CS 341: Algorithms
Lec 09:Minimum Spanning Trees

Armin Jamshidpey   Collin Roberts

Based on lecture notes by Éric Schost

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2025

Spanning trees

Definition:

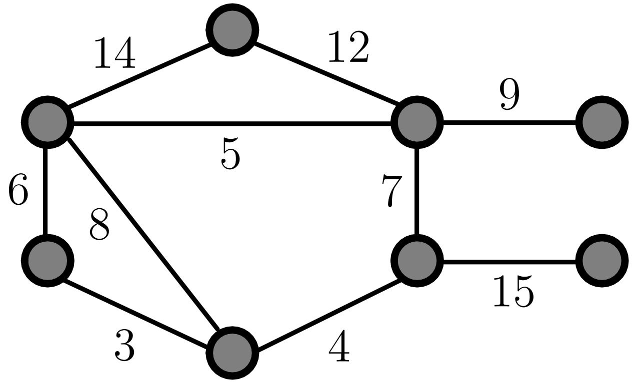
- $G = (V, E)$ is a connected graph
- a **spanning tree** in G is a tree of the form (V, A), with A a subset of E
- in other words: a tree with edges from E that covers all vertices
- examples: BFS tree, DFS tree

Now, suppose the edges have **weights** $w(e_i)$

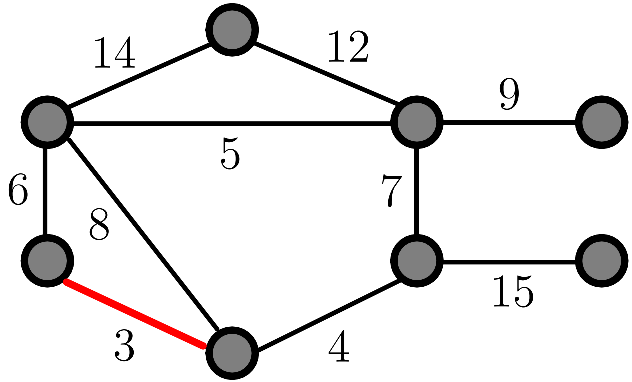
Goal:

