# CSE2310 Algorithm Design

## Lecture 3: Greedy Algorithms

Stefan Hugtenburg, Emir Demirović, and Mathijs de Weerdt



Algorithmics group — EEMCS — TU Delft

2023–2024 Q2

# You are here

## The course so far

- Introduction
- Greedy algorithms and proofs: scheduling

## Today's content

- Revisited problem: Minimum Spanning Trees
- New problems: Clustering

## The future

- Huffman's Optimal Encoding, Q&A
- Divide & Conquer algorithms
- Dynamic programming
- Network Flow

# Something new, something different

## 5 minutes – Let's see how this goes

### The Greedy Glossary

You have 5 minutes to write down/draw/paint/whatever the things you took away from the first week.

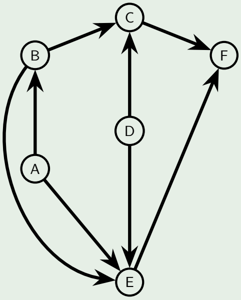
# Something new, something different

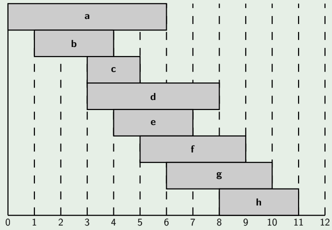
5 minutes – Let's see how this goes

## The Greedy Glossary

You have 5 minutes to write down/draw/paint/whatever the things you took away from the first week.

## Triggering memories

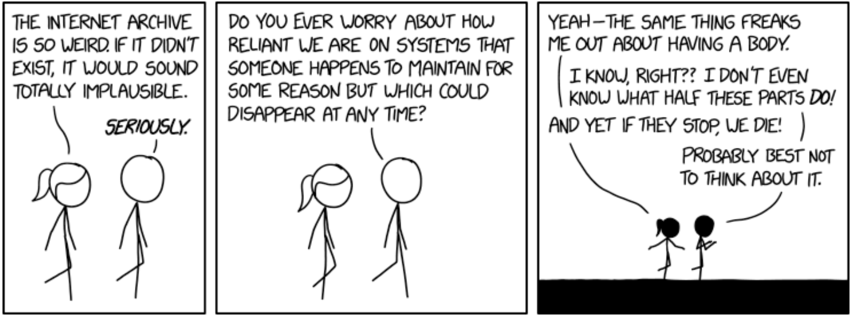



- Greedy algorithms: use smartly chosen order
- Greedy stays ahead: induction proof *and* by contradiction
- Exchange argument: contradiction proof about optimal solution most similar to greedy

# Back to the internet

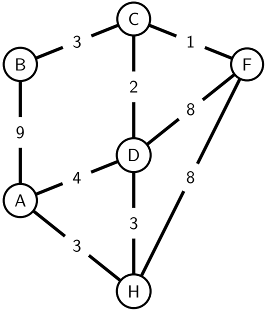
XKCD Internet Archive: `https://xkcd.com/2102/`



# Connecting machines

**Problem: Creating the internet**

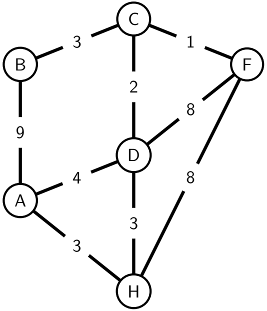
Given $n$ machines and $m$ possible connections between them, all with a cost $c(i)$ for every connection $i$.
What is the cheapest way to connect the machines such that a message can be sent from any machine to any other (i.e. the graph is strongly connected)?



# Connecting machines

**Problem: Creating the internet**

Given $n$ machines and $m$ possible connections between them, all with a cost $c(i)$ for every connection $i$.
What is the cheapest way to connect the machines such that a message can be sent from any machine to any other (i.e. the graph is strongly connected)?




**Question: Minimum cost?**

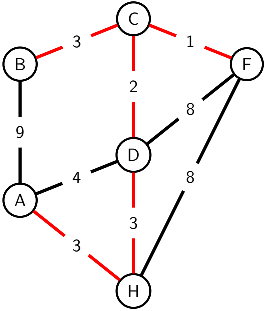
What is the minimum cost here?

A. 9
B. 12
C. 13
D. 20

# Connecting machines

**Problem: Creating the internet**

Given $n$ machines and $m$ possible connections between them, all with a cost $c(i)$ for every connection $i$.
What is the cheapest way to connect the machines such that a message can be sent from any machine to any other (i.e. the graph is strongly connected)?




**Question: Minimum cost?**

What is the minimum cost here?

- A. 9
- B. 12
- C. 13
- D. 20

# Minimum Spanning Tree (MST)

## Spanning Tree

A spanning tree $T$ of a graph $G = (V, E)$ is a subset of edges $T \subseteq E$, such that the graph $G' = (V, T)$ is strongly connected.

# Minimum Spanning Tree (MST)

## Spanning Tree

A spanning tree $T$ of a graph $G = (V, E)$ is a subset of edges $T \subseteq E$, such that the graph $G' = (V, T)$ is strongly connected.

## Minimum Spanning Tree

A Minimum Spanning Tree (MST) of a graph $G = (V, E)$ is spanning tree $T$, such that the cost $c(T) = \sum_{e \in T} w(e)$ is minimal.

That is, there is no spanning tree $T'$, such that $c(T') < c(T)$.

# How do we find this? Greedily?

**Question: How do we do it?**

How do we find a minimum spanning tree efficiently?

- A. For each vertex add cheapest edge, then join subtrees by adding cheapest edge.
- B. Add the cheapest edge to T that has exactly one endpoint in T.
- C. Add edges to T in ascending order of cost unless doing so would create a cycle.
- D. Start with all edges from G in T. Delete edges in descending order of cost unless doing so would disconnect T.
- E. All of the above.
- F. None of the above.
- G. I don't know.

