Algorithms and Data Structures

Module 2

Lecture 7

Greedy algorithms.
Minimum Spanning Tree Problem.
Kruskal’s algorithm.

Greedy strategy

Problem types

- <u>Decision problems</u>: answer ‘true’ or ‘false’
- <u>Search problems</u>: answer is an object which satisfies given conditions (a feasible solution)
- <u>Optimization problems</u>: answer is a feasible solution which is optimal with respect to a certain cost (weight) function.

Greedy algorithms

Key characteristics of a greedy algorithm:

- Can solve an optimization problem.
- Builds solution iteratively, adding one element after another.
- At each step, adds the element which is the best at the current situation.
- Does not revise the decisions (one-pass algorithm).

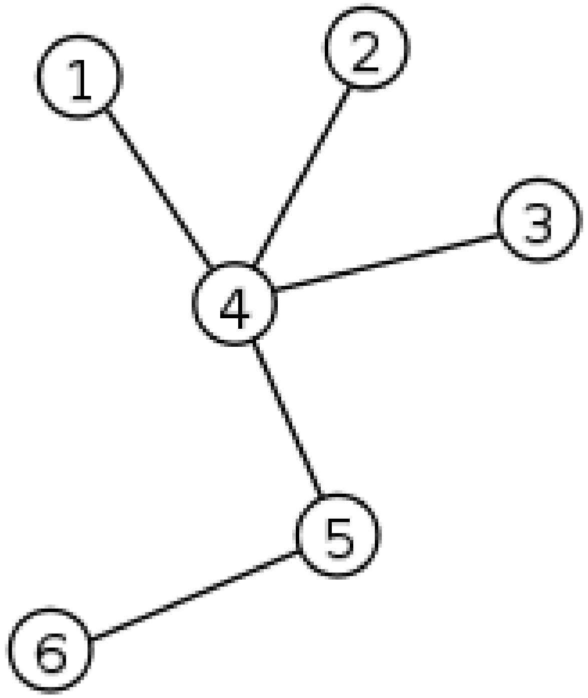
Greedy algorithms

- One can construct many different greedy algorithms for a problem.
- Greedy solution may be bad (not optimal).
- Greedy algorithms are usually efficient.

MST: definition

MST stands for 'Minimum Spanning Tree'.

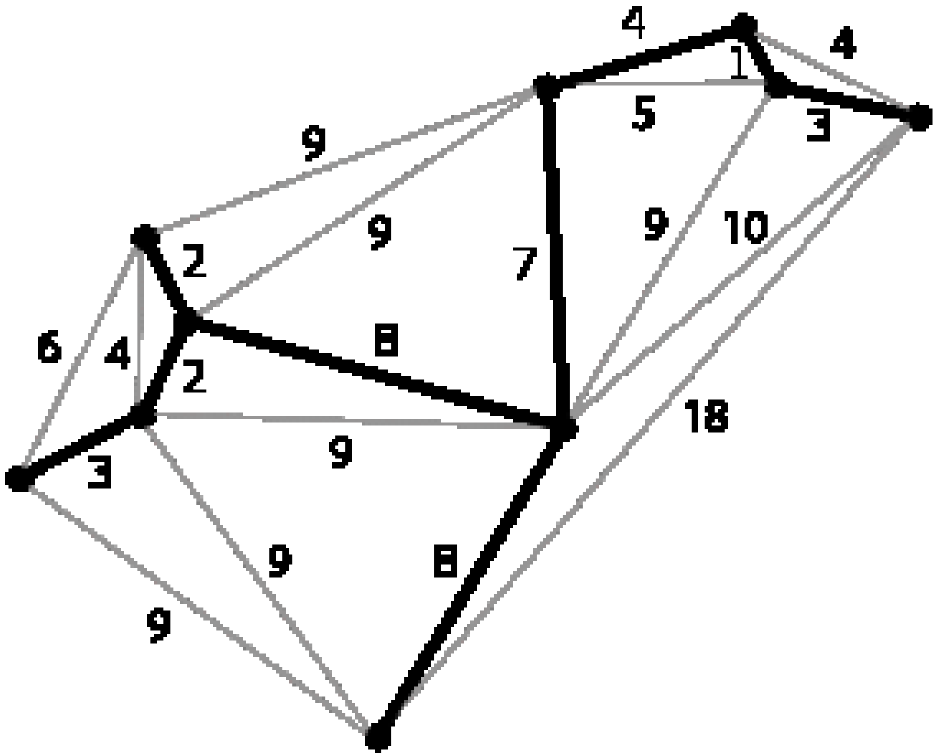
- A ***tree*** is a graph which is connected and has no (undirected) cycles.



MST: definition

MST stands for ‘Minimum Spanning Tree’.

- ***Spanning tree*** is a subgraph which is a tree and contains (spans) all ve the given graph.



