5.2 Minimale Spannbäume

$G = (V, E)$ sei **ungerichteter zusammenhängender Graph**
mit **positiven Kantengewichten** $w : E \to \mathbb{R}_{>0}$.

5.2 Minimale Spannbäume

$G = (V, E)$ sei **ungerichteter zusammenhängender Graph** mit **positiven Kantengewichten** $w : E \to \mathbb{R}_{>0}$.

Für $E' \subseteq E$ sei

$$w(E') = \sum_{e \in E'} w(e).$$

$G = (V, E)$ sei **ungerichteter zusammenhängender Graph** mit **positiven Kantengewichten** $w : E \to \mathbb{R}_{>0}$.

Für $E' \subseteq E$ sei

$$w(E') = \sum_{e \in E'} w(e).$$

Definition 5.12

Eine Kantenmenge $T \subseteq E$ heißt **Spannbaum** von G, wenn der Graph (V, T) ein Baum ist. Ein Spannbaum $T \subseteq E$ heißt **minimaler Spannbaum** von G, wenn es keinen Spannbaum von G mit einem kleineren Gewicht als $w(T)$ gibt.

5.2.1 Union-Find-Datenstrukturen

Union-Find-Datenstruktur

Speichere **disjunkte Mengen** $S_1, \ldots, S_k$

und für jede Menge S_i **Repräsentanten** $s_i \in S_i$.

5.2.1 Union-Find-Datenstrukturen

Union-Find-Datenstruktur

Speichere **disjunkte Mengen** $S_1, \ldots, S_k$

und für jede Menge S_i **Repräsentanten** $s_i \in S_i$.

- MAKE-SET(x): Erzeugt eine neue Menge $\{x\}$ mit Repräsentant x. Dabei darf x nicht bereits in einer anderen Menge enthalten sein.

5.2.1 Union-Find-Datenstrukturen

Union-Find-Datenstruktur

Speichere **disjunkte Mengen** $S_1, \ldots, S_k$
und für jede Menge S_i **Repräsentanten** $s_i \in S_i$.

- MAKE-SET(x): Erzeugt eine neue Menge $\{x\}$ mit Repräsentant x. Dabei darf x nicht bereits in einer anderen Menge enthalten sein.
- UNION(x, y): Falls zwei Mengen S_x und S_y mit $x \in S_x$ und $y \in S_y$ existieren, so werden diese entfernt und durch die Menge $S_x \cup S_y$ ersetzt. Der neue Repräsentant von $S_x \cup S_y$ kann ein beliebiges Element dieser vereinigten Menge sein.

5.2.1 Union-Find-Datenstrukturen

Union-Find-Datenstruktur

Speichere **disjunkte Mengen** $S_1, \ldots, S_k$
und für jede Menge S_i **Repräsentanten** $s_i \in S_i$.

- MAKE-SET(x): Erzeugt eine neue Menge $\{x\}$ mit Repräsentant x. Dabei darf x nicht bereits in einer anderen Menge enthalten sein.
- UNION(x, y): Falls zwei Mengen S_x und S_y mit $x \in S_x$ und $y \in S_y$ existieren, so werden diese entfernt und durch die Menge $S_x \cup S_y$ ersetzt. Der neue Repräsentant von $S_x \cup S_y$ kann ein beliebiges Element dieser vereinigten Menge sein.
- FIND(x): Liefert den Repräsentanten der Menge S mit $x \in S$ zurück.

5.2.1 Union-Find-Datenstrukturen

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1 : \{1\}$

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1:\{1\}$
MAKE-SET(2), MAKE-SET(3)	$1:\{1\}, 2:\{2\}, 3:\{3\}$

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1:\{1\}$
MAKE-SET(2), MAKE-SET(3)	$1:\{1\}, 2:\{2\}, 3:\{3\}$
MAKE-SET(4), MAKE-SET(5)	$1:\{1\}, 2:\{2\}, 3:\{3\}, 4:\{4\}, 5:\{5\}$

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1:\{1\}$
MAKE-SET(2), MAKE-SET(3)	$1:\{1\}$, $2:\{2\}$, $3:\{3\}$
MAKE-SET(4), MAKE-SET(5)	$1:\{1\}$, $2:\{2\}$, $3:\{3\}$, $4:\{4\}$, $5:\{5\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1:\{1\}$
MAKE-SET(2), MAKE-SET(3)	$1:\{1\}$, $2:\{2\}$, $3:\{3\}$
MAKE-SET(4), MAKE-SET(5)	$1:\{1\}$, $2:\{2\}$, $3:\{3\}$, $4:\{4\}$, $5:\{5\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(1,2)	$2:\{1,2\}$, $3:\{3\}$, $4:\{4\}$, $5:\{5\}$