# So many options

**Prim(Jarník)**

Like Dijkstra! Repeatedly pick the smallest edge out of our cloud fringe.

**Kruskal**

Repeatedly pick the smallest edge, add it if it doesn't make a cycle.

**Reverse Delete**

Remove the most expensive edge unless it disconnects the graph.

**Borůvska**

For each vertex add the cheapest edge, then merge subtrees using cheapest edges.

TU Delft

# So many options

**Prim(Jarník)**

Like Dijkstra! Repeatedly pick the smallest edge out of our cloud fringe.

**Kruskal**

Repeatedly pick the smallest edge, add it if it doesn't make a cycle.

**Reverse Delete**

Remove the most expensive edge unless it disconnects the graph.

**Borůvska**

For each vertex add the cheapest edge, then merge subtrees using cheapest edges.

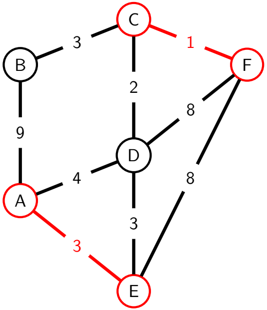
**Question: Prove it to me!**

But how can we be sure these are all always correct?

# A nice property (cut!)


Image From: *WikiHow*

# A first look

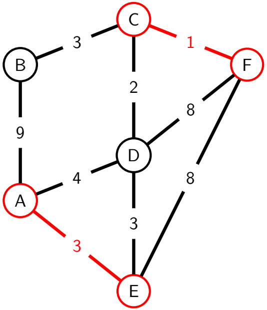



**Question: So...**

Let an MST-in-construction connect already $S = \{A, C, E, F\}$. What edge **must** be added?

A. $(A, B)$
B. $(C, D)$
C. $(D, F)$
D. $(E, F)$
E. I don't know.

# A first look



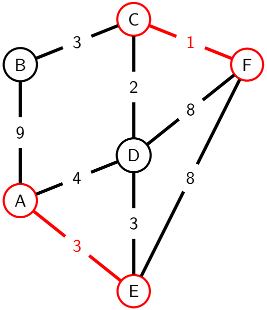

**Question: So...**

Let an MST-in-construction connect already $S = \{A, C, E, F\}$. What edge **must** be added?

**Answer: So old school!**

The edge $(C, D)$ must be in the MST to connect a node $(D)$ cheapest.

# A first look




**Question: So...**

Let an MST-in-construction connect already $S = \{A, C, E, F\}$. What edge **must** be added?

**Answer: So old school!**

The edge $(C, D)$ must be in the MST to connect a node $(D)$ cheapest.

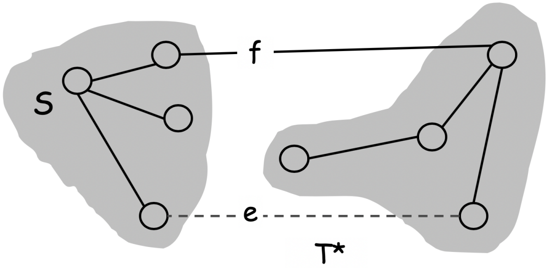
**The Cut Property**

Let $S$ be any subset of nodes $V$, and let $e$ be the min cost edge with exactly one endpoint in $S$ (i.e., the *cutset*). Then there is an MST $T$ that contains $e$.

# A proof

## The Cut Property

Let $S$ be any subset of $V$, and let $e$ be the min cost edge with exactly one endpoint in $S$ (i.e., the *cutset*). Then there is an MST $T$ that contains $e$.



# A proof

## The Cut Property

Let $S$ be any subset of $V$, and let $e$ be the min cost edge with exactly one endpoint in $S$ (i.e., the *cutset*). Then there is an MST $T$ that contains $e$.

## Proof.

Let $T$ be an MST. If $e \in T$ we're done. If $e \notin T$:
Adding $e$ to $T$ must create some cycle.
Therefore there is another edge $f (\neq e)$ that connect $S$ to $V - S$.

TU Delft

# A proof

**The Cut Property**

Let $S$ be any subset of $V$, and let $e$ be the min cost edge with exactly one endpoint in $S$ (i.e., the *cutset*). Then there is an MST $T$ that contains $e$.

**Proof.**

Let $T$ be an MST. If $e \in T$ we're done. If $e \notin T$:
Adding $e$ to $T$ must create some cycle.
Therefore there is another edge $f (\neq e)$ that connect $S$ to $V - S$.
Define a tree $T'$ to be $T$ with $f$ removed and $e$ added.
Since $w(e) \leq w(f)$, $T'$ has cost that is no larger and is also a spanning tree.
Thus we have now an MST that contains $e$. $\square$

# Dijkstra and Prim

**Two people discovered this**

Both Dijkstra and Prim independently discovered the next algorithm.

Image from *Wikimedia*

Image from *ITHistory*

# Jarník

**Thieves! They stole it from usss!**

Actually, mathematician Vojtěch Jarník invented it about 30 years before.

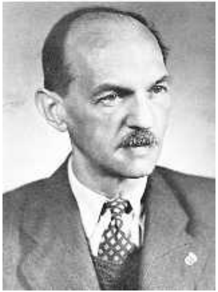


Image from: *web.math.muni.cz*

# Dijkstra already has his

**Prim-Jarník algorithm**

Similar to Dijkstra's algorithm for finding shortest paths: we grow a cloud ($S$), by repeatedly adding the smallest edge out of the cloud to the collection. This results in an MST! (Based on the Cut-Property)

# Dijkstra already has his

### Prim-Jarník algorithm

Similar to Dijkstra's algorithm for finding shortest paths: we grow a cloud ($S$), by repeatedly adding the smallest edge out of the cloud to the collection. This results in an MST! (Based on the Cut-Property)

### No love for Jarník

Unfortunately for Jarník, it is often just called Prim's algorithm.

