Greedy Algorithms and Data Compression.

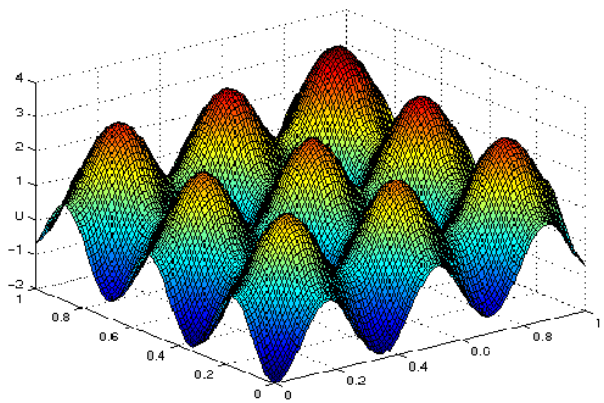
Curs 2015

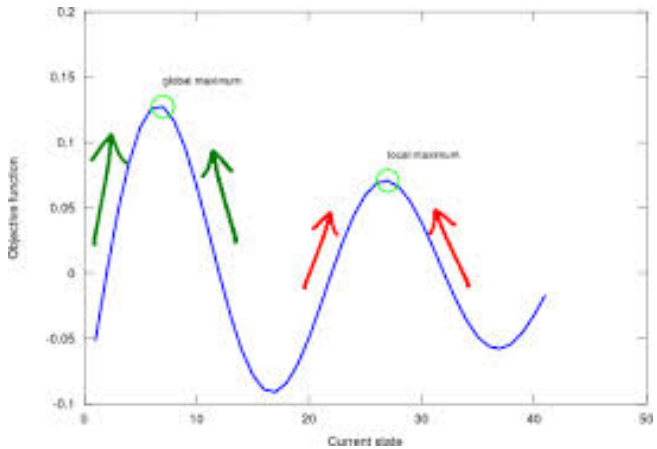
Greedy Algorithms

An optimization problem: Given of $(\mathfrak{S}, f)$, where $\mathfrak{S}$ is a set of feasible elements and $f : \mathfrak{S} \to \mathbb{R}$ is the objective function, find a $u \in \mathfrak{S}$, which optimizes (maximizes o minimizes) f.

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

Greedy algorithms do not always yield optimal solutions, but for some problems they are very efficient.





Greedy Algorithms

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

Greedy Algorithms

For the greedy strategy to work well, 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.

Buying of n software licenses

A company needs to obtain licenses for n different pieces of software. Due to budged restriction, they can only obtain at most one license per month. Each license is currently selling for 100 euros, however they become more expensive each month.

If licenses are labelled as $1, 2, 3 \ldots, n$, the cost of license j increases by a factor $r_j > 1$ each month (so if the company buys license j in t months, the cost will be $100 \cdot r_j^t$). Assume all price rates are distinct ($r_i \neq r_j$ for $i \neq j$).

INPUT: Given a set of monthly appreciations $r_1, r_2, \ldots, r_n$

QUESTION: Compute the order in which to buy the licenses so that the total amount of money spend is minimized.

The buying of n software licenses: Solution

Idea: Sort $\{r_i\}$! Increasing or decreasing order?
Try by small example: let $r_1 = 2, r_2 = 3, r_3 = 4$:
Buying in increasing order has cost $100(2 + 3^2 + 4^3) = 7500$,
in decreasing order has cost $100(4 + 3^2 + 2^3) = 2100$

Therefore increasing order not the best!
So the greedy algorithm seems to be sorting $\{r_i\}$ by decreasing order.

Buy now the license with highest r, next month the license with the second most expensive r, etc.

The complexity of this algorithm is: $T(n) = O(n \lg n)$

Correctness

Proof by contradiction

Let S the solution given by the greedy algorithm, let O be a different optimal solution.

As $S \neq O$ there are at least two months t and $t+1$ s.t. for S:$r_t < r_{t+1}$.

The cost of months t and $t+1$ for O: $100(r_t^t + r_{t+1}^{t+1})$

If we use the S algorithm: $100(r_{t+1}^t + r_t^{t+1})$

Therefore: as $r_i > 0 \ \forall i$ and $r_i < r_{i+1}$.

$r_t^t + r_{t+1}^{t+1} - (r_{t+1}^t + r_t^{t+1}) \Rightarrow r_{t+1}^t(r_{t+1} - 1) - r_t^t(r_t - 1) > 0$

and the cost of the solution S is smaller than the cost of O.

Exercise

Try the greedy approach to a variation of the previous problem:
We want to sell licenses and we must exactly sell one every month.
The cost of each license depreciates $r_i < 1$ per month. (i.e the rates are now < 1 instead of > 1)

Try as input $(3/4, 1/2, 1/100)$
Does the greedy approach work?

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 the items to maximize the profit, within allowed weight W

Example.

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

$Tw = 28$

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 i
while total weight taken $\leq W$ **do**
  Take the maximum amount of the next i
end while

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

Example.

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

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

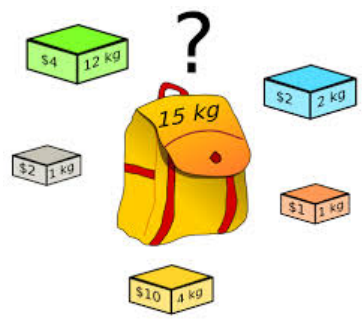
Greedy does not always work (2).

0-1 Knapsack
INPUT:a set $I = \{i\}_1^n$ of 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
Cost	V:	60	100	120
Weight	w:	10	20	30
v/w	:	6	5	4

with



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

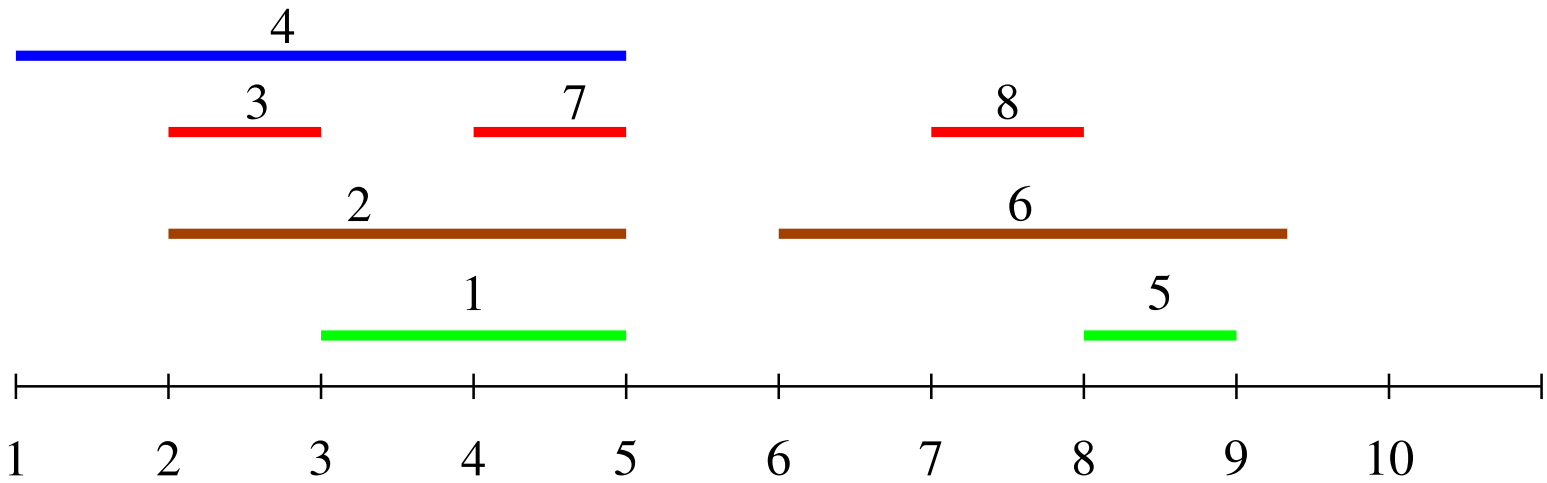
The activity selection problem

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, $\mathfrak{S} = \{A | A \subset S, A \text{ is compatible } \}$ (where A is a subset of activities) and $f(A) = |A|$.

Example.

Activity (A):	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,

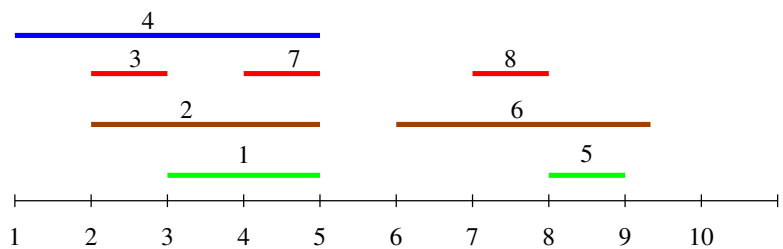
- A local criteria to allow the selection,
- a condition to determine if a partial solution can be completed,
- a procedure to test that we have the optimal solution.

The Activity Selection problem.

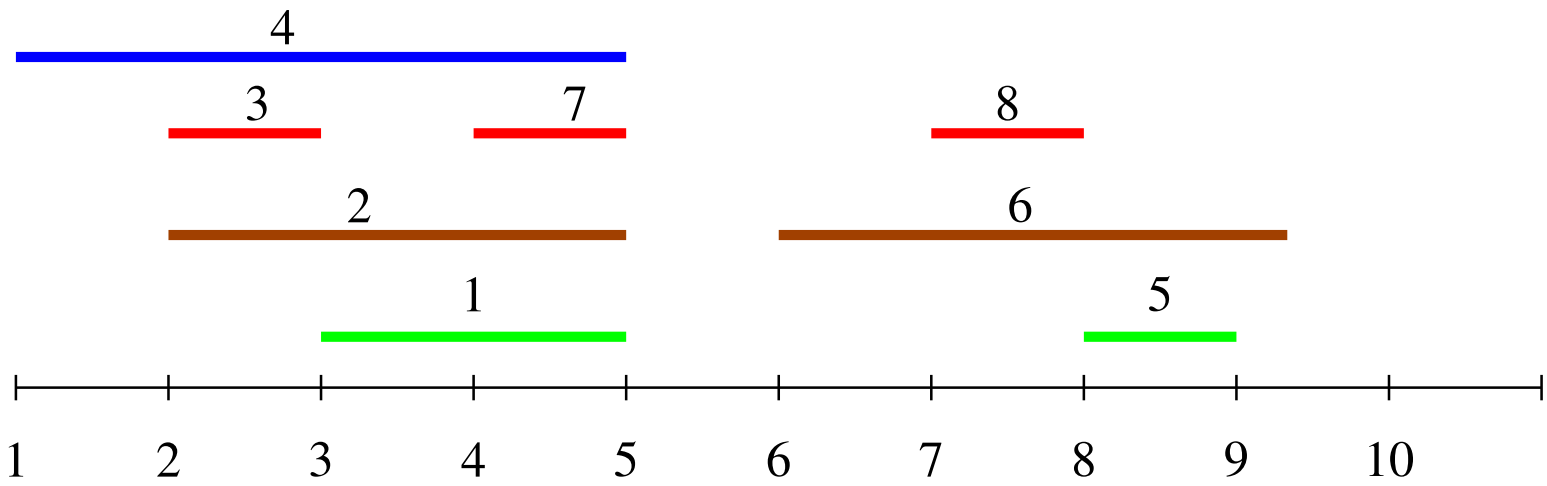
Given a set A of activities, wish to maximize the number of compatible activities.

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 (but the algorithm gives the first one). If we were asking for maximum occupancy 456 will be a solution.

How would you modify the previous algorithm to deal with the 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 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.