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1:\{1\}$
MAKE-SET(2), MAKE-SET(3)	$1:\{1\}$, $2:\{2\}$, $3:\{3\}$
MAKE-SET(4), MAKE-SET(5)	$1:\{1\}$, $2:\{2\}$, $3:\{3\}$, $4:\{4\}$, $5:\{5\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(1,2)	$2:\{1,2\}$, $3:\{3\}$, $4:\{4\}$, $5:\{5\}$
UNION(3,4)	$2:\{1,2\}$, $3:\{3,4\}$, $5:\{5\}$

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1:\{1\}$
MAKE-SET(2), MAKE-SET(3)	$1:\{1\}$, $2:\{2\}$, $3:\{3\}$
MAKE-SET(4), MAKE-SET(5)	$1:\{1\}$, $2:\{2\}$, $3:\{3\}$, $4:\{4\}$, $5:\{5\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(1,2)	$2:\{1,2\}$, $3:\{3\}$, $4:\{4\}$, $5:\{5\}$
UNION(3,4)	$2:\{1,2\}$, $3:\{3,4\}$, $5:\{5\}$
FIND(1)	Ausgabe: 2, keine Zustandsänderung

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1:\{1\}$
MAKE-SET(2), MAKE-SET(3)	$1:\{1\}, 2:\{2\}, 3:\{3\}$
MAKE-SET(4), MAKE-SET(5)	$1:\{1\}, 2:\{2\}, 3:\{3\}, 4:\{4\}, 5:\{5\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(1,2)	$2:\{1,2\}, 3:\{3\}, 4:\{4\}, 5:\{5\}$
UNION(3,4)	$2:\{1,2\}, 3:\{3,4\}, 5:\{5\}$
FIND(1)	Ausgabe: 2, keine Zustandsänderung
UNION(1,5)	$2:\{1,2,5\}, 3:\{3,4\}$

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1:\{1\}$
MAKE-SET(2), MAKE-SET(3)	$1:\{1\}, 2:\{2\}, 3:\{3\}$
MAKE-SET(4), MAKE-SET(5)	$1:\{1\}, 2:\{2\}, 3:\{3\}, 4:\{4\}, 5:\{5\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(1,2)	$2:\{1,2\}, 3:\{3\}, 4:\{4\}, 5:\{5\}$
UNION(3,4)	$2:\{1,2\}, 3:\{3,4\}, 5:\{5\}$
FIND(1)	Ausgabe: 2, keine Zustandsänderung
UNION(1,5)	$2:\{1,2,5\}, 3:\{3,4\}$
FIND(5)	Ausgabe: 2, keine Zustandsänderung

Beispiel:

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1:\{1\}$
MAKE-SET(2), MAKE-SET(3)	$1:\{1\}, 2:\{2\}, 3:\{3\}$
MAKE-SET(4), MAKE-SET(5)	$1:\{1\}, 2:\{2\}, 3:\{3\}, 4:\{4\}, 5:\{5\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(1,2)	$2:\{1,2\}, 3:\{3\}, 4:\{4\}, 5:\{5\}$
UNION(3,4)	$2:\{1,2\}, 3:\{3,4\}, 5:\{5\}$
FIND(1)	Ausgabe: 2, keine Zustandsänderung
UNION(1,5)	$2:\{1,2,5\}, 3:\{3,4\}$
FIND(5)	Ausgabe: 2, keine Zustandsänderung
UNION(5,4)	$3:\{1,2,3,4,5\}$

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	8	4	1	1	4	8	4

$$\{1, 5, 6\}, \{2, 4, 7, 9\}, \{3, 8\}$$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	8	4	1	1	4	8	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	8	4	1	1	4	8	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	8	4	1	1	4	8	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	4	4	1	1	4	8	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	4	4	1	1	4	8	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	4	4	1	1	4	8	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	4	4	1	1	4	8	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	4	4	1	1	4	8	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	4	4	1	1	4	8	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	4	4	1	1	4	4	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	4	4	1	1	4	4	4

UNION$(9, 8)$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	4	4	1	1	4	4	4

$\{1, 5, 6\}, \{2, 3, 4, 7, 8, 9\}$

5.2.1 Union-Find-Datenstrukturen

Implementierung als Feld

- **Annahme:** MAKE-SET wird für $1, \ldots, n$ jeweils einmal aufgerufen.

  n als Parameter bei Initialisierung gegeben.
- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.
- FIND(x): Gib $A[x]$ aus.
- UNION(x, y): Durchlaufe Feld A. Für i mit $A[i] = A[y]$ setze $A[i] = A[x]$.

