Greedy Algorithms

Hee-Kap Ahn
Graduate School of Artificial Intelligence
Dept. Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)

Textbooks: Algorithms by Jeff Erickson /Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.

Selecting Breakpoints

Consider the following scheduling problem.

- Road trip from Pohang to Seoul along fixed route.
- Fuel capacity $= C$. Refueling stations at certain points along the way.
- Goal: make as few refueling stops as possible.

Greedy algorithm (optimal): Go as far as you can before refueling.

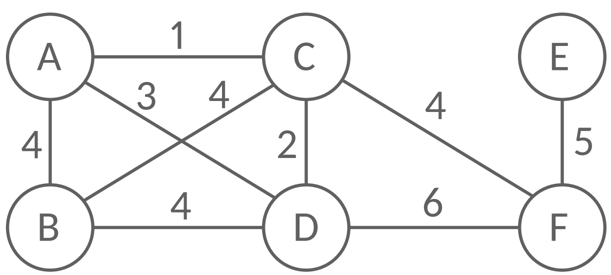
```
function SelectBreakpoints(B)
Sort breakpoints so that 0 = b_0 ≤ b_1 ≤ ··· ≤ b_n = L
S = {0}
x = 0
while x ≠ b_n do
    let p be the largest integer such that b_p ≤ x + C
    if b_p = x then
        return "no solution"
    x = b_p
    S = S ∪ {p}
return S
```

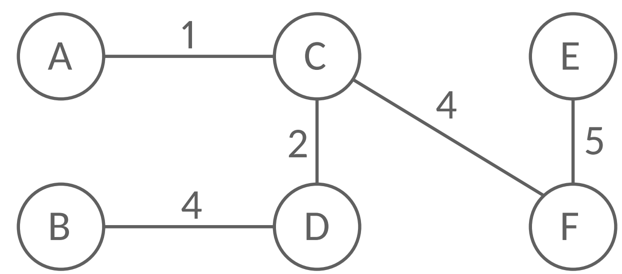
Greedy Algorithms

thinking ahead vs. choosing immediate advantage

Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obivious and immediate benefit.

What is the cheapest possible (connected) network of the following graph?

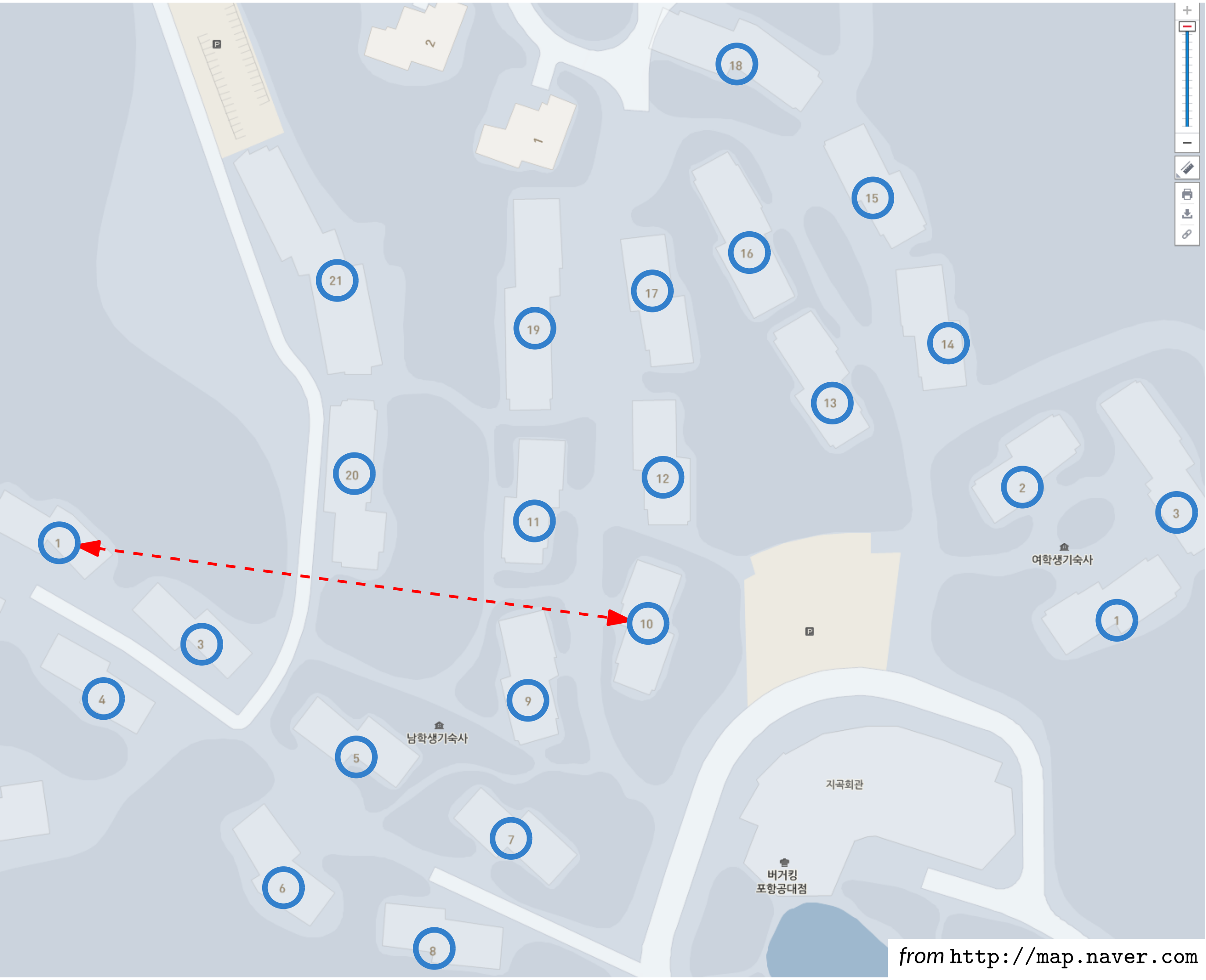




Property 1 Removing a cycle edge does not disconnect a graph.

Connected and acyclic graphs $\Rightarrow$ *Trees* (undriected graphs).

Minimum Spanning Trees



All pairwise distances are given.

Minimum Spanning Trees

from `http://map.naver.com`

When Euclidean distances are used.

Minimum Spanning Trees

from `http://map.naver.com`

When geodesic distances are used.

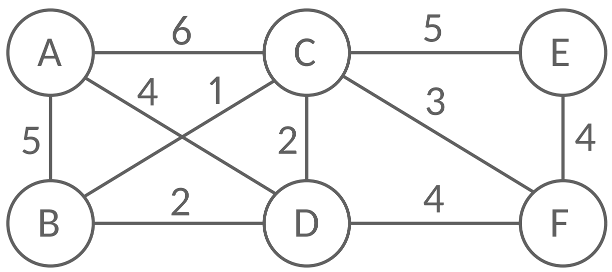
Input: An undirected graph $G = (V, E)$; edge weights w_e.
Output: A tree $T = (V, E')$, with $E' \subseteq E$, that minimizes

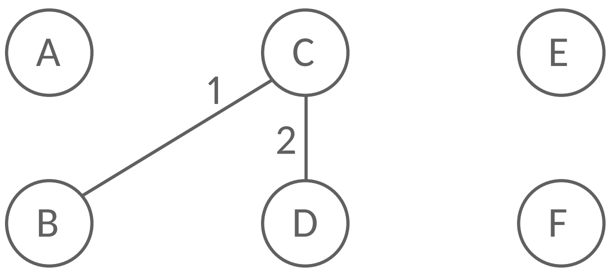
$$\text{weight}(T) = \sum_{e \in E'} w_e.$$

A greedy approach (Kruskal's algorithm): starts with the empty graph and selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

That is, take the most obvious immediate advantage in every decision!





Minimum Spanning Trees - Kruskal

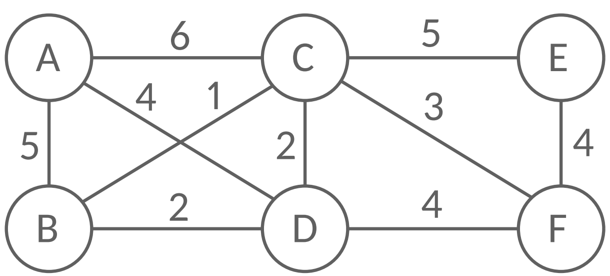
Input: An undirected graph $G = (V, E)$; edge weights w_e.
Output: A tree $T = (V, E')$, with $E' \subseteq E$, that minimizes

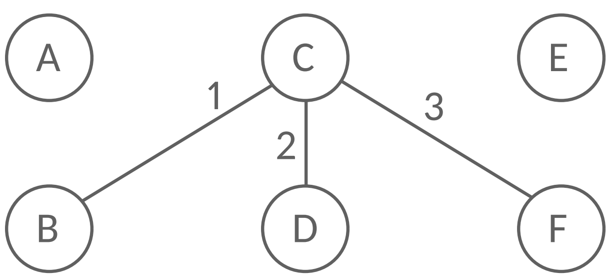
$$\text{weight}(T) = \sum_{e \in E'} w_e.$$

A greedy approach (Kruskal's algorithm): starts with the empty graph and selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

That is, take the most obvious immediate advantage in every decision!