MST: definition

MST stands for ‘Minimum Spanning Tree’.

- ***Minimum Spanning Tree*** is a spanning tree of a graph which has the minimum weight among all spanning trees of the graph.

Weighted graph G=(V,E), $w: E \rightarrow R$

Weight of a spanning tree: total weight of edges in the tree.

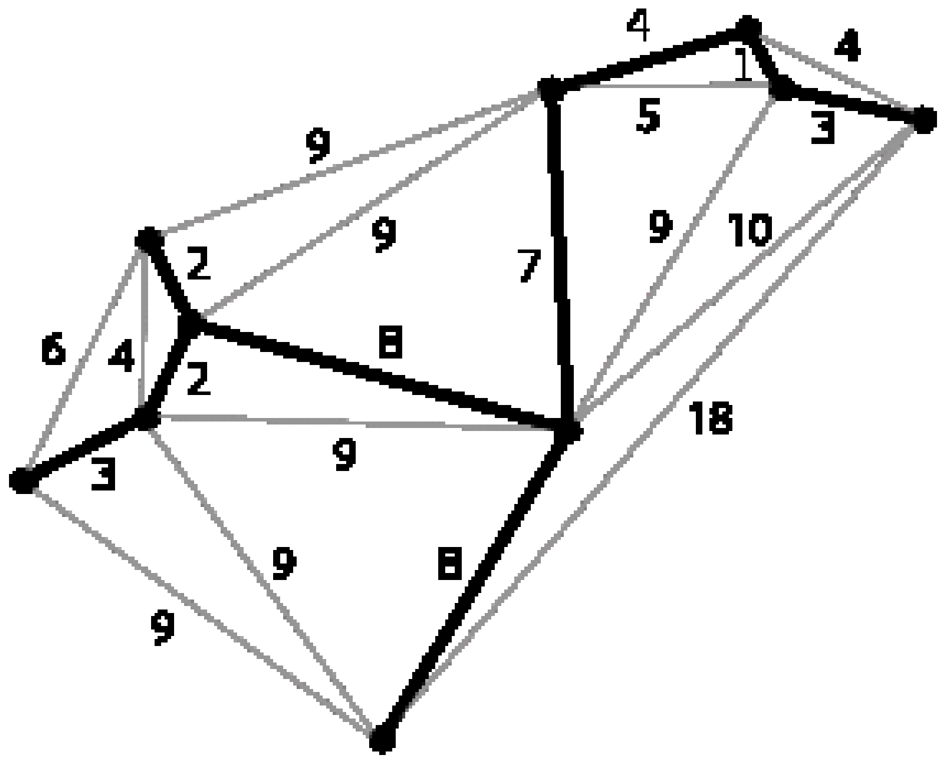
MST: definition

Weighted graph G=(V,E), $w: E \rightarrow R$ (weights, costs)

Weight of a spanning tree: total weight of edges in the tree.

Weight = 3+2+2+8+8+7+4+1+3

= 38



MST: algorithms

How can we search for a MST for the given graph?

- Brute force.

  Cayley's formula: a complete graph with n vertices contains n^{n-1} spanning trees.

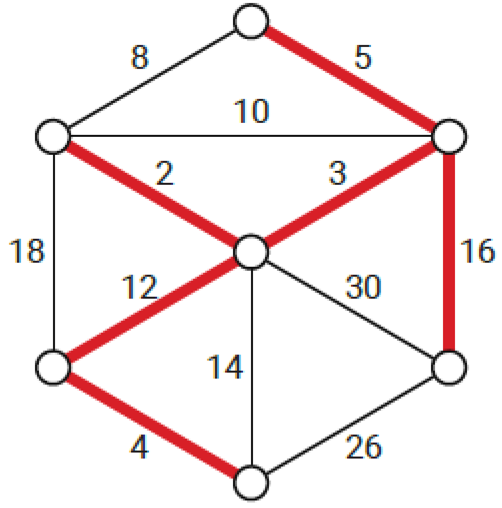
- A greedy strategy: start with an empty subgraph; add the *lightest* edge such that it does not create a cycle on the subgraph.

