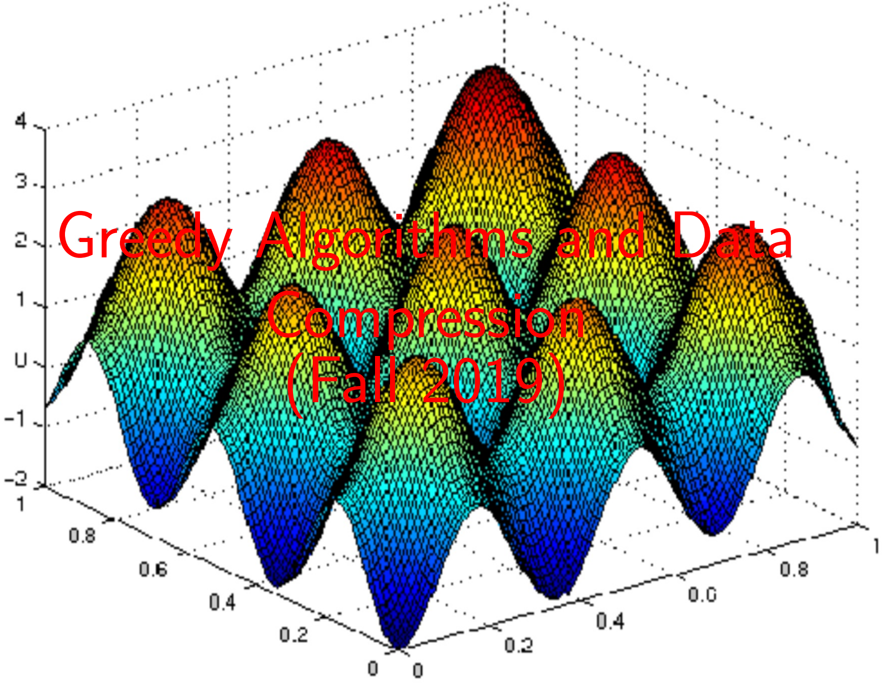
Greedy Algorithms and Data Compression (Fall 2019)



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: Given as input a set $I = \{i\}_1^n$ of items that can be fractioned, each i with weight w_i and value v_i together with a maximum total weight W permissible. We want to 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?

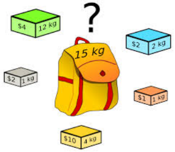
0-1 Knapsack

0-1 Knapsack Given as input a set I of n items that can NOT be fractioned, each i with weight w_i and value v_i, together with maximum weight W permissible. We want to select the items, which maximize the profit, within allowed weight W.

Greedy does not work for 0-1 Knapsack

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 $S = \{1, 2, \dots, n\}$ 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.

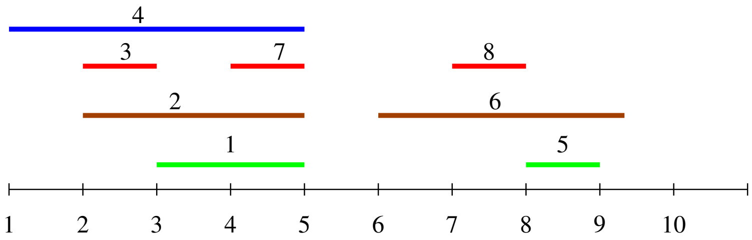
The INTERVAL SCHEDULING problem

The INTERVAL SCHEDULING (aka Activity Selection problem): Given as input a set $S = \{1, 2, \dots, n\}$ of activities to be processed by a single resource, where each activity i has a start time s_i and a finish time f_i, with $f_i > s_i$. We want to 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,

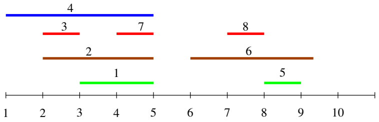
- A local criteria to allow the selection,
- a condition to determine if a partial solution can be completed,
- a procedure to test that we have the optimal solution.

