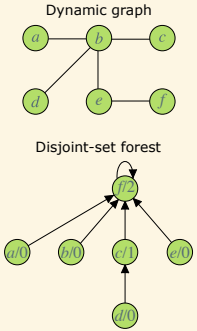
Lesson 03: Disjoint-Set Forests and Kruskal's Minimum-Spanning-Tree Algorithm

Michael T. Gastner (2 March 2023)



Disclaimer: These slides are based on and occasionally quote from 'Introduction to Algorithms' (3rd ed.) by Cormen et al. (2009), MIT Press.

Review of Previous Lesson

Depth-First Search and Identifying Strongly Connected Components

Last time, we studied depth-first search, a graph traversal algorithm, which starts at a source vertex and explores vertices along branches as far as possible before backtracking. The branches form the trees in a so-called depth-first forest.

Information about the finishing times of vertices during depth-first search are useful for identifying strongly connected components in directed graphs. The idea is to run depth-first search a second time but now on the transpose of the input graph. By choosing source vertices in descending order of finishing times calculated during the first search, the trees in the depth-first forest at the end of the second search correspond to the strongly connected components.

2 / 26

Learning Objectives

By the end of this lesson, you should be able to ...

- Recall the motivation for disjoint-set data structures.
- Apply union-find operations to disjoint-set forests.
- State the growth of the worst-case running time of constructing a disjoint-set forest as a function of the number of nodes and union-find operations.
- Explain what a minimum spanning tree is.
- Recall the motivation for Kruskal's minimum-spanning-tree algorithm.
- Apply Kruskal's algorithm to a given graph.
- State the growth of the running time of Kruskal's algorithm as a function of the number of vertices and the number of edges.

3 / 26

Dynamic Disjoint Sets

Connected Components in Undirected Graphs

Last time, we learnt that we can calculate connected components in undirected graphs with depth-first search. However, depth-first search assumes that the graph is static.

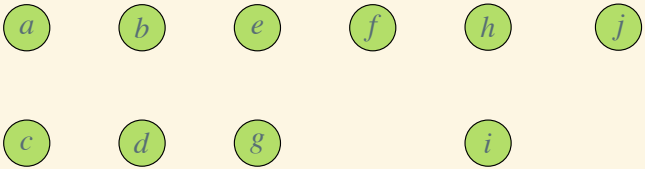
Suppose instead that edges are added to a graph one by one, and we want to know the connected components at all stages of the growth process.

We could run depth-first search repeatedly after each new edge is added to the graph. However, this strategy is inefficient. Let us take a look at an example to gain intuition.

4 / 26

Dynamic Disjoint Sets

Example



Consider the vertices $a, b, \ldots, j$ shown above. Without any edges, each vertex is in its own connected component.

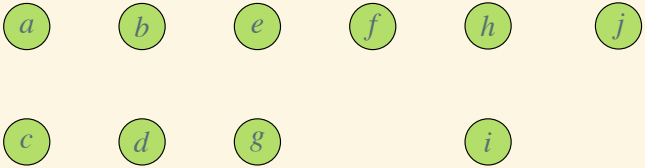
Let us insert the edges (b, d), (e, g), (a, c), (h, i), (a, b), (e, f) and (b, c) one by one.

The next few slides show how the connected components change as edges are inserted into the graph.

5 / 26

Dynamic Disjoint Sets

Example

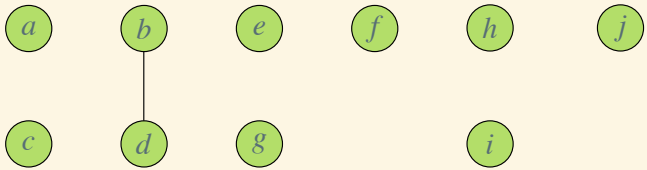


Edge inserted	Collection of disjoint sets (i.e. connected components)									
none	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)										
(e, g)										
(a, c)										
(h, i)										
(a, b)										
(e, f)										
(b, c)										