TU Delft

# The pseudo code

```
function PRIMJARNIK(G)
    d[v] ← ∞ for all v ∈ V.
    s ∈ V is some vertex (arbitrarily chosen)
    d[s] ← 0.
    c ← 0
    while V is not empty do                     ▷ Q. How to repeatedly get the minimum here efficiently?
        m ← arg min_{v∈V} d[v]                  ▷ Take node from V with minimal distance from explored (Cut property!)
        c ← c + d[m]                            ▷ And add edge to tree
        Remove m from V
        if d[m] = ∞ then
            return ∞                            ▷ We cannot make an MST!
        for every e ← (m, u) ∈ E do             ▷ Check all neighbours
            if w(e) < d[u] then
                d[u] ← w(e)                     ▷ Update the shortest distance
    return c
```

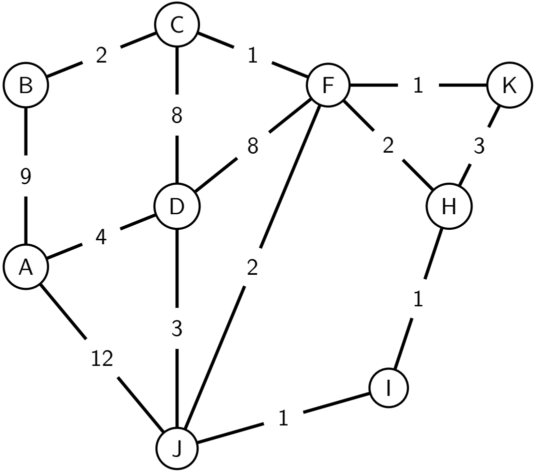
# The pseudo code

**function** ~~PRIM~~DIJKSTRAJARNIK$(G)$
    $d[v] \leftarrow \infty$ for all $v \in V$.
    $s \in V$ is some vertex (arbitrarily chosen)
    $d[s] \leftarrow 0$.

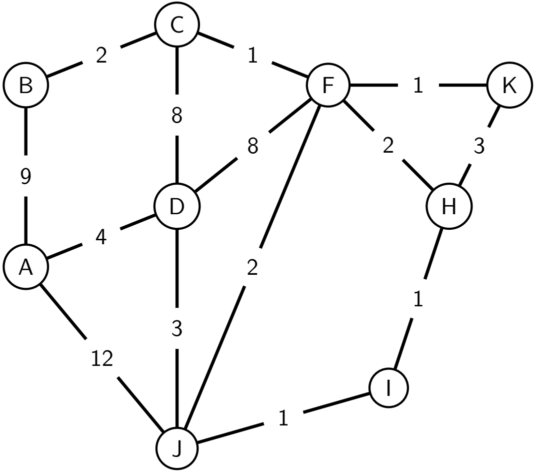
    **while** V is not empty **do** ▷ Q. How to repeatedly get the minimum here efficiently?
        $m \leftarrow \arg\min_{v \in V} d[v]$ ▷ Take node from $V$ with minimal distance from $s$

        Remove $m$ from $V$
        **if** $d[m] = \infty$ **then**
            **return** $\infty$ ▷ We cannot find a path!
        **for** every $e \leftarrow (m, u) \in E$ **do** ▷ Check all neighbours
            **if** $d[m] + w(e) < d[u]$ **then**
                $d[u] \leftarrow d[m] + w(e)$ ▷ Update the shortest distance
    **return** $d$

# Let's apply PrimJarnik!



# Let's apply PrimJarnik!




**Question: So how long is it?**

What is the cost of the minimum spanning tree of this graph?

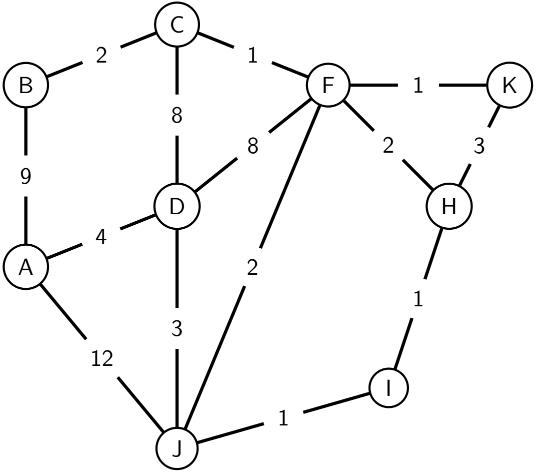
A. 10

B. 12

C. 15

D. 18

# Let's apply PrimJarnik!




**Question: So how long is it?**

What is the cost of the minimum spanning tree of this graph?

A. 10
B. 12
C. 15
D. 18

# Runtime?

**Just like Wybe**

The runtime is the same as Dijkstra's, so

- $\Theta((|V| + |E|) \log |V|)$ when we use a priority queue, which is
- $\Theta(|E| \log |V|)$ for connected graphs.

# Cycle Property


Image By: *Vector Open Stock*

# Cycle Property

## The Cycle Property

Let $C$ be any cycle in $G$, and let $e$ be the max cost edge in $C$. Then $e$ is not in any MST $T$.[a]

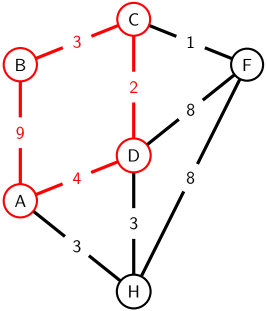
[a] We assume unique edge costs – which is easy to guarantee by distorting the input.

# Cycle Property

## The Cycle Property

Let $C$ be any cycle in $G$, and let $e$ be the max cost edge in $C$. Then $e$ is not in any MST $T$.[a]

[a] We assume unique edge costs – which is easy to guarantee by distorting the input.