Minimum Spanning Trees - Kruskal

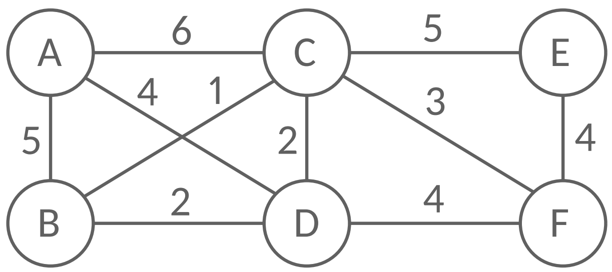
Input: An undirected graph $G = (V, E)$; edge weights w_e.
Output: A tree $T = (V, E')$, with $E' \subseteq E$, that minimizes

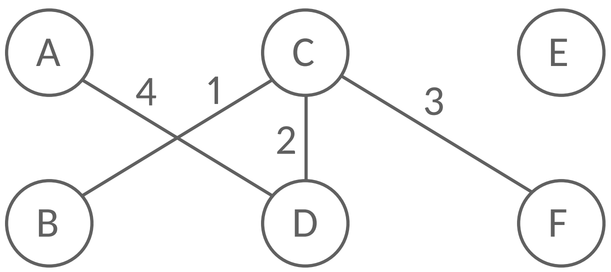
$$\text{weight}(T) = \sum_{e \in E'} w_e.$$

A greedy approach (Kruskal's algorithm): starts with the empty graph and selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

That is, take the most obvious immediate advantage in every decision!





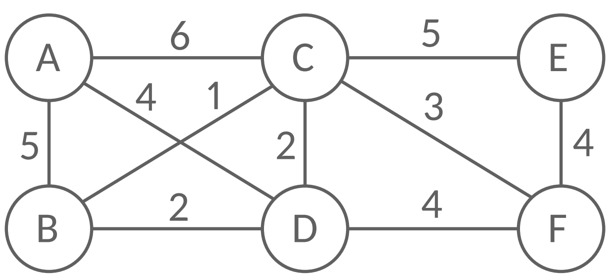
Input: An undirected graph $G = (V, E)$; edge weights w_e.
Output: A tree $T = (V, E')$, with $E' \subseteq E$, that minimizes

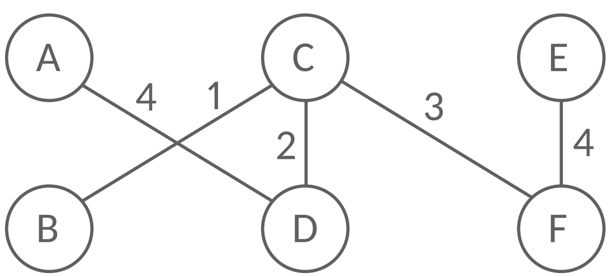
$$\text{weight}(T) = \sum_{e \in E'} w_e.$$

A greedy approach (Kruskal's algorithm): starts with the empty graph and selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

That is, take the most obvious immediate advantage in every decision!





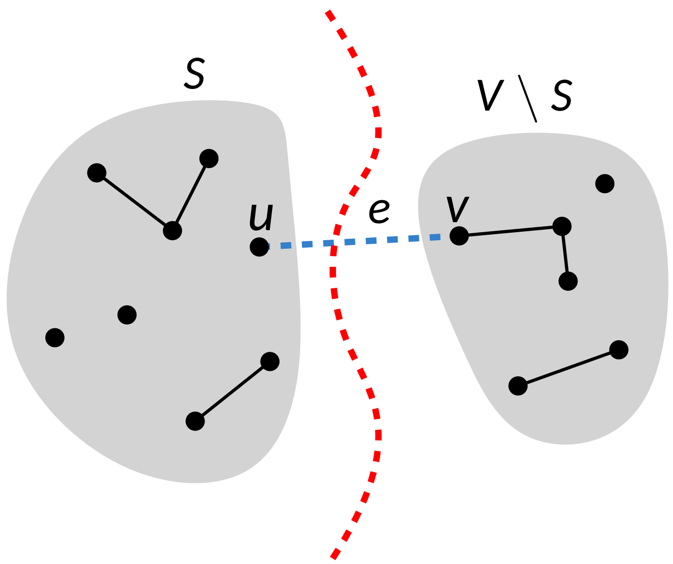
Minimum Spanning Trees - Kruskal

A greedy approach (Kruskal's algorithm): starts with the empty graph and selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

The correctness of Kruskal's method follows from a certain cut property.

Cut property Suppose edges in $X \subset E$ are part of a minimum spanning tree T of $G = (V, E)$. Pick any subset of nodes S for which no edge in X crosses between S and $V \setminus S$, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.

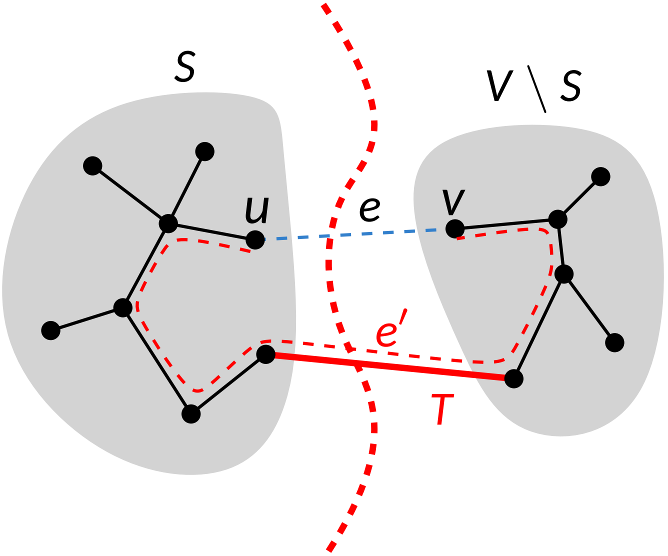


Cut property

Cut property Suppose edges in $X \subset E$ are part of a minimum spanning tree T of $G = (V, E)$. Pick any subset of nodes S for which no edge in X crosses between S and $V \setminus S$, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.

Proof. Assume $e = (u, v) \notin T$. We can construct a different MST T' containing $X \cup \{e\}$ by altering T slightly.

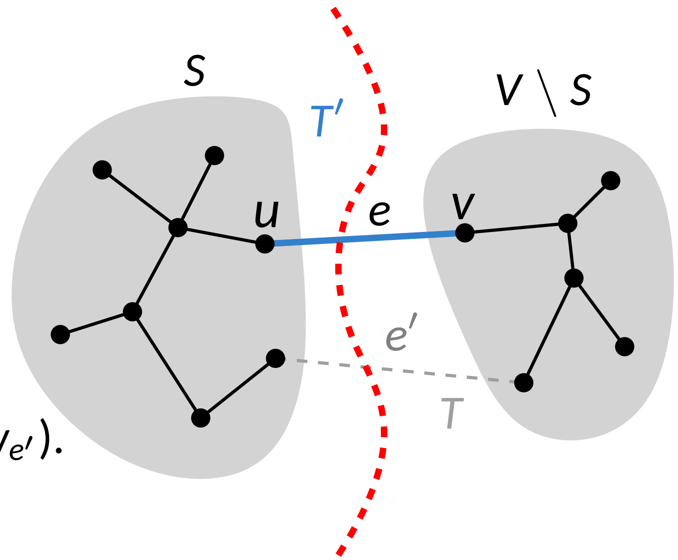
- u and v are connected by a path in T which contains an edge e' crossing S and $V \setminus S$.



Cut property Suppose edges in $X \subset E$ are part of a minimum spanning tree T of $G = (V, E)$. Pick any subset of nodes S for which no edge in X crosses between S and $V \setminus S$, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.

Proof. Assume $e = (u, v) \notin T$. We can construct a different MST T' containing $X \cup \{e\}$ by altering T slightly.

- u and v are connected by a path in T which contains an edge e' crossing S and $V \setminus S$.
- Construct a tree T' from T - remove e' and add e.
- T' is a spanning tree with $\text{cost}(T') \leqslant \text{cost}(T)$ ($\because w_e \leqslant w_{e'}$). So T' is an MST of G.



Minimum Spanning Trees - Kruskal

A greedy approach (Kruskal's algorithm): starts with the empty graph and selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

function **Kruskal**(G, w)

for all $u \in V$ **do**
    makeset(u)
$X = \{\}$
Sort the edges in E by weight
for all edges $(u, v) \in E$, in increasing order of weight **do**
    if find$(u) \neq$ find(v) **then**
        add edge (u, v) to X
        union(u, v)

At any given moment,

- the set X forms a partial solution, a collection of trees.
- the next edge e to be added is the lightest one among edges connecting two of these trees.

Minimum Spanning Trees - Kruskal

A greedy approach (Kruskal's algorithm): starts with the empty graph and selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

function **Kruskal**(G, w)

for all $u \in V$ **do**
  makeset(u)
$\boxed{X = \{\}}$
Sort the edges in E by weight
for all edges $(u, v) \in E$, in increasing order of weight **do**
  if $\boxed{\text{find}(u) \neq \text{find}(v)}$ **then**
    add edge (u, v) to X
    $\boxed{\text{union}(u, v)}$

$|V|$ `makeset`, $2|E|$ `find`, $|V| - 1$ `union` operations.

At any given moment,

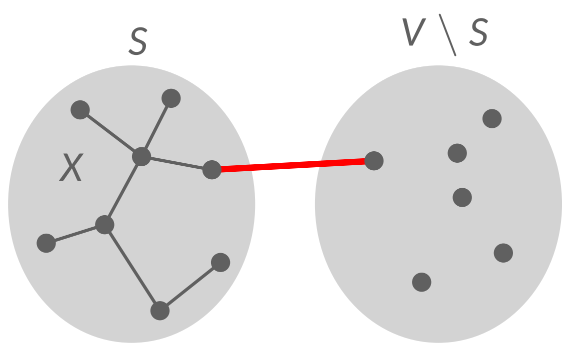
- the set X forms a partial solution, a collection of trees.
- the next edge e to be added is the lightest one among edges connecting two of these trees.

