Making Kruskal's algorithm fast
COMS20010 (Algorithms II)

John Lapinskas, University of Bristol

Implementing Kruskal's algorithm

Algorithm: KRUSKAL

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

Sort the edges by weight as $e_1, \ldots, e_m$, with $w(e_1) \leq \cdots \leq w(e_m)$.
Let $T \leftarrow (V, \emptyset)$ be the empty tree on V.
for $i = 1$ *to* m **do**
    if $T + e_i$ *has no cycles* **then**
        Let $T \leftarrow T + e_i$.
Return T.

Implementing Kruskal's algorithm

Algorithm: KRUSKAL

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

```
1 Sort the edges by weight as e_1, ..., e_m, with w(e_1) ≤ ··· ≤ w(e_m).
2 Let T ← (V, ∅) be the empty tree on V.
3 for i = 1 to m do
4     if T + e_i has no cycles then
5         Let T ← T + e_i.
6 Return T.
```

Lines 1, 2 and 6 take $O(|E| \log |E|)$ time, and lines 3–5 repeat $|E|$ times.

We *could* implement line 4 with BFS...

Implementing Kruskal's algorithm

Algorithm: KRUSKAL

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

1 Sort the edges by weight as $e_1, \dots, e_m$, with $w(e_1) \leq \dots \leq w(e_m)$.
2 Let $T \leftarrow (V, \emptyset)$ be the empty tree on V.
3 **for** $i = 1$ *to* m **do**
4   **if** $T + e_i$ *has no cycles* **then**
5   Let $T \leftarrow T + e_i$.
6 Return T.

Lines 1, 2 and 6 take $O(|E| \log |E|)$ time, and lines 3–5 repeat $|E|$ times.

We *could* implement line 4 with BFS... but this would take $\Theta(|E|)$ time, giving us a worst-case running time of $\Theta(|E|^2)$. That's bad.

Implementing Kruskal's algorithm: Take 2

Idea: Joining two tree components with an edge will never add a cycle, and adding an edge inside a tree component will always add one.

So when we consider an edge e_i to T, we just need to make sure both endpoints aren't in the same component — this implementation will work:

Algorithm: $\textsc{Kruskal}$

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

```
Sort the edges by weight as e_1, ..., e_m, with w(e_1) ≤ ··· ≤ w(e_m).
Let T ← (V, ∅) be the empty tree on V.
Let C ← the set of T's components.
for i = 1 to m do
    Let C_1 and C_2 be the components containing e_i's endpoints in C.
    if C_1 ≠ C_2 then
        Let T ← T + e_i.
        Merge C_1 and C_2 in C.
Return T.
```

The key problem

Algorithm: KRUSKAL

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

Sort the edges by weight as $e_1, \dots, e_m$, with $w(e_1) \leq \cdots \leq w(e_m)$.
Let $T \leftarrow (V, \emptyset)$ be the empty tree on V.
Let $\mathcal{C} \leftarrow$ the set of T's components.
for $i = 1$ *to* m **do**
    Let C_1 and C_2 be the components containing e_i's endpoints in $\mathcal{C}$.
    if $C_1 \neq C_2$ **then**
        Let $T \leftarrow T + e_i$.
        Merge C_1 and C_2 in $\mathcal{C}$.
Return T.

The key problem

Algorithm: $\textsc{Kruskal}$

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

Sort the edges by weight as $e_1, \dots, e_m$, with $w(e_1) \leq \cdots \leq w(e_m)$.
Let $T \leftarrow (V, \emptyset)$ be the empty tree on V.
Let $\mathcal{C} \leftarrow$ the set of T's components.
for $i = 1$ *to* m **do**
    Let C_1 and C_2 be the components containing e_i's endpoints in $\mathcal{C}$.
    if $C_1 \neq C_2$ **then**
        Let $T \leftarrow T + e_i$.
        Merge C_1 and C_2 in $\mathcal{C}$.
Return T.

But how do we implement $\mathcal{C}$?

The key problem

Algorithm: Kruskal

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

```
Sort the edges by weight as e_1, ..., e_m, with w(e_1) ≤ ··· ≤ w(e_m).
Let T ← (V, ∅) be the empty tree on V.
Let 𝒞 ← the set of T's components.
for i = 1 to m do
    Let C_1 and C_2 be the components containing e_i's endpoints in 𝒞.
    if C_1 ≠ C_2 then
        Let T ← T + e_i.
        Merge C_1 and C_2 in 𝒞.
Return T.
```

But how do we implement $\mathcal{C}$?