**Question: So...**

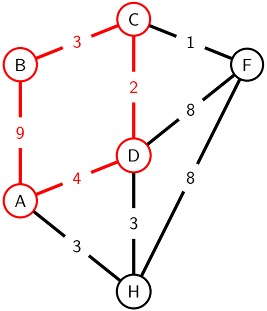
Let $C = \{A, B, D, C\}$. So what edge cannot be in an MST?

# Cycle Property

**The Cycle Property**

Let $C$ be any cycle in $G$, and let $e$ be the max cost edge in $C$. Then $e$ is not in any MST $T$.[a]

[a] We assume unique edge costs – which is easy to guarantee by distorting the input.




**Question: So...**

Let $C = \{A, B, D, C\}$. So what edge cannot be in an MST?

**Answer: Breaking up the alphabet**

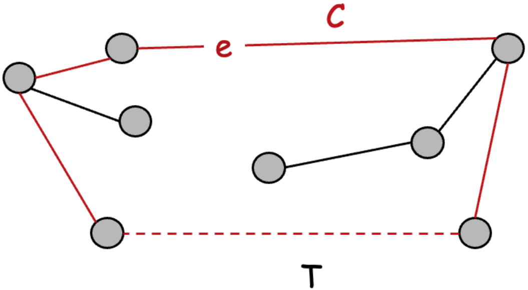
The edge $(A, B)$ must not be in the MST as a result.

# A proeof

**The Cycle Property**

Let $C$ be any cycle in $G$, and let $e$ be the max cost edge in $C$. Then $e$ is not in any MST $T$.[a]

[a] We assume unique edge costs – which is easy to guarantee by distorting the input.

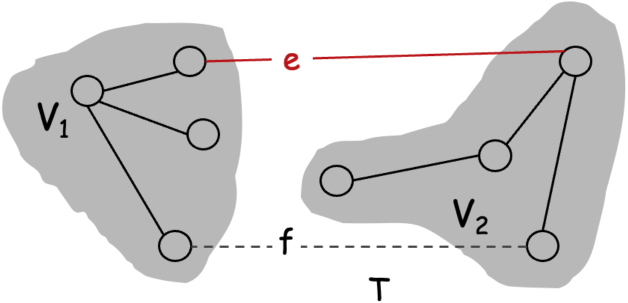


# A proeof

## The Cycle Property

Let $C$ be any cycle in $G$, and let $e$ be the max cost edge in $C$. Then $e$ is not in any MST $T$.[a]

[a] We assume unique edge costs – which is easy to guarantee by distorting the input.



# A proeof

## The Cycle Property

Let $C$ be any cycle in $G$, and let $e$ be the max cost edge in $C$. Then $e$ is not in any MST $T$.[a]

[a] We assume unique edge costs – which is easy to guarantee by distorting the input.

## Proof.

For the sake of contradiction, let $T$ be an MST of the graph such that $e = \{u, v\} \in T$. Remove $e$ from the tree. This breaks it into two parts $V_1$ and $V_2$ that are not connected, with $u \in V_1$ and $v \in V_2$.

# A proeof

## The Cycle Property

Let $C$ be any cycle in $G$, and let $e$ be the max cost edge in $C$. Then $e$ is not in any MST $T$.[a]

[a] We assume unique edge costs – which is easy to guarantee by distorting the input.

## Proof.

For the sake of contradiction, let $T$ be an MST of the graph such that $e = \{u, v\} \in T$.
Remove $e$ from the tree. This breaks it into two parts $V_1$ and $V_2$ that are not connected, with $u \in V_1$ and $v \in V_2$.
But since $C$ was a cycle, there must an edge $f$ in the graph to connect $V_1$ and $V_2$.

# A proeof

## The Cycle Property

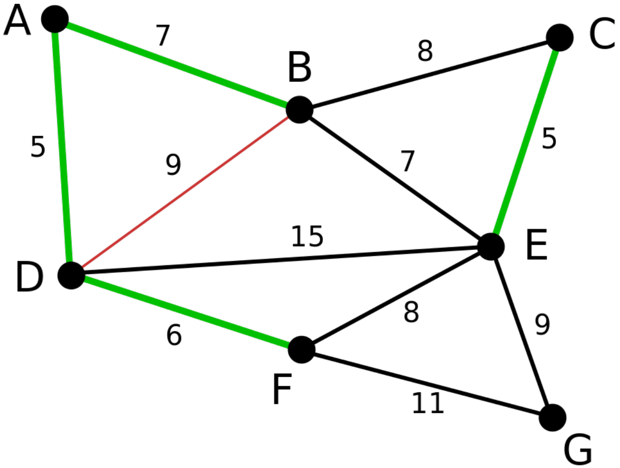
Let $C$ be any cycle in $G$, and let $e$ be the max cost edge in $C$. Then $e$ is not in any MST $T$.[a]

[a] We assume unique edge costs – which is easy to guarantee by distorting the input.

## Proof.

For the sake of contradiction, let $T$ be an MST of the graph such that $e = \{u, v\} \in T$.
Remove $e$ from the tree. This breaks it into two parts $V_1$ and $V_2$ that are not connected, with $u \in V_1$ and $v \in V_2$.
But since $C$ was a cycle, there must an edge $f$ in the graph to connect $V_1$ and $V_2$.
And $w(f) < w(e)$ (by choice of $e$), so add $f$ to T. Now for $T' = (T - \{e\}) \cup \{f\}$: $c(T') < c(T)$, so $T$ can't be an MST. Contradiction. So $e$ can't be in any MST. $\square$

# Kruskal



# The alternative to Prim

**Kruskal's algorithm**

First sort the edges by weight. Now repeatedly add the smallest weight edge (according to the *cut* property), unless it creates a cycle then we discard it (in line with the *cycle* property).