The Activity Selection problem

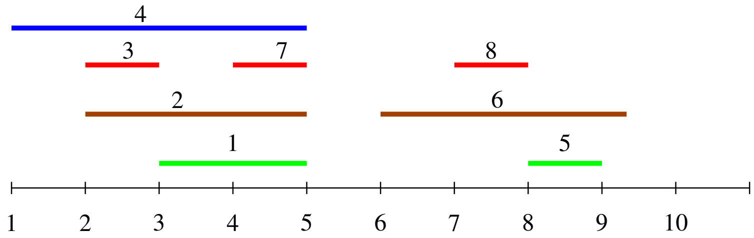
Given a set A of activities, wish to maximize the number of compatible activities.

```
**activity selection** $A$
Sort $A$ by increasing order of $f_i$
Let $a_1, a_2, \dots, a_n$ 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 Selection 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.

Could you modify the previous algorithm to solve the Maximum Occupancy problem?

Theorem

The previous algorithm produces an optimal solution to the Activity Selectionproblem.

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

Proof.

Given $A = \{1, \dots, n\}$ sorted by finishing time, we must show there is an optimal solution that begins with activity 1. Let $S = \{k, \dots, m\}$ 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 □

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.

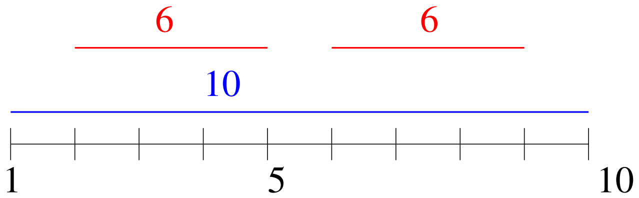
Weighted Activity Selection problem: Given as input a set $A = \{1, 2, \dots, n\}$ of activities to be processed by a single resource, where each activity i has a start time s_i and a finish time f_i, with $f_i > s_i$, and a weight w_i. We want to find the set of mutually compatible activities such that it maximizes $\sum_{i \in S} w_i$

```
**weighted activity selection** $A$
$S = \emptyset$
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 textWeighted Activity Selection 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.

THE GRAVEYARD OF PAST DEADLINES
ASAP
LIKE, YESTERDAY
NOON-ISH
NO LATER THAN TUESDAY 6:15 E.S.T.
FIRST THING IN THE MORNING
FRIDAY AT 7 A.M.

Minimize Lateness

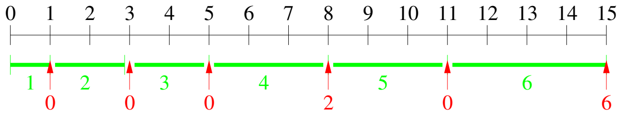
- We have a single processor
- We have n jobs such that job i:
  - requires t_i units of processing time,
  - it has to be finished by time d_i,
- Lateness of i:

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

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

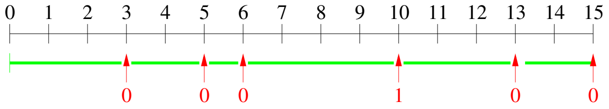
In this case, job 2 should be processed first, which doesn't minimise lateness.

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

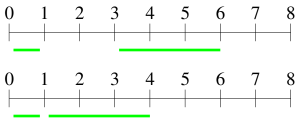
Complexity and idle time

Time complexity

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

Idle steps

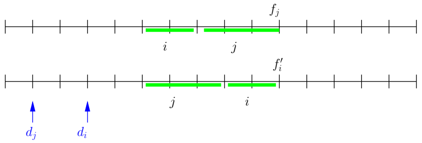
From an optimal schedule with idle steps, we always can eliminate gaps to obtain another optimal schedule:



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. $\square$

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 $\hat{S} = S$.

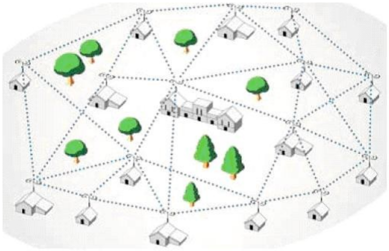
If number inversions in $\hat{S}$ is > 0, let $i - j$ be an adjacent inversion.