MST: algorithms

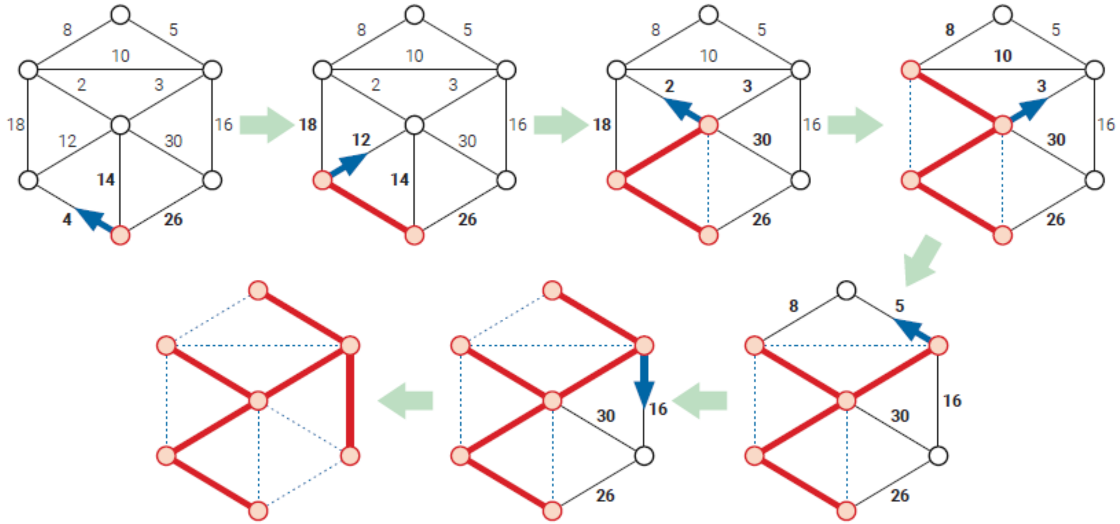
A greedy strategy: start with an empty subgraph; add the *lightest* edge such that it does not create a cycle on the subgraph (the lightest *safe* edge).

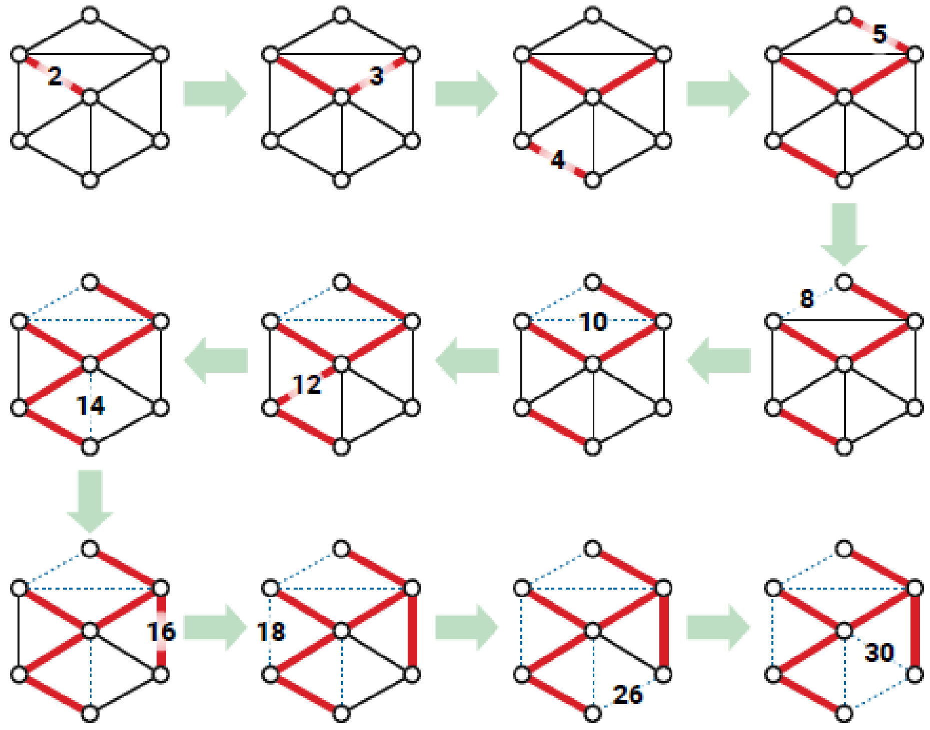
- Kruskal’s algorithm: build a *spanning* forest, adding edges until there is one component (tree).
- Prim’s algorithm: build the *tree*, adding edges until it spans the graph.

MST: algorithms



Prim's algorithm





Kruskal's algorithm

http://jeffe.cs.illinois.edu/teaching/algorithms/

Kruskal's algorithm

Given a connected graph $G(V, E)$, $|V| = n, |E| = m$.

1. $T = \emptyset$
2. Sort the set of edges by increasing their weights.
3. Scan the sequence of edges. For each edge:
   - If the current edge is safe: add this edge to T.
   - Otherwise: just skip this edge (=do nothing with it).