5 / 26

Dynamic Disjoint Sets

Example

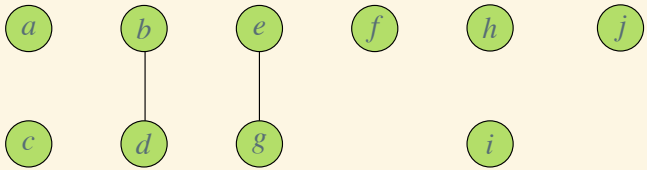


Edge inserted	Collection of disjoint sets (i.e. connected components)									
none	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)										
(a, c)										
(h, i)										
(a, b)										
(e, f)										
(b, c)										

5 / 26

Dynamic Disjoint Sets

Example

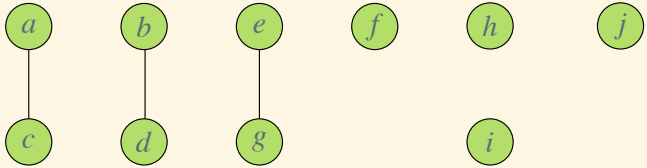


Edge inserted	Collection of disjoint sets (i.e. connected components)									
none	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)										
(h, i)										
(a, b)										
(e, f)										
(b, c)										

5 / 26

Dynamic Disjoint Sets

Example

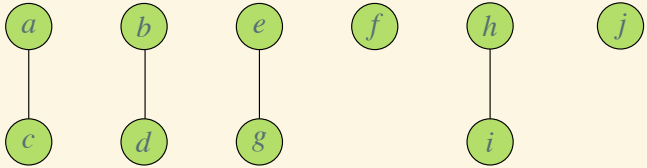


Edge inserted	Collection of disjoint sets (i.e. connected components)									
none	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h, i)										
(a, b)										
(e, f)										
(b, c)										

5 / 26

Dynamic Disjoint Sets

Example

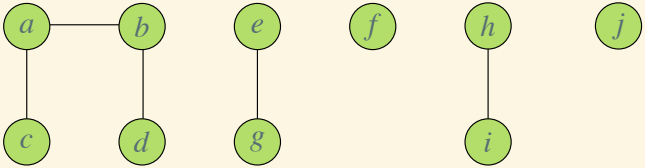


Edge inserted	Collection of disjoint sets (i.e. connected components)									
none	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(a, b)										
(e, f)										
(b, c)										

5 / 26

Dynamic Disjoint Sets

Example

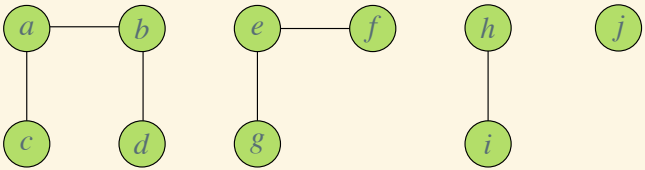


Edge inserted	Collection of disjoint sets (i.e. connected components)									
none	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)										
(b,c)										

5 / 26

Dynamic Disjoint Sets

Example

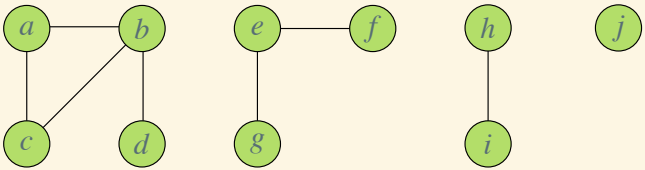


Edge inserted	Collection of disjoint sets (i.e. connected components)									
none	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(a, b)	$\{a, b, c, d\}$				$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(e, f)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
(b, c)										

5 / 26

Dynamic Disjoint Sets

Example



Edge inserted	Collection of disjoint sets (i.e. connected components)									
none	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(a, b)	$\{a, b, c, d\}$				$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(e, f)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

5 / 26

Disjoint-Set Data Structures

Storage for a Collection of Sets and a Representative of Each Set

A **disjoint-set data structure** maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.

We identify each set by a **representative**, which is some member of the set. In our applications, it does not matter which member is used as a representative.