Beispiel:

1	2	3	4	5	6	7	8	9
1	4	4	4	1	1	4	4	4

$\{1, 5, 6\}, \{2, 3, 4, 7, 8, 9\}$

Laufzeit von UNION: $\Theta(n)$.

Implementierung als Feld mit verketteten Listen

Idee: Zusätzliche Listen $L[1, \ldots, n]$, wobei $L[i]$ die Elemente enthält, die i repräsentiert.

Implementierung als Feld mit verketteten Listen

Idee: Zusätzliche Listen $L[1, \ldots, n]$, wobei $L[i]$ die Elemente enthält, die i repräsentiert.

- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.

  Erzeuge Feld $L[1, \ldots, n]$ mit $L[i] \to i \to \mathrm{null}$.

  Erzeuge Feld $\mathrm{size}[1, \ldots, n]$ mit $\mathrm{size}[i] = 1$.

Implementierung als Feld mit verketteten Listen

Idee: Zusätzliche Listen $L[1, \ldots, n]$, wobei $L[i]$ die Elemente enthält, die i repräsentiert.

- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.

  Erzeuge Feld $L[1, \ldots, n]$ mit $L[i] \to i \to \mathrm{null}$.

  Erzeuge Feld $\mathrm{size}[1, \ldots, n]$ mit $\mathrm{size}[i] = 1$.
- FIND(x): Gib $A[x]$ aus.

Implementierung als Feld mit verketteten Listen

Idee: Zusätzliche Listen $L[1, \ldots, n]$, wobei $L[i]$ die Elemente enthält, die i repräsentiert.

- INIT(n): Erzeuge Feld $A[1, \ldots, n]$ mit $A[i] = i$.

  Erzeuge Feld $L[1, \ldots, n]$ mit $L[i] \to i \to \mathrm{null}$.

  Erzeuge Feld $\mathrm{size}[1, \ldots, n]$ mit $\mathrm{size}[i] = 1$.
- FIND(x): Gib $A[x]$ aus.

5.2.1 Union-Find-Datenstrukturen

UNION(x, y)

1 $i = \text{FIND}(x); j = \text{FIND}(y);$

2 **if** $(\text{size}[i] > \text{size}[j])$ Vertausche i und j.

3 **for each** $(z \in L[i])$ $\{ A[z] = j; \}$

4 Hänge $L[i]$ an $L[j]$ an.

5 $\text{size}[j] = \text{size}[j] + \text{size}[i];$

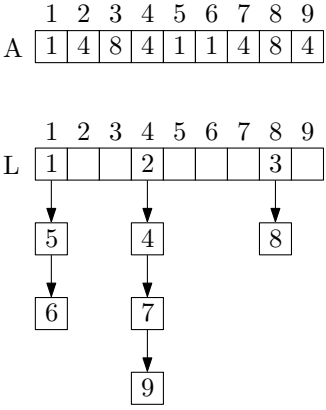
6 $L[i] =$ **null**;

7 $\text{size}[i] = 0;$

5.2.1 Union-Find-Datenstrukturen

UNION(x, y)

1 $i = \text{FIND}(x); j = \text{FIND}(y);$
2 **if** $(\text{size}[i] > \text{size}[j])$ Vertausche i und j.
3 **for each** $(z \in L[i])$ $\{ A[z] = j; \}$
4 Hänge $L[i]$ an $L[j]$ an.
5 $\text{size}[j] = \text{size}[j] + \text{size}[i];$
6 $L[i] =$ **null**;
7 $\text{size}[i] = 0;$

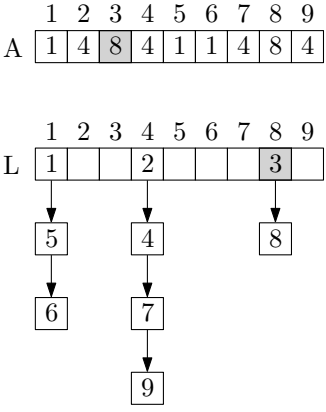


Beispiel: UNION$(8, 9)$

$i = \text{FIND}(8) = 8 \quad j = \text{FIND}(9) = 4$

5.2.1 Union-Find-Datenstrukturen

```
UNION(x, y)
1   i = FIND(x); j = FIND(y);
2   if (size[i] > size[j]) Vertausche i und j.
3   for each (z ∈ L[i]) { A[z] = j; }
4   Hänge L[i] an L[j] an.
5   size[j] = size[j] + size[i];
6   L[i] = null;
7   size[i] = 0;
```