# The pseudocode

**function** $\textsc{Kruskal}(G)$
    sort $E$ ascendingly; label them $e_1$ through $e_m$
    $T \leftarrow \emptyset$
    $i \leftarrow 0$; $k \leftarrow 0$
    **while** $k < |V| - 1$ **do**
        **if** $T \cup \{e_i\}$ does not have a cycle **then**
            $k \leftarrow k + 1$
            $T \leftarrow T \cup \{e_i\}$
    **return** $T$

# The pseudocode

**function** $\textsc{Kruskal}(G)$
  sort $E$ ascendingly; label them $e_1$ through $e_m$
  $T \leftarrow \emptyset$
  $i \leftarrow 0; k \leftarrow 0$
  **while** $k < |V| - 1$ **do**
    **if** $T \cup \{e_i\}$ does not have a cycle **then**
      $k \leftarrow k + 1$
      $T \leftarrow T \cup \{e_i\}$
  **return** $T$

**Question: Cycle detection**

How do we efficiently determine if adding an edge creates a cycle?

# The pseudocode

**function** $\textsc{Kruskal}(G)$
  sort $E$ ascendingly; label them $e_1$ through $e_m$
  $T \leftarrow \emptyset$
  $i \leftarrow 0$; $k \leftarrow 0$
  **while** $k < |V| - 1$ **do**
    **if** $T \cup \{e_i\}$ does not have a cycle **then**
      $k \leftarrow k + 1$
      $T \leftarrow T \cup \{e_i\}$
  **return** $T$

**Question: Cycle detection**

How do we efficiently determine if adding an edge creates a cycle?

**Just an idea?**

Let's try a simple id-based idea.

# To each their own

## Id-based cycle detection

Every node starts with their own id as their '*cycle id*'.
If we want to put two nodes together (i.e., add an edge between them), this is only allowed if their cycle ids are different.
We then update both ids to be the maximum of the two ids.

# To each their own

### Id-based cycle detection

Every node starts with their own id as their '*cycle id*'.
If we want to put two nodes together (i.e., add an edge between them), this is only allowed if their cycle ids are different.
We then update both ids to be the maximum of the two ids.

### Just like Ralph

Break this strategy by creating a graph for which this would not be efficient.

# To each their own

**Id-based cycle detection**

Every node starts with their own id as their '*cycle id*'.
If we want to put two nodes together (i.e., add an edge between them), this is only allowed if their cycle ids are different.
We then update both ids to be the maximum of the two ids.

**Just like Ralph**

Break this strategy by creating a graph for which this would not be efficient.

**Even the fix doesn't fix it**

If we update the whole tree every time, then this requires $O(|V|)$ time to add every edge. This would make it much worse than PrimJarník.

# Union-Find

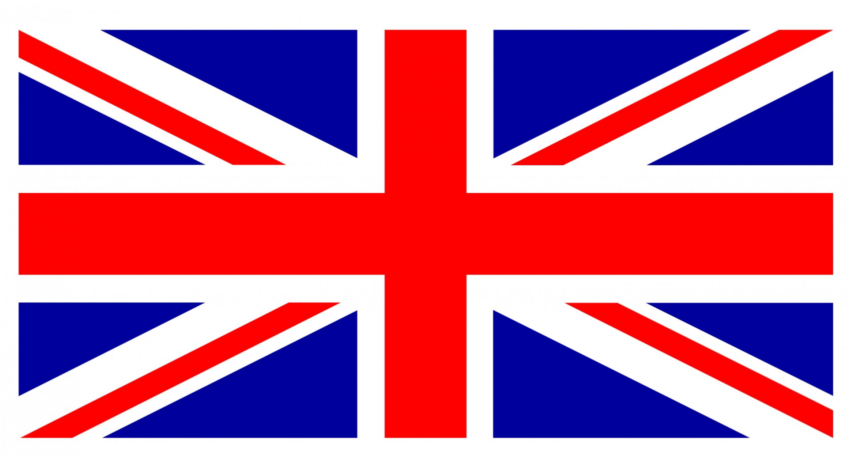


Image By: *Karen Arnold*

# A new data structure

**Union-Find**

The *Union-Find* (also known as *Disjoint-Set*) data structure can track elements partitioned into disjoint subsets. It provides near-constant time operations merge (*union*) and contains (*find*) operations.

# A new data structure

**Union-Find**

The *Union-Find* (also known as *Disjoint-Set*) data structure can track elements partitioned into disjoint subsets. It provides near-constant time operations merge (*union*) and contains (*find*) operations.

**Just what we need!**

That's exactly what we need! Kruskal creates a bunch of sets (connected components), which we only connect if they don't contain duplicates (as that would indicate a cycle).

# How does it work?

**Three operations to cover us**

- `MakeSet(n)` which creates $n$ initial sets, all containing 1 element.
- `Find(x)` which returns the id of the set that $x$ is in.
- `Union(x,y)` which merges the sets that $x$ and $y$ are in.

# MakeSet and Find

**function** MakeSet($n$)
    set ← array of size $n$, where set[i] ← i ▷ Often called 'representative'
    rank ← array of size $n$, where rank[i] ← 0

# MakeSet and Find

**function** $\text{MAKESET}(n)$
  set $\leftarrow$ array of size $n$, where set[i] $\leftarrow$ i ▷ Often called 'representative'
  rank $\leftarrow$ array of size $n$, where rank[i] $\leftarrow$ 0

**function** $\text{FIND}(x)$
  **if** set[x] $\neq$ x **then**
    set[x] $\leftarrow$ FIND(set[x]) ▷ Apply what we call 'path compression'
  **return** set[x]

# Union

```
function UNION(x,y)
    xSet ← FIND(x)
    ySet ← FIND(y)
    if xSet = ySet then
        return False            ▷ Nothing to merge
    else if rank[xSet] < rank[ySet] then
        set[xSet] ← ySet
    else
        set[ySet] ← xSet
```

# Union

```
function UNION(x,y)
    xSet ← FIND(x)
    ySet ← FIND(y)
    if xSet = ySet then
        return False            ▷ Nothing to merge
    else if rank[xSet] < rank[ySet] then
        set[xSet] ← ySet
    else
        set[ySet] ← xSet
        if rank[xSet] = rank[ySet] then
            rank[xSet] ← rank[xSet] + 1
    return True
```