Minimum Spanning Trees - Prim

Prim's algorithm: choose the next edge s.t. the intermediate set X of edges forms a subtree, that is, the subtree X grows by the lightest edge between a vertex $v \in S$ and a vertex $z \notin S$.

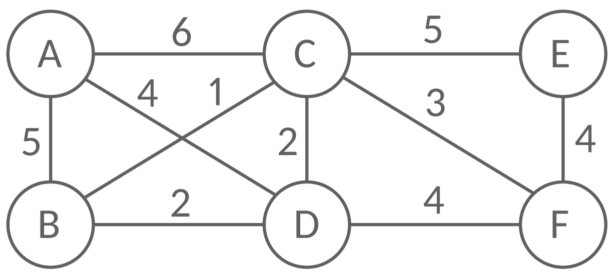
function **Prim**(G, w)

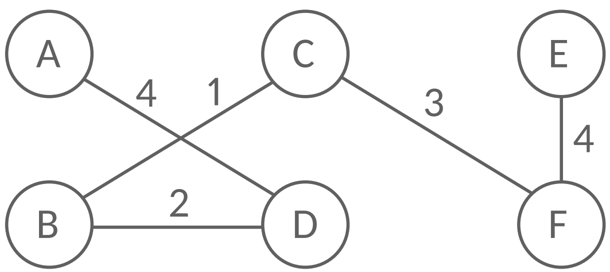
for all $u \in V$ **do**
  cost$(u) = \infty$; prev$(u) = \texttt{nil}$;
Pick any initial node u_0 and set cost$(u_0) = 0$
$H =$ makequeue(V)
while H is not empty **do**
  $v =$ deletemin(H)
  for each edge $(v, z) \in E$ **do**
    if cost$(z) > w(v, z)$ **then**
      cost$(z) = w(v, z)$; prev$(z) = v$;
      decreasekey(H, z)



$O(|E| \log |V|)$

Prim's algorithm: choose the next edge s.t. the intermediate set X of edges forms a subtree, that is, the subtree X grows by the lightest edge between a vertex $v \in S$ and a vertex $z \notin S$.





Set S	A	B	C	D	E	F
{}	$0/\texttt{nil}$	$\infty/\texttt{nil}$	$\infty/\texttt{nil}$	$\infty/\texttt{nil}$	$\infty/\texttt{nil}$	$\infty/\texttt{nil}$
A		$5/A$	$6/A$	$4/A$	$\infty/\texttt{nil}$	$\infty/\texttt{nil}$
A, D		$2/D$	$2/D$		$\infty/\texttt{nil}$	$4/D$
A, D, B			$1/B$		$\infty/\texttt{nil}$	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

cost$(z)/$prev(z)

Datastructure for Disjoint Sets

Maintaining a collection of disjoint sets under the operation of **union**:

- makeset(x): create a new set containing x.
- find(x): return the *root* node of the set containing x.
- union(x, y): form a union of two sets containing x and y.

For a node x,

- $\pi(x)$ denotes a pointer to its parent, and
- rank(x) denotes the height of the subtree hanging from x.

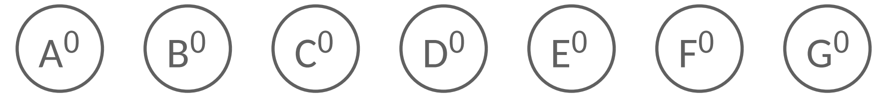
```
function makeset(x)
  π(x) ← x
  rank(x) ← 0
```

```
function find(x)
  while x ≠ π(x) do
      x ← π(x)
  return x
```

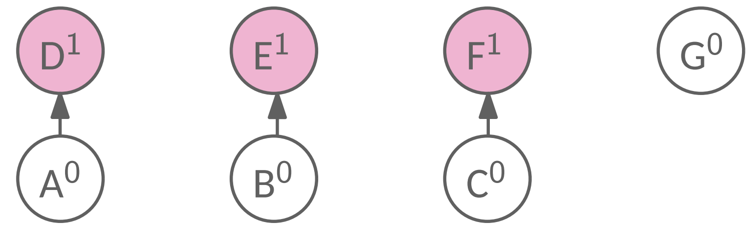
```
function union(x, y)
  r_x ← find(x); r_y ← find(y)
  if r_x = r_y then
      return
  if rank(r_x) > rank(r_y) then
      π(r_y) ← r_x
  else
      π(r_x) ← r_y
      if rank(r_x) = rank(r_y) then
          rank(r_y) ← rank(r_y) + 1
```

Datastructure for Disjoint Sets

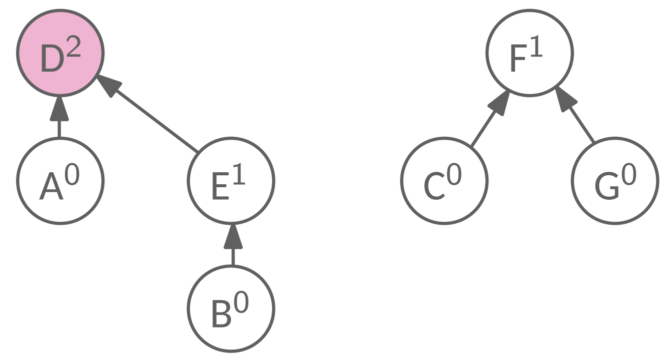
After $\texttt{makeset}(A), \texttt{makeset}(B), \ldots, \texttt{makeset}(G)$:



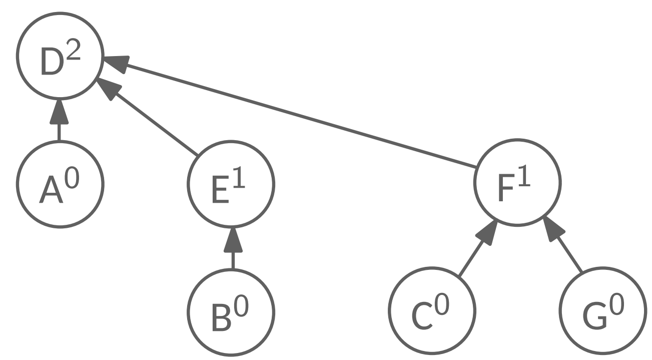
After $\texttt{union}(A, D), \texttt{union}(B, E), \texttt{union}(C, F)$:



After $\texttt{union}(C, G), \texttt{union}(E, A)$:



After $\texttt{union}(B, G)$:



Properties of union and find DS

A few properties of union-find data structures.

1. For any x, $\text{rank}(x) \leqslant \text{rank}(\pi(x))$.
2. If x is not a root, $\text{rank}(x)$ will never change again.

   Rank changes only for roots. A nonroot node never becomes a root.
3. Any root node of rank k has at least 2^k nodes in its tree.
4. If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.

   $\rightarrow$ the max. rank is $\log n$, so the tree height $\leqslant \log n$.

Property 3. Proof by induction on k.

- It holds for $k = 0$.
- Assume that the claim holds for $k - 1$.
- A node x of rank k is created only by merging two roots of rank $k - 1$, each subtree consisting of $\geqslant 2^{k-1}$ nodes. Thus, x has $\geqslant 2^k$ nodes in its subtree.

Kruskal's algorithm takes

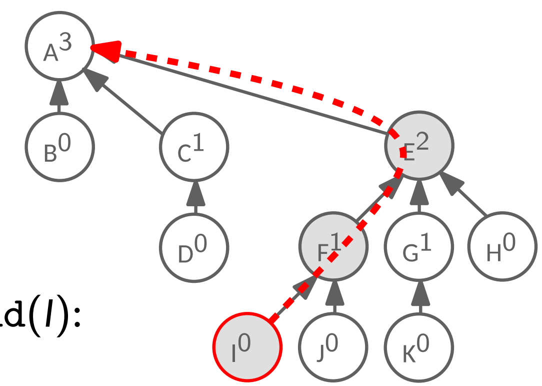
- $O(|E| \log |V|)$ for sorting, plus
- $O(|E| \log |V|)$ for union and find operations.

