cdot 3 + .05 \cdot 3 = 2.23$

is that the best? (the maximal compression)

Optimal prefix code.

Given a text, an optimal prefix code is a prefix code that minimizes the total number of bits needed to encode the text.

Note that an optimal encoding minimizes α.

Intuitively, in the T of an optimal prefix code, symbols with high frequencies should have small depth ans symbols with low frequency should have large depth.

The search for an optimal prefix code is the search for a T, which minimizes the α.

Characterization of optimal prefix trees.

A binary tree T is full if every interior node has two sons.

Lemma

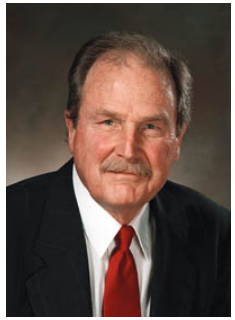
The binary prefix tree corresponding to an optimal prefix code is full.

Proof.

Let T be the prefix tree of an optimal code, and suppose it contains a u with a son v.
If u is the root, construct T' by deleting u and using v com root.
T' will yield a code with less bits to code the symbols.
Contradiction to optimality of T.
If u is not the root, let w be the father of u. Construct T' by deleting u and connecting directly v to w. Again this decreases the number of bits, contradiction to optimality of T. $\square$

Greedy approach: Huffman code

Greedy approach due to David Huffman (1925-99) in 1952, while he was a PhD student at MIT



Wish to produce a labeled binary full tree, in which the leaves are as close to the root as possible. Moreover symbols with low frequency will be placed deeper than the symbol with high frequency.

Greedy approach: Huffman code

- Given S assume we computed $f(x)$ for every $x \in \Sigma$
- Sort the symbols by increasing f. Keep the dynamic sorted list in a priority queue Q.
- Construct a tree in bottom-up fashion, take two first elements of Q join them by a new *virtual node* with f the sum of the f's of its sons, and place the new node in Q.
- When Q is empty, the resulting tree will be prefix tree of an optimal prefix code.

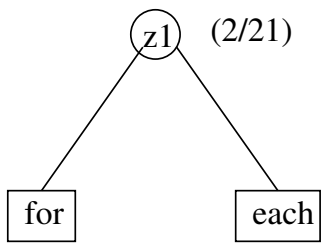
Huffman Coding: Construction of the tree.

Huffman Σ, S
Given Σ and S {compute the frequencies $\{f\}$}
Construct priority queue Q of Σ, ordered by increasing f
while $Q \neq \emptyset$ **do**
  create a new node z
  $x =$Extract-Min (Q)
  $y =$Extract-Min (Q)
  make x, y the sons of z
  $f(z) = f(x) + f(y)$
  Insert (Q, z)
end while

If Q is implemented by a Heap, the algorithm has a complexity $O(n \lg n)$.

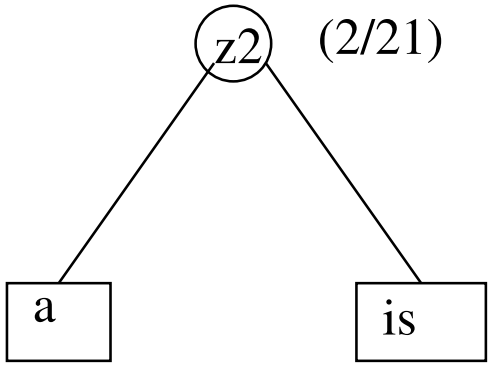
Example

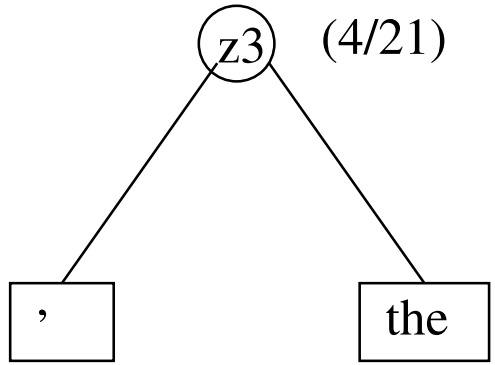
Consider the text: *for each rose, a rose is a rose, the rose*.
with $\Sigma = \{$for/ each/ rose/ a/ is/ the/ ,/ b$\}$
Frequencies: f(for) $= 1/21$, f(rose) $= 4/21$, f(is) $= 1/21$,
f(a) $= 2/21$, f(each) $= 1/21$, f(,) $= 2/21$, f(the) $= 1/21$,
f(b) $= 9/21$.
Priority Queue:
Q=(for(1/21), each(1/21), a(1/21), is(1/21), ,(2/21), the(2/21),
rose(4/21), b(9/21))