# Union

**function** UNION(x,y)
    xSet $\leftarrow$ FIND(x)
    ySet $\leftarrow$ FIND(y)
    **if** xSet $=$ ySet **then**
        **return** False    $\triangleright$ Nothing to merge
    **else if** rank[xSet] $<$ rank[ySet] **then**
        set[xSet] $\leftarrow$ ySet
    **else**
        set[ySet] $\leftarrow$ xSet
        **if** rank[xSet] $=$ rank[ySet] **then**
            rank[xSet] $\leftarrow$ rank[xSet] $+ 1$
    **return** True

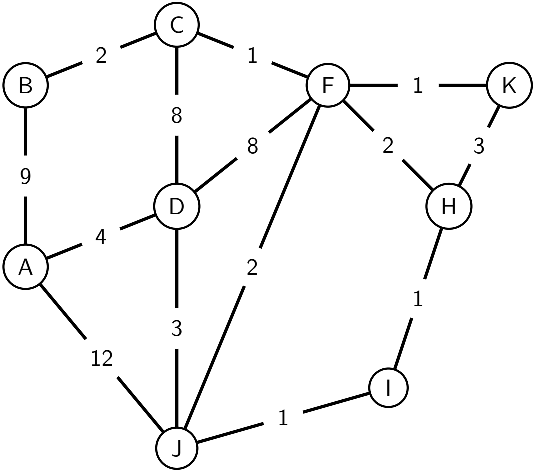
**Question: Union-Find practice**

Given 10 items, apply the following operations.

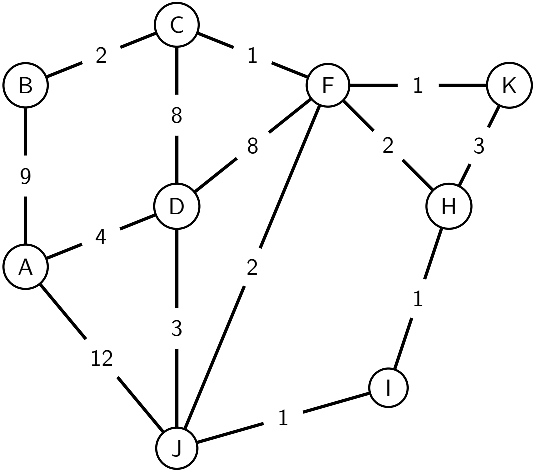
1. `Union(1,8)`
2. `Union(2,4)`
3. `Union(4,7)`
4. `Union(3,5)`
5. `Union(9,5)`
6. `Union(5,2)`

TUDelft

# Let's apply Kruskal!



# Let's apply Kruskal!

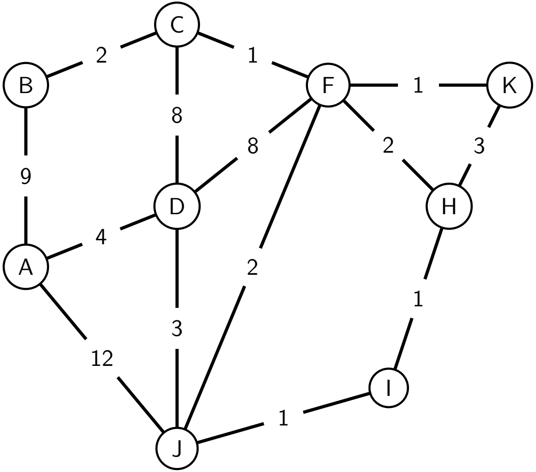



**Question: So how long is it?**

How many edges are considered, before Kruskal is done?

- A. 9
- B. 10 (inc. cost 4)
- C. 11
- D. 12

# Let's apply Kruskal!




**Question: So how long is it?**

How many edges are considered, before Kruskal is done?

- A. 9
- B. 10 (inc. cost 4)
- C. 11
- D. 12

# Runtime?

## Runtime

Union-Find allows $m$ union/find operations on $n$ sets in $O(m \log n)$ time.*
For Kruskal we have $|V|$ sets, on which we do at most $|E|$ union/find operations.
Furthermore sorting the edges is $\Theta(|E| \log |E|)$.
So the runtime is $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$ time.

* Actually, Union-Find is even faster; close to constant (inverse Ackermann).

# To summarize

**PrimJarník**

Like Dijkstra! Repeatedly pick the smallest edge out of our cloud.

**Kruskal**

Repeatedly pick the smallest edge, add it if it doesn't make a cycle.

**Runtimes**

Both offer pretty good runtimes of $O(|E| \log |V|)$, but each excel at their own bit.
Kruskal is excellent when edges are already sorted for instance.
PrimJarník can be improved (with a more advanced heap-based PQ) to run in $\Theta(|E| + |V| \log |V|)$ time, which makes it better when $|E| >> |V|$.

# Clusters!

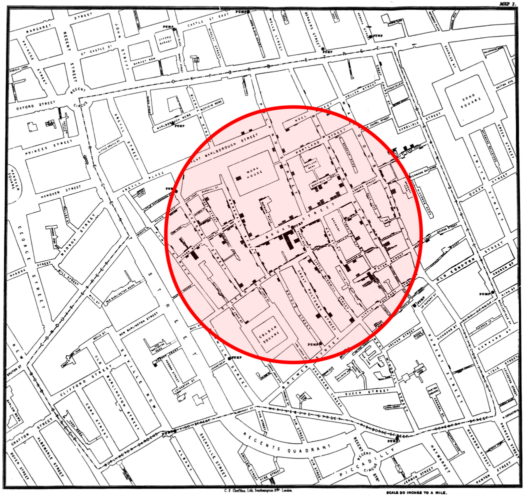



Image from: *Wikipedia*

# k-Clustering of Maximum spacing

## k-Clustering

Divide objects into $k$ non-empty groups.