Path Compression

If the time for sorting doesn't count,

- we keep the tree short by *path compression*, that is,
- for each find(x), we make the nodes on the path to point the root.

find(I):



function **find**(x)

if $x \neq \pi(x)$ **then**
    $\pi(x) \leftarrow$ find($\pi(x)$)
return $\pi(x)$

~~function **find**(x)~~

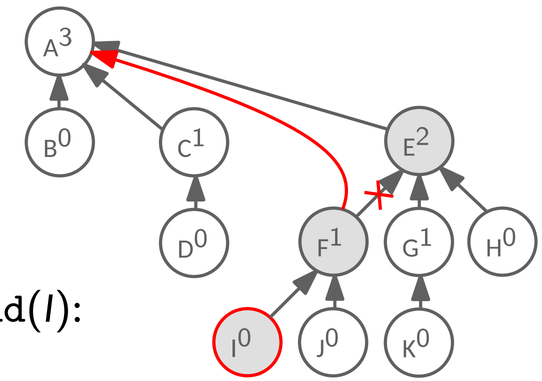
~~**while** $x \neq \pi(x)$ **do**~~
~~ $x \leftarrow \pi(x)$~~
~~return x~~

Path Compression

If the time for sorting doesn't count,

- we keep the tree short by *path compression*, that is,
- for each find(x), we make the nodes on the path to point the root.

find(I):



function **find**(x)

if $x \neq \pi(x)$ **then**

  $\pi(x) \leftarrow \text{find}(\pi(x))$

return $\pi(x)$

~~function **find**(x)~~

~~**while** $x \neq \pi(x)$ **do**~~

~~$x \leftarrow \pi(x)$~~

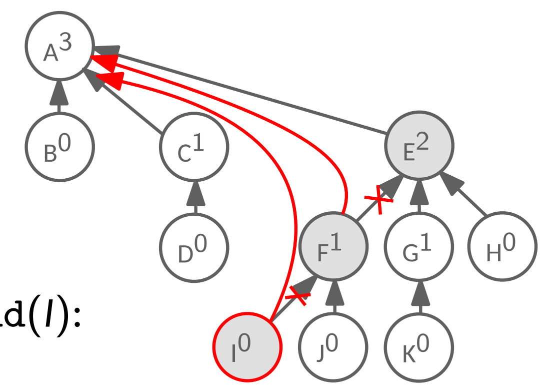
~~return x~~

Path Compression

If the time for sorting doesn't count,

- we keep the tree short by *path compression*, that is,
- for each find(x), we make the nodes on the path to point the root.

`find(`I`):`



function **find**(x)

if $x \neq \pi(x)$ **then**

$\quad \pi(x) \leftarrow \text{find}(\pi(x))$

return $\pi(x)$

~~function **find**(x)~~

~~**while** $x \neq \pi(x)$ **do**~~

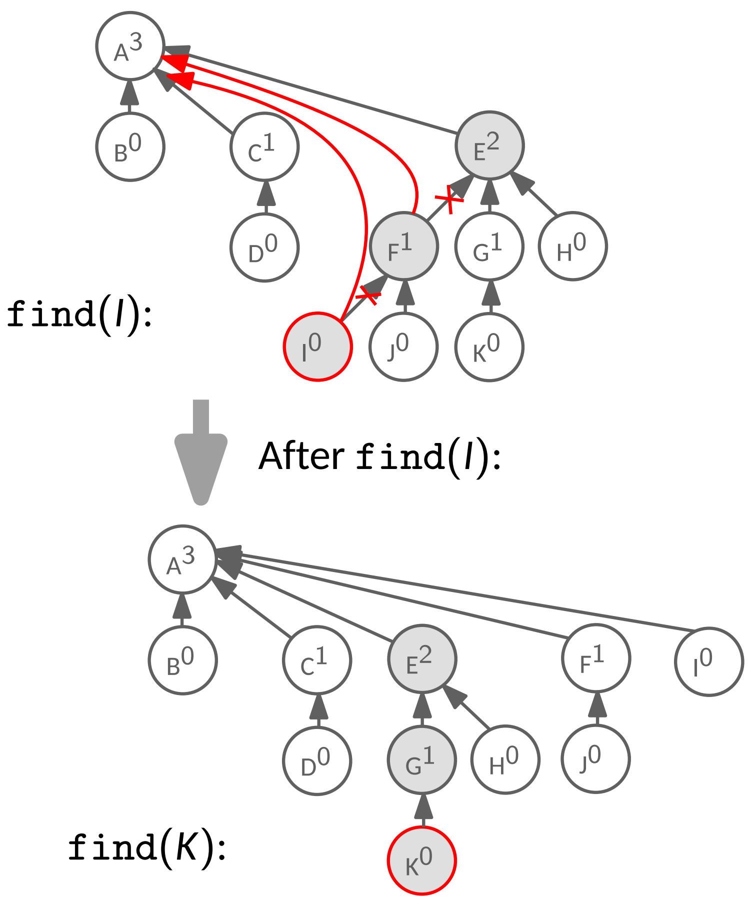
~~$\quad x \leftarrow \pi(x)$~~

~~return x~~

Path Compression

If the time for sorting doesn't count,

- we keep the tree short by *path compression*, that is,
- for each find(x), we make the nodes on the path to point the root.

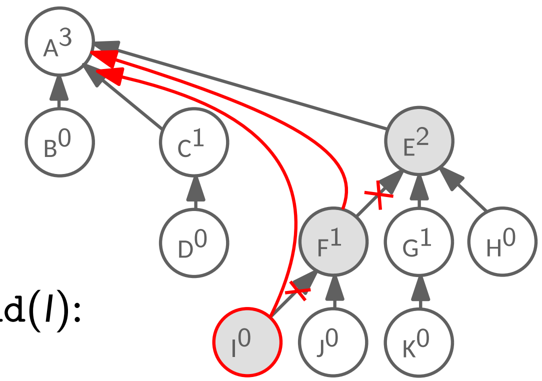


function **find**(x)
if $x \neq \pi(x)$ **then**
  $\pi(x) \leftarrow \text{find}(\pi(x))$
return $\pi(x)$

~~function **find**(x)~~
~~**while** $x \neq \pi(x)$ **do**~~
  ~~$x \leftarrow \pi(x)$~~
~~return x~~

If the time for sorting doesn't count,
- we keep the tree short by *path compression*, that is,
- for each find(x), we make the nodes on the path to point the root.

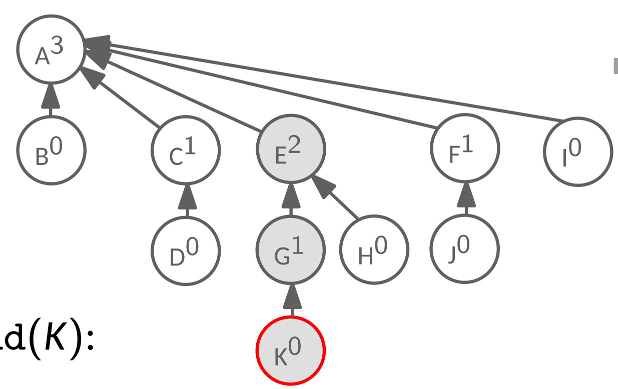
find(I):



function **find**(x)
if $x \neq \pi(x)$ **then**
$\quad \pi(x) \leftarrow \text{find}(\pi(x))$
return $\pi(x)$

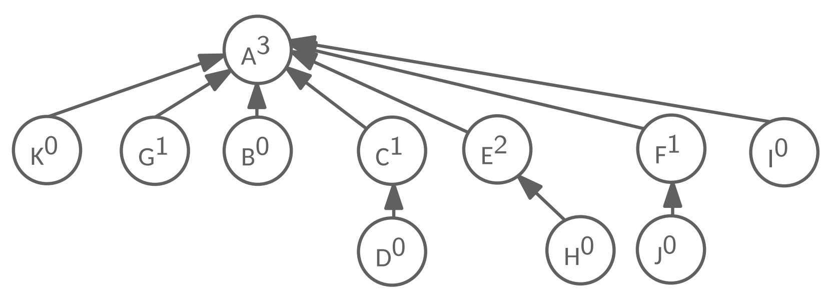
~~function **find**(x)~~
~~**while** $x \neq \pi(x)$ **do**~~
~~$\quad x \leftarrow \pi(x)$~~
~~return x~~

After find(I):



find(K):

After find(K):



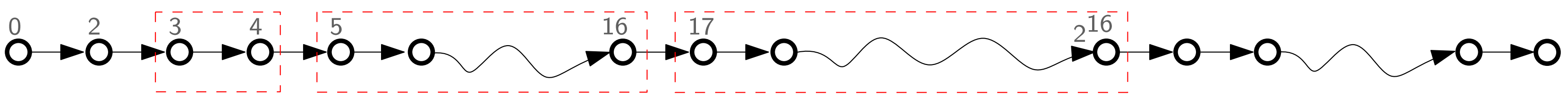
Path Compression

We now look at an intermixed sequence of find and union operations, starting from an empty data structure.

- # union operations is at most $n - 1$ for n nodes.
- We analyze the average time per operation $\rightarrow$ amortized cost

Ranks, ranging $0 \leqslant \text{rank}(x) \leqslant \log n$, form groups of interval $\{k+1, k+2, \ldots, 2^k\}$ for k, a power of 2, from 0.

$$\underset{2^0}{\{1\}}, \underset{2^1}{\{2\}}, \underset{2^2}{\{3,4\}}, \underset{2^4}{\{5,6,\ldots,16\}}, \underset{2^{16}=65536}{\{17,18,\ldots,2^{16}\}}, \underset{2^{2^{16}}}{\{65537,65538,\ldots,2^{65536}\}}, \ldots$$



groups $= \log^* n$

$\log^* n$: the number of successive log operations that need to be applied to n to bring it down to 1 (or below). In other words,

$$\log^* n = \begin{cases} 0 & \text{for } n \leqslant 1, \\ 1 + \log^*(\log_2 n) & \text{for } n > 1. \end{cases}$$

$$\log^* 2^{2^2} = 1 + \log^* 2^2 = 1 + 1 + \log^* 2 = 1 + 1 + 1 + \log^* 1 = 3.$$

Path Compression

Each find operation is given $\log^* n$ dollars in its pocket.

Each node x is given 2^k dollars if $\text{rank}(x)$ lies in the interval $\{k+1, \dots, 2^k\}$.

- # nodes with rank $> k$ is $\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leqslant \frac{n}{2^k}$,
- the total money given to nodes in this interval is $\leqslant n$ dollars, and
- **the total money given to all nodes is $\leqslant n \log^* n$ dollars.**

Path Compression

Each find operation is given $\log^* n$ dollars in its pocket.