Exchanging i and j does not increase lateness and decrease the number of inversions.

Therefore, max lateness $S \leq$ max lateness $\hat{S}$. $\square$

Network construction: Minimum Spanning Tree

- We have a set of locations $V = \{v_1, \dots, v_n\}$,
- 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 $\sum_{e \in T(E)} w(e)$.



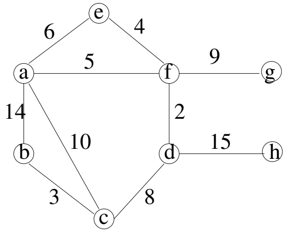
Network construction: Minimum Spanning Tree

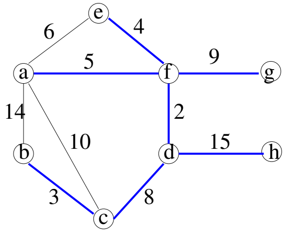
- We have a set of locations $V = \{v_1, \ldots, v_n\}$,
- we want to build a communication network on top of them
- we want that any v_i can communicate with any v_j,
- for any pair (v_i, v_j) there is a cost $w(v_i, v_j)$ of building a direct link,
- if E is the set of all possible edges ($|E| \leq n(n-1)/2$), 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 problem (MST)

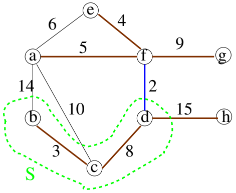
Given as input an edge weighted graph $G = (V, E)$, $|V| = n, \forall e \in E, w(e) \in \mathbb{R}$.
Find a tree T with $V(T) = V$ and $E(T) \subseteq E$, such that it minimizes $w(T) = \sum_{e \in E(T)} w(e)$.





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.

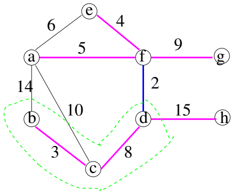
Theorem (The cut rule)

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 $e \not\in T$. 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. $\square$



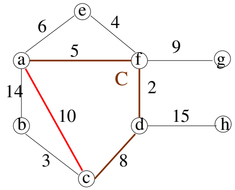
The cycle rule (Red rule)

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 $\{v_i, \dots, v_l\}$, then removing the max weighted edge gives a a better solution. $\square$



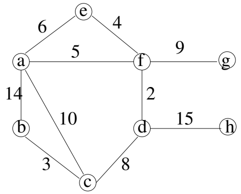
C=cycle spanning {a,c,d,f}

This doesn't mean that the max. weighted edge would not be in the MST.

Generic greedy for MST: Apply blue and/or red rules

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.

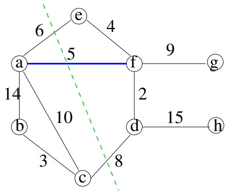


Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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.

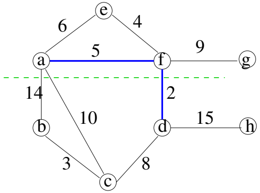


Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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.

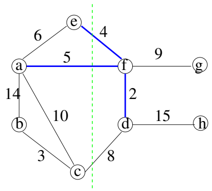


Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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.

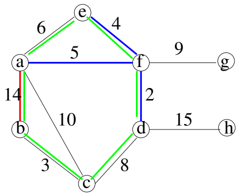


Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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.



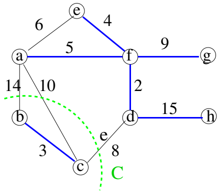
Greedy scheme:
Given G, $V(G) = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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 e ends are in different blue trees, 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

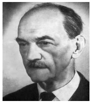
There has been extensive work to obtain the most efficient algorithm to find a MST in a given graph:

- 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.