- a spanning tree with **minimal weight**

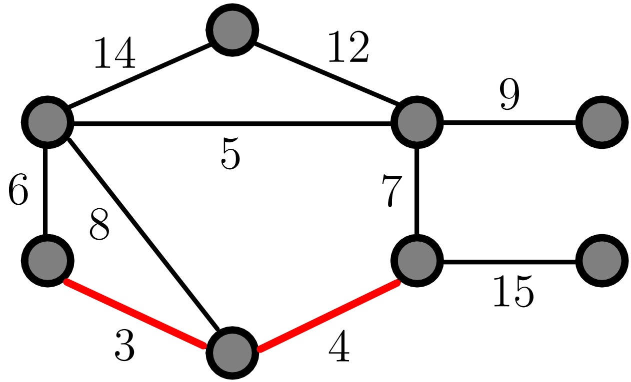
Example



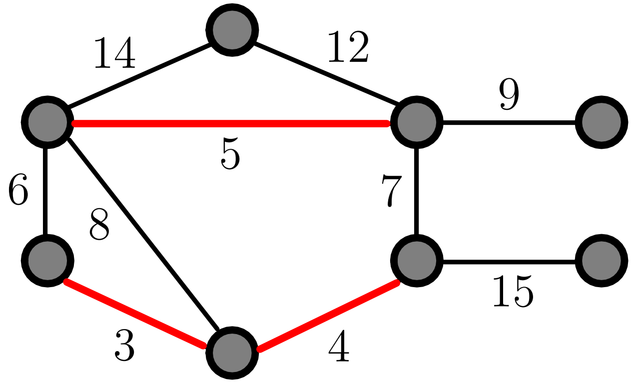
Example



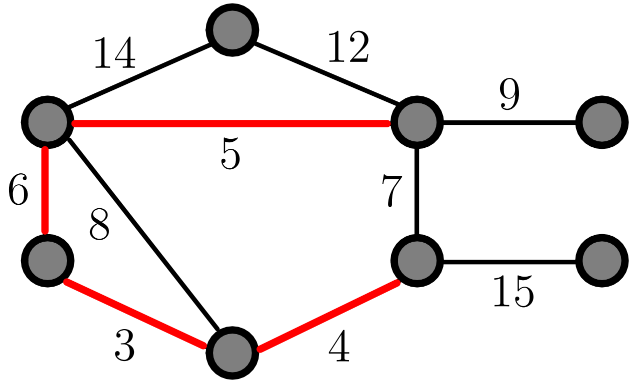
Example



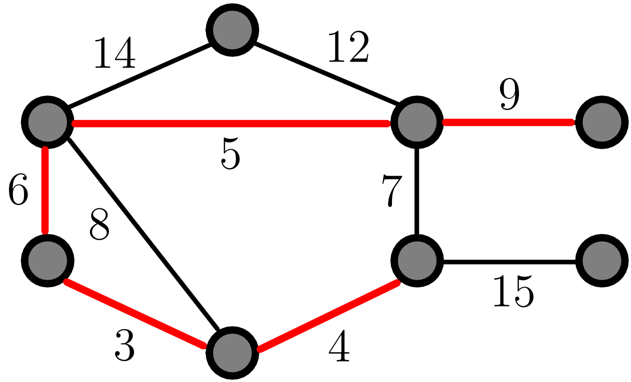
Example



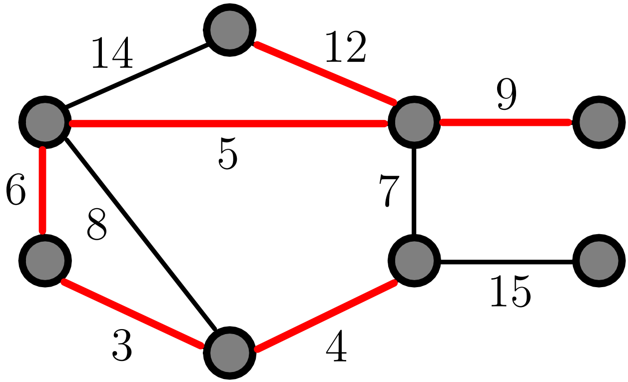
Example



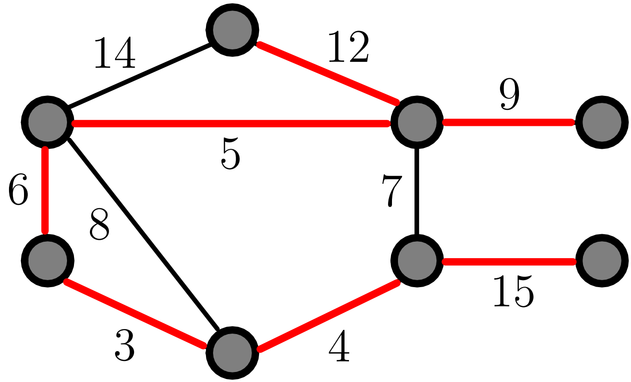
Example



Example



Example



Kruskal's algorithm

GreedyMST(G)

1. $A \leftarrow [\ ]$
2. sort edges by increasing weight
3. **for** $k = 1, \ldots, m$ **do**
4.   **if** e_k does not create a cycle in A **then**
5.   append e_k to A

Augmenting sets without cycles

Claim

Let G be a connected graph, and let A be a subset of the edges of G.

If (V, A) has no cycle and $|A| < n - 1$, then one can find an edge e not in A such that $A \cup \{e\}$ still has no cycle.

Augmenting sets without cycles

Claim

Let G be a connected graph, and let A be a subset of the edges of G.

If (V, A) has no cycle and $|A| < n - 1$, then one can find an edge e not in A such that $A \cup \{e\}$ still has no cycle.

Proof

- in any graph, #vertices $-$ #con. comp. $\leq$ #edges
- for (V, A), this gives $\boldsymbol{n - c < n - 1}$ so $\boldsymbol{c > 1}$
- take any edge on a path that connects two components.

Properties of the output

Claim

If the output is $A = [e_1, \ldots, e_r]$, then (V, A) is a spanning tree (and $r = n - 1$)

Proof:

- of course, (V, A) has no cycle.
- suppose (V, A) is not a spanning tree. Then, there exists an edge e not in A, such that $(V, A \cup \{e\})$ still has no cycle.
- **Case 1:** $\boldsymbol{w(e) < w(e_1)}$. Impossible, since e_1 is the element with the smallest weight.
- **Case 2:** $\boldsymbol{w(e_i) < w(e) < w(e_{i+1})}$. Impossible: at the moment we inserted e_{i+1}, we decided not to include e. This means that e created a loop with $e_1, \ldots, e_i$.
- **Case 3:** $\boldsymbol{w(e_r) < w(e)}$. Impossible: we would have included it in A, since there is no loop in $A \cup \{e\}$.

Exchanging edges

Claim

Let (V, A) and (V, T) be two spanning trees, and let e be an edge in T but not in A.