Each node x is given 2^k dollars if $\mathrm{rank}(x)$ lies in the interval $\{k+1, \ldots, 2^k\}$.

- # nodes with rank $> k$ is $\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leqslant \frac{n}{2^k}$,
- the total money given to nodes in this interval is $\leqslant n$ dollars, and
- **the total money given to all nodes is $\leqslant n \log^* n$ dollars.**

Consider the chain of nodes by find(u), and let x be a node in the chain. There are two categories of nodes x on the chain, depending on the parents $\pi(x)$:

A. x and $\pi(x)$ belong to different intervals.

find(u) pays 1 dollar using its pocket money.
At most $\log^* n$ such nodes.

B. x and $\pi(x)$ belong to the same interval.

x pays 1 dollar using its pocket money.

Each find operation is given $\log^* n$ dollars in its pocket.

Each node x is given 2^k dollars if $\text{rank}(x)$ lies in the interval $\{k+1, \ldots, 2^k\}$.

- # nodes with rank $> k$ is $\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leqslant \frac{n}{2^k}$,
- the total money given to nodes in this interval is $\leqslant n$ dollars, and
- **the total money given to all nodes is $\leqslant n \log^* n$ dollars.**

Consider the chain of nodes by find(u), and let x be a node in the chain. There are two categories of nodes x on the chain, depending on the parents $\pi(x)$:

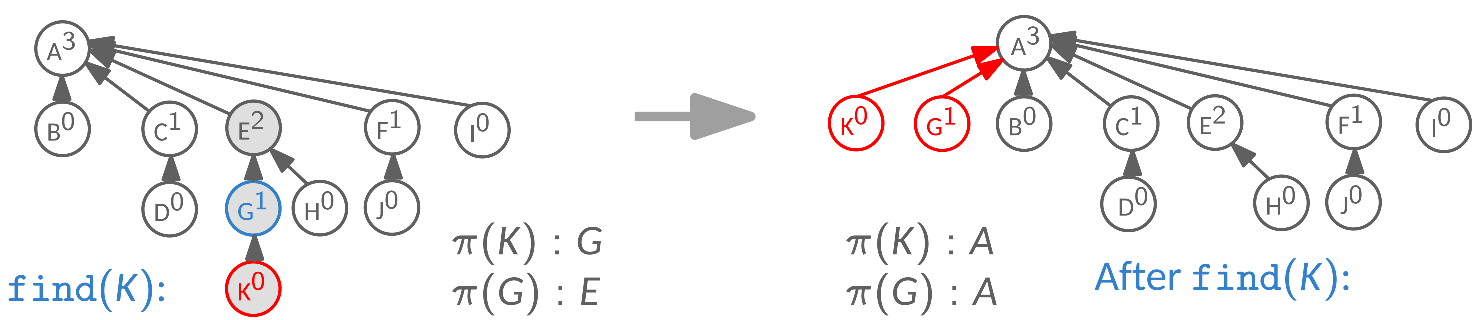
A. x and $\pi(x)$ belong to different intervals.

find(u) pays 1 dollar using its pocket money.
At most $\log^* n$ such nodes.

B. x and $\pi(x)$ belong to the same interval.

x pays 1 dollar using its pocket money.

Each time x pays 1 dollar, $\pi(x)$ changes to the root node of higher rank.
($\text{rank}(\pi(x))$ increases at least by 1.)



Path Compression

Each find operation takes $O(\log^* n)$ steps plus some additional amount of time.

Consider the chain of nodes by find(u), and let x be a node in the chain. There are two categories of nodes x on the chain, depending on the parents $\pi(x)$:

A. x and $\pi(x)$ belong to different intervals.

find(u) pays 1 dollar using its pocket money. At most $\log^* n$ such nodes.

B. x and $\pi(x)$ belong to the same interval.

x pays 1 dollar using its pocket money.

(1) Each time x pays 1 dollar, $\pi(x)$ changes to a node of higher rank.

(2) If rank(x) lies in the interval $\{k+1, k+2, \ldots, 2^k\}$, it has to pay at most 2^k dollars before rank$(\pi(x))$ is in a higher interval;

(3) Once rank$(\pi(x))$ is in a higher group than rank(x), it remains so. Thus, x and $\pi(x)$ belong to A and x never has to pay again! (Instead, find pays.)

(4) Once a root node becomes nonroot, it stays as nonroot and its rank remains the same.

An intermixed sequence of m find and $n-1$ union operations can be done in $O(m \log^* n + n \log^* n) = O(m \log^* n)$ time $(m \geqslant n)$.

Path Compression

- (1972) Fischer derived an upper bound of $(m \log \log n)$.
- (1973) Hopcroft and Ullman improved the bound to $O(m \log^* n)$.
- (1975) Tarjan obtained the actual worst-case bound, $\Theta(m\alpha(m, n))$, where $\alpha(m, n)$ is a functional inverse of Ackerman's function which grows very slow. For example, $\alpha(m, n) \leqslant 3$ for $n < 2^{16} = 65,536$.

Remarks. Path compression requires two passes over the find path, one to find the tree root and another to perform the compression. Tarjan and Leeuwen studied a number of one-pass variants, some of which run in $O(m\alpha(m, n))$ time. For example, the following program implements *path halving*.

```
function find(x)
while π(π(x)) ≠ π(x) do
    x ← π(x) ← π(π(x))
return π(x)
```

Is the bound, $O(m\alpha(m, n))$ tight? Yes, there are sequences of set operations that actually take $\Omega(m\alpha(m, n))$ time (Tarjan 1975).

Interval Scheduling

Consider the following scheduling problem.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

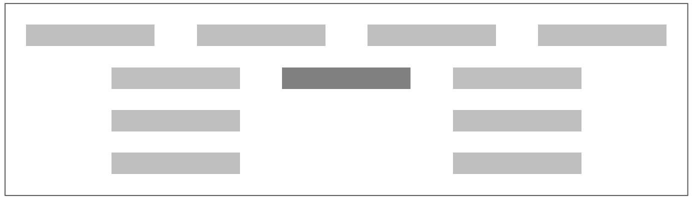
Interval Scheduling

Consider the following scheduling problem.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

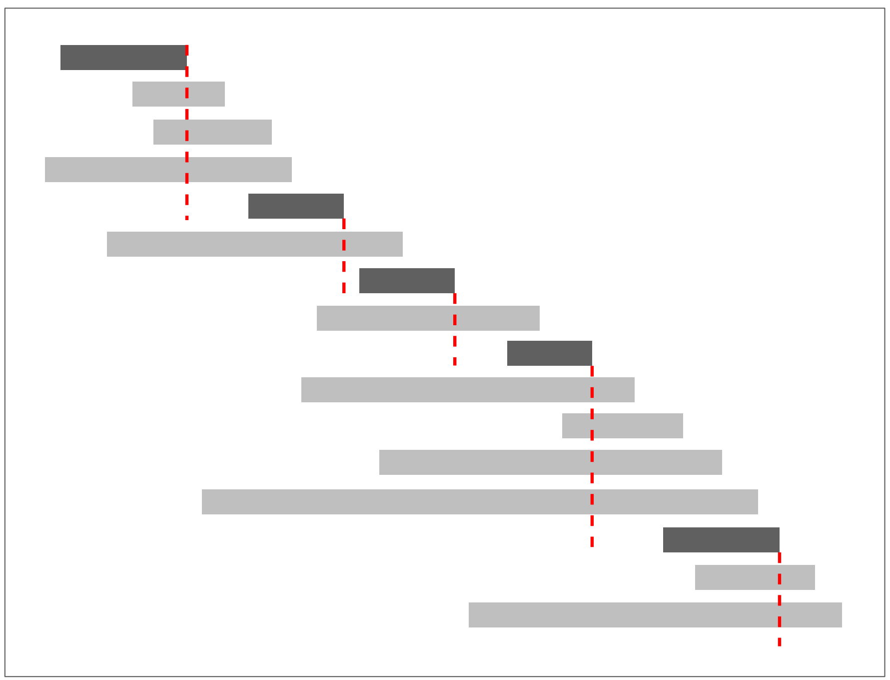
Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

- Earliest start time: in ascending order of start time s_j.
- Earliest finish time: in ascending order of finish time f_j.
- Shortest interval: in ascending order of interval length $f_j - s_j$.
- Fewest conflicts: in ascending order of conflicts c_j.



Interval Scheduling

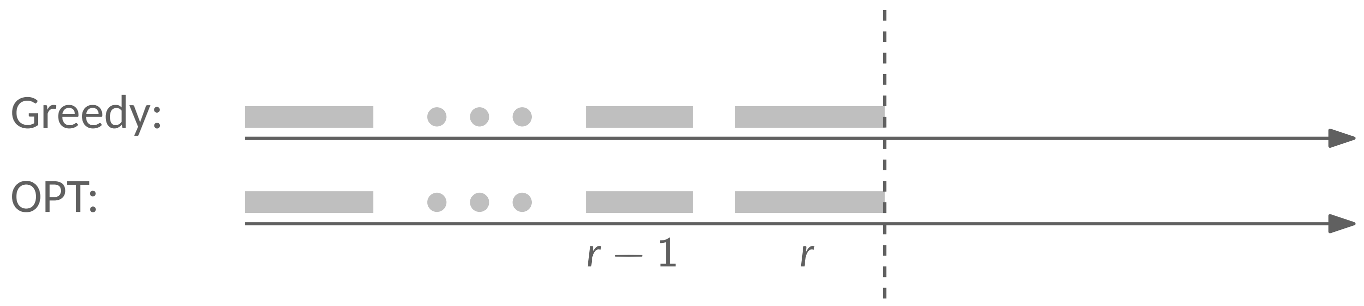
Earliest finish time. Consider jobs in ascending order of finish time f_j. Take each job provided it is compatible with the ones already taken.