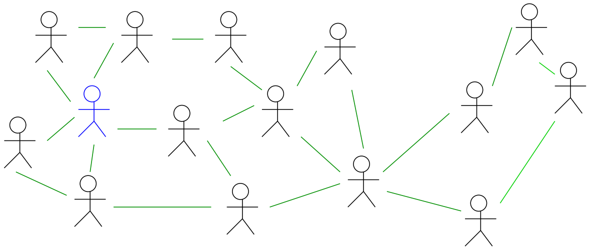
Basic algorithms

Use the greedy

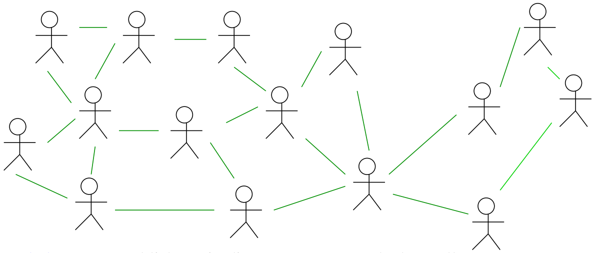
- Jarnik-Prim (Serial centralized) Starting from a vertex v, grows T adding each time the lighter edge already connected to a vertex in T, using the blue's rule. Uses a priority queue (usually a heap) to store the edges to be added and retrieve the lighter one.
- Kruskal (Serial distributed) Considers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

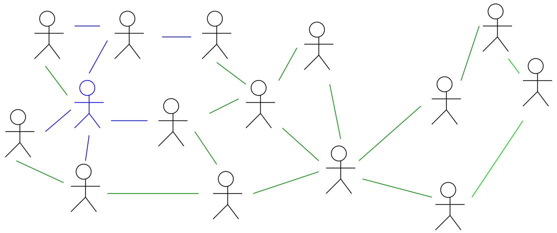


Jarnik–Prim: How blue man can spread his message to everybody

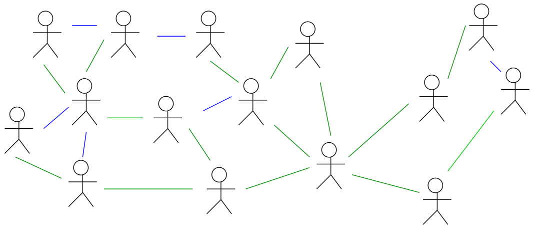


Kruskal: How to stablish a min distance cost network about all men

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.

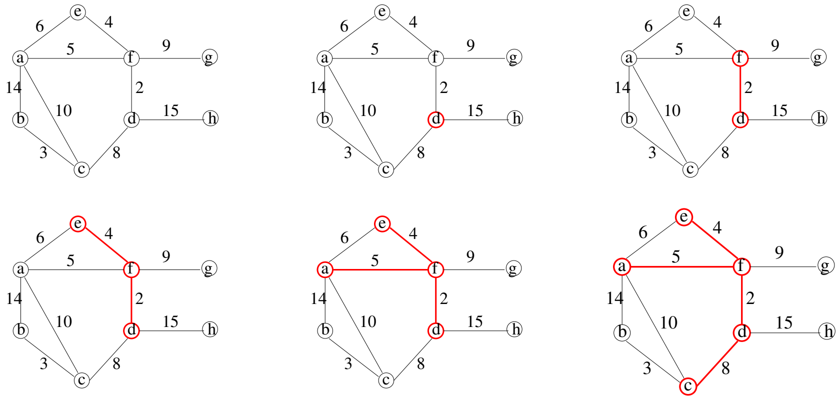
For every $v \in V - T$, let $k[v] =$ minimum weight of any edge connecting v to any vertex in T.

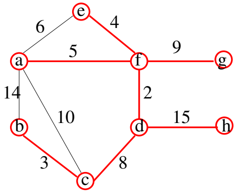
Start with $k[v] = \infty$ for all v.

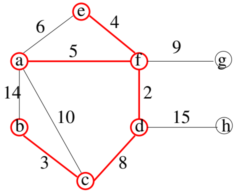
For $v \in T$, let $\pi[v]$ be the parent of v. During the algorithm
$T = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$

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.