However, if we ask for the representative of a dynamic set twice without modifying the set between the requests, we demand that we get the same answer both times.

6 / 26

Disjoint-Set Operations

Make-Set, Union and Find-Set

We represent members of the sets by objects. Letting x denote an object, we wish to support the following operations applied to objects :

- Make-Set(x) creates a new set whose only member is x.
- Union(x, y) unites the dynamic sets that contain x and y into a new set that is the union of these two sets.
- Find-Set(x) returns a pointer to the representative of the set containing x.

We assume that all sets are disjoint before and after these operations.

7 / 26

Disjoint-Set Forests

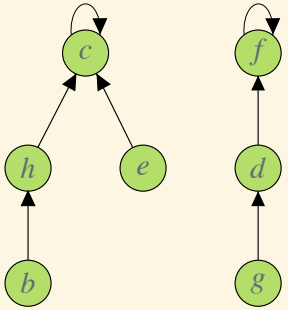
Rooted Trees Represent Sets

A **disjoint-set forest** is a type of disjoint-set data structure that allows for efficient implementations of MAKE-SET, UNION and FIND-SET.

In a disjoint-set forest, sets are represented by rooted trees, with each node[a] containing one member and each tree corresponding to one set.

The root of the tree contains the representative of the set and is its own parent.

[a] Following Cormen et al. (2009), the term 'node' is used for a vertex in a rooted tree.



Disjoint-set forest with two trees representing the sets $\{b, c, e, h\}$, with c as the representative, and $\{d, f, g\}$, with f as the representative.

8 / 26

Node Attributes in Disjoint-Set Forests

Parent and Rank

We designate the **parent** of node x by $x.p$.

We also maintain an integer **rank**, denoted $x.rank$, which is an upper bound on the height of x (i.e. the number of edges in the longest simple path between x and a descendant leaf).

When $\textsc{Make-Set}(x)$ creates a set containing only x, we make x the root and assign a rank of 0 to x.

$\textsc{Make-Set}(x)$

1 $x.p = x$ // Make x the root.
2 $x.rank = 0$

9 / 26

Disjoint-Set Operations for Disjoint-Set Forests

Pseudocode for FIND-SET

Whenever FIND-SET is called, it applies **path compression** (i.e. every node along the search path to x will be directly connected to the root by the end of the operation).

```
FIND-SET(x)
1  if x ≠ x.p
2        x.p = FIND-SET(x.p)
3  return x.p
```


10 / 26

Disjoint-Set Operations for Disjoint-Set Forests

Pseudocode for UNION and LINK

When UNION merges two sets, we must distinguish between two cases:

- If the roots of the trees have unequal rank, we make the root with higher rank the parent of the root with lower rank, but the ranks themselves remain unchanged.
- If the roots have equal ranks, we arbitrarily choose one of the roots as the parent and increment its rank.

```
UNION(x, y)
1  rx = FIND-SET(x)
2  ry = FIND-SET(y)
3  if rx ≠ ry
4        LINK(rx, ry)
```

```
LINK(x, y)
1  if x.rank > y.rank
2        y.p = x
3  else x.p = y
4        if x.rank == y.rank
5              y.rank = y.rank + 1
```


11 / 26

Disjoint-Set Operations for Disjoint-Set Forests

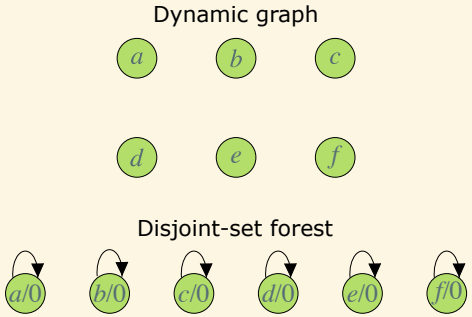
Example

Consider a dynamic undirected graph with $V = \{a, b, ..., f\}$.

Start from an empty graph.

Then add the edges (b, c), (b, d), (e, f), (b, e) and (b, a) one by one.

The numbers in the disjoint-set forest on the right are the ranks of the vertices.