Earliest finish time. Consider jobs in ascending order of finish time f_j. Take each job provided it is compatible with the ones already taken.

Theorem The greedy algorithm above is optimal.

Proof. Assume greedy is not optimal, and let's see what happens.
Let $i_1, i_2, \ldots, i_k$ denote the set of jobs selected by greedy. Let $j_1, j_2, \ldots, j_m$ denote the set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

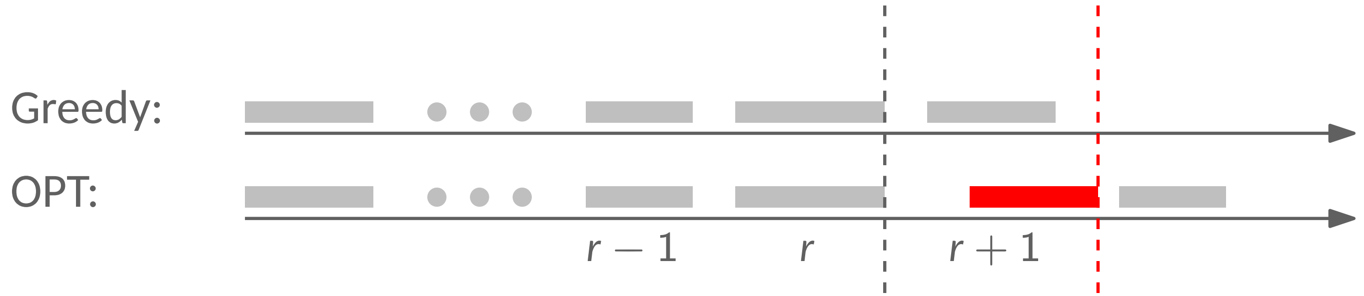


Earliest finish time. Consider jobs in ascending order of finish time f_j. Take each job provided it is compatible with the ones already taken.

Theorem The greedy algorithm above is optimal.

Proof. Assume greedy is not optimal, and let's see what happens.
Let $i_1, i_2, \ldots, i_k$ denote the set of jobs selected by greedy. Let $j_1, j_2, \ldots, j_m$ denote the set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

By the greedy choice, job i_{r+1} finishes before j_{r+1}. So in the optimal solution, we can replace job j_{r+1} with job i_{r+1}. The resulting solution is still feasible and optimal, but contradicts the maximality of r.



Interval Partitioning

There are n lectures. Lecture j starts at s_j and finishes at f_j.
Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same classroom.

The depth of a set of open intervals is the maximum number of intervals that contain any given time. Then the number of classrooms needed $\geqslant$ depth.

Does there always exist a schedule equal to depth of intervals?

function **IntervalPartition**(x, y)

Sort lectures by starting time
$d = 0$
for $j = 1$ to n **do**
  if lecture j is compatible with a classroom k opened so far **then**
    schedule lecture j in classroom k
  else
    allocate a new classroom $d + 1$
    schedule lecture j in classroom $d + 1$
    $d = d + 1$

Implementation: $O(n \log n)$. For each classroom k, maintain the finish time of the last job added. Keep the classrooms in a priority queue.

Interval Partitioning

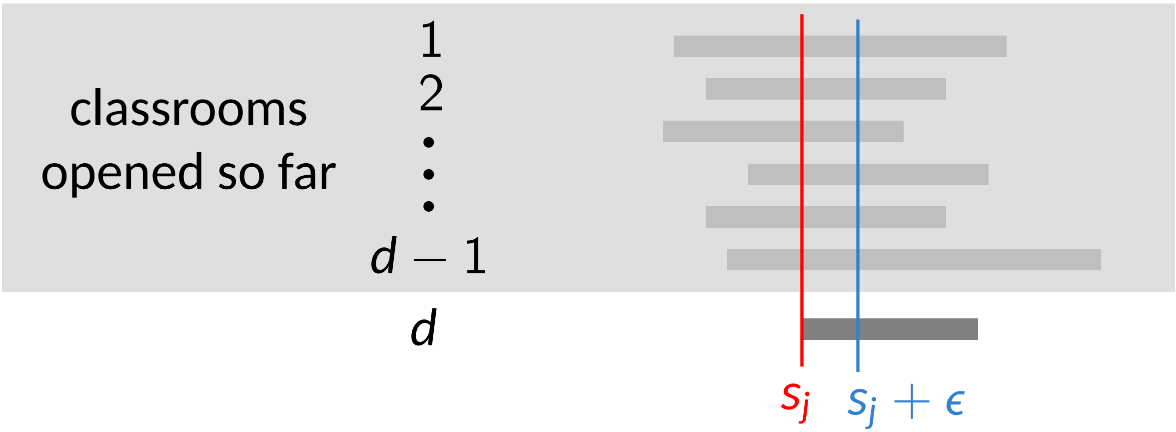
Theorem. Greedy algorithm is optimal.

Proof. Let d be the number of classrooms that the greedy algorithm allocates.

Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all $d-1$ other classrooms.

Since we sorted lectures by their start time, all these incompatibilities are caused by lectures that start earlier than or at s_j.

Thus, we have d lectures overlapping at time $s_j + \varepsilon$, which implies that any valid schedule uses at least d classrooms.



Huffman Encoding

Consider a string consisting of 130 million characters and the alphabet $\{A, B, C, D\}$.
`CDADBBADDDAABCCACDBBABABBCDCDCCCDDACCBDADCAB...`
Most economic way to write this long string *in binary*?

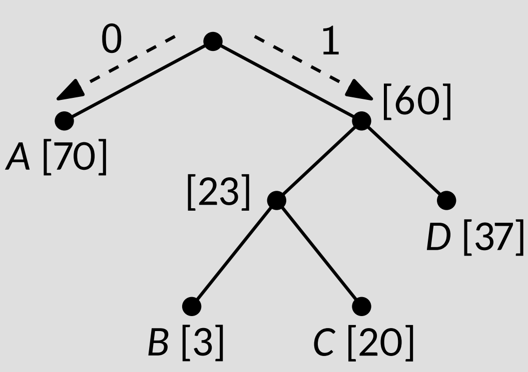
- **Two bits per symbol.** $\{A = 00, B = 01, C = 10, D = 11\} \Rightarrow$ 260 Mbits.
  `10110011010100111111000001...`
- Any **better encoding** than this?

What about using **variable-length encoding** with respect to the frequency.

But we need to devise a way to guarantee that decoding is **unique** $\Rightarrow$ **prefix-free**!

(No code word in the system is a prefix of any other code word in the system.)

Symbol	Codeword
A	0
B	100
C	101
D	11



A prefix-free encoding and its coding tree (full binary).

Huffman Encoding

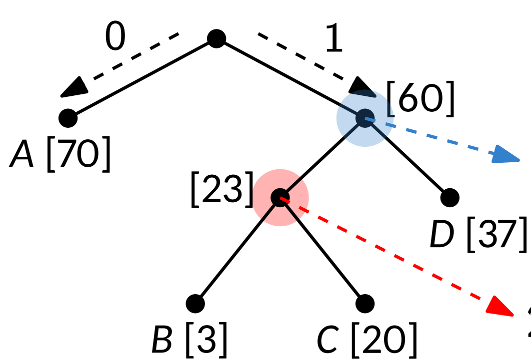
Given the frequencies $f_1, \dots, f_n$ of n symbols, find the optimal coding tree?
In other words, we want to find a (*full* binary) coding tree minimizing

$$\text{cost of tree} = \sum_{i=1}^{n} f_i \cdot (\text{depth(\# bits) of } i\text{th symbol in tree})$$

Another interpretation of the cost function is to define the *frequency* $f(v)$ of an internal node v as $f(v) = \sum_{\text{leaf } i \text{ in the subtree of } v} f_i$.

Then, $f(v)$ = number of times v is visited during encoding and decoding. Thus,

$$\text{cost of tree} = \sum_{\text{nonroot internal node } v} f(v) + \sum_{i=1}^{n} f_i$$



$60 = 3 + 20 + 37$

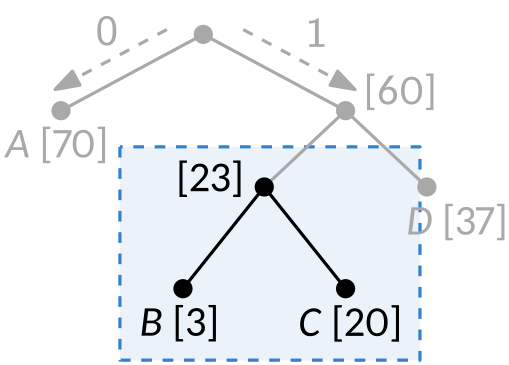
$23 = 3 + 20$

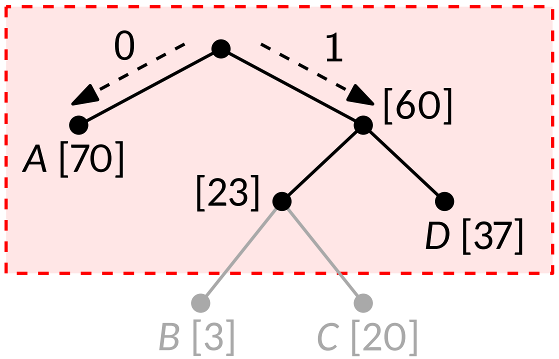
$\text{cost of tree} = 70*1+3*3+20*3+37*2 = 213$

$\text{cost of tree} = 23+60+70+3+20+37 = 213$

Huffman: Merge the two least frequent letters and recurse.