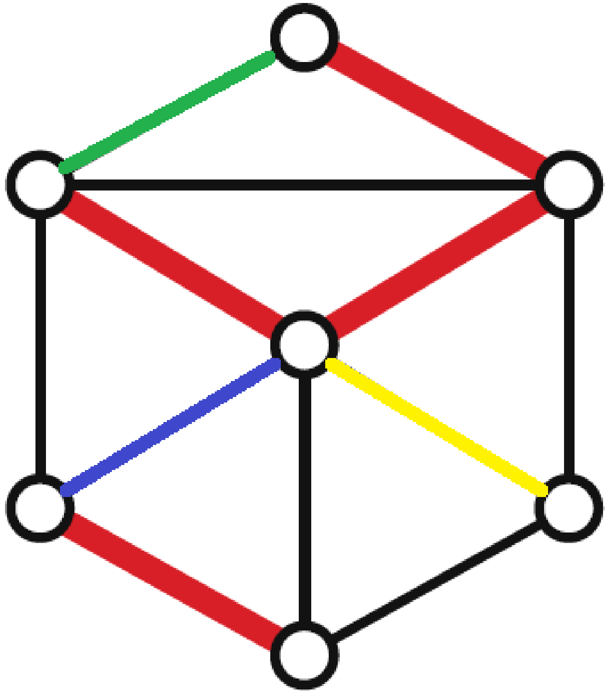
Kruskal's algorithm

Given a connected graph $G(V, E)$, $|V| = n, |E| = m$.

1. $T = \emptyset$ $O(n)$
2. Sort the set of edges by increasing their weights. $O(n \log n)$
3. Scan the sequence of edges. For each edge: *m* iterations
   - If the current edge is safe: add this edge to T. ???
   - Otherwise: just skip this edge (=do nothing with it).

Kruskal's algorithm: safety check

Given the graph $G(V, E)$, spanning forest T and the current edge $e = (u, v) \in E$, how can we check whether e is safe (=adding e to T does not create a cycle)?



Red edges belong to T.

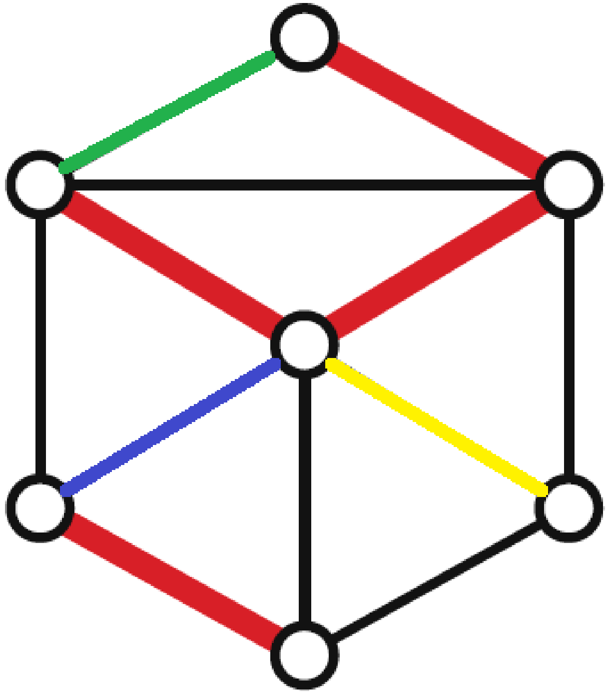
The blue and yellow edges are safe.

The green edge is unsafe.

Kruskal’s algorithm: safety check

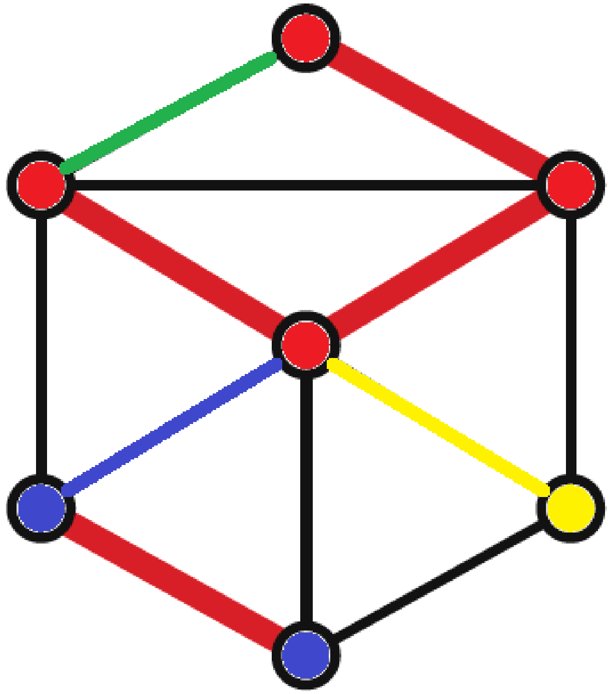
Given the graph $G(V, E)$, spanning forest T and the current edge $e = (u, v) \in E$, how can we check whether e is safe?

<u>Naïve approach</u>: add the new edge and check graph $G \cup \{e\}$ for presence of cycles. Algorithm: a modification of DFS. Complexity: $O(m) = O(n^2)$ for each check and $O(m^2) = O(m^4)$ for the total time.



Kruskal's algorithm: safety check

Given the graph $G(V, E)$, spanning forest T and the current edge $e = (u, v) \in E$, how can we check whether e is safe (=adding e to T does not create a cycle)?