A linked list for each component?

The key problem

Algorithm: KRUSKAL

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

```
Sort the edges by weight as e_1, ..., e_m, with w(e_1) ≤ ··· ≤ w(e_m).
Let T ← (V, ∅) be the empty tree on V.
Let 𝒞 ← the set of T's components.
for i = 1 to m do
    Let C_1 and C_2 be the components containing e_i's endpoints in 𝒞.
    if C_1 ≠ C_2 then
        Let T ← T + e_i.
        Merge C_1 and C_2 in 𝒞.
Return T.
```

But how do we implement $\mathcal{C}$?

A linked list for each component? Then merging will take $O(1)$ time, but finding C_1 and C_2 could take $\Omega(|V|)$ time, giving a runtime of $\Omega(|V||E|)$.

The key problem

Algorithm: KRUSKAL

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

```
Sort the edges by weight as e_1, ..., e_m, with w(e_1) ≤ ··· ≤ w(e_m).
Let T ← (V, ∅) be the empty tree on V.
Let 𝒞 ← the set of T's components.
for i = 1 to m do
    Let C_1 and C_2 be the components containing e_i's endpoints in 𝒞.
    if C_1 ≠ C_2 then
        Let T ← T + e_i.
        Merge C_1 and C_2 in 𝒞.
Return T.
```

But how do we implement $\mathcal{C}$?

A linked list for each component? Then merging will take $O(1)$ time, but finding C_1 and C_2 could take $\Omega(|V|)$ time, giving a runtime of $\Omega(|V||E|)$.

An array for each component?

The key problem

Algorithm: KRUSKAL

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

```
Sort the edges by weight as e_1, ..., e_m, with w(e_1) ≤ ··· ≤ w(e_m).
Let T ← (V, ∅) be the empty tree on V.
Let 𝒞 ← the set of T's components.
for i = 1 to m do
    Let C_1 and C_2 be the components containing e_i's endpoints in 𝒞.
    if C_1 ≠ C_2 then
        Let T ← T + e_i.
        Merge C_1 and C_2 in 𝒞.
Return T.
```

But how do we implement $\mathcal{C}$?

A linked list for each component? Then merging will take $O(1)$ time, but finding C_1 and C_2 could take $\Omega(|V|)$ time, giving a runtime of $\Omega(|V||E|)$.

An array for each component? Then finding C_1 and C_2 will take $O(1)$ time, but merging will take $\Omega(|V|)$, so we still get $\Omega(|V||E|)$ overall...

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

`MakeUnionFind`$(v_1, v_2, v_3, v_4, v_5, v_6)$;

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	2	3	4	5	6
$\{v_1\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_5\}$	$\{v_6\}$

`MakeUnionFind`$(v_1, v_2, v_3, v_4, v_5, v_6)$;

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	2	3	4	5	6
$\{v_1\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_5\}$	$\{v_6\}$

`FindSet`(v_5);

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	2	3	4	5	6
$\{v_1\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_5\}$	$\{v_6\}$

`FindSet`(v_5); Returns 5.

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	2	3	4	5	6
$\{v_1\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_5\}$	$\{v_6\}$

`Union`(v_1, v_2);

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	3	4	5	6
$\boldsymbol{\{v_1, v_2\}}$	$\{v_3\}$	$\{v_4\}$	$\{v_5\}$	$\{v_6\}$

`Union`(v_1, v_2);

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	3	4	5	6
$\{v_1, v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_5\}$	$\{v_6\}$

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	3	4	5	6
$\{v_1, v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_5\}$	$\{v_6\}$

`Union`(v_3, v_5);

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	4	**7**	6
$\{v_1, v_2\}$	$\{v_4\}$	$\boldsymbol{\{v_3, v_5\}}$	$\{v_6\}$

`Union`(v_3, v_5);

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	4	7	6
$\{v_1, v_2\}$	$\{v_4\}$	$\{v_3, v_5\}$	$\{v_6\}$

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	4	7	6
$\{v_1, v_2\}$	$\{v_4\}$	$\{v_3, v_5\}$	$\{v_6\}$

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

1	4	7	6
$\{v_1, v_2\}$	$\{v_4\}$	$\{v_3, v_5\}$	$\{v_6\}$

`Union`(v_4, v_2);