12 / 26

Disjoint-Set Operations for Disjoint-Set Forests

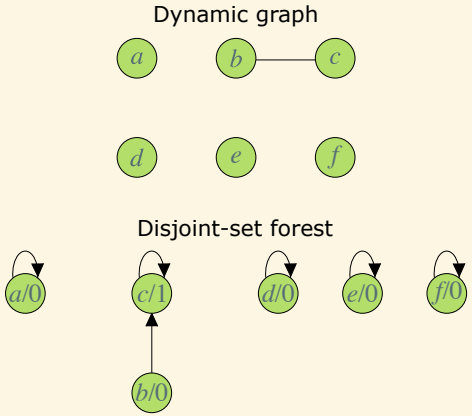
Example

Consider a dynamic undirected graph with $V = \{a, b, ..., f\}$.

Start from an empty graph.

Then add the edges (b, c), (b, d), (e, f), (b, e) and (b, a) one by one.

The numbers in the disjoint-set forest on the right are the ranks of the vertices.



12 / 26

Disjoint-Set Operations for Disjoint-Set Forests

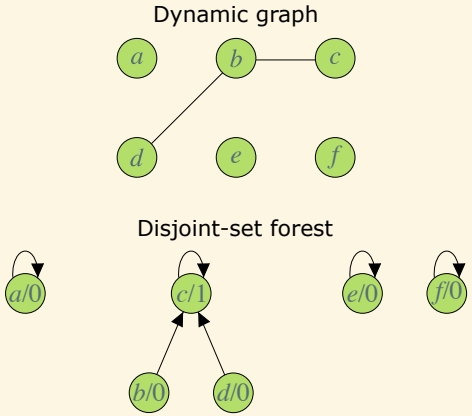
Example

Consider a dynamic undirected graph with $V = \{a, b, ..., f\}$.

Start from an empty graph.

Then add the edges (b, c), (b, d), (e, f), (b, e) and (b, a) one by one.

The numbers in the disjoint-set forest on the right are the ranks of the vertices.



12 / 26

Disjoint-Set Operations for Disjoint-Set Forests

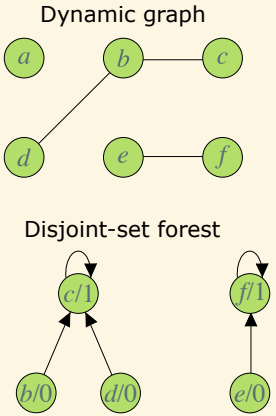
Example

Consider a dynamic undirected graph with $V = \{a, b, ..., f\}$.

Start from an empty graph.

Then add the edges (b, c), (b, d), (e, f), (b, e) and (b, a) one by one.

The numbers in the disjoint-set forest on the right are the ranks of the vertices.



12 / 26

Disjoint-Set Operations for Disjoint-Set Forests

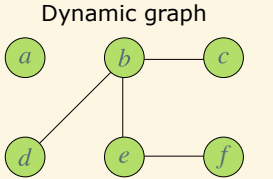
Example

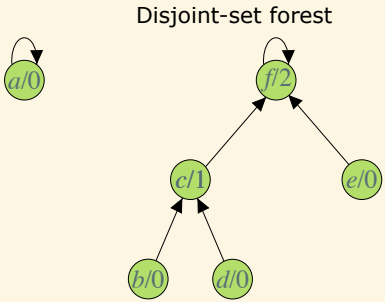
Consider a dynamic undirected graph with $V = \{a, b, ..., f\}$.

Start from an empty graph.

Then add the edges (b, c), (b, d), (e, f), (b, e) and (b, a) one by one.

The numbers in the disjoint-set forest on the right are the ranks of the vertices.





12 / 26

Disjoint-Set Operations for Disjoint-Set Forests

Example

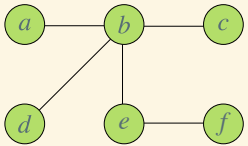
Consider a dynamic undirected graph with $V = \{a, b, ..., f\}$.

Start from an empty graph.