Then there exists an edge e' in A but not in T such that $(V, T + e' - e)$ is still a spanning tree. **Bonus:** e' is on the cycle that e creates in A.

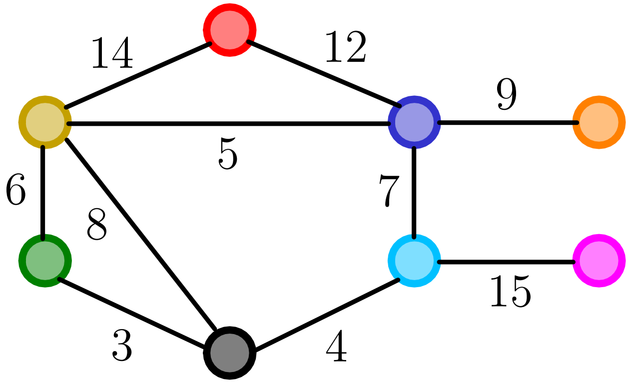
Proof:

- write $e = \{v, w\}$
- $(V, A + e)$ contains a cycle $\boldsymbol{c = v, w, \ldots, v}$
- removing e from T splits $(V, T - e)$ into two connected components T_1, T_2
- $\boldsymbol{c}$ starts in T_1, crosses over to T_2, so it contains another edge e' between T_2 and T_1
- e' is in A, but not in T
- $(V, T + e' - e)$ is a spanning tree

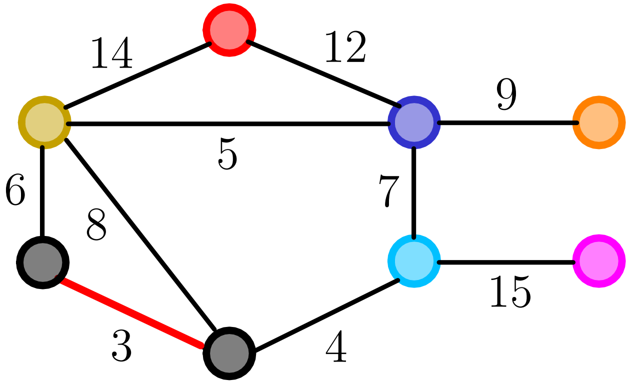
Correctness: exchange argument

- let A be the output of the algorithm
- let (V, T) be **any** spanning tree
- if $T \neq A$, let e be an edge in T but not in A
- so there is an edge e' in A but not in T such that $\boldsymbol{(V, T + e' - e)}$ is a spanning tree, **and** e' is on the cycle that e creates in A
- during the algorithm, we considered e but rejected it, because it created a cycle in A
- all other elements in this cycle have smaller (or equal) weight
- so $\boldsymbol{w(e') \leq w(e)}$
- so $T' = T + e' - e$ has weight $\leq w(T)$, and **one more common element** with A
- keep going