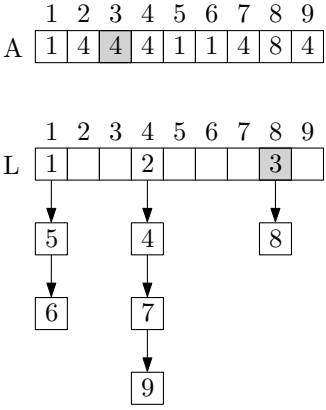
Beispiel: UNION$(8, 9)$

$i =$FIND$(8) = 8 \quad j =$FIND$(9) = 4$

5.2.1 Union-Find-Datenstrukturen

UNION(x, y)

1 $i = \text{FIND}(x); j = \text{FIND}(y);$
2 **if** $(\text{size}[i] > \text{size}[j])$ Vertausche i und j.
3 **for each** $(z \in L[i])$ $\{ A[z] = j; \}$
4 Hänge $L[i]$ an $L[j]$ an.
5 $\text{size}[j] = \text{size}[j] + \text{size}[i];$
6 $L[i] =$ **null**;
7 $\text{size}[i] = 0;$

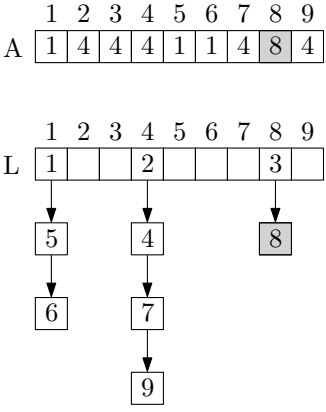


Beispiel: UNION$(8, 9)$

$i = \text{FIND}(8) = 8 \quad j = \text{FIND}(9) = 4$

5.2.1 Union-Find-Datenstrukturen

```
UNION(x, y)
1   i = FIND(x); j = FIND(y);
2   if (size[i] > size[j]) Vertausche i und j.
3   for each (z ∈ L[i]) { A[z] = j; }
4   Hänge L[i] an L[j] an.
5   size[j] = size[j] + size[i];
6   L[i] = null;
7   size[i] = 0;
```

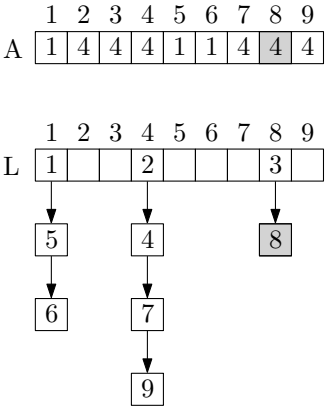


Beispiel: UNION$(8, 9)$

$i =$FIND$(8) = 8 \quad j =$FIND$(9) = 4$

5.2.1 Union-Find-Datenstrukturen

```
UNION(x, y)
1   i = FIND(x); j = FIND(y);
2   if (size[i] > size[j]) Vertausche i und j.
3   for each (z ∈ L[i]) { A[z] = j; }
4   Hänge L[i] an L[j] an.
5   size[j] = size[j] + size[i];
6   L[i] = null;
7   size[i] = 0;
```



Beispiel: UNION$(8, 9)$

$i =$FIND$(8) = 8 \quad j =$FIND$(9) = 4$

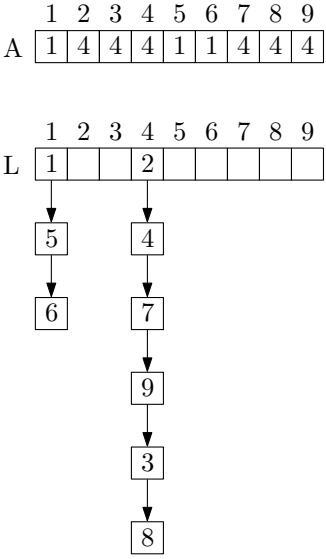
5.2.1 Union-Find-Datenstrukturen

UNION(x, y)

1 $i = \text{FIND}(x); j = \text{FIND}(y);$
2 **if** $(\text{size}[i] > \text{size}[j])$ Vertausche i und j.
3 **for each** $(z \in L[i])$ $\{ A[z] = j; \}$
4 Hänge $L[i]$ an $L[j]$ an.
5 $\text{size}[j] = \text{size}[j] + \text{size}[i];$
6 $L[i] =$ **null**;
7 $\text{size}[i] = 0;$