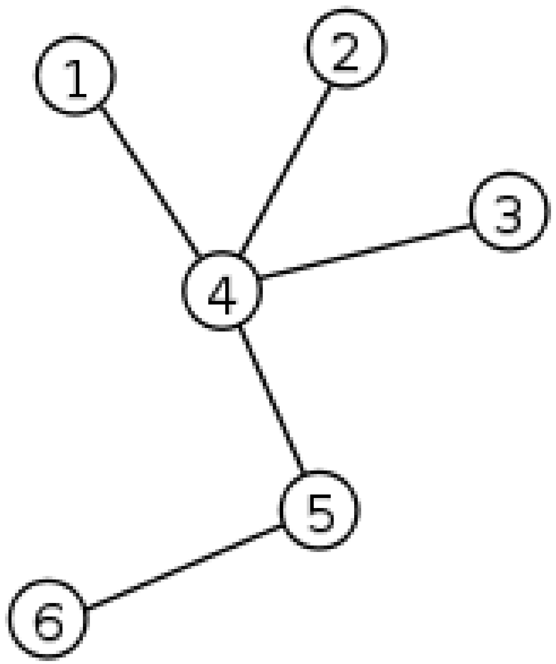


Rule: $e = (u, v)$ is safe iff its endpoints u and v belong to different components of T; otherwise e is unsafe.

Kruskal's algorithm: safety check

<u>Theorem</u> (properties of trees).

A graph $G(V, E)$ is a tree iff any of the following equivalent conditions hold:



1) *G* is connected and acyclic (contains no cycles).
2) *G* is acyclic, and a simple cycle is formed if any edge is added to *G*.
3) *G* is connected, but would become disconnected if any single edge is removed from *G*.
4) Any two vertices in *G* can be connected by a unique simple path.
5) *G* is connected and has $n - 1$ edges ($n = |V|$).
6) *G* has no simple cycles and has $n - 1$ edges.

Kruskal's algorithm: safety check

Rule: $e = (u, v)$ is safe iff its endpoints u and v belong to different components of *T*; otherwise e is unsafe.

$\Rightarrow$We need to keep a component ID for each vertex, and we also need to update this information after adding a new edge to the forest.

$\Rightarrow$ We need a Union-Find (Merge-Find) data structure that keeps a collection of disjoint subsets (components) of a set and implements operations:

- `MakeSet(v)`: creates a set $\{v\}$.
- `Find(v)`: returns the unique ID of the subset containing v.
- `Union(u,v)`: unions (merges) subsets containing u and v to a single subset.

Kruskal’s algorithm: safety check

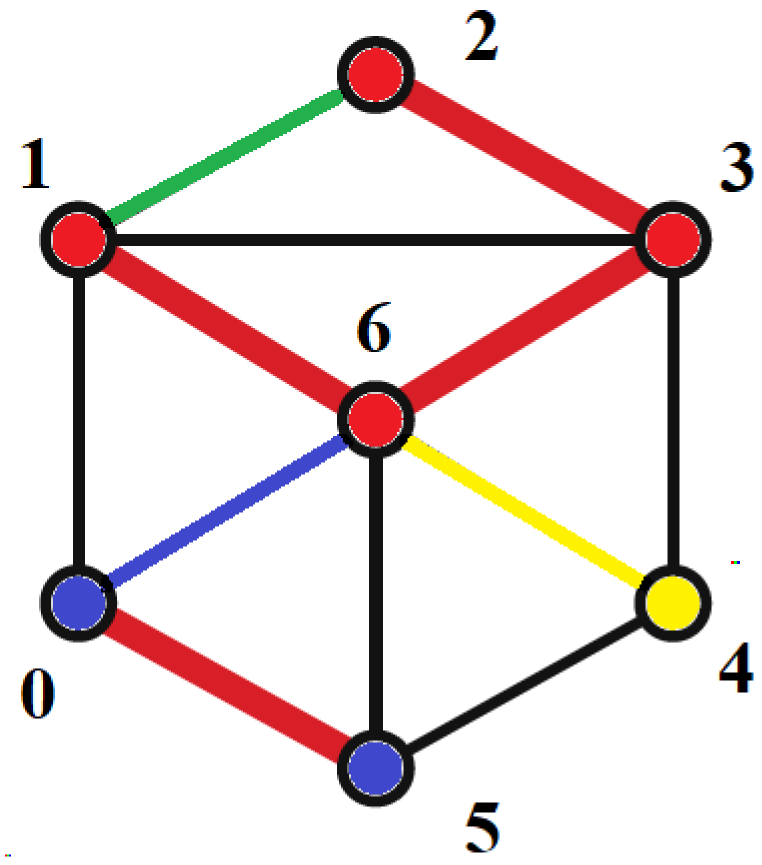
Simple implementation of Union-Find

Array `Component[1..n]`, *i*-th item contains the ID of the component currently containing the *i*-th vertex. We will use an index of a certain vertex as the component’s ID.