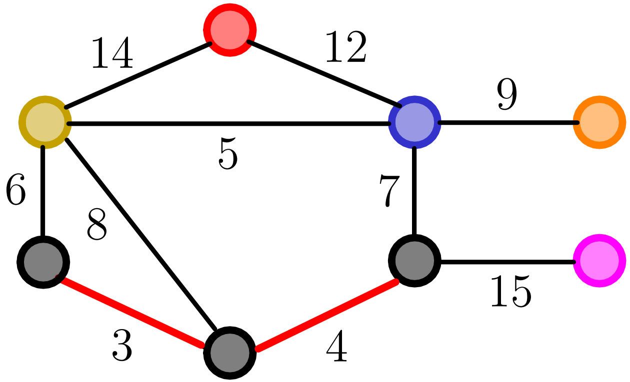
Merging connected sets of vertices



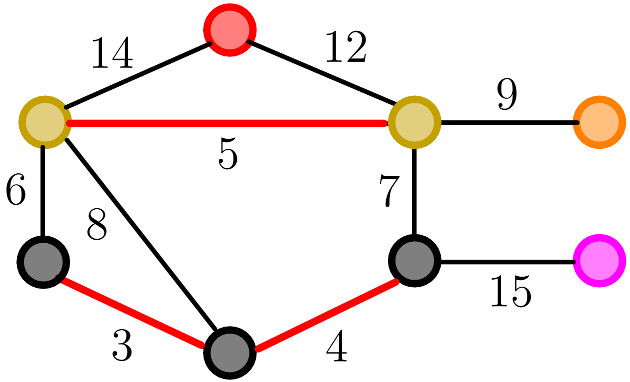
Merging connected sets of vertices



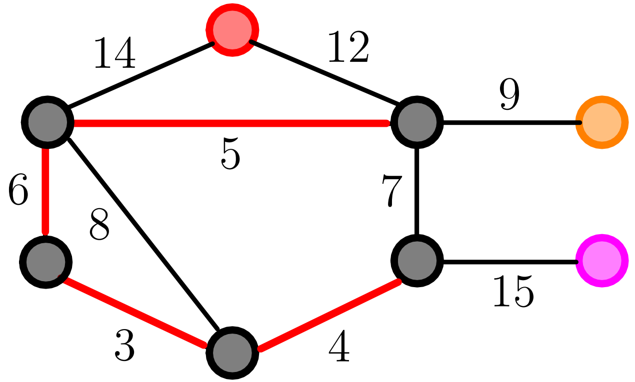
Merging connected sets of vertices



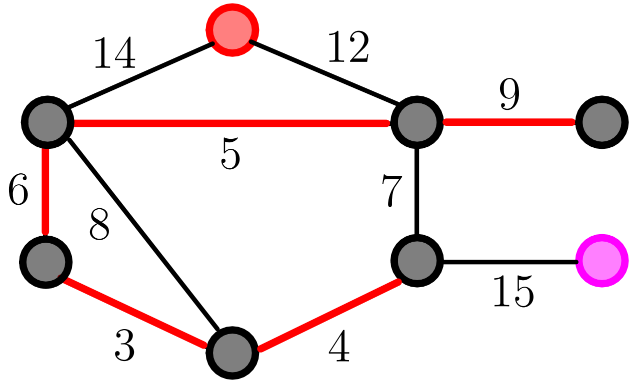
Merging connected sets of vertices



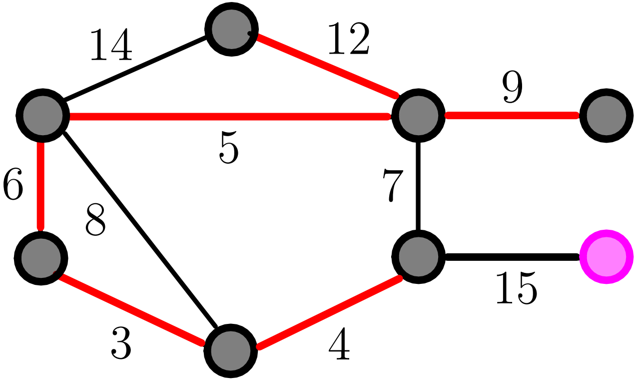
Merging connected sets of vertices



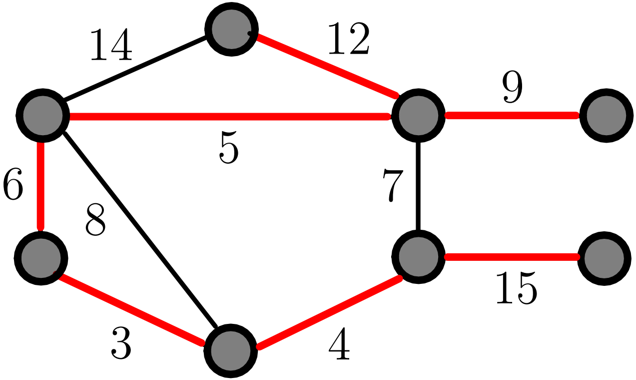
Merging connected sets of vertices



Merging connected sets of vertices



Merging connected sets of vertices



Data structures

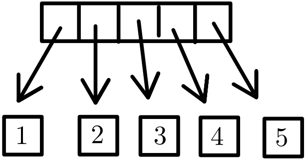
Operations on **disjoint sets of vertices:**

- **Find:** identify which set contains a given vertex
- **Union:** replace two sets by their union

```
GreedyMST_UnionFind(G)
    T ← [ ]
    U ← {{v_1}, …, {v_n}}
    sort edges by increasing weight
    for k = 1, …, m do
        if U.Find(e_k.1) ≠ U.Find(e_k.2) then
            U.Union(U.Find(e_k.1), U.Find(e_k.2))
            append e_k to T
```