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

42	7	6
$\{v_1, v_2, v_4\}$	$\{v_3, v_5\}$	$\{v_6\}$

`Union`(v_4, v_2);

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

42	7	6
$\{v_1, v_2, v_4\}$	$\{v_3, v_5\}$	$\{v_6\}$

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

42	7	6
$\{v_1, v_2, v_4\}$	$\{v_3, v_5\}$	$\{v_6\}$

`Union`(v_5, v_6);

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

42

$\{v_1, v_2, v_4\}$

$\boldsymbol{\{v_3, v_5, v_6\}}$

`Union`(v_5, v_6);

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

42

$\{v_1, v_2, v_4\}$

$\{v_3, v_5, v_6\}$

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

42

$\{v_1, v_2, v_4\}$

$\{v_3, v_5, v_6\}$

`FindSet`(v_2);

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

42

$\{v_1, v_2, v_4\}$

$\{v_3, v_5, v_6\}$

`FindSet`(v_2); Returns 42.

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

42

$\{v_1, v_2, v_4\}$

$\{v_3, v_5, v_6\}$

`FindSet`(v_5);

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

42 

$\{v_1, v_2, v_4\}$ $\{v_3, v_5, v_6\}$

`FindSet`(v_5); Returns .

Note that `Union` may affect set identifiers unpredictably!

The solution

We need to use a **union-find** data structure, also known as a **disjoint-set** or **merge-find** data structure. It supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$.
- `Union`(x, y): Merge the set containing x and the set containing y.
- `FindSet`(x): Returns a unique identifier for the set containing x.

42

$\{v_1, v_2, v_4\}$

$\{v_3, v_5, v_6\}$

`FindSet`(v_5); Returns .

Note that `Union` may affect set identifiers unpredictably!

`MakeUnionFind` takes $O(|X|)$ time, and `Union` and `FindSet` take $O(\log |X|)$ time. (It is also possible to add elements dynamically, but we won't need to.) So if we use this for $\mathcal{C}$...

Implementing Kruskal's algorithm: Third time lucky!

Algorithm: KRUSKAL

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

Sort the edges by weight as $e_1, \dots, e_m$, with $w(e_1) \leq \cdots \leq w(e_m)$.
Let $T \leftarrow (V, \emptyset)$ be the empty tree on V.
Let $\mathcal{C} = \texttt{MakeUnionFind}(V)$.
for $i = 1$ *to* m **do**
    Write $e_i \to \{u_i, v_i\}$.
    if $\mathcal{C}.\texttt{FindSet}(u_i) \neq \mathcal{C}.\texttt{FindSet}(v_i)$ **then**
        Let $T \leftarrow T + e_i$.
        Call $\mathcal{C}.\texttt{Union}(u_i, v_i)$.
Return T.

Implementing Kruskal's algorithm: Third time lucky!

Algorithm: KRUSKAL

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

1 Sort the edges by weight as $e_1, \dots, e_m$, with $w(e_1) \leq \cdots \leq w(e_m)$.
2 Let $T \leftarrow (V, \emptyset)$ be the empty tree on V.
3 Let $\mathcal{C} = \texttt{MakeUnionFind}(V)$.
4 **for** $i = 1$ *to* m **do**
5   Write $e_i \rightarrow \{u_i, v_i\}$.
6   **if** $\mathcal{C}.\texttt{FindSet}(u_i) \neq \mathcal{C}.\texttt{FindSet}(v_i)$ **then**
7   Let $T \leftarrow T + e_i$.
8   Call $\mathcal{C}.\texttt{Union}(u_i, v_i)$.
9 Return T.

Now line 3 takes $O(|V|)$ time, and each iteration of lines 6 and 8 takes $O(\log|V|)$ time.

Implementing Kruskal's algorithm: Third time lucky!

Algorithm: KRUSKAL

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

1 Sort the edges by weight as $e_1, \dots, e_m$, with $w(e_1) \leq \cdots \leq w(e_m)$.
2 Let $T \leftarrow (V, \emptyset)$ be the empty tree on V.
3 Let $\mathcal{C} = \texttt{MakeUnionFind}(V)$.
4 **for** $i = 1$ *to* m **do**
5   Write $e_i \rightarrow \{u_i, v_i\}$.
6   **if** $\mathcal{C}.\texttt{FindSet}(u_i) \neq \mathcal{C}.\texttt{FindSet}(v_i)$ **then**
7   Let $T \leftarrow T + e_i$.
8   Call $\mathcal{C}.\texttt{Union}(u_i, v_i)$.
9 Return T.