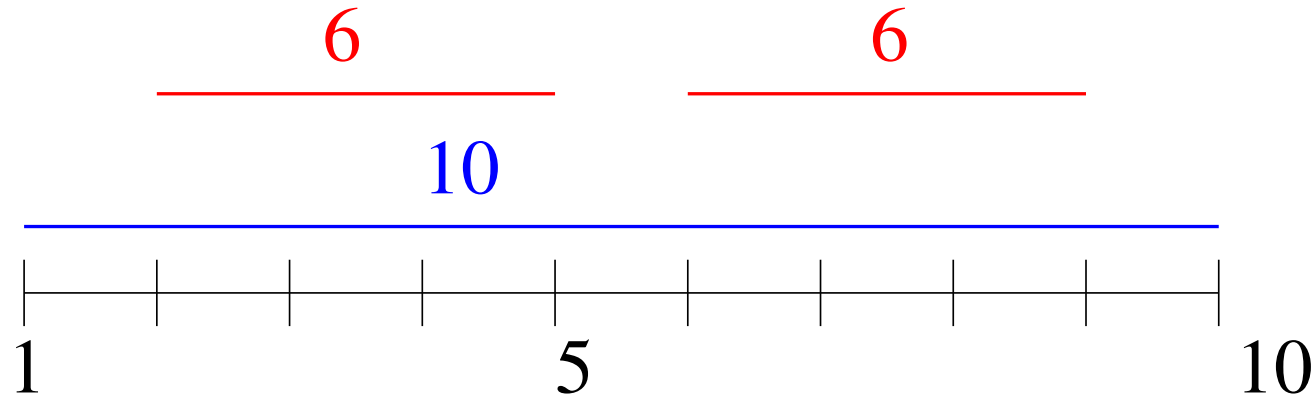
Weighted Activity Selection Problem
INPUT: a set $S = \{1, 2, \ldots, n\}$ of activities to be processed by a single resource. Each activity i has a start time s_i and a finish time f_i, with $f_i > s_i$, and a weight w_i.
QUESTION: Find the set of mutually compatible such that it maximizes $\sum_{i \in S} w_i$

$S := \emptyset$
sort $W = \{w_i\}$by decreasing value
choose the max. weight w_m
add $m = (l_m, u_m)$ to S
remove all intervals overlapping with m
while there are intervals **do**
  repeat the greedy procedure
end while
return S

Prove the correctness of the greedy algorithm

Greedy does not always work.

The previous greedy does not always solve the problem!



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

Minimize Lateness problem

We have a single resource and n requests to use the resource, each request i taking a time t_i.

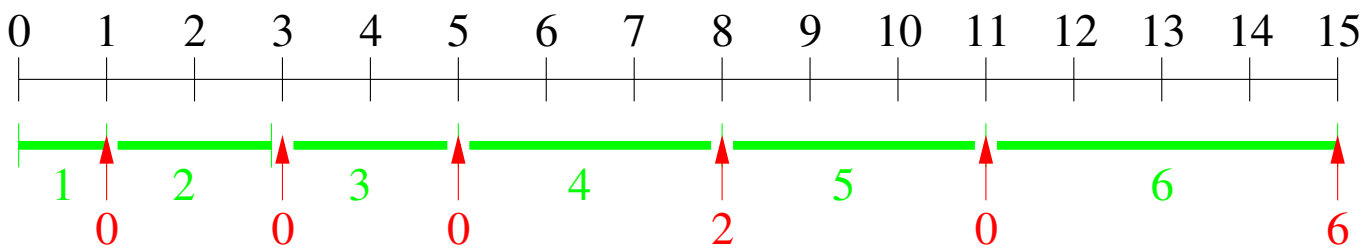
In contrast to the previous problem, each request instead of having an starting and finishing time, it has a deadline d_i. The goal is to schedule the tasks as to minimize the total amount the total time exceeding the deadlines.

Minimize Lateness

- We have a single processor that can process 1 job at the time
- We have n jobs such that job i:
  - requires t_i units of processing time,
  - it has to be finished by time d_i,
  - it starts at s_i (therefore finishes at $f_i = s_i + t_i$)
- define lateness of i: $L_i = \max\{0, f_i - d_i\}$
- we want to $\min_i \max L_i$

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 lateness.



Minimize Lateness

Schedule jobs according to some ordering

(1.-) Sort in increasing order of t_i:

i	t_i	d_i
1	1	50
2	5	5

(2.-) Sort in increasing order of $d_i - t_i$:

i	t_i	d_i
1	1	5
2	2	5

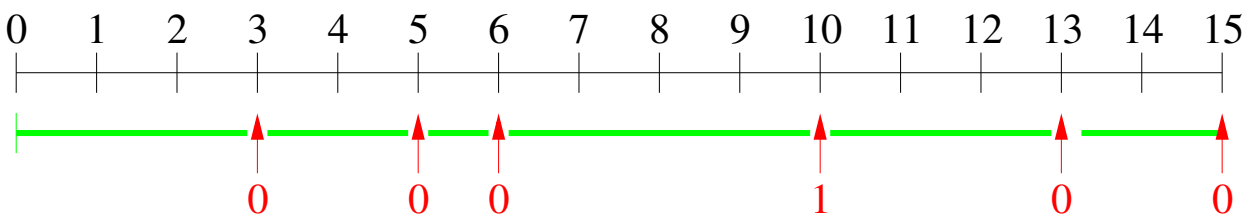
Minimize Lateness: Greedy Algorithm

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

Process urgent jobs first!

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

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



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

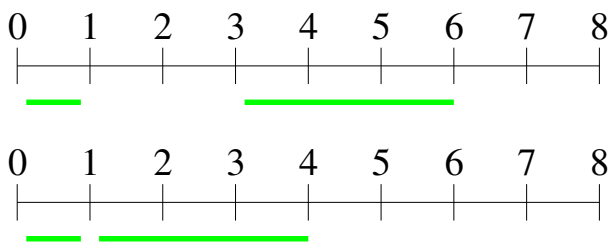
Complexity and idle time

Time complexity

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

Idle steps

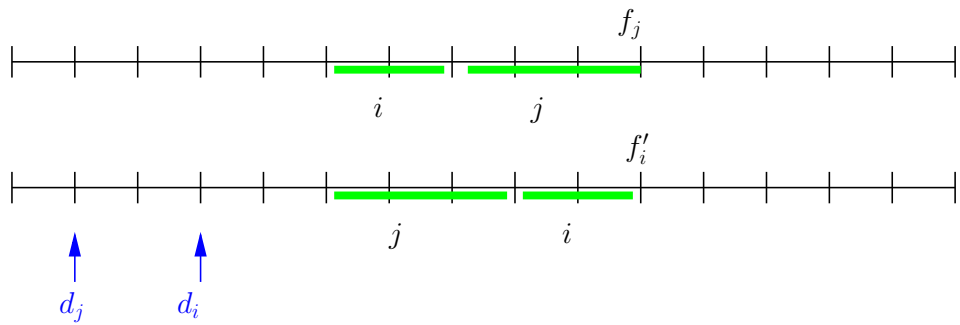
From an optimal schedule with idle steps, we always can eliminate gaps to obtain another optimal schedule:



There exists an optimal schedule with no idle steps.

Inversions and exchange argument

We say an schedule has an inversion if job i is scheduled before job 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.
Note, $\forall k \neq i, j \; L'_k = L_k$ and $L'_i = L_i$
If job j is late: $L'_j = f'_j - d_j = f_i - d_i \leq f_i - d_i \leq L_i$ □

Correctness of LatenessA

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

Theorem

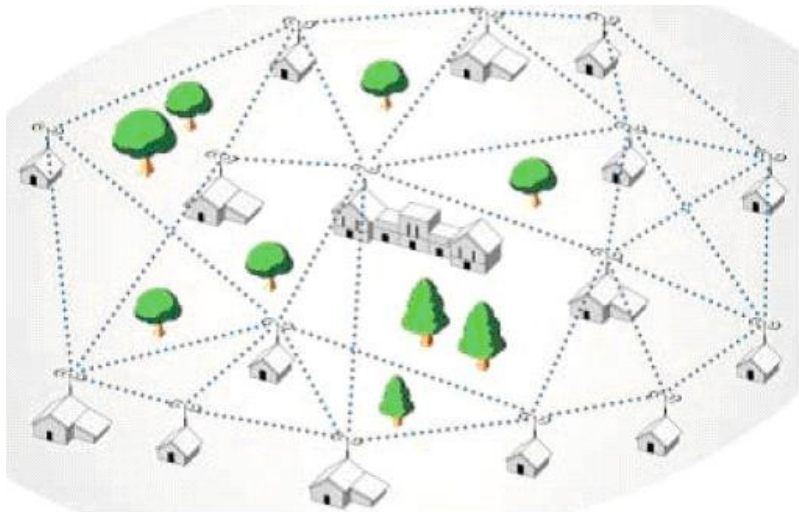
Algorithm LatenessA returns an optimal schedule S.

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

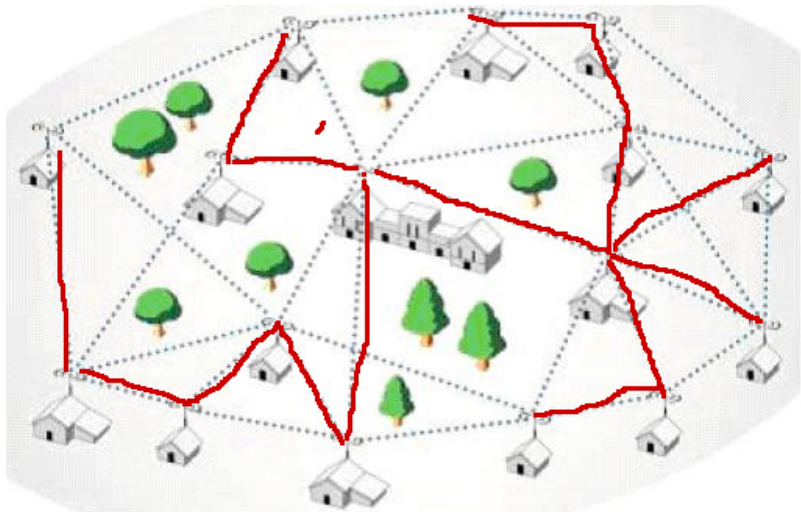
- We have a set of locations $V = \{v_1, \ldots, v_n\}$,
- we want to build a communication network on top of them
- we want that any v_i can communicate with any v_j,
- for any pair (v_i, v_j) there is a cost $w(v_i, v_j)$ of building a direct link,
- if E is the set of all $n^2/2$ possible edges, we 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

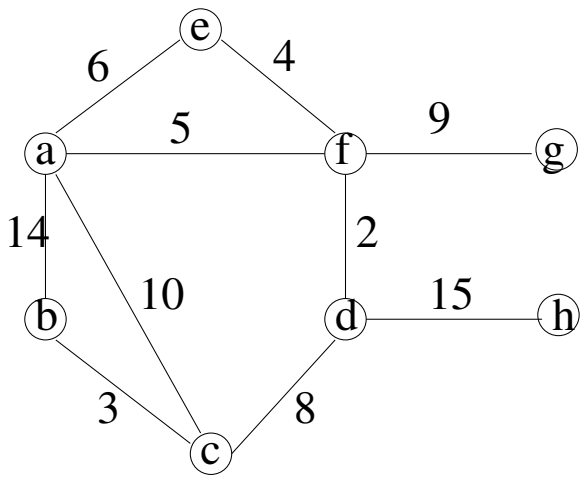
- We have a set of locations $V = \{v_1, \ldots, v_n\}$,
- we want to build a communication network on top of them
- we want that any v_i can communicate with any v_j,
- for any pair (v_i, v_j) there is a cost $w(v_i, v_j)$ of building a direct link,
- if E is the set of all $n^2/2$ possible edges, we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

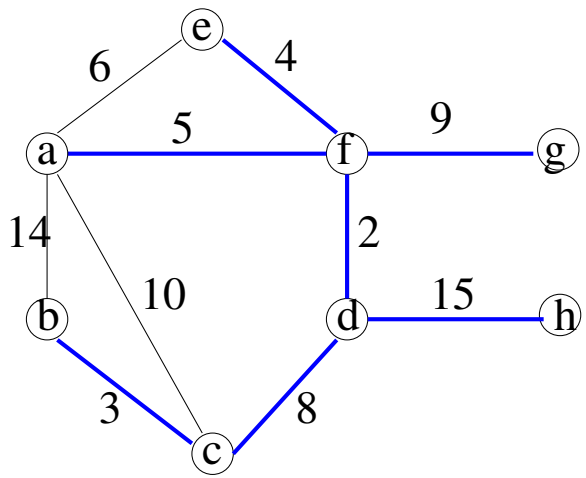
Construct the MST



Minimum Spanning Tree (MST).