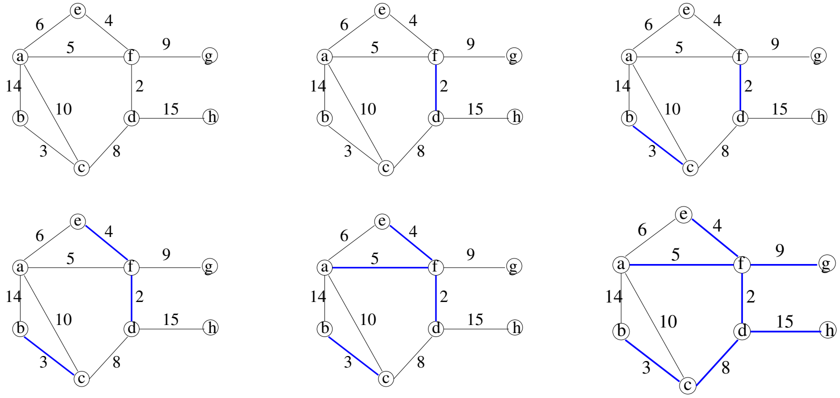
J. Kruskal, 1956

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.



Using Union-Find for Kruskal

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

Union-Find implementation for Kruskal

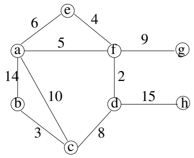
```
**MST** $(G(V,E), w, r)$, $|V| = n, |E| = m$
Sort $E$ by increasing weight: $\{e_1, \dots, e_m\}$
$T = \emptyset$
**for all** $v \in V$ **do**
    MAKESET($v$)
**end for**
**for** $i = 1$ **to** $m$ **do**
    Chose $e_i = (u, v)$ in order from $E$
    **if** FIND($x$) $\neq$ Find($y$) **then**
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

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

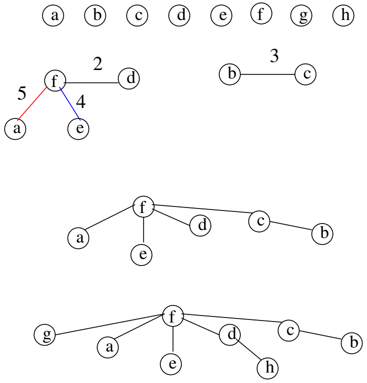
But due to the sorting instruction, the overall complexity of Kuskal still is $O(m \lg n)$.

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 approximation algorithms

- Many times the Greedy strategy yields a feasible solution with value which is near to the optimum solution.
- In many practical cases, when finding the global optimum is hard, the greedy may yield a *good enough* feasible solution: An approximation to the optimal solution.
- An approximation algorithm for the problem always computes a close valid output. Heuristics also could yield approximated solutions, but they are not algorithms.
- Greedy is one of the algorithmic techniques that produce approximations algorithms, but it is not the only one.

Greedy and approximation algorithms

- For any optimization problem, let $c(*)$ be the value of the optimization function, let $\mathcal{A}px$ be the approximation algorithm and let OPT be the exact algorithm, which could have exponential cost.
- For any instance a we want to design an approximation algorithm that do produce a solution $c(\mathcal{A}px(e))$, as close as possible to the unknown optimal solution $c(\text{OPT}\,(e))$.
- Giving a problem that it is not in P, therefore we don't know if it has polynomial time algorithms, we want to design algorithms that are fast (polynomial) and that output solutions as close as it is possible to $c(\text{OPT}\,(e))$.

Formal definition

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

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

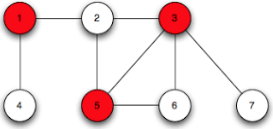
α is called the approximation ratio.

Notice, α measures the factor by which the cost output of $\mathcal{A}px$ exceeds that of $\mathrm{OPT}(e)$, on any input.

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

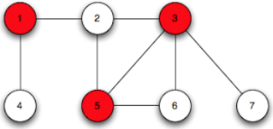
An easy example: VERTEX COVER problem

Recall the problem of 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.

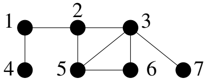


An easy example: Vertex Cover problem

Recall the problem of 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.

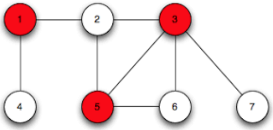