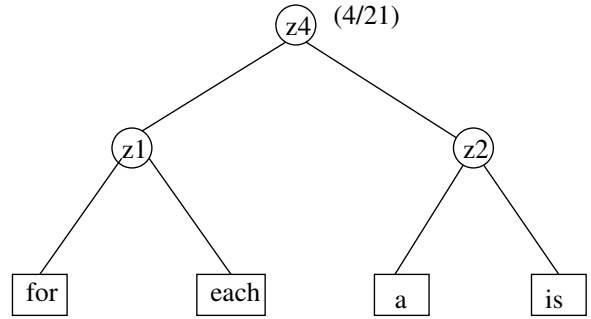


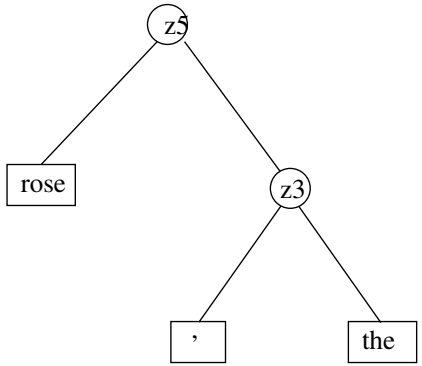
Q=(a(1/21), is(1/21), ,(2/21), the(2/21), z1(2/21), rose(4,21),
b(9/21))

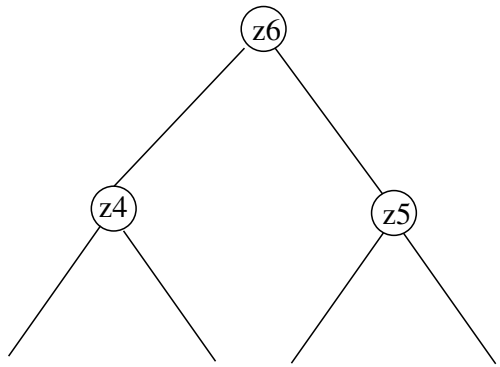
Example.

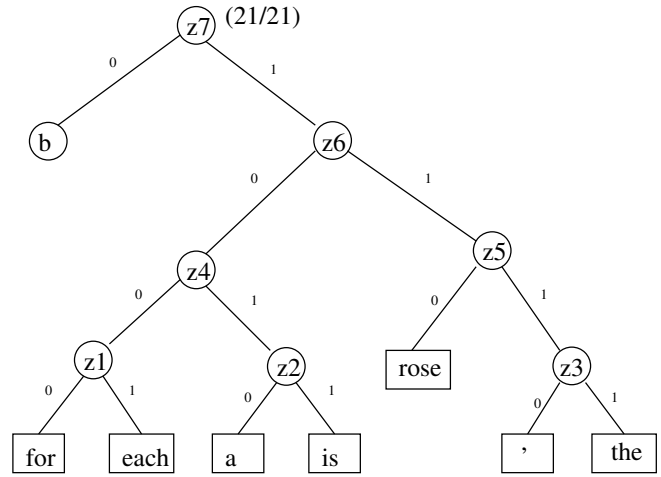

z2
(2/21)
a
is


z3
(4/21)
,
the


z4
(4/21)
z1
z2
for
each
a
is


z5
rose
z3
,
the


z6
z4
z5


z7
(21/21)
0
1
b
z6
0
1
z4
z5
0
1
0
1
z1
z2
rose
z3
0
1
0
1
0
1
for
each
a
is
,
the

Example

Therefore *for each rose, a rose is a rose, the rose* is Huffman codified:
100001001011011100101001100101101010011011100111110110

Notice with a fix code we will use 4 bits per symbol $\Rightarrow$ 84 bits instead of the 53 we use.

The solution is not unique!

Why does the Huffman's algorithm produce an optimal prefix code?

Correctness.

Theorem (Greedy property)

Let Σ be an alphabet, and x, y two symbols with the lowest frequency. Then, there is an optimal prefix code in which the code for x and y have the same length and differ only in the last bit.

Proof.

For T optimal with a and b sibilings at max. depth. Assume $f(b) \leq f(a)$. Construct T' by exchanging x with a and y with b. As $f(x) \leq f(a)$ and $f(y) \leq f(b)$ then $B(T') \leq B(T)$. $\square$

Theorem (Optimal substructure)

Assume T' is an optimal prefix tree for $(\Sigma - \{x, y\}) \cup \{z\}$ where x, y are symbols with lowest frequency, and z has frequency $f(x) + f(y)$. The T obtained from T' by making x and y children of z is an optimal prefix tree for Σ.

Proof.

Let T_0 be any prefix tree for Σ. Must show $B(T) \leq B(T_0)$.
We only need to consider T_0 where x and y are siblings. Let T_0' be obtained by removing x, y from T_0. As T_0' is a prefix tree for $(\Sigma - \{x, y\}) \cup \{z\}$, then $B(T_0') \geq B(T')$.
Comparing T_0 with T_0' we get,
$B(T_0') + f(x) + f(y) = B(T_0)$ and $B(T') + f(x) + f(y) = B(T)$,
Putting together the three identities, we get $B(T) \leq B(T_0)$. $\square$

Optimality of Huffman

Huffman is optimal under assumptions:

- The compression is lossless, i.e. *uncompressing the compressed file yield the original file*.
- We must know the alphabet beforehand (characters, words, etc.)
- We must pre-compute the frequencies of symbols, i.e. read the data twice

For certain applications is very slow (on the size n of the input text)