Symbol	Codeword
A	0
B	100
C	101
D	11





Lemma. Let x and y be the two least frequent letters. There is an optimal code tree in which x and y are siblings.

Proof. Let T be an optimal tree, with depth d. Because T is a full binary tree, it has two leaves at depth d that are siblings. Suppose they are not x and y, but some other letters a and b.

Let T' be the code tree obtained by swapping x and a, and let $\Delta = d - \text{depth}_T(x)$. Then $\text{cost}(T') = \text{cost}(T) + \Delta \cdot (f[x] - f[a])$.

Since $f[x] \leq f[a]$ and $\Delta \geq 0$, $\text{cost}(T') \leq \text{cost}(T)$. Since T is an optimal code tree, T' is also an optimal code tree.

Similarly, by swapping y and b, we can get another optimal code tree T''. Then x and y are siblings in T''.

Huffman Encoding

Lemma. Every Huffman code is an optimal prefix-free binary code.

Proof. $f[1:n]$: Input frequencies such that $f[1]$ and $f[2]$ are the two smallest frequencies. By the previous lemma, **1 and 2 are deepest siblings** in some optimal code tree for $f[1:n]$.

Let T' be the Huffman tree for $f[3:n+1]$, an optimal code tree for $f[3:n+1]$, where $f[n+1] = f[1] + f[2]$. Let T be the coding tree obtained from T' **by replacing the leaf $n+1$ with an internal node with two child nodes 1 and 2.**

We show that T is optimal for $f[1:n]$ by expressing $\text{cost}(T)$ in terms of $\text{cost}(T')$.

$$\begin{aligned}
\text{cost}(T) &= \sum_{i=1}^{n} f[i] \cdot \text{depth}(i) \qquad \text{depth}(i) = \text{depth of the leaf labeled } i \text{ in either } T \text{ or } T'. \\
&= \sum_{i=3}^{n+1} f[i] \cdot \text{depth}(i) + f[1] \cdot \text{depth}(1) + f[2] \cdot \text{depth}(2) - f[n+1] \cdot \text{depth}(n+1) \\
&= \text{cost}(T') + (f[1] + f[2]) \cdot \text{depth}(T) - f[n+1] \cdot (\text{depth}(T) - 1) \\
&= \text{cost}(T') + f[1] + f[2] + (f[1] + f[2] - f[n+1]) \cdot (\text{depth}(T) - 1) \\
&= \text{cost}(T') + f[1] + f[2]
\end{aligned}$$

Minimizing $\text{cost}(T)$ is equivalent to minimizing $\text{cost}(T')$. Attaching leaves labeled 1 and 2 to the leaf in T' labeled $n+1$ gives an optimal code tree for $f[1:n]$.

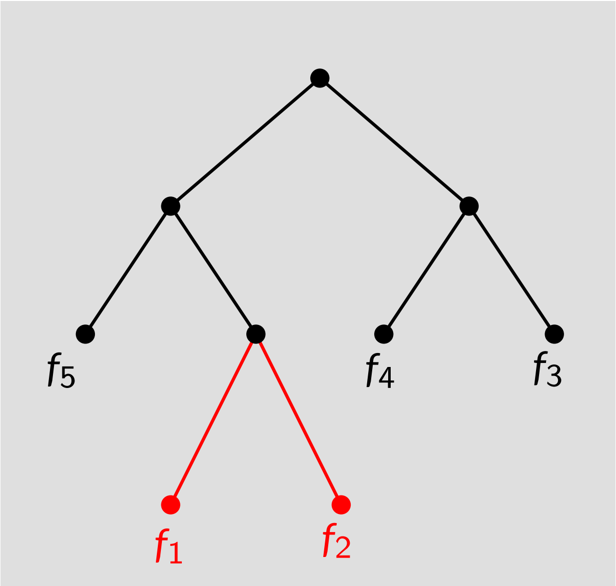
Huffman Encoding

$$\text{cost of tree} = \sum_{i=1}^{n} f_i \cdot (\text{depth(\# bits) of } i\text{th symbol in tree}) \tag{1}$$

$$= \sum_{\text{nonroot internal node } v} f(v) + \sum_{i=1}^{n} f_i \tag{2}$$

Huffman: Merge the two least frequent letters and recurse.

(1): two symbols with the smallest frequencies must be at the **bottom** of the optimal tree.
(Otherwise we can always find a better coding tree.)



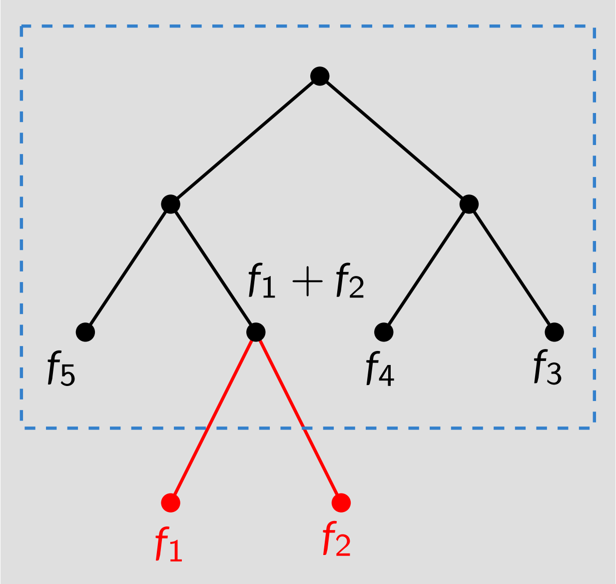
$$\text{cost of tree} = \sum_{i=1}^{n} f_i \cdot (\text{depth(\# bits) of } i\text{th symbol in tree}) \tag{1}$$

$$= \sum_{\text{nonroot internal node } v} f(v) + \sum_{i=1}^{n} f_i \tag{2}$$

Huffman: Merge the two least frequent letters and recurse.

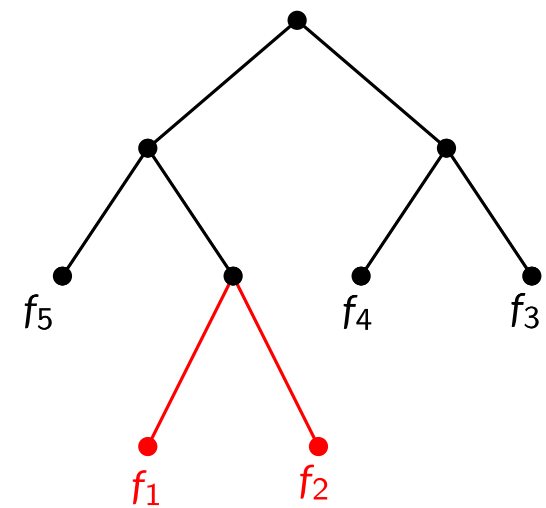
(1): two symbols with the smallest frequencies must be at the **bottom** of the optimal tree.
(Otherwise we can always find a better coding tree.)

(2): any tree with sibling-leaves f_1 and f_2 has cost $f_1 + f_2$ plus the cost for a tree with $n - 1$ leaves of frequences $(f_1 + f_2), f_3, f_4, \ldots, f_n$.



function **Huffman**(f)

makequeue(H)
for $i = 1$ to n **do**
  insert($H, i, f[i]$) (* number i with key $f[i]$ *)
for $k = n + 1$ to $2n - 1$ **do**
  $\boxed{i = \text{deletemin}(H), j = \text{deletemin}(H)}$ (1)
  Create a node numbered k with child nodes i, j.
  $f[k] = f[i] + f[j]$
  insert($H, k, f(k)$)

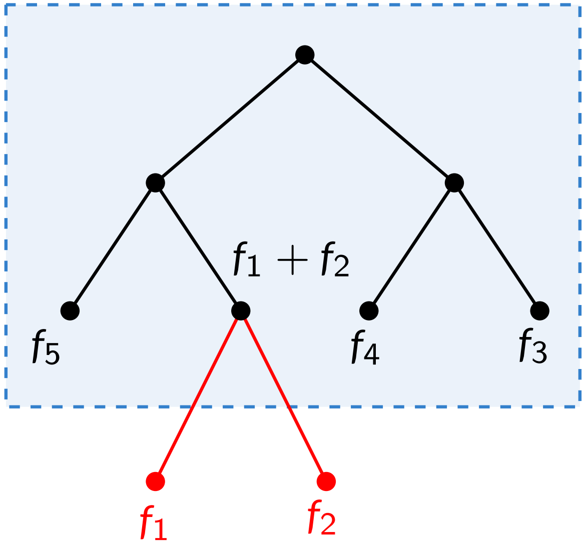


(1): two symbols with the smallest frequencies must be at the **bottom** of the optimal tree.