- MakeSet(*i*): `Component[i]=i`.
- Find(*i*): return `Component[i]`.
- Union(i,j): scan the array and for each *k*:

```
if Component[k]==Component[i] then Component[k]=Component[j]
```

Kruskal's algorithm: safety check



index	0	1	2	3	4	5	6
value	0	1	1	1	4	0	1

Kruskal's algorithm: safety check

- MakeSet(*i*): `Component[i]=i.` $O(1)$
- Find(*i*): return `Component[i].` $O(1)$
- Union(i,j): scan the array and for each *k*: $O(n)$

```
if Component[k]==Component[i] then Component[k]=Component[j]
```

For building MST we call Union $O(m)$ times => the total time for safety check is $O(mn)$.

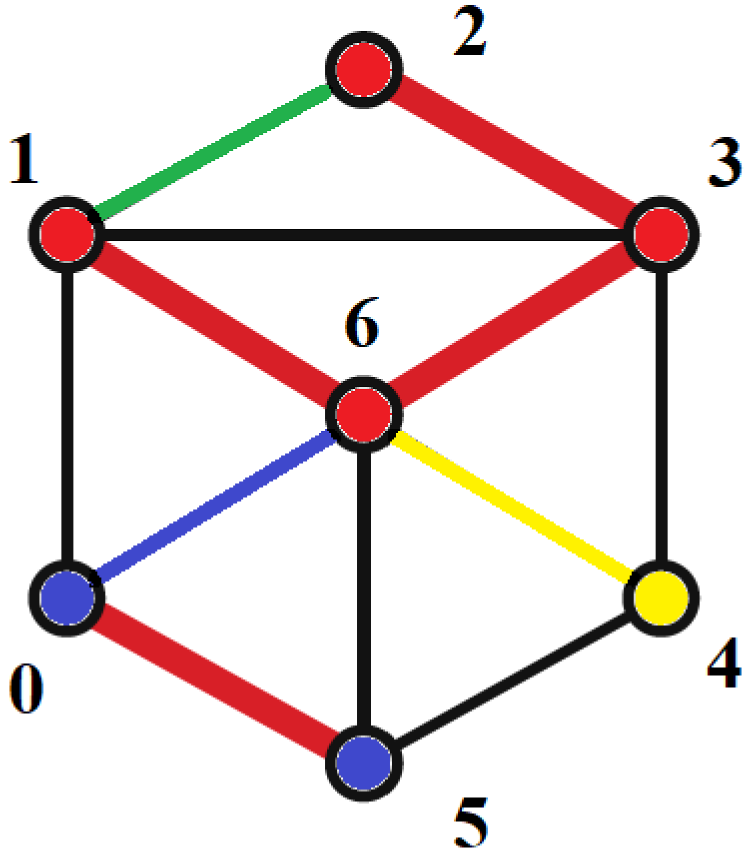
The total time complexity of Kruskal's algorithm: $O(mn)$.

Kruskal's algorithm: safety check

An improved version of the array-based implementation.

1) Explicitly maintain a list of vertices in each component. => create a list of dynamic lists of vertex indices. When two components are being merged, merge the corresponding lists as well ($O(1)$ time).
2) Explicitly count the sizes of components and when two components are being merged, elements of the smaller component take the ID of the larger component.

Kruskal’s algorithm: safety check



index	0	1	2	3	4	5	6
value	0	1	1	1	4	0	1

List of components

index (ID)	0	1	2	3	4	5	6
size	2	4	0	0	1	0	0
vertices	0	1			4		
	5	2					
		3					
		6					

Kruskal's algorithm: safety check

Theorem. For the improved version of the array-based implementation.

1) `MakeSet` takes $O(1)$ time / operation; total time is $O(n)$.
2) `Find` takes $O(1)$ time / operation; total time is $O(m)$.
3) Any sequence of k `Union` operations takes at most $O(k \log k)$ time; total time is $O(n \log n)$.

NB: For (3) we consider *amortized* time complexity instead of worst-case complexity of a single operation.

The total time complexity of Kruskal's algorithm: $O(m \log m)$.