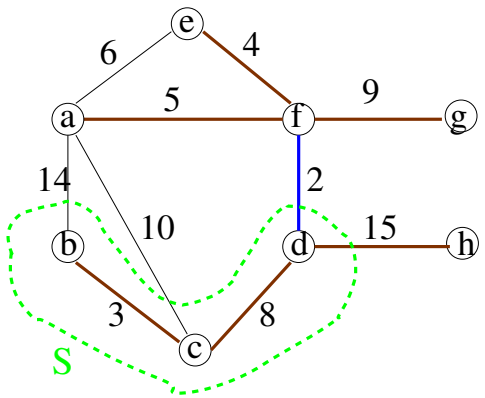
INPUT: An edge weighted graph $G = (V, E)$,
$|V| = n, \forall e \in E, w(e) \in \mathbb{R}$,
QUESTION: Find a tree T with $V(T) = V$ and $E(T) \subseteq E$, such that it minimizes $w(T) = \sum_{e \in E(T)} w(e)$.





Some previous definitions

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

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)

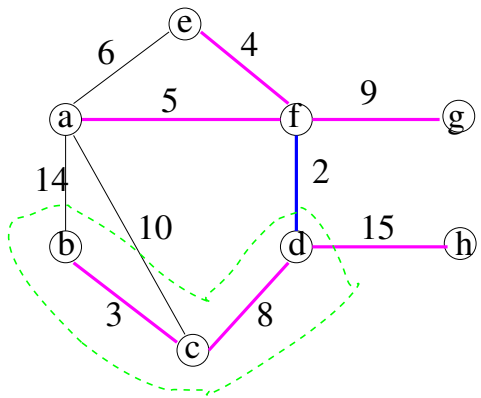
The MST problem has the property of the optimal substructure: Given an optimal MST T, removing middle edge e yields T_1 and T_2 which are optimal.

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 $e \in T$.

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. □



The cycle rule (Red rule)

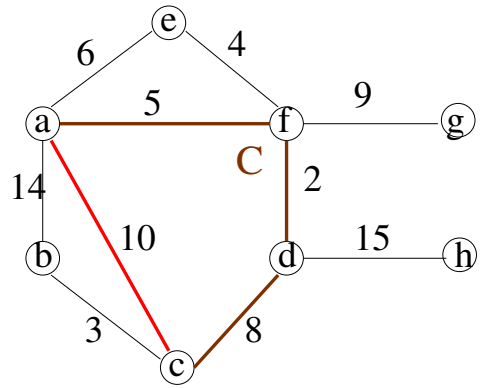
The MST problem has the property that the the optimal solution must be a tree.

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.

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 to . 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. □



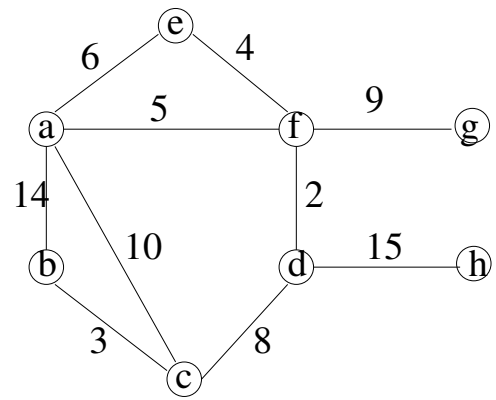
C=cycle spanning {a,c,d,f}

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 a non-colored edge with min weight and paint it blue

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



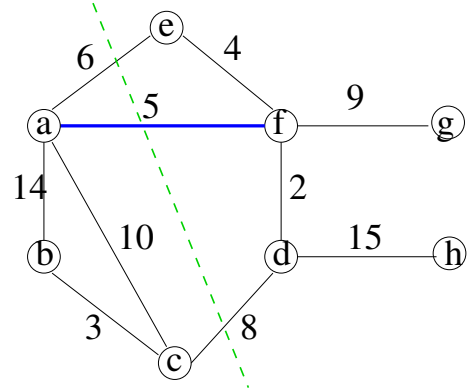
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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 an non-colored edge in C with max weight and paint it blue.



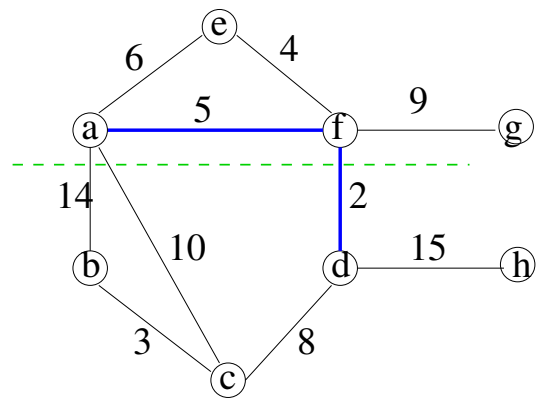
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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 an non-colored edge in C with max weight and paint it blue.



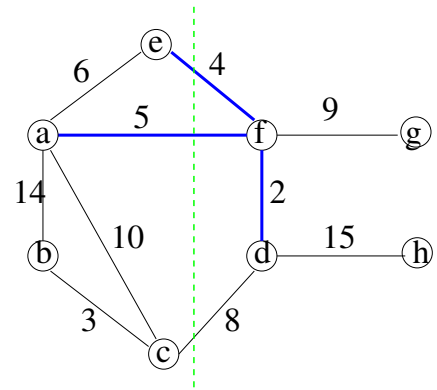
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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 an non-colored edge in C with max weight and paint it blue.



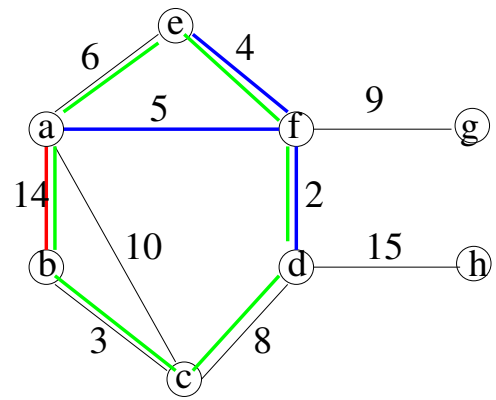
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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 an non-colored edge in C with max weight and paint it blue.



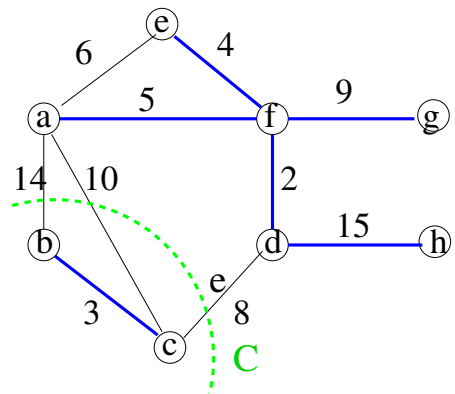
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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 MSP implementation

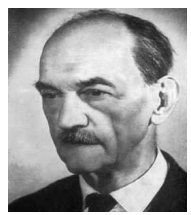
There has been extensive work to obtain the most efficient algorithm to find a MST in a given graph:

- O. Borůvka gave the first greedy algorithm for the MSP in 1926. V. Jarník 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.
- 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 .
- In 1997 B. Chazelle gave an $O(m\alpha(n))$ algorithm.

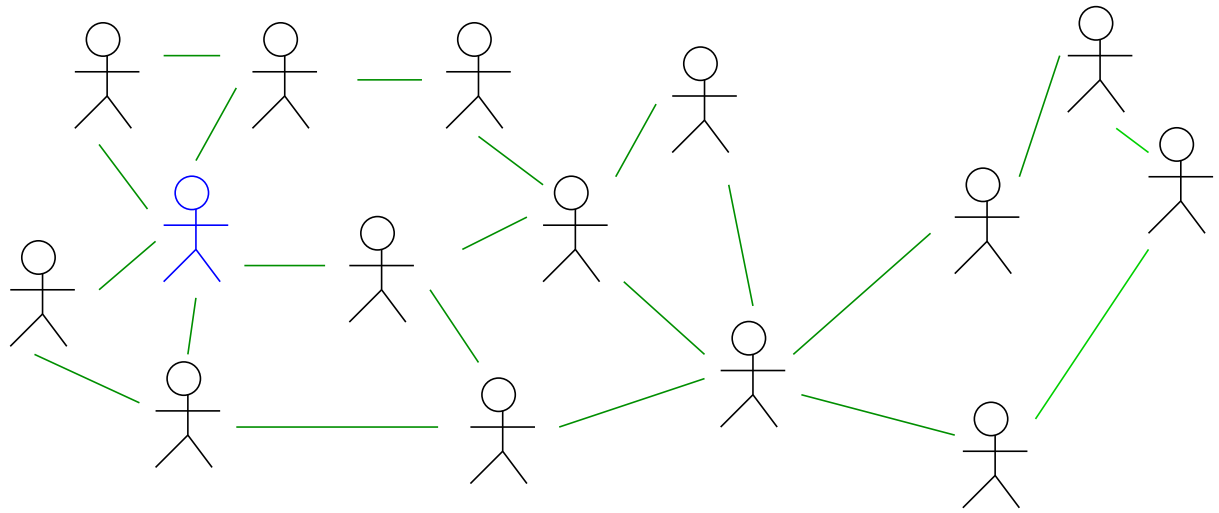
Basic algorithms

Use the greedy

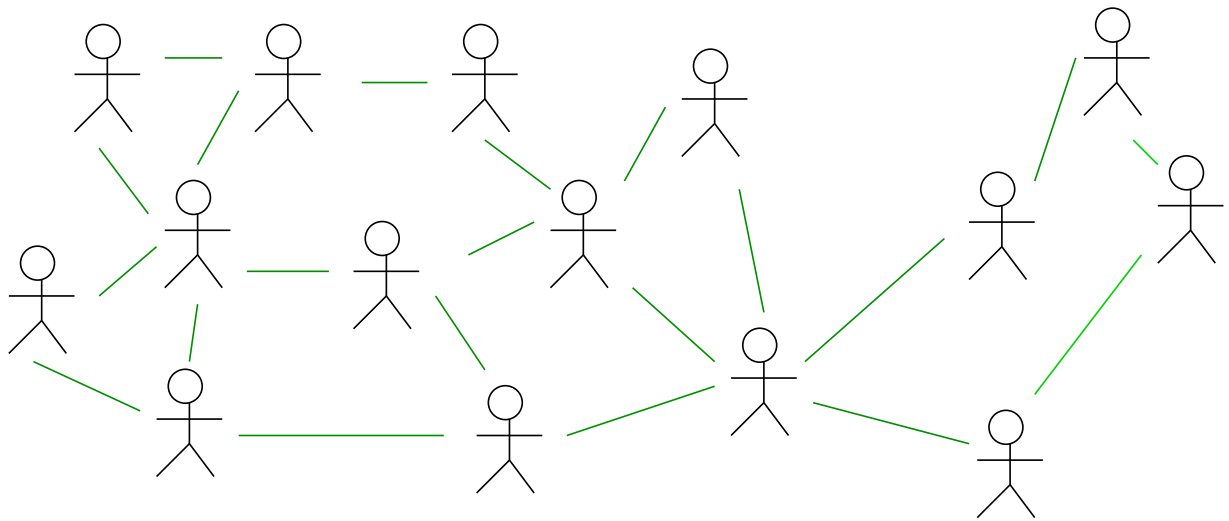
- Jarník-Prim (Serial centralized) Starting from a vertex v, grows T adding each time the lighter edge already connected to a vertex in T, using the blue's rule. Uses a heap to store the edges to be added and retrieve the lighter one.
- Kruskal (Serial distributed) Considers every edge and grows a forest 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.