## Some notion of distance

- Identity of indiscernibles: $d(p_i, p_j) = 0$ iff $p_i = p_j$
- Non-negativity: $d(p_i, p_j) \geq 0$
- Symmetry: $d(p_i, p_j) = d(p_j, p_i)$

# k-Clustering of Maximum spacing

**k-Clustering**

Divide objects into $k$ non-empty groups.

**Some notion of distance**

- Identity of indiscernibles: $d(p_i, p_j) = 0$ iff $p_i = p_j$
- Non-negativity: $d(p_i, p_j) \geq 0$
- Symmetry: $d(p_i, p_j) = d(p_j, p_i)$

**Spacing between clusters**

*Spacing* is the minimum distance between any pair of points in different clusters.

**Problem: Clustering of maximum spacing**

Given $n$ objects and an integer $k < n$, find a $k$-clustering of maximum spacing.

# k-Clustering

**Question: How do we do this?**

How can we efficiently divide objects into $k$ non-empty groups such that the minimal distance between groups (spacing) is maximized?

- A. $k-1$ times: consider all possible partitions of a cluster, maximize spacing.
- B. Build a minimal spanning tree and delete the $k-1$ most expensive edges.
- C. Grow minimal spanning tree fragments (with Kruskal) and stop after $n-k$ edges.
- D. Merge objects that are closest and replace them with one in between. Stop when exactly $k$ objects are left.
- E. I don't know.

# Not ideal

**A**

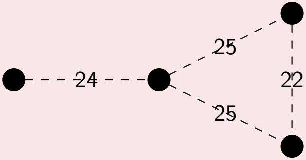
$k-1$ times consider all possible partitions of a cluster. Wait.... That's $O(2^n)$ work!

# Not ideal

**A**

$k-1$ times consider all possible partitions of a cluster. Wait.... That's $O(2^n)$ work!

**D**

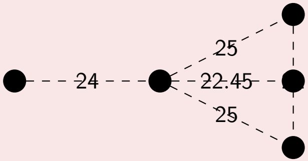


# Not ideal

## A

$k - 1$ times consider all possible partitions of a cluster. Wait.... That's $O(2^n)$ work!

## D



# C I told you so!

## Single-link k-clustering

- Form graph (without edges) on vertex set
  - This corresponds to $n$ initial clusters.
- Find the closest pair of objects from different clusters.
- Add edge between them.
- Repeat $n - k$ times until there are exactly $k$ clusters left.

# C I told you so!

## Single-link k-clustering

- Form graph (without edges) on vertex set
  - This corresponds to $n$ initial clusters.
- Find the closest pair of objects from different clusters.
- Add edge between them.
- Repeat $n - k$ times until there are exactly $k$ clusters left.

## Hang on a second...

That's exactly Kruskal's algorithm! Provided we stop it early.

# C I told you so!

## Single-link k-clustering

- Form graph (without edges) on vertex set
  - This corresponds to $n$ initial clusters.
- Find the closest pair of objects from different clusters.
- Add edge between them.
- Repeat $n - k$ times until there are exactly $k$ clusters left.

## Hang on a second...

That's exactly Kruskal's algorithm! Provided we stop it early.

## Just like answer B

Equivalent to finding an MST and deleting the $k - 1$ most expensive edges (answer B)

# Greedy clustering algorithm

Algorithm idea: Delete the $k - 1$ most expensive edges from the MST.

**Question: Space!**

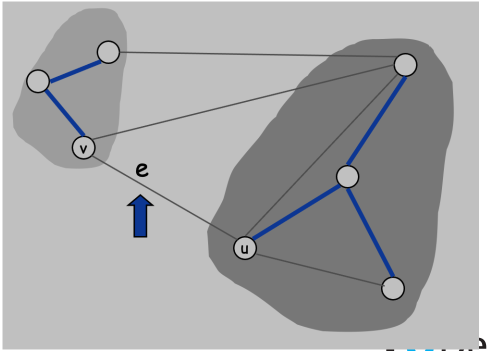
What is the spacing of the resulting clustering?

# Greedy clustering algorithm

Algorithm idea: Delete the $k-1$ most expensive edges from the MST.

**Question: Space!**

What is the spacing of the resulting clustering?

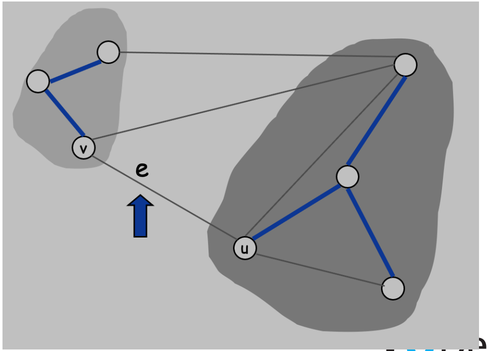


# Greedy clustering algorithm

Algorithm idea: Delete the $k-1$ most expensive edges from the MST.

**Question: Space!**

What is the spacing of the resulting clustering?




**Answer: Expensive stuff**

The cost/length of the $k-1$ most expensive edge in the MST.

There is no shorter edge between the clusters: the Cut property tells us the shortest edge in the cutset is in the MST.

# Greedy clustering algorithm: Analysis of optimality

**Theorem**

Let $C^*$ denote the clustering $C_1^*, \ldots, C_k^*$ formed by deleting the $k-1$ most expensive edges of a MST by Kruskal. $C^*$ is a clustering of *maximum* spacing.