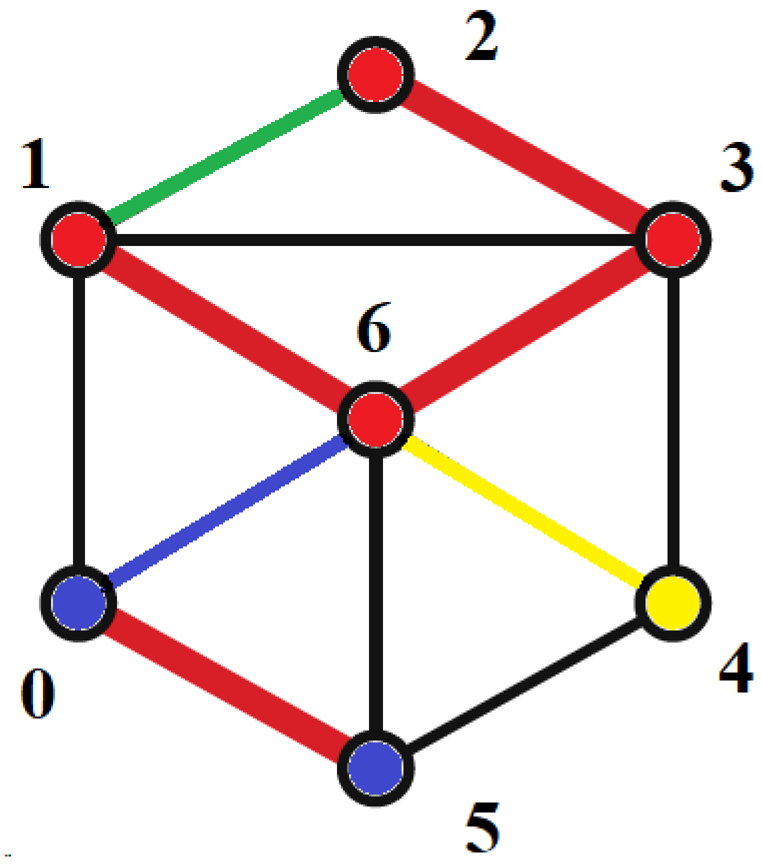
Kruskal’s algorithm: safety check

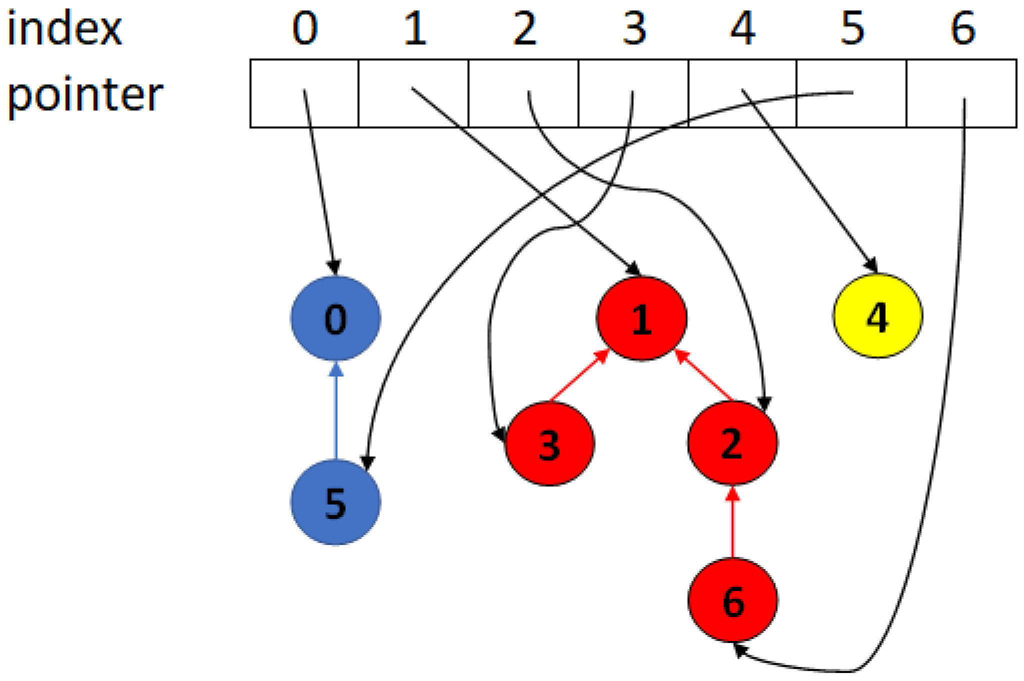
Reversed Trees – a better data structure for Union-Find

Keep a component as a dynamic tree structure:

- Each node of the tree represents a single element of the component (= a vertex of graph G).
- Each node of the tree, except the root, has a pointer to its parent node.