Jarník-Prim vs. Kruskal

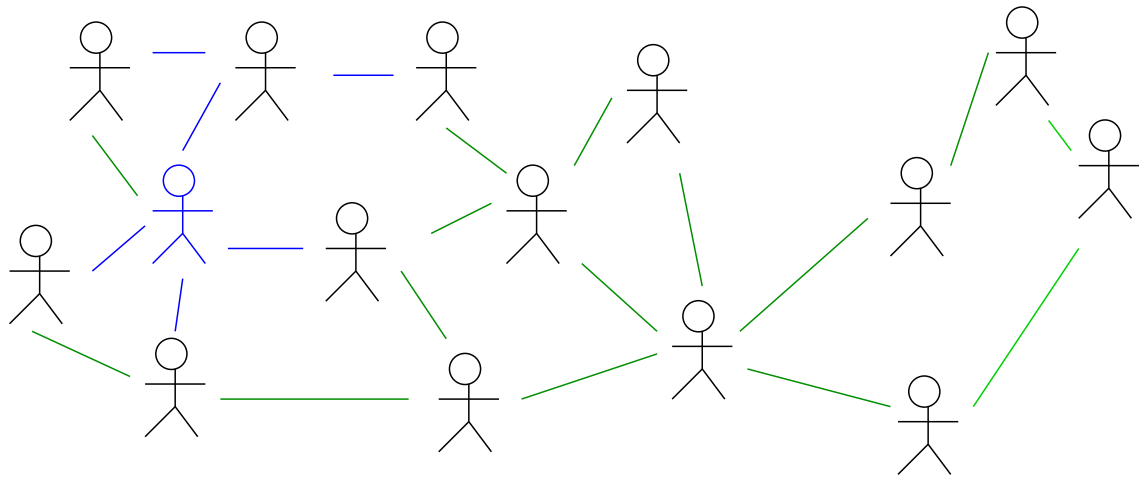


Jarnik–Prim: How blue man can spread his message to everybody

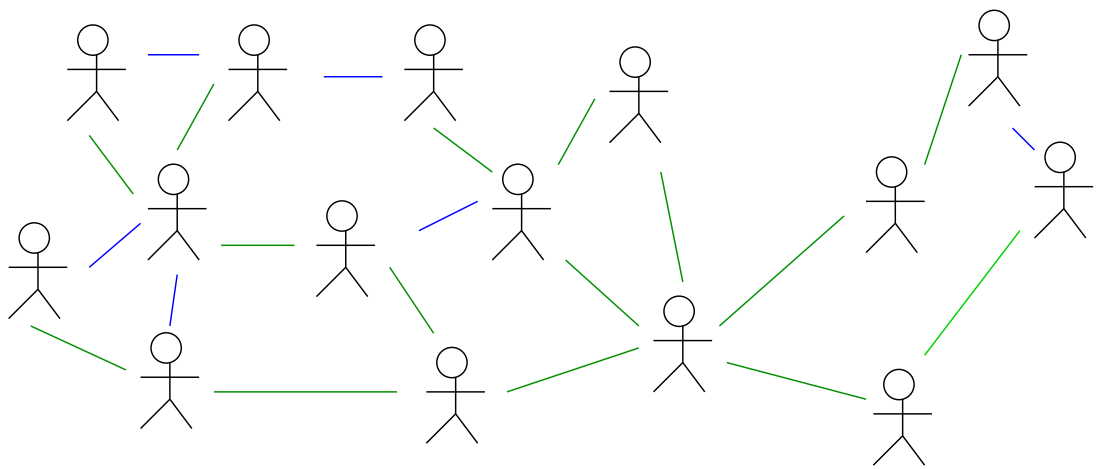


Kruskal: How to stablish a min distance cost network about all men

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.

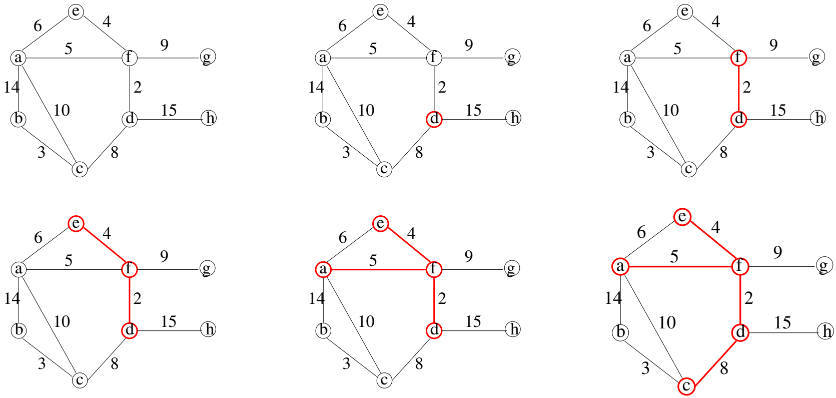
For every $v \in V - T$, let $k[v] =$ minimum weight of any edge connecting v to any vertex in T.

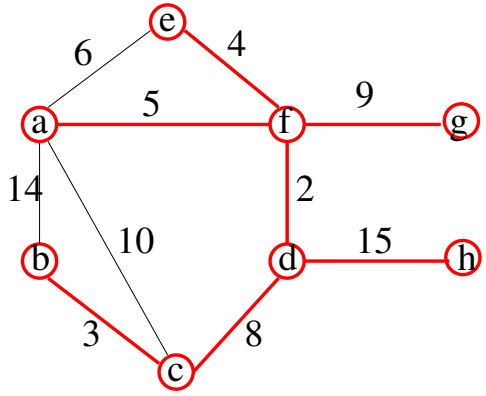
Start with $k[v] = \infty$ for all v.

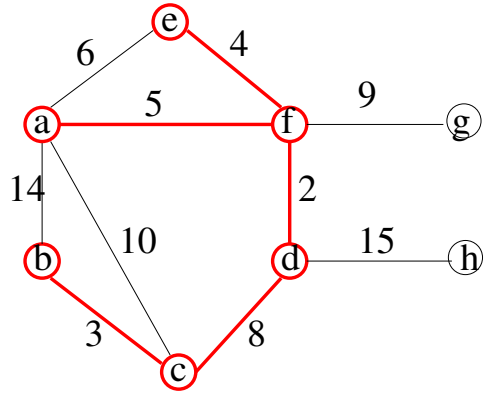
For $v \in T$, let $\pi[v]$ be the parent of v. During the algorithm $T = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$

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

Example.


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





$w(T) = 52$

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.

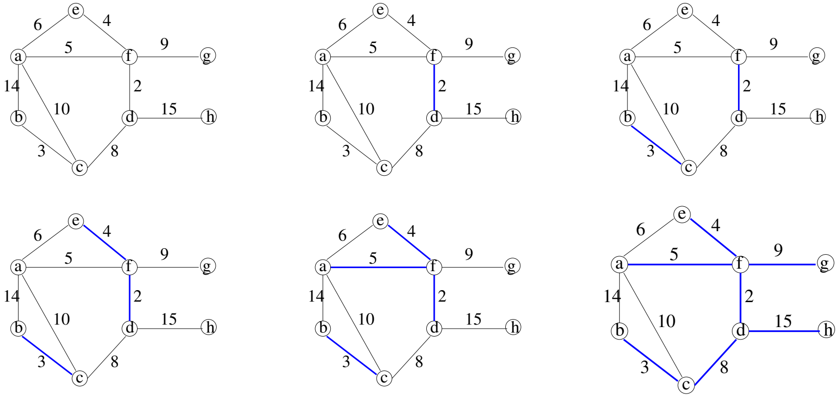
J. Kruskal, 1956

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.
Useful to implement the adding of edges to T we use Union-Find data structure.

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

A data structure: Union-Find

B. Galler, M. Fisher: An improved equivalence algorithm. ACM Comm., 1964

A disjoint-set data structure that maintains a collection $\{S_1, \ldots, S - k\}$ of disjoint dynamic sets, each set identified by a representative.

Union find supports three operations on a set of :
Make-set (x): creates a new set containing x.
Union (x, y): Merge the sets containing x and y.
Find x: find to which set x belongs.

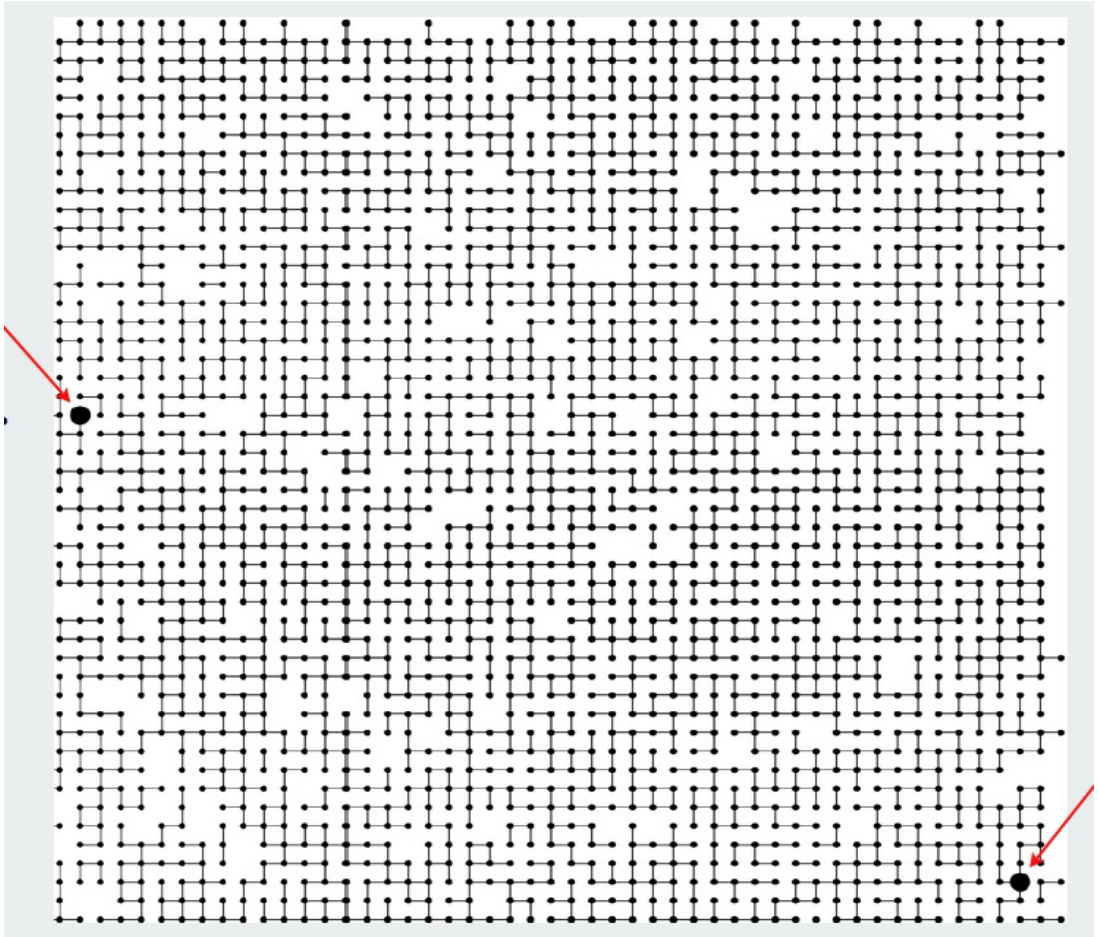
Union-find manipulates all types of objects: names, integers, web-pages, computes ID, ...