Beispiel: UNION$(8, 9)$

$i = \text{FIND}(8) = 8 \quad j = \text{FIND}(9) = 4$



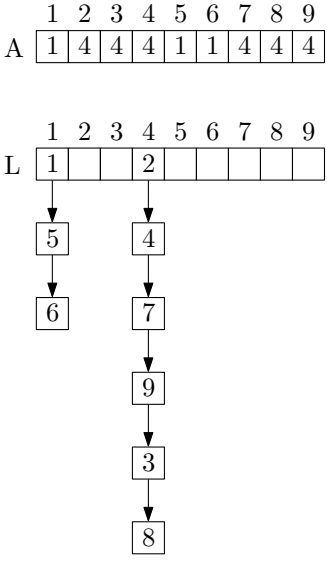
5.2.1 Union-Find-Datenstrukturen

UNION(x, y)
1 $i = \text{FIND}(x); j = \text{FIND}(y);$
2 **if** $(\text{size}[i] > \text{size}[j])$ Vertausche i und j.
3 **for each** $(z \in L[i])$ $\{ A[z] = j; \}$
4 Hänge $L[i]$ an $L[j]$ an.
5 $\text{size}[j] = \text{size}[j] + \text{size}[i];$
6 $L[i] =$ **null**;
7 $\text{size}[i] = 0;$

Beispiel: UNION$(8, 9)$

$i = \text{FIND}(8) = 8 \quad j = \text{FIND}(9) = 4$

Laufzeit: $\Theta(\min\{\text{size}[i], \text{size}[j]\})$



5.2.1 Union-Find-Datenstrukturen

Lemma 5.13

Jede Folge von $(n-1)$ UNION- und f FIND-Operationen kann in Zeit $O(n\log(n)+f)$ durchgeführt werden, wobei n die Anzahl an Elementen bezeichnet.

Lemma 5.13

Jede Folge von $(n-1)$ UNION- und f FIND-Operationen kann in Zeit $O(n \log(n) + f)$ durchgeführt werden, wobei n die Anzahl an Elementen bezeichnet.

Beweis: Gesamtlaufzeit der FIND-Operationen: $O(f)$

Lemma 5.13

Jede Folge von $(n-1)$ UNION- und f FIND-Operationen kann in Zeit $O(n \log(n) + f)$ durchgeführt werden, wobei n die Anzahl an Elementen bezeichnet.

Beweis: Gesamtlaufzeit der FIND-Operationen: $O(f)$

Laufzeit einer UNION-Operation: $\leq c \cdot (\min\{\text{size}[i], \text{size}[j]\})$

Lemma 5.13

Jede Folge von $(n-1)$ UNION- und f FIND-Operationen kann in Zeit $O(n\log(n)+f)$ durchgeführt werden, wobei n die Anzahl an Elementen bezeichnet.

Beweis: Gesamtlaufzeit der FIND-Operationen: $O(f)$

Laufzeit einer UNION-Operation: $\leq c \cdot (\min\{\mathrm{size}[i], \mathrm{size}[j]\})$

Für $i \in \{1, \ldots, n-1\}$ bezeichne **X_i die kleinere Menge** bei der i-ten UNION-Operation.

Lemma 5.13

Jede Folge von $(n-1)$ UNION- und f FIND-Operationen kann in Zeit $O(n \log(n) + f)$ durchgeführt werden, wobei n die Anzahl an Elementen bezeichnet.

Beweis: Gesamtlaufzeit der FIND-Operationen: $O(f)$

Laufzeit einer UNION-Operation: $\leq c \cdot (\min\{\text{size}[i], \text{size}[j]\})$

Für $i \in \{1, \ldots, n-1\}$ bezeichne **X_i die kleinere Menge** bei der i-ten UNION-Operation.

Laufzeit aller UNION-Operationen:

$$\leq c \cdot \sum_{i=1}^{n-1} |X_i| = c \cdot \sum_{i=1}^{n-1} \sum_{x \in X_i} 1 = c \cdot \sum_{j=1}^{n} |\{X_i \mid j \in X_i\}|.$$

Lemma 5.13

Jede Folge von $(n-1)$ UNION- und f FIND-Operationen kann in Zeit $O(n \log(n) + f)$ durchgeführt werden, wobei n die Anzahl an Elementen bezeichnet.

Beweis: Gesamtlaufzeit der FIND-Operationen: $O(f)$
Laufzeit einer UNION-Operation: $\leq c \cdot (\min\{\text{size}[i], \text{size}[j]\})$

Für $i \in \{1, \ldots, n-1\}$ bezeichne **X_i die kleinere Menge** bei der i-ten UNION-Operation.
Laufzeit aller UNION-Operationen:

$$\leq c \cdot \sum_{i=1}^{n-1} |X_i| = c \cdot \sum_{i=1}^{n-1} \sum_{x \in X_i} 1 = c \cdot \sum_{j=1}^{n} |\{X_i \mid j \in X_i\}|.$$

Nachdem ein Element **s mal in der kleineren Menge** einer UNION-Operation lag, liegt es in einer Menge der **Größe mindestens 2^s**.