```
GreedyVC $G = (V, E)$
$E' = E$, $S = \emptyset$,
while $E' \neq \emptyset$ do
    Pick $e \in E'$, say $e = (u, v)$
    $S = S \cup \{u, v\}$,
    $E' = E' - \{(u, v) \cup \{\text{edges incident to } u, v\}\}$
end while
return $S$.
```

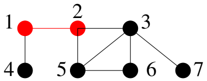


An easy example: VERTEX COVER problem

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.

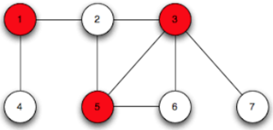


```
**GreedyVC** $G = (V, E)$
$E' = E$, $S = \emptyset$,
**while** $E' \neq \emptyset$ **do**
    Pick $e \in E'$, say $e = (u, v)$
    $S = S \cup \{u, v\}$,
    $E' = E' - \{(u, v) \cup \{\text{edges incident to } u, v\}\}$
**end while**
**return** $S$.
```

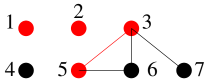


An easy example: VERTEX COVER problem

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.

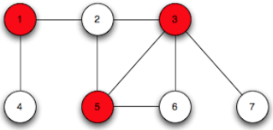


```
GreedyVC $G = (V, E)$
$E' = E$, $S = \emptyset$,
while $E' \neq \emptyset$ do
    Pick $e \in E'$, say $e = (u, v)$
    $S = S \cup \{u, v\}$,
    $E' = E' - \{(u, v) \cup \{\text{edges incident to } u, v\}\}$
end while
return $S$.
```

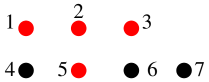


An easy example: VERTEX COVER problem

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, $|\mathcal{A}px(e)| \leq 2|\text{OPT}\,(e)\,|$.

Proof: We use induction to prove $|\mathcal{A}px(e)| \leq 2|\text{OPT}\,(e)\,|$. Notice for every $\{u, v\}$ we add to $\mathcal{A}px(e)$, either u or v are in $\text{OPT}\,(e)$.

Base: If $V = \emptyset$ then $|\mathcal{A}px(e)| = |\text{OPT}\,(e)\,| = 0$.

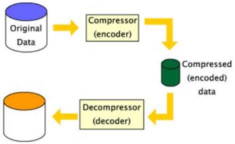
Hipothesis: $|\mathcal{A}px(e - \{u, v\})| \leq 2|\text{OPT}\,(e - \{u, v\})\,|$. Then,

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

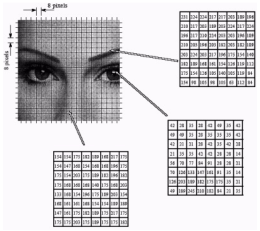
The decision problem for Vertex Cover is NP-complete. Moreover, unless P=NP, vertex cover can't be approximated within a factor $\alpha \leq 1.36$

Data Compression problem

Given as input a text $\mathcal{T}$ over an finite alphabet Σ. We want to 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.



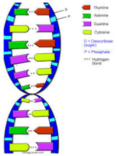
231	224	224	217	217	203	189	196
210	217	203	189	203	224	217	224
196	217	210	224	203	203	196	189
210	203	196	203	182	203	182	189
203	224	203	217	196	175	154	140
182	189	168	161	154	126	119	112
175	154	126	105	140	105	119	84
154	98	105	98	105	63	112	84

42	28	35	28	42	49	35	42
49	49	35	28	35	35	35	42
42	21	21	28	42	35	42	28
21	35	35	42	42	28	28	14
56	70	77	84	91	28	28	21
70	126	133	147	161	91	35	14
126	203	189	182	175	175	35	21
49	189	245	210	182	84	21	35

154	154	175	182	189	168	217	175
154	147	168	154	168	168	196	175
175	154	203	175	189	182	196	182
175	168	168	168	140	175	168	203
133	168	154	196	175	189	203	154
168	161	161	168	154	154	189	189
147	161	175	182	189	175	217	175
175	175	203	175	189	175	175	182

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):
000101100110100000101

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

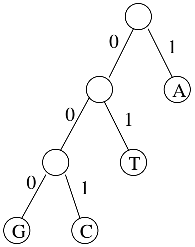
Prefix tree.

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

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

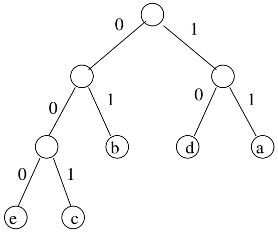
Given ϕ, $\alpha = \sum_{x \in \Sigma} f(x)|\phi(x)|$ is the average number of bits required per symbol.

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

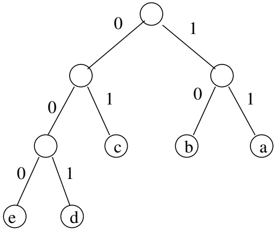


$\alpha = .32 \cdot 2 + .25 \cdot 2 + .20 \cdot 3 + .18 \cdot 2 + .05 \cdot 3 = 2.25$
In average, ϕ_1 reduces the bits per symbol over the fixed-length code from 3 to 2.25, about 25%

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

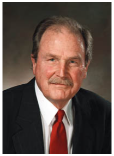
The binary prefix tree corresponding to an optimal prefix code is full.

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. □

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.

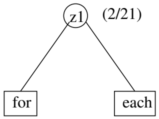
Huffman Coding: Construction of the tree.