Union-find can be used in a myriad of applications, for ex. to compute the connected components of a graph or implement Kruskal.

Network connectivity

Union: connect u to v

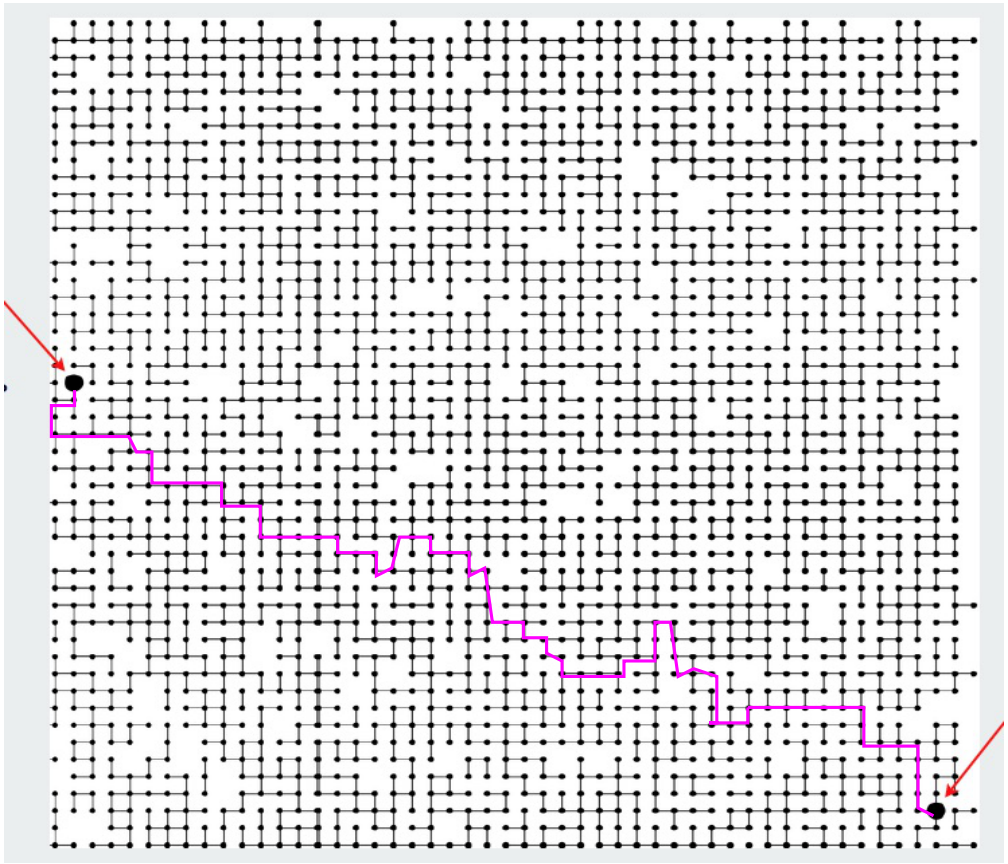
find: is there a path from u to v



Network connectivity

Union: connect u to v

find: is there a path from u to v

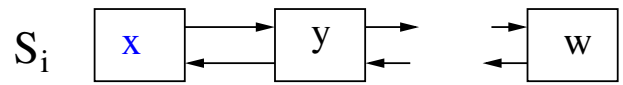


Design efficient implementation for union-find

Union-find implementation: First idea

Given $\{S_1, \ldots, S - k\}$, use double links

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



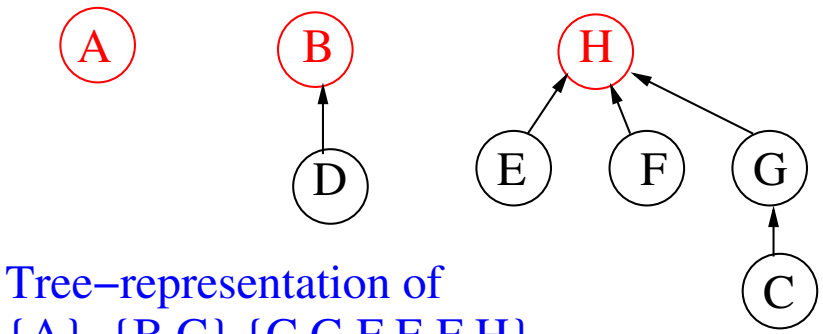
- Make-set (x): Initializes x as a lone list. Wost time $\Theta(1)$.
- Union (x, y): Goues to the tail of S_x and points to head of S_y, concatenating both lists. Worst time $\Theta(n)$
- Find (x): Goes left from x to the head of S_x. Worst case $\Theta(n)$.

Can we do it more efficient?

Union-find implementation: Forest of Trees

Represent each set as a tree of elements:

- The root contains the representative.
- Make-set (x): $\Theta(1)$
- Find (x): find the root of the tree containing x. $\Theta(\text{height})$
- Union (x, y): make the root on one tree point to the root of the other. How?

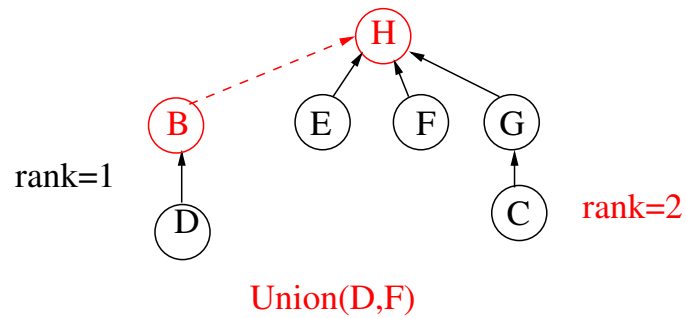


Tree–representation of
{A}, {B,C},{C,G,F,E,F,H}

Link by rank

Any singleton element has rank=0
Union rule: Link the root of smaller rank tree to the root of larger rank tree.

Think of At the beginning, rank of x= height of the subtree rooted at x 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. The new representant is the representant of the highest rank tree. This takes Θ(height) steps.

Link by rank

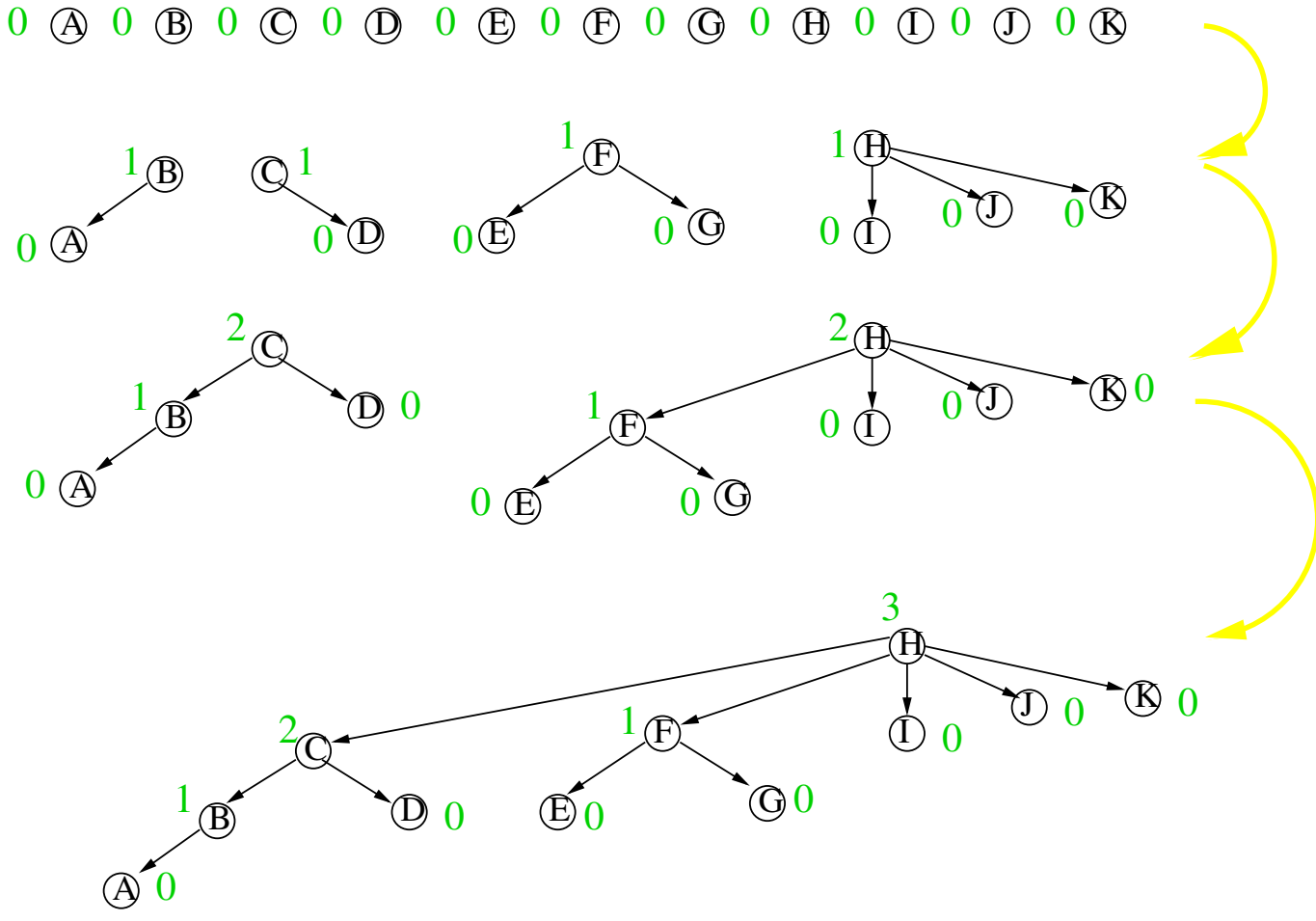
Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if a tie, increase rank of new root by 1.

Make-set (x)
$\text{parent}(x) = x$
$\text{rank}(x) = 0$

Find (x)
while $(x \neq \text{parent}(x)$
do
$x = \text{parent}(x)$
end while

Union (x, y)
$r_x =$ **Find**(x)
$r_y =$ **Find**(y)
if $r_x = r_y$ **then**
Stop
else if $(rank)(r_x) > (rank)(r_y)$ **then**
$\text{parent}(r_y) = r_x$
else if $(rank)(r_x) < (rank)(r_y)$ **then**
$\text{parent}(r_x) = r_y$
else
$\text{parent}(r_x) = r_y$
$(rank)(r_y) = (rank)(r_y) + 1$
end if

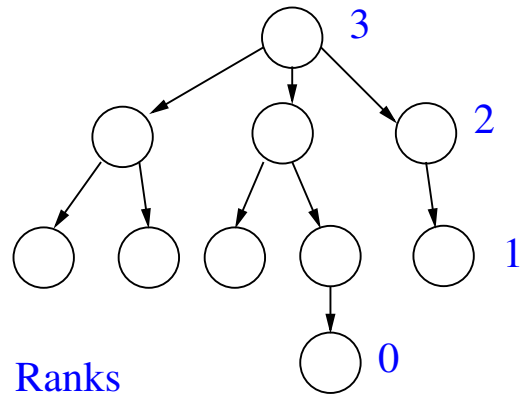
Example construction Union-find



Properties of Link by rank

P1.- If x is not a root then $\text{rank}(x) < \text{rank}(\text{parent}(x))$

P2.- If $\text{parent}(x)$ changes then $\text{rank}(\text{parent}(x))$ increases



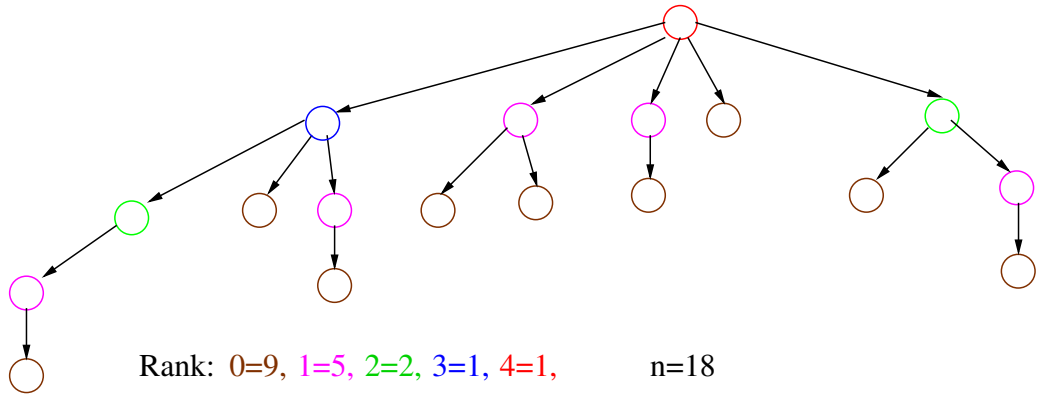
P3.- Any root of rank k has $\geq 2^k$ descendants.