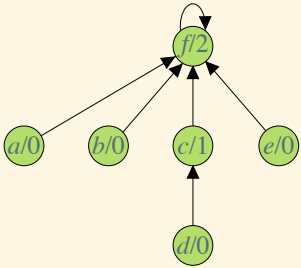
Then add the edges (b, c), (b, d), (e, f), (b, e) and (b, a) one by one.

The numbers in the disjoint-set forest on the right are the ranks of the vertices.

Dynamic graph



Disjoint-set forest



12 / 26

Running Time of Disjoint-Set Operations

Almost Linear in the Number of Edges

The running time of all disjoint-forest operations starting from an empty graph with $|V|$ vertices to a graph with $|E|$ edges is $O(E\,\alpha(V))$, where $\alpha(V)$ is a very slowly growing function.

In all practical situations, $\alpha(V) \leq 4$. Thus, we can view the running time as almost linear in $|E|$.

13 / 26

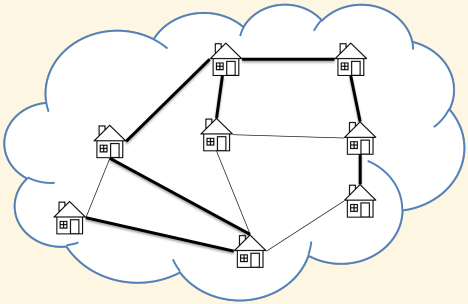
Minimum Spanning Trees

Connect All Vertices Such That the Total Edge Weight Is Minimised

Imagine that a cable company has to run a cable that is supposed to connect all the houses in a town to the Internet.

To reduce costs, the company would like to minimise the length of cable.

The solution to their problem is a **minimum spanning tree**.



14 / 26

Minimum Spanning Trees

Weighted Graphs

To construct a minimum spanning tree, we need a graph $G(V, E)$ and edge weights.

For example, V could be the set of all the houses in a town, and E contains all pairs of houses that could be directly connected to each other by a cable.

For each edge $(u, v) \in E$, we have a weight $w(u, v) \geq 0$ specifying the cost (e.g. measured in distance or man-hours).

15 / 26

Minimum Spanning Trees

Formal Definition

We wish to find a subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is minimised.

We can assume that the solution T is acyclic (i.e. a tree) because removing an edge from a cycle would never increase the total weight while the remaining edges would still connect all of the vertices.

A graph T that satisfies these constraints is, thus, called a **minimum spanning tree**.

16 / 26

Kruskal's algorithm

Growing a Forest of Small Edges Until It Connects All Vertices

Kruskal's algorithm is a strategy to solve the minimum-spanning-tree problem.

It maintains a set A of edges that forms a forest (i.e. a set of trees) at all times. By adding edges in ascending order of weight, the forest eventually connects all vertices. However, we must not add any edge that would create a cycle.

To detect whether an edge would create a cycle, we can use the operations MAKE-SET, FIND-SET and UNION that we defined above. The pseudocode is on the next page.

17 / 26

Kruskal's Algorithm

Pseudocode

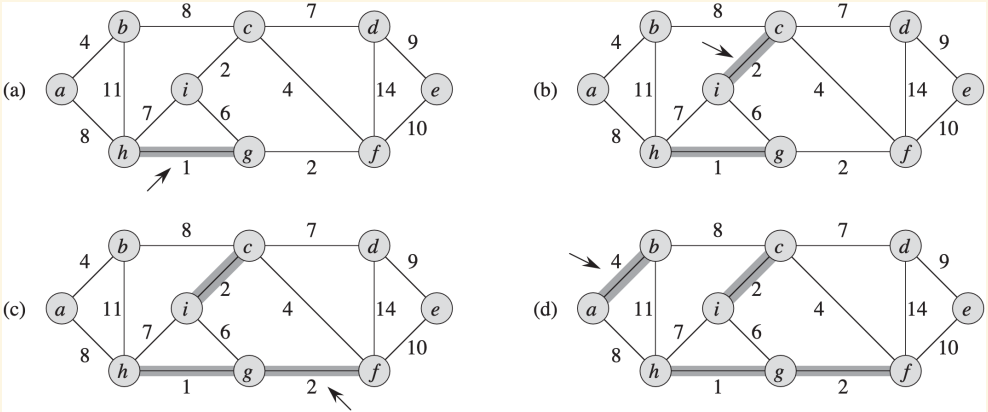
MST-KRUSKAL(G, w)

```
1  A = ∅
2  for each vertex v ∈ G.V
3        MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6        if FIND-SET(u) ≠ FIND-SET(v)
7              A = A ∪ {(u, v)}
8              UNION(u, v)
9  return A
```