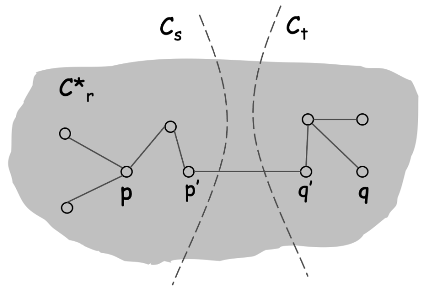
# Greedy clustering algorithm: Analysis of optimality

**Theorem**

Let $C^*$ denote the clustering $C_1^*, \ldots, C_k^*$ formed by deleting the $k-1$ most expensive edges of a MST by Kruskal. $C^*$ is a clustering of *maximum* spacing.

**Proof.**

Idea: show that every other clustering has a smaller (or equal) spacing than $C^*$ (defined as $d^*$).



# Greedy clustering algorithm: Analysis of optimality

**Theorem**

Let $C^*$ denote the clustering $C_1^*, \ldots, C_k^*$ formed by deleting the $k-1$ most expensive edges of a MST by Kruskal. $C^*$ is a clustering of *maximum* spacing.

**Proof.**

- The spacing of $C^*$ is the length $d^*$ of the $(k-1)$th most expensive edge.

# Greedy clustering algorithm: Analysis of optimality

## Theorem

Let $C^*$ denote the clustering $C_1^*, \ldots, C_k^*$ formed by deleting the $k-1$ most expensive edges of a MST by Kruskal. $C^*$ is a clustering of *maximum* spacing.

## Proof.

- The spacing of $C^*$ is the length $d^*$ of the $(k-1)$th most expensive edge.
- Let $C$ denote some other arbitrary clustering $C_1, \ldots, C_k$.

elft

# Greedy clustering algorithm: Analysis of optimality

**Theorem**

Let $C^*$ denote the clustering $C_1^*, \ldots, C_k^*$ formed by deleting the $k-1$ most expensive edges of a MST by Kruskal. $C^*$ is a clustering of *maximum* spacing.

**Proof.**

- The spacing of $C^*$ is the length $d^*$ of the $(k-1)$th most expensive edge.
- Let $C$ denote some other arbitrary clustering $C_1, \ldots, C_k$.
- Let $p, q$ be in the same cluster in $C^*$, say $C_r^*$, but in different clusters in $C$, say $C_s$ and $C_t$.

lft

# Greedy clustering algorithm: Analysis of optimality

**Theorem**

Let $C^*$ denote the clustering $C_1^*, \ldots, C_k^*$ formed by deleting the $k-1$ most expensive edges of a MST by Kruskal. $C^*$ is a clustering of *maximum* spacing.

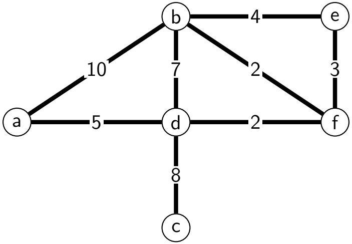
**Proof.**

- The spacing of $C^*$ is the length $d^*$ of the $(k-1)$th most expensive edge.
- Let $C$ denote some other arbitrary clustering $C_1, \ldots, C_k$.
- Let $p, q$ be in the same cluster in $C^*$, say $C_r^*$, but in different clusters in $C$, say $C_s$ and $C_t$.
- Some edge $(p', q')$ on the $p-q$ path in $C_r^*$ spans two different clusters in $C$.
- All edges on this path have length $\leq d^*$ since Kruskal chose them.
- So spacing of $C$ is $\leq d^*$ since $p'$ and $q'$ are in different clusters.
- Since $C$ is arbitrary, this holds for all $C$.

$\square$

# Old exam question: On clustering

5 minutes (+5 minutes)




The edge weights represent the distances between the vertices; if there is no edge, the distance is $\infty$. What is the spacing for a 3-clustering with maximum spacing?

A. 4

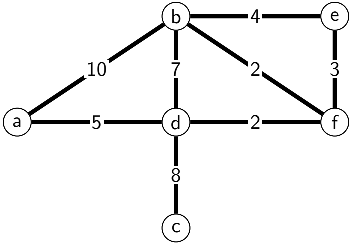
B. 5

C. 8

D. 23

# Old exam question: On clustering

5 minutes (+5 minutes)




The edge weights represent the distances between the vertices; if there is no edge, the distance is $\infty$. What is the spacing for a 3-clustering with maximum spacing?

**Answer: One more than 4, one less than 6**

The maximum-spacing clustering is to make the groups $\{a\}, \{c\}$, and $\{b, e, d, f\}$. The spacing is then equal to 5 as this is the smallest distance between any two points of different sets (between $\{a\}$ and $\{b, e, d, f\}$).

# You are here

## The course so far

- Introduction
- Greedy algorithms and proofs: scheduling

## Today's content

- Revisited problem: Minimum Spanning Trees
- New problems: Clustering

## The future

- Huffman's Optimal Encoding, Q&A
- Divide & Conquer algorithms
- Dynamic programming
- Network Flow

# What is still unclear?

**Question: After every lecture...**

Give us some homework and tell us:
What is still unclear after attending today's lecture?

# Homework for this week

- Before next lecture:
    - Study Chapter 4:
        - MST (Ch. 4.5-4.6)
        - Clustering (Ch. 4.7)
    - Do all skills of module Greedy until "Lecture 4" (for your chosen path)
    - **Think about your questions for the Q&A (this Friday)!**

# Homework for this week

- Before next lecture:
  - Study Chapter 4:
    - MST (Ch. 4.5-4.6)
    - Clustering (Ch. 4.7)
  - Do all skills of module Greedy until "Lecture 4" (for your chosen path)
  - **Think about your questions for the Q&A (this Friday)!**
- Next TA check:
  - *Experiments!* (Mountain Climber): November 23
  - *Greedy Triathlon*: November 25
- Next peer review:
  - November 23 (tomorrow during the lab)

# CSE2310 Algorithm Design

## Lecture 3: Greedy Algorithms

Stefan Hugtenburg, Emir Demirović, and Mathijs de Weerdt



Algorithmics group — EEMCS — TU Delft

2023–2024 Q2