```
**Huffman** $\Sigma, S$
Given $\Sigma$ and $S$ {compute the frequencies $\{f\}$}
Construct priority queue $Q$ of $\Sigma$, 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 $O(n \lg n)$.

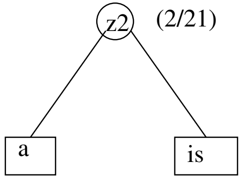
Example

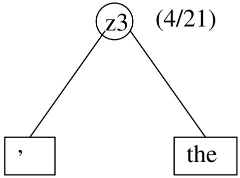
Consider the text: *for each rose, a rose is a rose, the rose*
with $\Sigma = \{$for/ each/ rose/ a/ is/ the/ ,/ $\flat$ $\}$
Frequencies: f(for) $= 1/21$, f(rose) $= 4/21$, f(is) $= 1/21$,
f(a) $= 2/21$, f(each) $= 1/21$, f(,) $= 2/21$, f(the) $= 1/21$,
$f(\flat) = 9/21$.
Priority Queue:
Q=(for(1/21),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

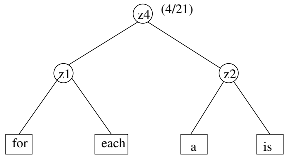


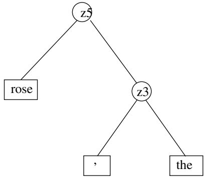
Q=(a(1/21), is(1/21), ,(2/21), the(2/21), z1(2/21), rose(4,21), $\flat$
(9/21))

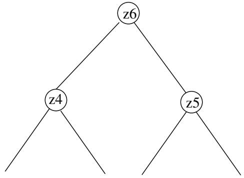
Example.

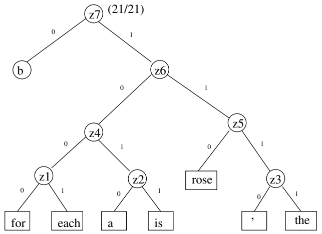

z2
(2/21)
a
is


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:
10000100101101110010100110010110101001101110011110110

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)