18 / 26

Kruskal's Algorithm

Illustration

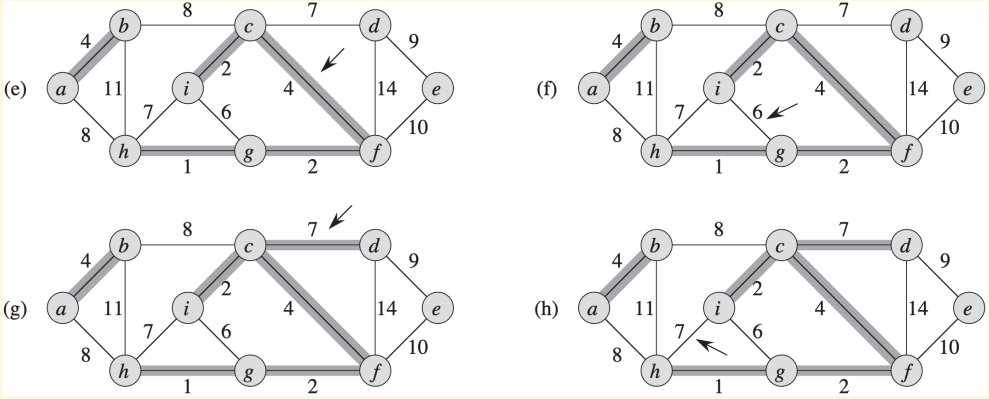


19 / 26

Kruskal's Algorithm

Illustration

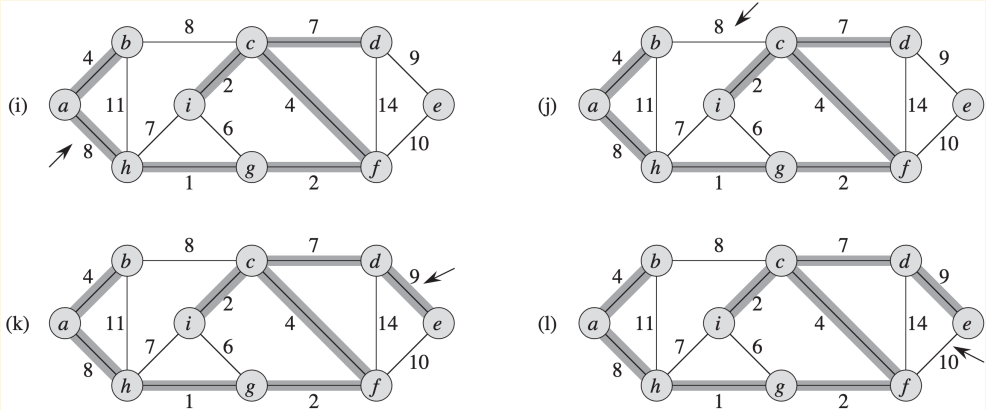
Shaded edges belong to the forest being grown. An arrow points to the edge under consideration at each step of the algorithm.



20 / 26

Kruskal's Algorithm

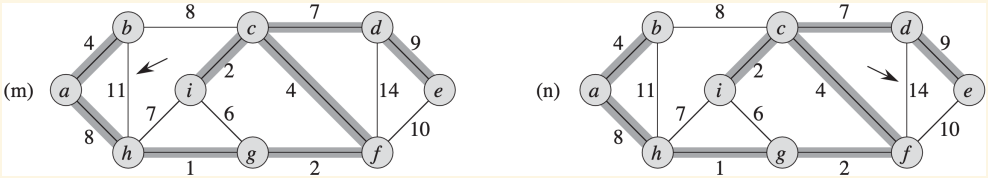
Illustration



21 / 26

Kruskal's Algorithm

Illustration



22 / 26

Kruskal's Algorithm

Exercise

Find the minimum spanning tree of the graph depicted below by applying Kruskal's algorithm.