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 $k - 1$ □

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

Properties of Link by rank

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



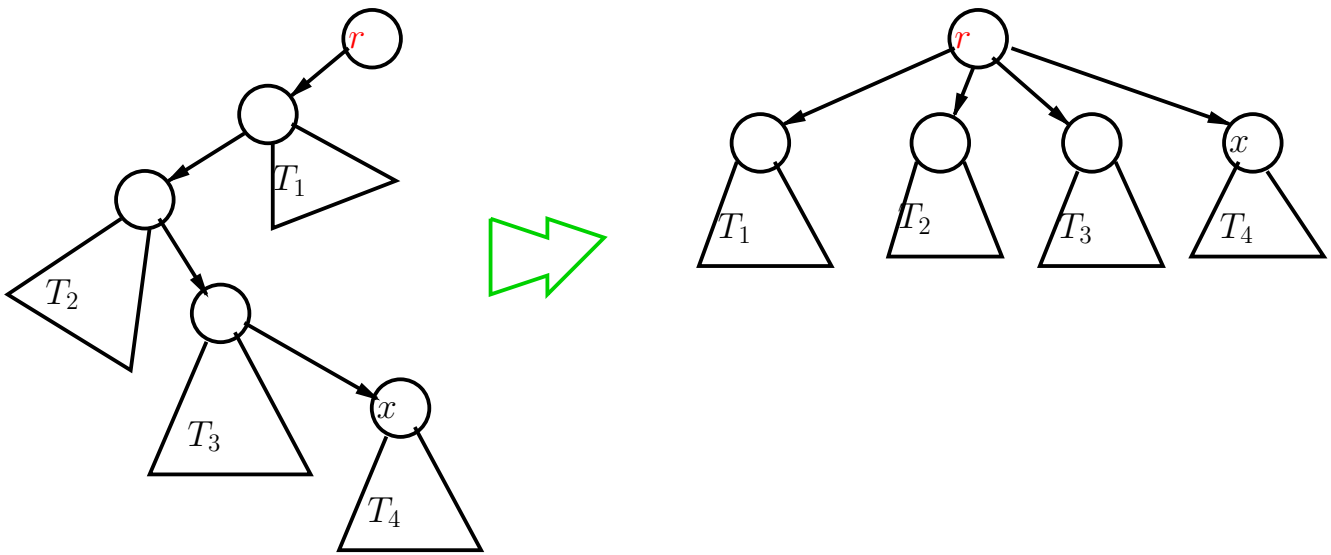
Theorem

Using Union-find with link-by-rank, the Union(x, y) and Find(x) operations take $O(\lg n)$ operations.

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

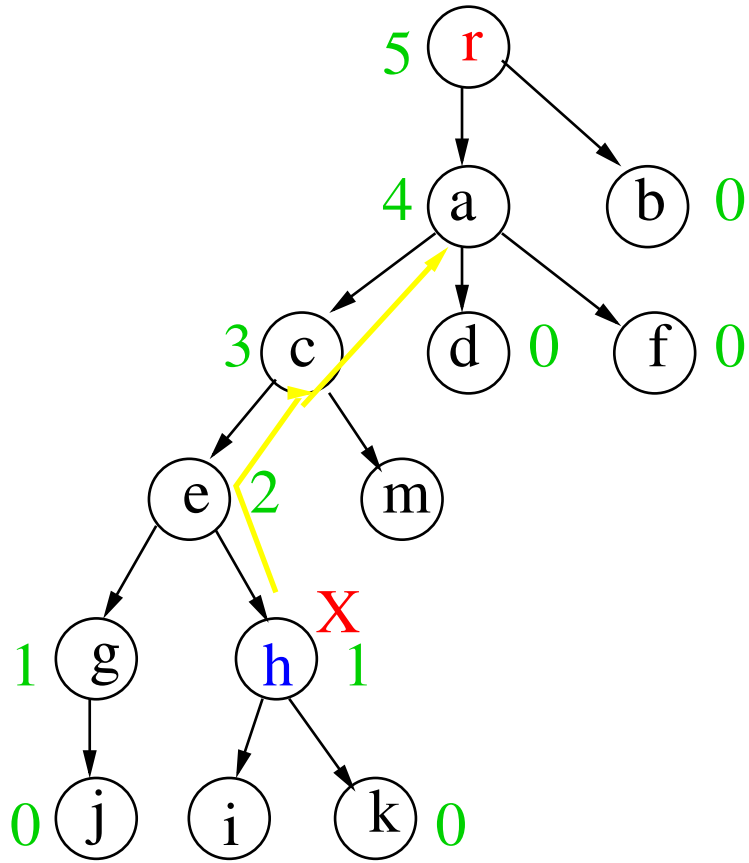
An improvement: Path compression

To improve the $O(\log n)$ bound per operation in union-bound, we keep to flatten the trees as much as possible.
We use the path compression: At each use of Fins(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.



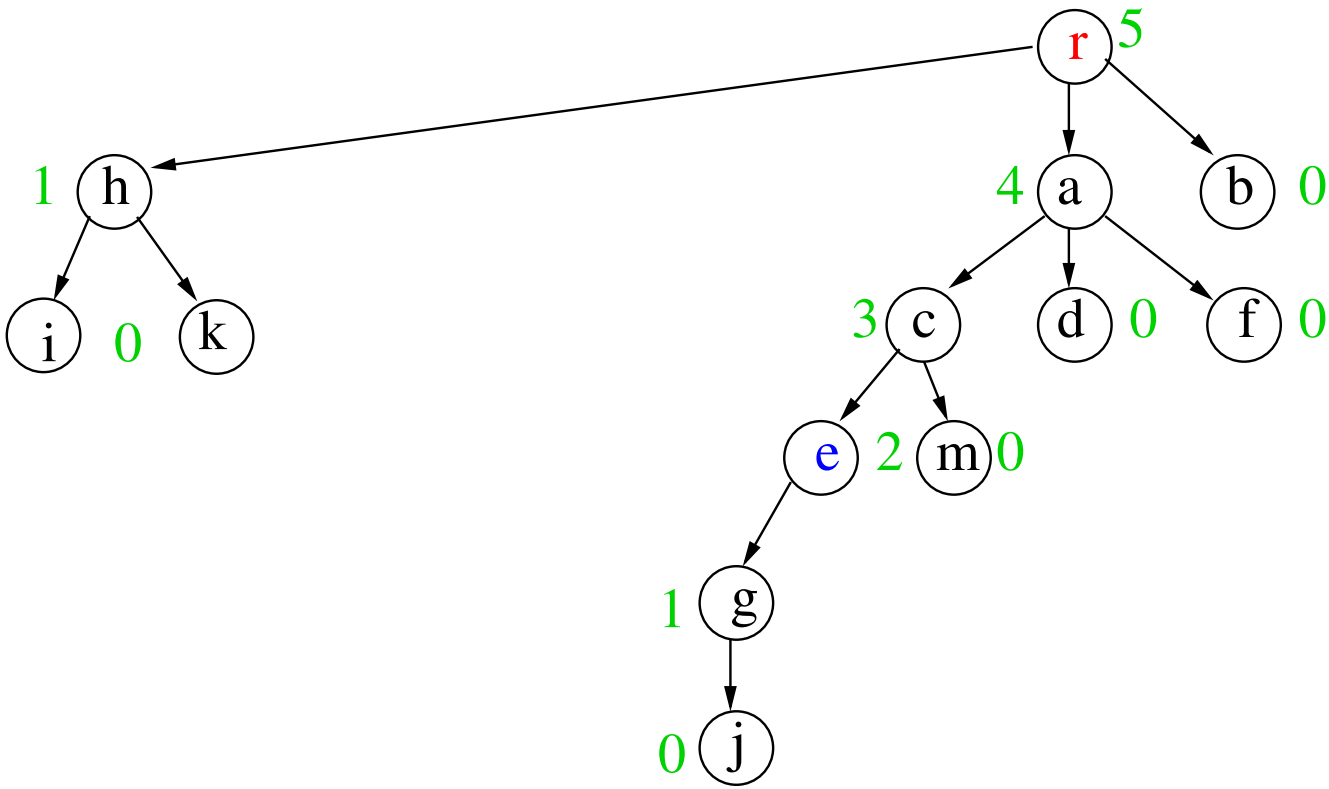
Path compression: Function

Find (x)
if $(x \neq \text{parent}(x)$
then
  $\text{parent}(x) =$
  Find $\text{parent}(x)$
  return $\text{parent}(x)$
end if



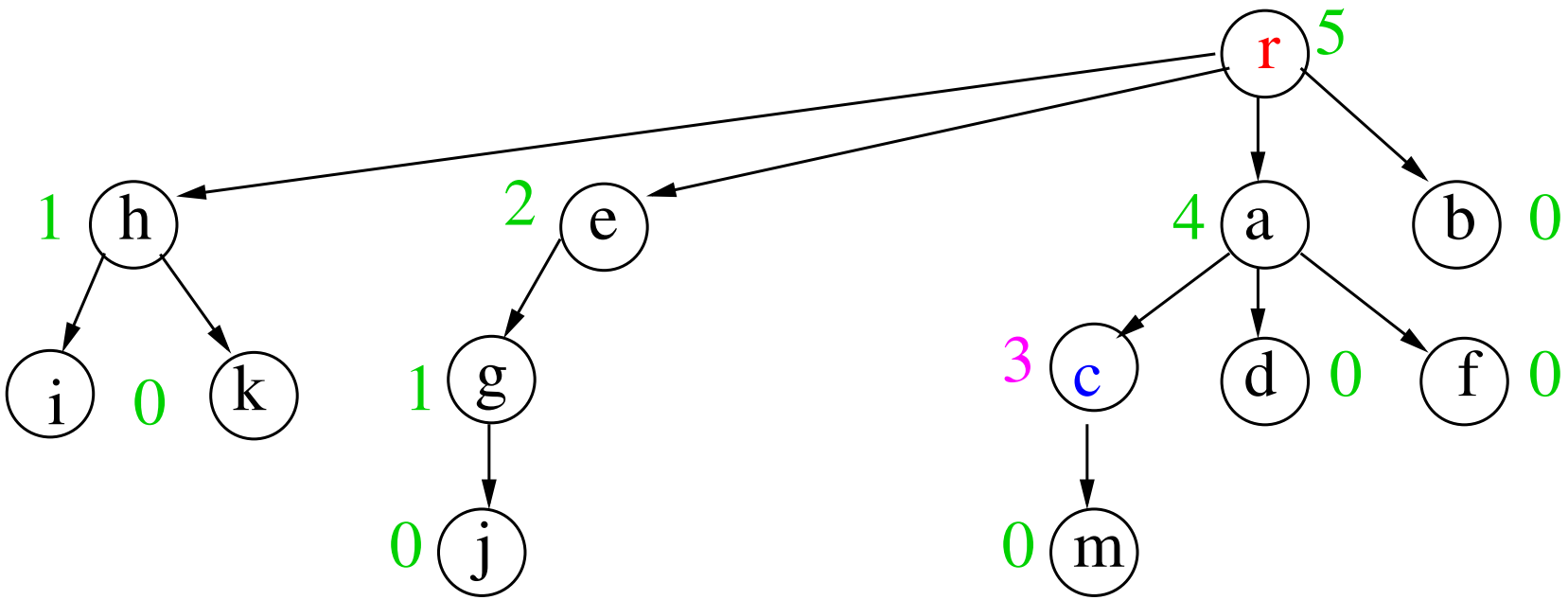
Path compression: Function

Find (x)
if $(x \neq \text{parent}(x)$ **then**
  $\text{parent}(x) =$ **Find** $\text{parent}(x)$
  return $\text{parent}(x)$
end if