Huffman Encoding

function **Huffman**(f)

makequeue(H)
for $i = 1$ to n **do**
  insert$(H, i, f[i])$ (* number i with key $f[i]$ *)
for $k = n + 1$ to $2n - 1$ **do**
  i = deletemin(H), j = deletemin(H) (1)
  Create a node numbered k with child nodes i, j.
  $f[k] = f[i] + f[j]$ (2)
  insert$(H, k, f(k))$

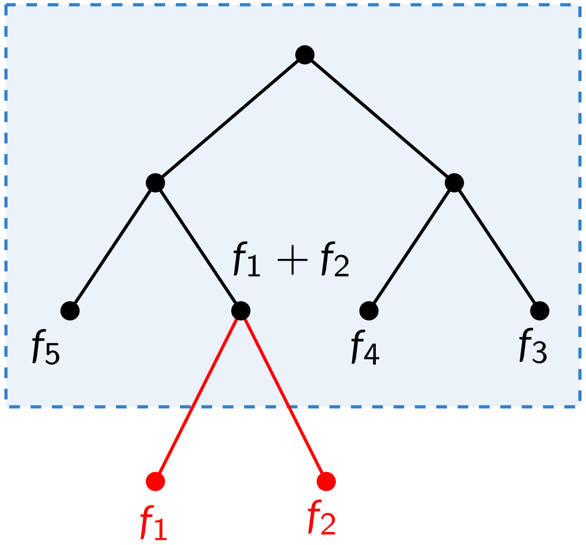


(1): two symbols with the smallest frequencies must be at the **bottom** of the optimal tree.

(2): any tree with sibling-leaves f_1 and f_2 has cost $f_1 + f_2$ plus the cost for a tree with $n - 1$ leaves of frequences $(f_1 + f_2), f_3, f_4, \ldots, f_n$.

function Huffman(f)

makequeue(H)
for $i = 1$ to n **do**
  insert($H, i, f[i]$) (* number i with key $f[i]$ *)
for $k = n + 1$ to $2n - 1$ **do**
  i = deletemin(H), j = deletemin(H) (1)
  Create a node numbered k with child nodes i, j.
  $f[k] = f[i] + f[j]$ (2)
  insert($H, k, f(k)$)



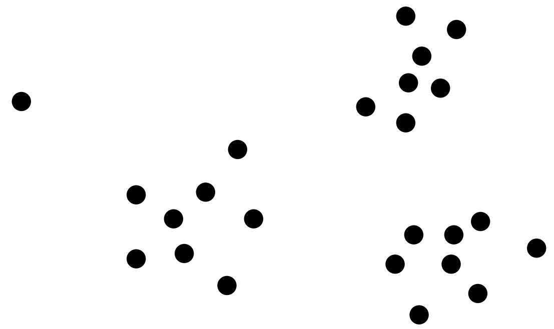
$O(n \log n)$ time if a binary heap is used.

(1): two symbols with the smallest frequencies must be at the **bottom** of the optimal tree.

(2): any tree with sibling-leaves f_1 and f_2 has cost $f_1 + f_2$ plus the cost for a tree with $n - 1$ leaves of frequences $(f_1 + f_2), f_3, f_4, \ldots, f_n$.

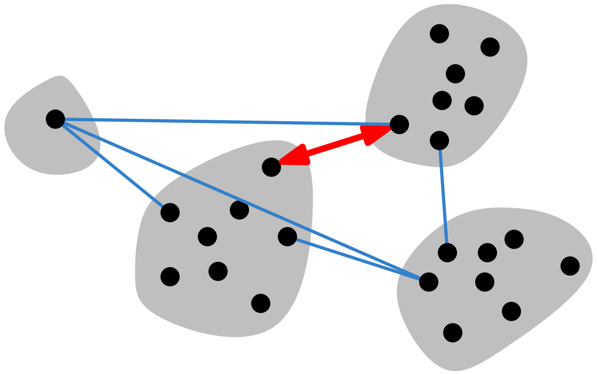
Clustering of Maximum Spacing

Clustering Given a set of n objects, classify them into *coherent* groups.



Clustering of Maximum Spacing

Clustering Given a set of n objects, classify them into *coherent* groups.



k**-clustering** divides objects into k nonempty groups.

Distance functions must reflect closeness of two objects.
Spacing is the min. distance between any pair of points in different clusters.

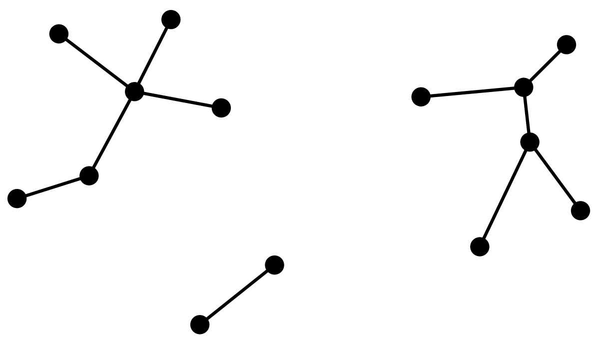
Clustering of max. spacing. Given k, find a k-clustering of maximum spacing.

Clustering of Maximum Spacing

Clustering of max. spacing. Given k, find a k-clustering of maximum spacing.

Single-link k-clustering algorithm.
- form a graph on the vertex set U, corresponding to n clusters.
- find the closest pair of objects s.t. each object is in a different cluster, and add an edge between them.
- repeat $n - k$ times until there are exactly k clusters.

Kruskal's algorithm, except we stop when there are k connected components.



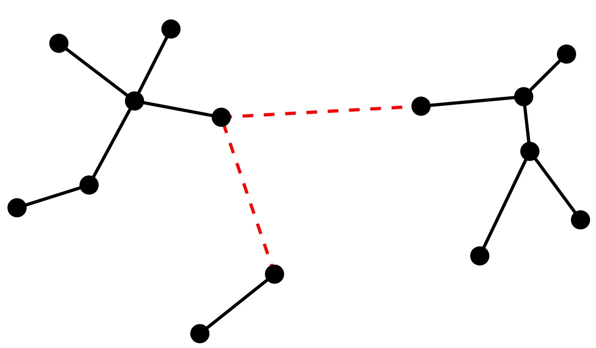
Clustering of Maximum Spacing

Clustering of max. spacing. Given k, find a k-clustering of maximum spacing.

Single-link k-clustering algorithm.
- form a graph on the vertex set U, corresponding to n clusters.
- find the closest pair of objects s.t. each object is in a different cluster, and add an edge between them.
- repeat $n - k$ times until there are exactly k clusters.

Kruskal's algorithm, except we stop when there are k connected components.

Equivalent to finding an MST and then deleting the $k - 1$ most expensive edges.

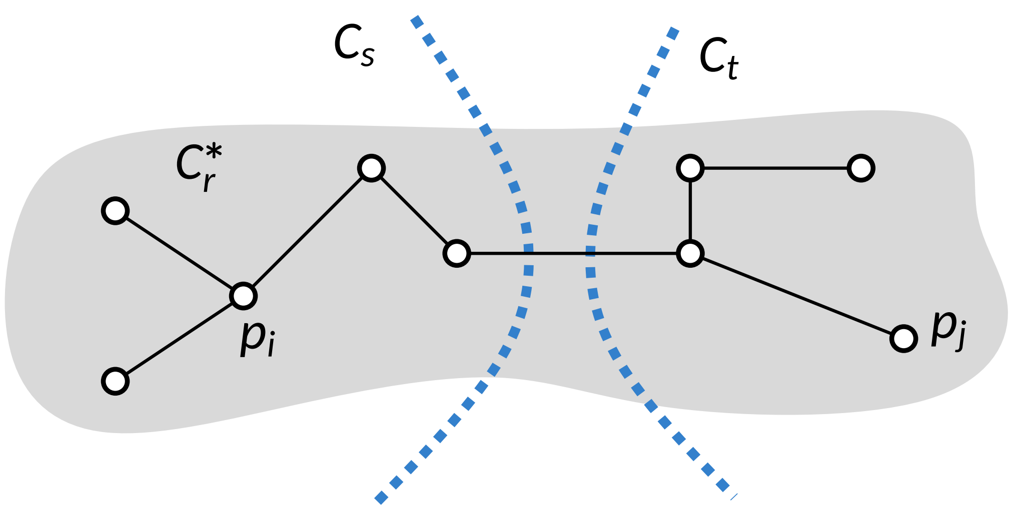


Clustering of Maximum Spacing

Theorem Let $\mathcal{C}^*$ denote the clustering $C_1^*, \ldots, C_k^*$ formed by deleting the k most expensive edges of a MST. $\mathcal{C}^*$ is a k-clustering of max. spacing.

Proof. The spacing of $\mathcal{C}^*$ is the length d^* of the $(k-1)$st most expensive edge. Let $\mathcal{C}$ denote some other clustering $C_1, \ldots, C_k$, and d denote its spacing.

Let p_i, p_j be two points in the same cluster in $\mathcal{C}^*$, say C_r^*, but in different clusters in $\mathcal{C}$, say $p_i \in C_s$ and $p_j \in C_t$.



Clustering of Maximum Spacing

Theorem Let $\mathcal{C}^*$ denote the clustering $C_1^*, \ldots, C_k^*$ formed by deleting the k most expensive edges of a MST. $\mathcal{C}^*$ is a k-clustering of max. spacing.

Proof. The spacing of $\mathcal{C}^*$ is the length d^* of the $(k-1)$st most expensive edge. Let $\mathcal{C}$ denote some other clustering $C_1, \ldots, C_k$, and d denote its spacing.

Let p_i, p_j be two points in the same cluster in $\mathcal{C}^*$, say C_r^*, but in different clusters in $\mathcal{C}$, say $p_i \in C_s$ and $p_j \in C_t$.

Consider the path from p_i to p_j in C_r^*. Some edge (p, q) on the path spans two different clusters, C_s and C_t in $\mathcal{C}$.

Since all edges on the path have length $\leqslant d^*$, we have $d \leqslant d^*$ ($\because$ p and q are in different clusters and therefore $d \leqslant |pq|$).