Lemma 5.13

Jede Folge von $(n-1)$ UNION- und f FIND-Operationen kann in Zeit $O(n \log(n) + f)$ durchgeführt werden, wobei n die Anzahl an Elementen bezeichnet.

Beweis:

Aus $2^s \leq n$ folgt $s \leq \log_2 n$. Also gilt für jedes Element j

$$|\{X_i \mid j \in X_i\}| \leq \log_2 n.$$

Lemma 5.13

Jede Folge von $(n-1)$ UNION- und f FIND-Operationen kann in Zeit $O(n \log(n) + f)$ durchgeführt werden, wobei n die Anzahl an Elementen bezeichnet.

Beweis:

Aus $2^s \leq n$ folgt $s \leq \log_2 n$. Also gilt für jedes Element j

$$|\{X_i \mid j \in X_i\}| \leq \log_2 n.$$

Also beträgt die Gesamtlaufzeit der UNION-Operationen höchstens

$$c \cdot \sum_{j=1}^{n} |\{X_i \mid j \in X_i\}| \leq c \cdot \sum_{j=1}^{n} \log_2 n = O(n \log n). \quad \square$$

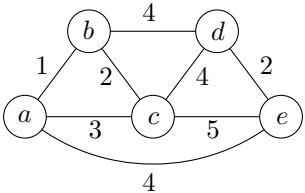
5.2.2 Algorithmus von Kruskal

KRUSKAL(G, w)

1 Teste mittels DFS, ob G zusammenhängend ist. Falls nicht, Abbruch.
2 **for each** $(v \in V)$ MAKE-SET(v);
3 $T = \emptyset$;
4 Sortiere die Kanten in E gemäß ihrem Gewicht.
Danach gelte $E = \{e_1, \ldots, e_m\}$ mit $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$.
Außerdem sei $e_i = (u_i, v_i)$.
5 **for** (**int** $i = 1$; $i \leq m$; i++) $\{$
6 **if** (FIND$(u_i) \neq$ FIND(v_i)) $\{$
7 $T = T \cup \{e_i\}$;
8 UNION(u_i, v_i);
9 $\}$
10 $\}$
11 **return** T

5.2.2 Algorithmus von Kruskal

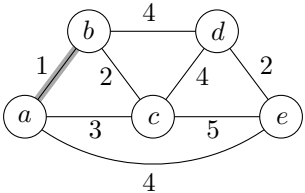
Beispiel:



$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}$

5.2.2 Algorithmus von Kruskal

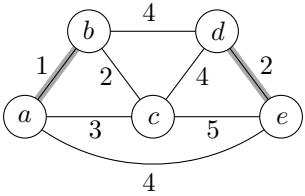
Beispiel:



$\{a, b\}, \{c\}, \{d\}, \{e\}$

5.2.2 Algorithmus von Kruskal

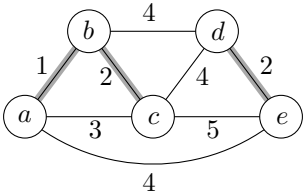
Beispiel:



$\{a, b\}, \{c\}, \{d, e\}$

5.2.2 Algorithmus von Kruskal

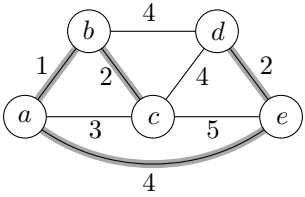
Beispiel:



$\{a, b, c\}, \{d, e\}$

5.2.2 Algorithmus von Kruskal

Beispiel:



$\{a, b, c, d, e\}$

5.2.2 Algorithmus von Kruskal

Theorem 5.14

Für zusammenhängende Graphen berechnet der Algorithmus von Kruskal einen minimalen Spannbaum.

5.2.2 Algorithmus von Kruskal

Theorem 5.14

Für zusammenhängende Graphen berechnet der Algorithmus von Kruskal einen minimalen Spannbaum.

Beweis:

Invariante:

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen am Ende des Schleifenrumpfes der for-Schleife den Zusammenhangskomponenten des Graphen (V, T).

5.2.2 Algorithmus von Kruskal

Theorem 5.14

Für zusammenhängende Graphen berechnet der Algorithmus von Kruskal einen minimalen Spannbaum.