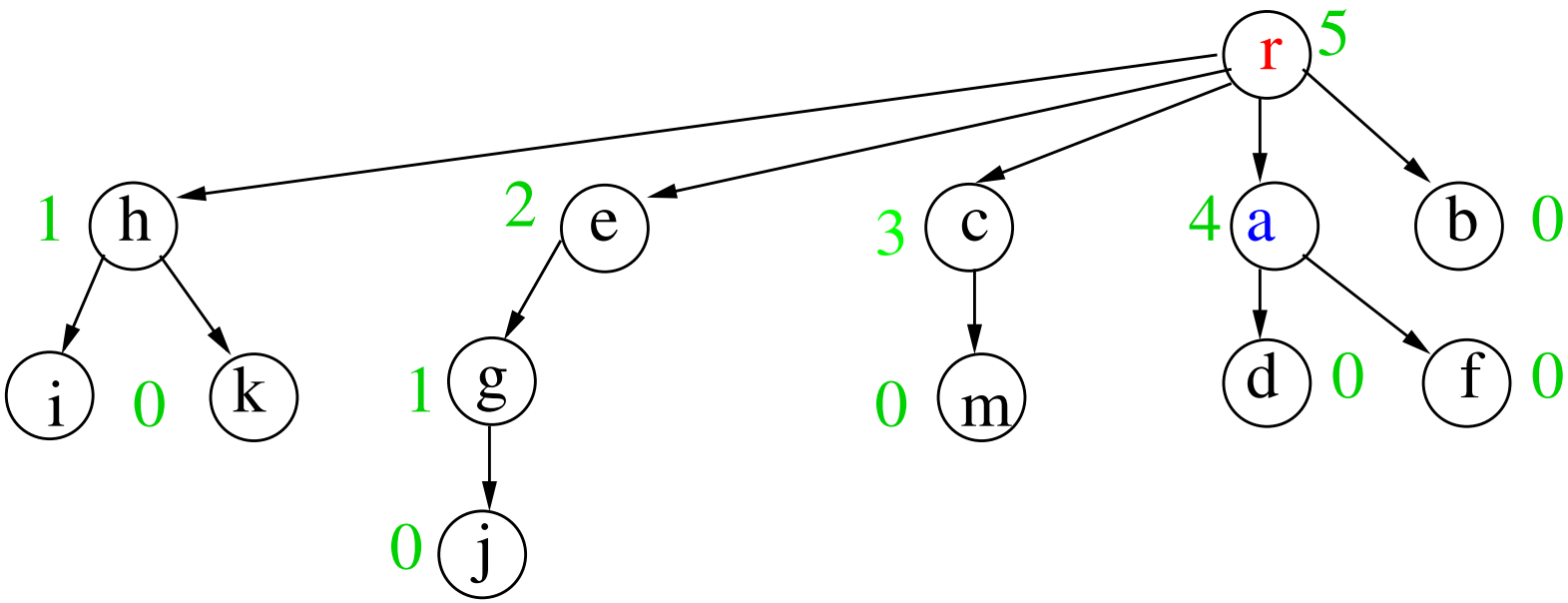
Path compression: Function

Find (x)
if $(x \neq \text{parent}(x)$ **then**
  $\text{parent}(x) =$ **Find** $\text{parent}(x)$
  return $\text{parent}(x)$
end if



Path compression: Function

Find (x)
if $(x \neq \text{parent}(x)$ **then**
  $\text{parent}(x) =$ **Find** $\text{parent}(x)$
  return $\text{parent}(x)$
end if



Path compression: Properties

1. If x is not a root then $\text{rank}(x) < \text{rank}(\text{parent}(x))$
2. The ranks of root nodes are unaltered by path compression
3. The ranks of all nodes are unchanged by path compression, those values can not longer be interpreted as tree heights.
4. If there are n elements, their rank values can range from 0 to $\lfloor \lg n \rfloor$
5. For any integer $k \geq 0$ there are $\leq n/2^k$ nodes with rank k.

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
[65537,2^{65536}]	5

Main result Analysis

Theorem

Starting from an empty data structure with h disjoint single sets, link-by-size with path compression performs any intermixed sequence of $m \geq n$ **Find** *and $n-1$* **Union** *operations in $m \lg^* n$ steps.*

Proof

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

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

Analysis

Divide non-zero ranks into the following intervals:
$\{1\}, \{2\}, \{3,4\}, \{5,\ldots,16\}, \{65537,\ldots,2^{65536}\}, \ldots \underbrace{\{k+1,\ldots,2^k\}}_{k\text{group}}$.

As the rank is between 0 and $\lfloor \lg n \rfloor$:

6.- Every non-zero rank in a UF with n elements falls within the first $\lg^* n$ intervals.

We give a credit of 2^k to any non-root node with rank within $\{k+1,\ldots,2^k\}$

As the number of nodes with rank $\geq k+1$ is $\leq \frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}$ then nodes in interval k need $\leq n$ credits, as there are $\leq \lg^* n$ intervals:

The number of credits given to all nodes is with rank $\leq n \lg^* n$.

Analysis

Consider the number of steps taken by a specific Find(x), that will be the number nodes between x and r
The rank strictly increases as you go up the tree. Two cases:

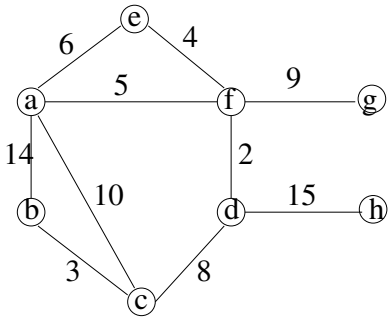
1. rank(parent(x)) is in a higher interval than rank(x).
   By 6 there are only $\lg^* n$ nodes in this case. So the work done on them takes $O(\lg^* n)$.
2. rank(parent(x)) is in the same interval than rank(x).
   Those nodes are charged 1 credit to follow their parent pointer
   $\therefore$ if rank(x) is in the interval $[k+1, \ldots, 2^k]$, the rank of its parent will be in a higher interval before x pays 2^k.
   As once rank(parent(x)) is in a higher interval than rank(x) it remains through all the process.

Back to Kruskal

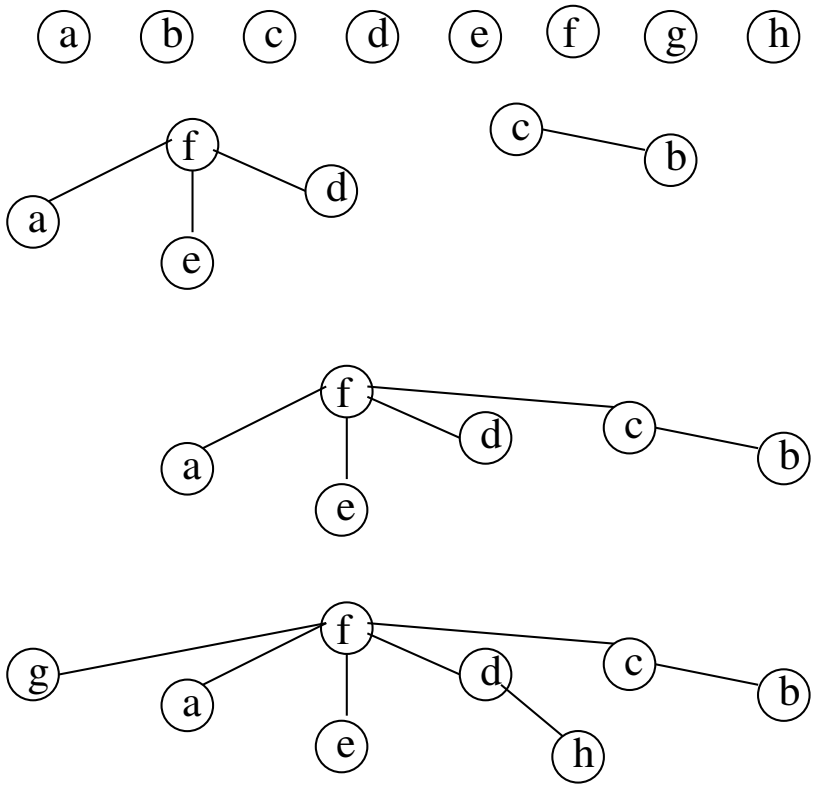
MST (G, w, r)
Sort E by increasing weight: $\{e_1, \ldots, e_m\}$
$T := \emptyset$
for all $v \in V$ **do**
  Make-set(v)
end for
for $i = 1$ **to** m **do**
  Chose $e_i = (u, v)$ in order from E
  if Find(x) $\neq$ Find(y) **then**
    $T := T \cup \{e_i\}$
    Union(u, v)
  end if
end for

Cost is dominated by $O(m \lg m)$

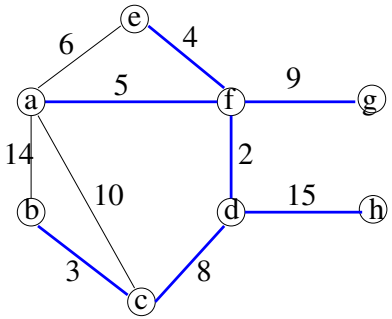
Example of Kruskal



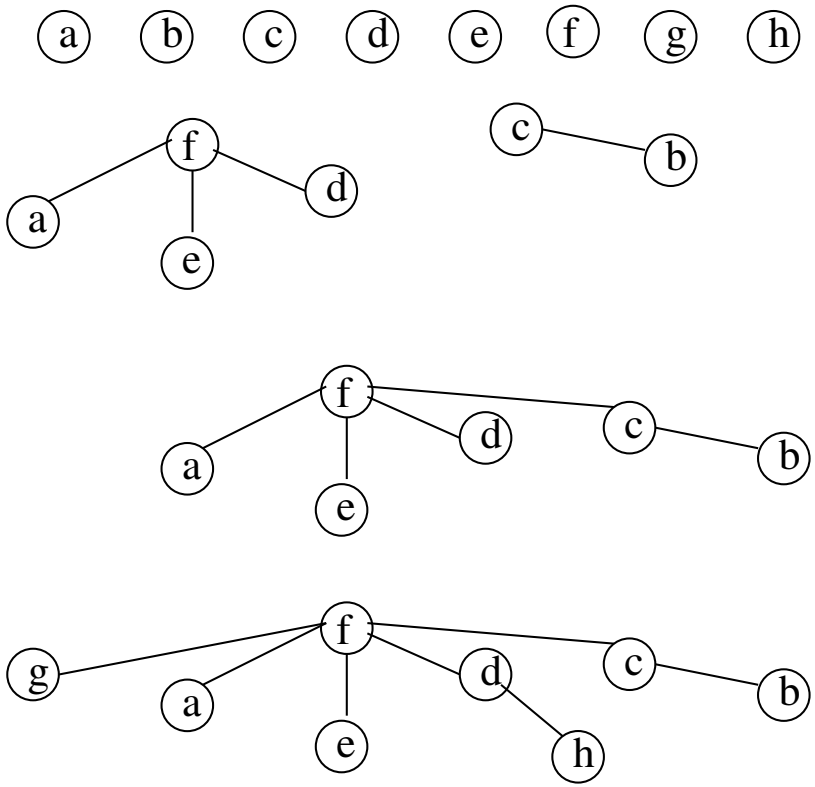
$$E = \{(f,d),(c,d),(e,f),(a,f),(a,e),(c,d),(f,g),(a,c),(a,b),(d,h)\}$$