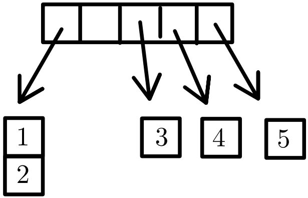
An OK solution

- U is an **array of linked lists**



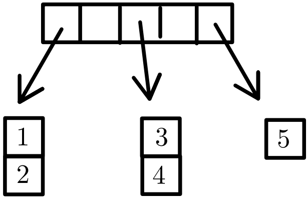
An OK solution

- U is an **array of linked lists**



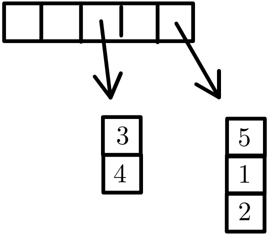
An OK solution

- U is an **array of linked lists**



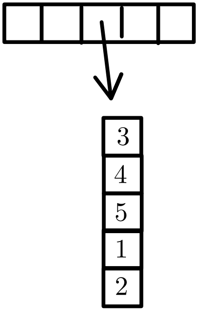
An OK solution

- U is an **array of linked lists**



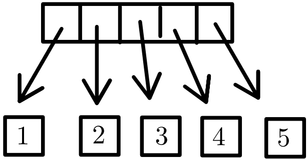
An OK solution

- U is an **array of linked lists**



An OK solution

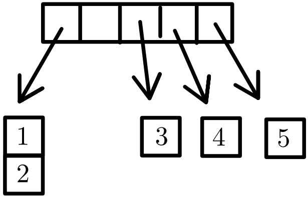
- U is an **array of linked lists**
- to do find, add an **array of indices**, $X[i] =$ set that contains i



$$X = [1, 2, 3, 4, 5]$$

An OK solution

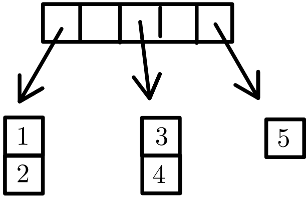
- U is an **array of linked lists**
- to do find, add an **array of indices**, $X[i] =$ set that contains i



$$X = [1, 1, 3, 4, 5]$$

An OK solution

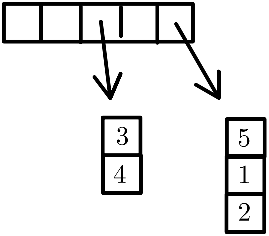
- U is an **array of linked lists**
- to do find, add an **array of indices**, $X[i]$ = set that contains i



$$X = [1, 1, 3, 3, 5]$$

An OK solution

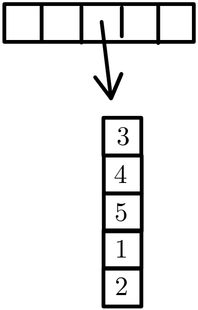
- U is an **array of linked lists**
- to do find, add an **array of indices**, $X[i]$ = set that contains i



$$X = [5, 5, 3, 3, 5]$$

An OK solution

- U is an **array of linked lists**
- to do find, add an **array of indices**, $X[i]$ = set that contains i



$$X = [3, 3, 3, 3, 3]$$

Analysis

Worst case:

- **Find** is $O(1)$
- **Union** traverses one of the linked lists, updates corresponding entries of X, concatenates two linked lists. Worst case $\Theta(n)$

Analysis

Worst case:

- **Find** is $O(1)$
- **Union** traverses one of the linked lists, updates corresponding entries of X, concatenates two linked lists. Worst case $\Theta(n)$

Kruskal's algorithm:

- sorting edges $\boldsymbol{O(m\log(m))}$
- $O(m)$ **Find**
- $O(n^2)$ **Union**

Worst case $\boldsymbol{O(m\log(m)+n^2)}$

A simple heuristics for Union

Modified Union

- each set in U keeps track of its size
- only traverse the **smaller list**
- also add a pointer to the **trail** of the lists to concatenate in $O(1)$

A simple heuristics for Union

Modified Union

- each set in U keeps track of its size
- only traverse the **smaller list**
- also add a pointer to the **trail** of the lists to concatenate in $O(1)$

Key observation: worst case for **one** union **still** $\Theta(n)$, but better total time.

- for any given vertex v, the size of the set containing v **at least doubles** when we update $X[v]$
- so $X[v]$ updated at most $\log(n)$ times
- so the **total** cost of union per vertex is $O(\log(n))$

A simple heuristics for Union

Modified Union

- each set in U keeps track of its size
- only traverse the **smaller list**
- also add a pointer to the **trail** of the lists to concatenate in $O(1)$

Key observation: worst case for **one** union **still** $\Theta(n)$, but better total time.

- for any given vertex v, the size of the set containing v **at least doubles** when we update $X[v]$
- so $X[v]$ updated at most $\log(n)$ times
- so the **total** cost of union per vertex is $O(\log(n))$

Conclusion: $\boldsymbol{O(n \log(n))}$ for all unions and $\boldsymbol{O(m \log(m))}$ total