Beweis:

Invariante:

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen am Ende des Schleifenrumpfes der for-Schleife den Zusammenhangskomponenten des Graphen (V, T).
2. Immer wenn die Abbruchbedingung der Schleife in Zeile 5 überprüft wird, gibt es eine Kantenmenge $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum von G ist. Insbesondere ist (V, T) azyklisch.

5.2.2 Algorithmus von Kruskal

Theorem 5.14

Für zusammenhängende Graphen berechnet der Algorithmus von Kruskal einen minimalen Spannbaum.

Beweis:

Invariante:

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen am Ende des Schleifenrumpfes der for-Schleife den Zusammenhangskomponenten des Graphen (V, T).
2. Immer wenn die Abbruchbedingung der Schleife in Zeile 5 überprüft wird, gibt es eine Kantenmenge $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum von G ist. Insbesondere ist (V, T) azyklisch.

Nach dem letzten Schleifendurchlauf gilt $i = m + 1$. Somit muss S leer sein und damit T ein minimaler Spannbaum.

5.2.2 Algorithmus von Kruskal

Invariante:

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen am Ende des Schleifenrumpfes der for-Schleife den Zusammenhangskomponenten des Graphen (V, T).
2. Immer wenn die Abbruchbedingung der Schleife in Zeile 5 überprüft wird, gibt es eine Kantenmenge $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum von G ist. Insbesondere ist (V, T) azyklisch.

Induktionsanfang (vor dem ersten Schleifendurchlauf):

1. Es gilt $T = \emptyset$. Die ZHK von (V, T) sind genau die einelementigen Mengen. Dies ist auch in der Union-Find-Struktur abgebildet.

5.2.2 Algorithmus von Kruskal

Invariante:

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen am Ende des Schleifenrumpfes der for-Schleife den Zusammenhangskomponenten des Graphen (V, T).
2. Immer wenn die Abbruchbedingung der Schleife in Zeile 5 überprüft wird, gibt es eine Kantenmenge $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum von G ist. Insbesondere ist (V, T) azyklisch.

Induktionsanfang (vor dem ersten Schleifendurchlauf):

1. Es gilt $T = \emptyset$. Die ZHK von (V, T) sind genau die einelementigen Mengen. Dies ist auch in der Union-Find-Struktur abgebildet.
2. Es gilt $T = \emptyset$ und $i = 1$. Wir können S als einen beliebigen minimalen Spannbaum von G wählen.

Induktionsschritt, erster Teil:

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen am Ende des Schleifenrumpfes der for-Schleife den Zusammenhangskomponenten des Graphen (V, T).

Invariante sei zu Beginn eines Schleifendurchlaufs erfüllt:

Induktionsschritt, erster Teil:

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen am Ende des Schleifenrumpfes der for-Schleife den Zusammenhangskomponenten des Graphen (V, T).

Invariante sei zu Beginn eines Schleifendurchlaufs erfüllt:

Falls wir Kante e_i nicht in T einfügen, so ändern sich weder die Mengen in der Union-Find-Datenstruktur noch die Zusammenhangskomponenten von (V, T).

Induktionsschritt, erster Teil:

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen am Ende des Schleifenrumpfes der for-Schleife den Zusammenhangskomponenten des Graphen (V, T).

Invariante sei zu Beginn eines Schleifendurchlaufs erfüllt:

Falls wir Kante e_i nicht in T einfügen, so ändern sich weder die Mengen in der Union-Find-Datenstruktur noch die Zusammenhangskomponenten von (V, T).

Fügen wir e_i in T ein, so fallen die Zusammenhangskomponenten von u_i und v_i in (V, T) zu einer gemeinsamen Komponente zusammen. Diese Veränderung bilden wir in der Union-Find-Datenstruktur korrekt ab.

Induktionsschritt, zweiter Teil:

2. Immer wenn die Abbruchbedingung der Schleife in Zeile 5 überprüft wird, gibt es eine Kantenmenge $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum von G ist.

Invariante sei zu Beginn eines Schleifendurchlaufs erfüllt:

Es gibt $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum ist.

Induktionsschritt, zweiter Teil:

2. Immer wenn die Abbruchbedingung der Schleife in Zeile 5 überprüft wird, gibt es eine Kantenmenge $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum von G ist.

Invariante sei zu Beginn eines Schleifendurchlaufs erfüllt:

Es gibt $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum ist.

1. Fall: $T \cup \{e_i\}$ enthält Kreis. e_i wird nicht zu T hinzugefügt.

Es gilt $e_i \notin S$, denn sonst wäre $T \cup S$ kein Baum.

Also gilt $S \subseteq \{e_{i+1}, \ldots, e_m\}$.