Numbers near the edges symbolise edge weights.

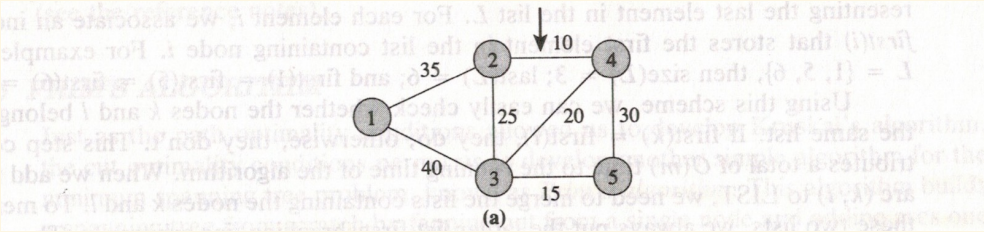


Image from Ahuja et al. (1993), Network Flows, Prentice Hall, Upper Saddle River, NJ.

23 / 26

Kruskal's Algorithm

Solution to Exercise

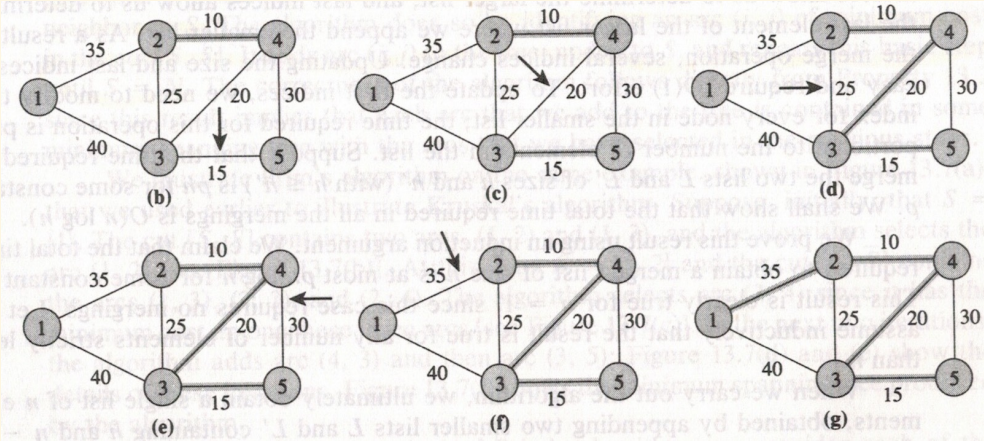


Image from Ahuja et al. (1993), Network Flows, Prentice Hall, Upper Saddle River, NJ.

24 / 26

Running Time of Kruskal's Algorithm
$O(E \log V)$

If we use a disjoint-set forest implementation to represent the components in set A, then it takes $O(E\,\alpha(V))$ time to construct the full minimum spanning tree.

It takes additional $O(E \log E)$ time to sort the edges on line 4 of MST-KRUSKAL.

Because $|E| < |V|^2$ in a simple graph, we have $\log |E| = O(\log V)$. Moreover $\alpha(|V|) = O(\log V)$.

In conclusion, Kruskal's algorithm needs $O(E \log V)$ time.

25 / 26

Outlook and Conclusion

This Lesson: Kruskal's Algorithm. Next Lesson: Prim's Algorithm

In this lesson, we learnt how to find minimum spanning trees using Kruskal's algorithm, which maintains a forest comprising all vertices. The forest grows into a single tree connecting all the vertices by adding edges.

In the next lesson, we will learn an alternative algorithm, known as Prim's algorithm, which maintains a single tree. Starting from an empty tree, edges are added one by one until the tree connects all vertices.

26 / 26