Now line 3 takes $O(|V|)$ time, and each iteration of lines 6 and 8 takes $O(\log |V|)$ time.

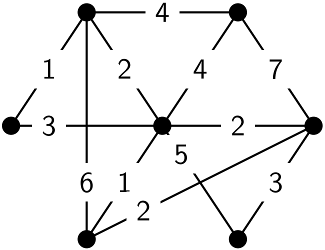
So overall, since G is connected and $|E| \geq |V| - 1$, the running time is $O(|E| \log |V|)$ — exactly what we got from Prim's algorithm!

Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.



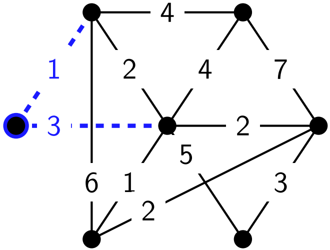
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



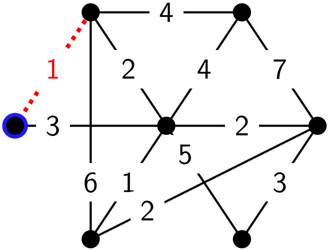
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



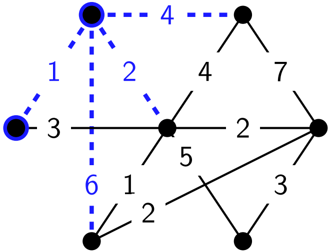
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



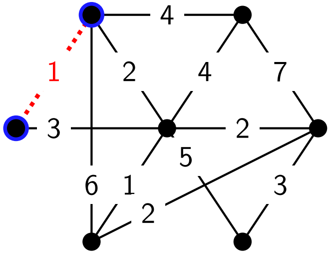
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



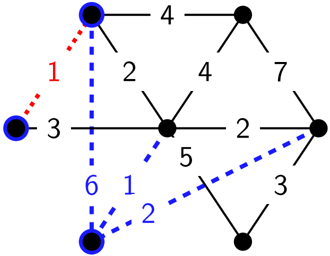
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



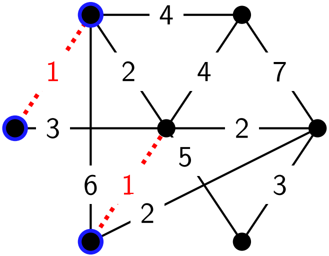
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



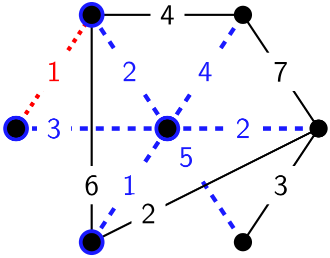
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



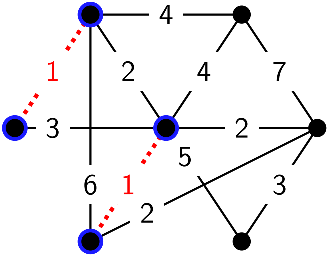
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



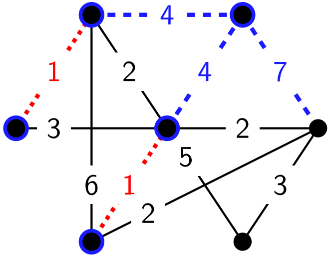
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



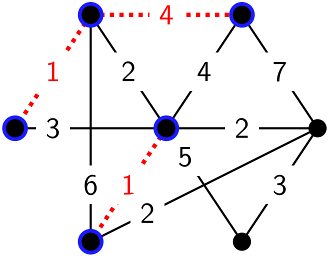
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



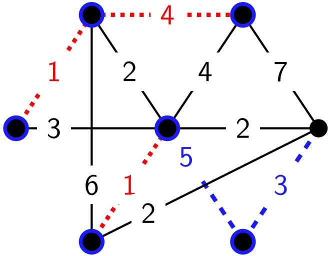
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



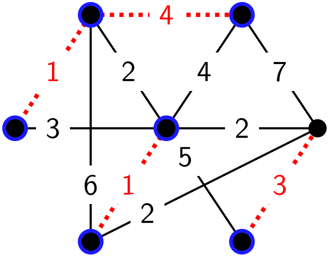
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