Kruskal’s algorithm: safety check



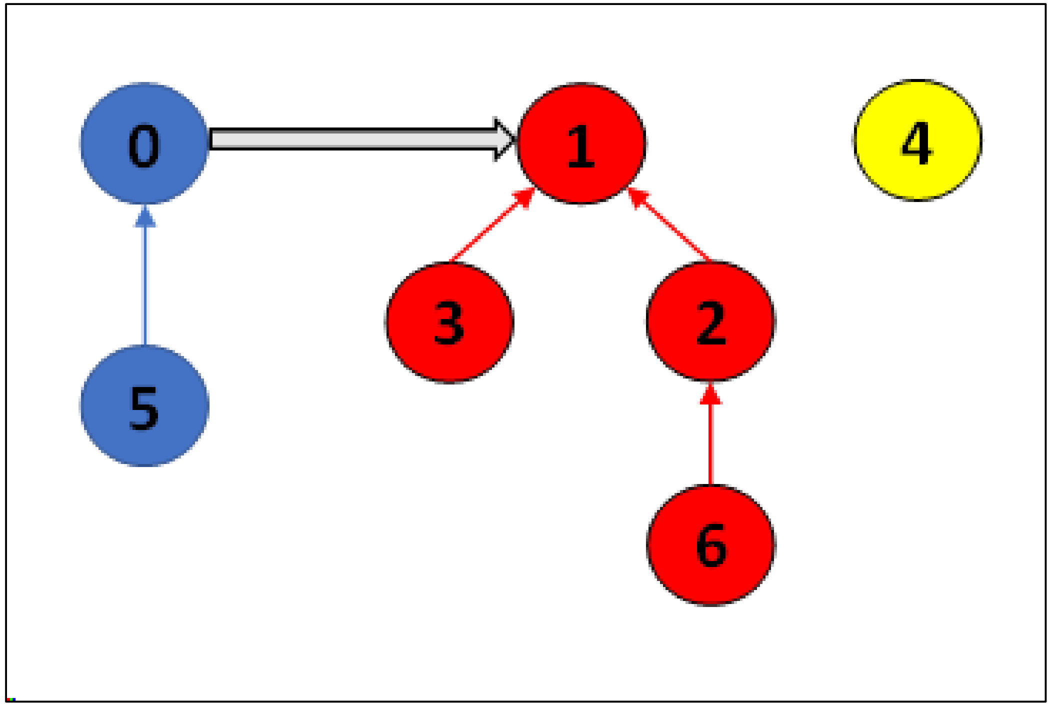


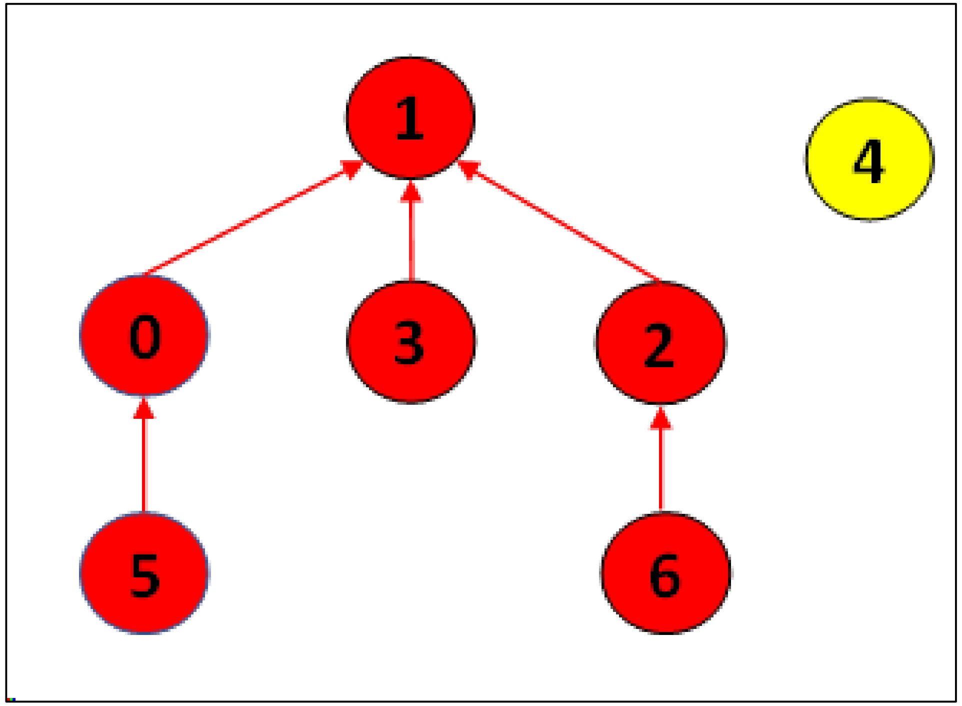
<u>Track the depths of components' trees</u>

index (ID)	0	1	2	3	4	5	6
depth	1	2	0	0	0	0	0

Kruskal’s algorithm: safety check

Union(2,5)





Kruskal’s algorithm: safety check

<u>Theorem</u>. For the reversed trees implementation:

1) `MakeSet` takes $O(1)$ time / operation; total time is $O(n)$.

2) `Find` takes $O(\log n)$ time / operation; total time is $O(m \log n)$.

3) `Union` takes $O(1)$ time / operation; total time is $O(n)$.

The total time complexity of Kruskal’s algorithm: $O(m \log m)$.