Example of Kruskal

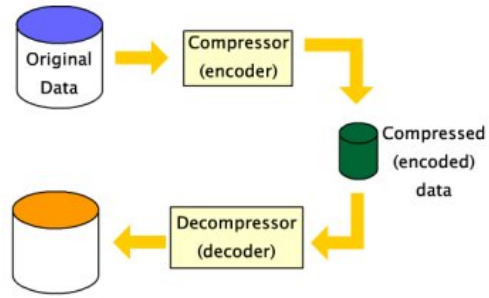


$$E = \{(f,d),(c,d),(e,f),(a,f),(a,e),(c,d),(f,g),(a,c),(a,b),(d,h)\}$$

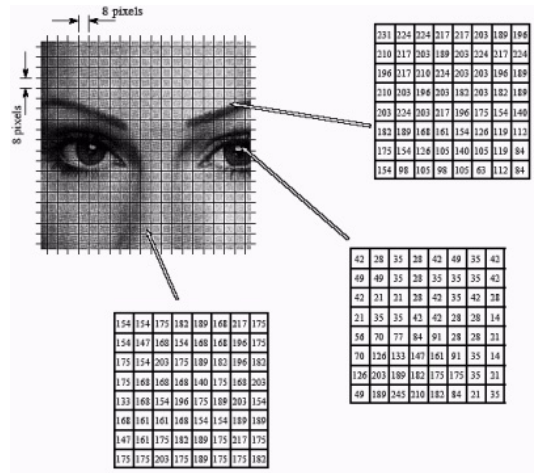


Data Compression

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

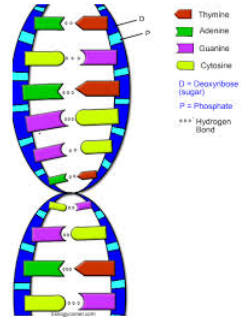


The goal of data compression is to reduce the time to transmit large files, and to reduce the space to store them.
If we are using variable-length encoding we need a system easy to encode and decode.



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

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

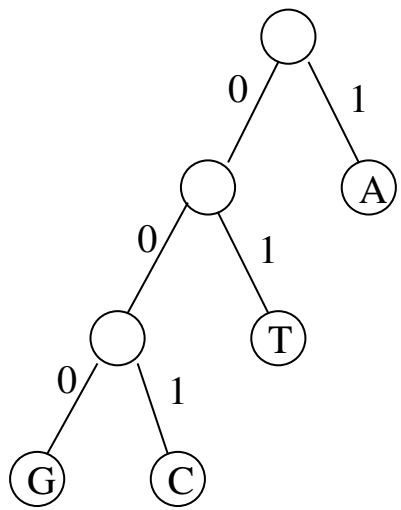
Prefix tree.

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

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

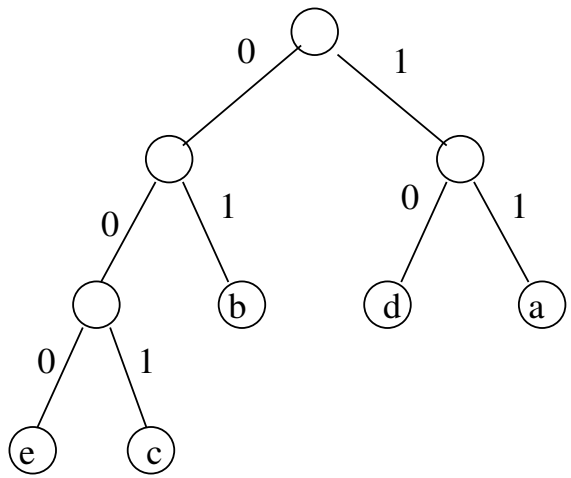
Given ϕ, $\alpha = \sum_{x \in \Sigma} f(x)|\phi(x)|$ is the average number of bits required per symbol.

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:
$\phi_1(a) = 11, \phi_1(b) = 01, \phi_1(c) = 001, \phi_1(d) = 10, \phi_1(e) = 000$

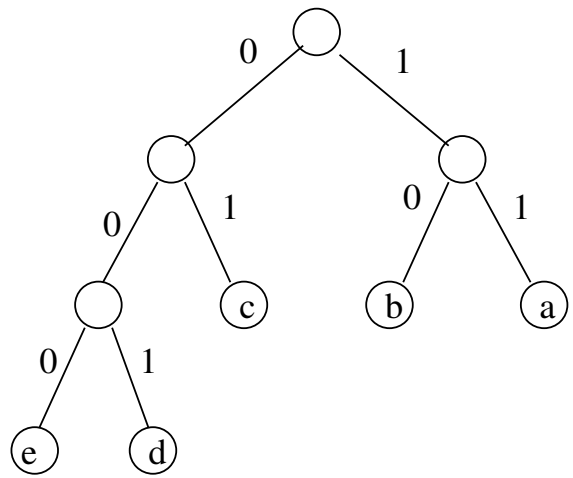


$\alpha = .32 \cdot 2 + .25 \cdot 2 + .20 \cdot 3 + .18 \cdot 2 + .05 \cdot 3 = 2.25$
In average, ϕ_1 reduces the bits per symbol over the fixed-length code from 3 to 2.25, about 25%

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

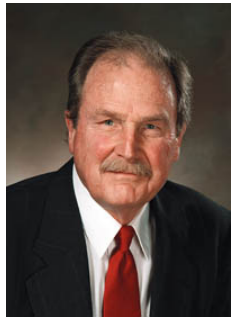
The binary prefix tree corresponding to an optimal prefix code is full.

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.

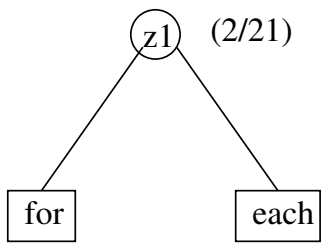
Huffman Coding: Construction of the tree.

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 $O(n \lg n)$.

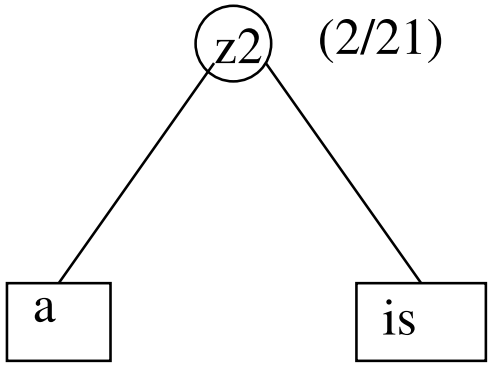
Example

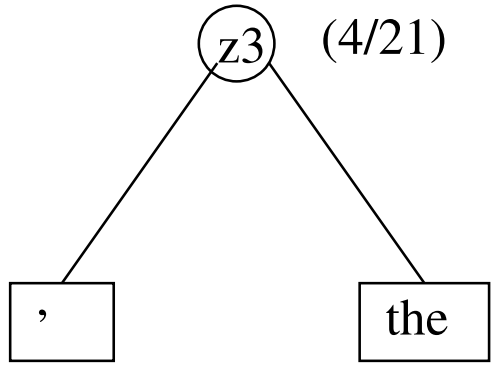
Consider the text: *for each rose, a rose is a rose, the rose*.
with $\Sigma = \{$for/ each/ rose/ a/ is/ the/ ,/ b$\}$
Frequencies: f(for) $= 1/21$, f(rose) $= 4/21$, f(is) $= 1/21$,
f(a) $= 2/21$, f(each) $= 1/21$, f(,) $= 2/12$, f(the) $= 1/12$,
f(b) $= 9/21$.
Priority Queue:
Q=(for(1/21), each(1/21), a(1/21), is(1/21), ,(2/21), the(2/21), rose(4/21), b(9/21))

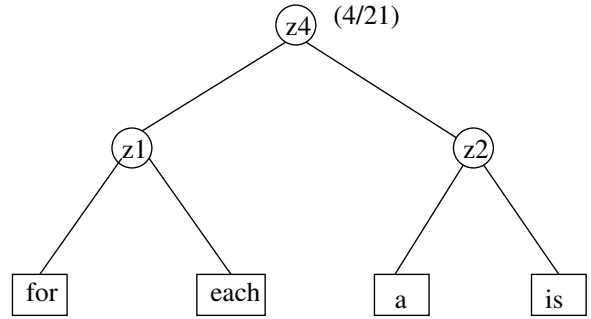


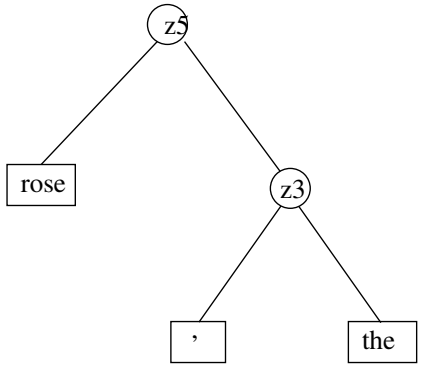
Q=(a(1/21), is(1/21), ,(2/21), the(2/21), z1(2/21), rose(4,21), b(9/21))

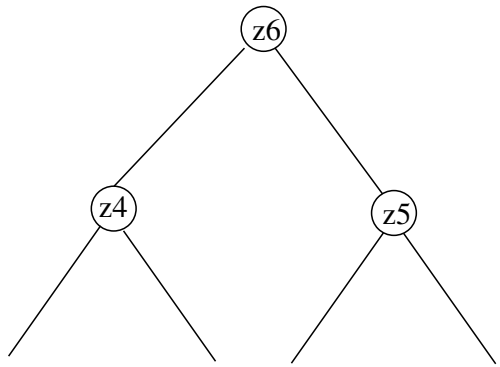
Example.

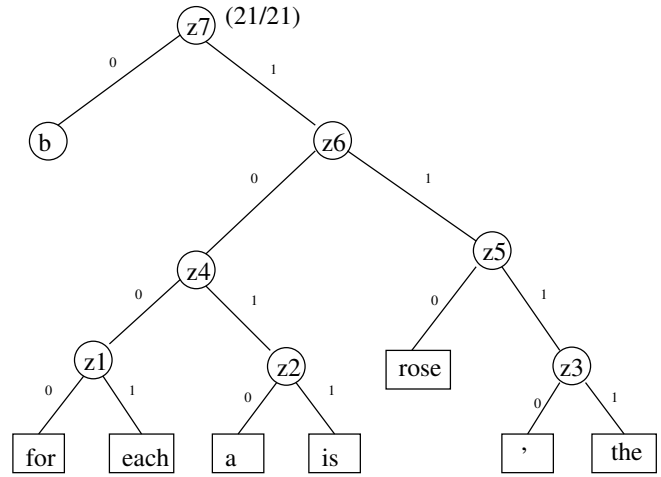

z2
(2/21)
a
is


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.

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 $B(T) \leq B(T_0)$. $\square$