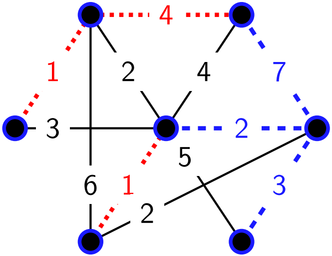
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1



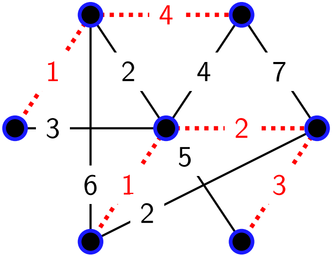
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 1

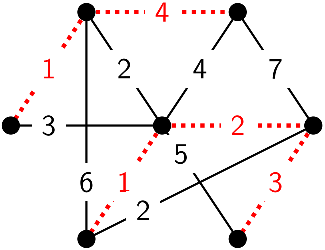


Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 2



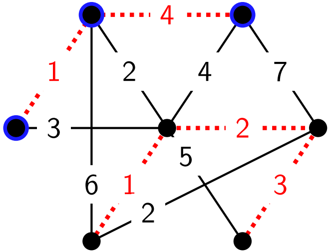
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 2



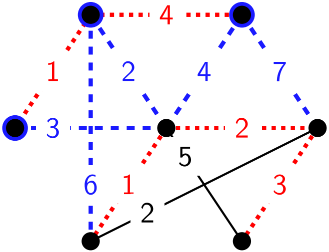
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 2



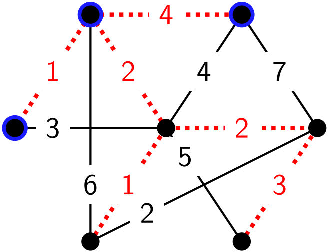
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 2



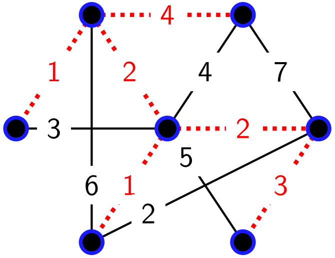
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 2



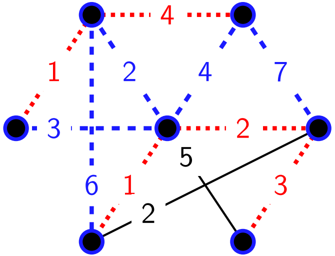
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 2



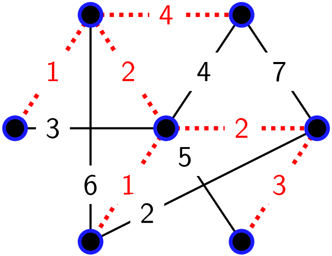
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Step 2



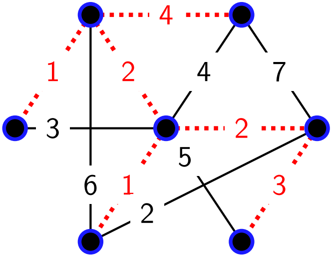
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Done!



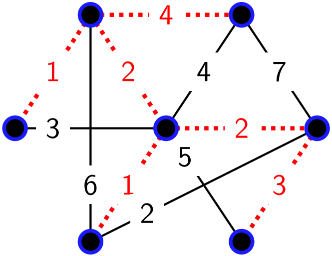
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Done!



Most modern algorithms for minimum spanning tree are variants of Borůvka's algorithm...

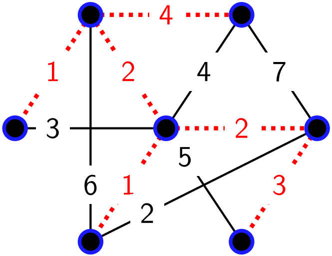
Non-examinable: Borůvka's algorithm

Neither Kruskal's algorithm and Prim's algorithm parallelise effectively.

But Borůvka's original algorithm, from 40 years earlier, works nicely.

At each step, it **simultaneously** finds and adds the cheapest edge out of **each component** of the output tree T.

Done!



Most modern algorithms for minimum spanning tree are variants of Borůvka's algorithm...and they use a union-find data structure to keep track of the components! So it is useful, after all.