Induktionsschritt, zweiter Teil:

2. Immer wenn die Abbruchbedingung der Schleife in Zeile 5 überprüft wird, gibt es eine Kantenmenge $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum von G ist.

Invariante sei zu Beginn eines Schleifendurchlaufs erfüllt:

Es gibt $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum ist.

1. Fall: $T \cup \{e_i\}$ enthält Kreis. e_i wird nicht zu T hinzugefügt.

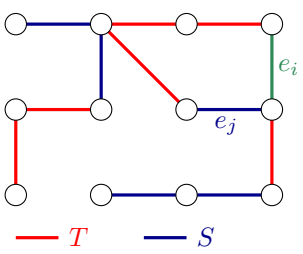
Es gilt $e_i \notin S$, denn sonst wäre $T \cup S$ kein Baum.

Also gilt $S \subseteq \{e_{i+1}, \ldots, e_m\}$.

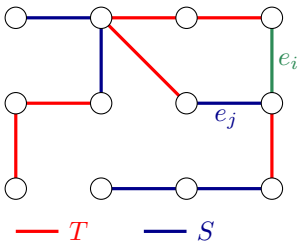
2. Fall: $T \cup \{e_i\}$ kreisfrei. e_i wird zu T hinzugefügt. Gilt $e_i \in S$, so ist nichts zu zeigen. Sei also $e_i \notin S$.

Da $(V, T \cup S)$ zusammenhängend, muss $T \cup S \cup \{e_i\}$ einen Kreis C enthalten. Es muss Kante $e_j \in S \cap C$ geben.

Behauptung $T' := T \cup (S \setminus \{e_j\}) \cup \{e_i\}$ ist MST.



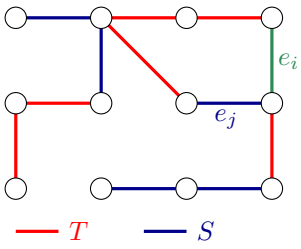
2. Fall: $T \cup \{e_i\}$ kreisfrei. e_i wird zu T hinzugefügt. Gilt $e_i \in S$, so ist nichts zu zeigen. Sei also $e_i \notin S$.
Da $(V, T \cup S)$ zusammenhängend, muss $T \cup S \cup \{e_i\}$ einen Kreis C enthalten. Es muss Kante $e_j \in S \cap C$ geben.
Behauptung $T' := T \cup (S \setminus \{e_j\}) \cup \{e_i\}$ ist MST.



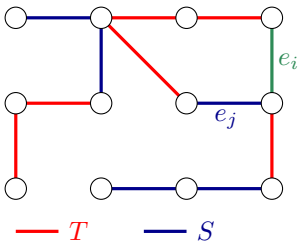
2. Fall: $T \cup \{e_i\}$ kreisfrei. e_i wird zu T hinzugefügt. Gilt $e_i \in S$, so ist nichts zu zeigen. Sei also $e_i \notin S$.
Da $(V, T \cup S)$ zusammenhängend, muss $T \cup S \cup \{e_i\}$ einen Kreis C enthalten. Es muss Kante $e_j \in S \cap C$ geben.
Behauptung $T' := T \cup (S \setminus \{e_j\}) \cup \{e_i\}$ ist MST.

Beobachtungen:

- (V, T') ist zusammenhängend.



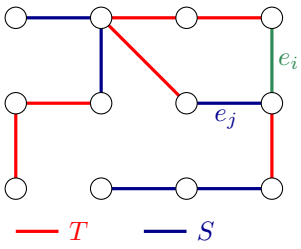
2. Fall: $T \cup \{e_i\}$ kreisfrei. e_i wird zu T hinzugefügt. Gilt $e_i \in S$, so ist nichts zu zeigen. Sei also $e_i \notin S$.
Da $(V, T \cup S)$ zusammenhängend, muss $T \cup S \cup \{e_i\}$ einen Kreis C enthalten. Es muss Kante $e_j \in S \cap C$ geben.
Behauptung $T' := T \cup (S \setminus \{e_j\}) \cup \{e_i\}$ ist MST.



Beobachtungen:

- (V, T') ist zusammenhängend.
- (V, T') ist kreisfrei.

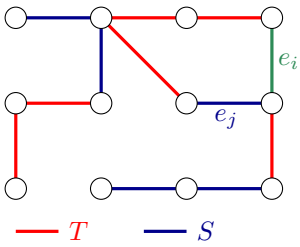
2. Fall: $T \cup \{e_i\}$ kreisfrei. e_i wird zu T hinzugefügt. Gilt $e_i \in S$, so ist nichts zu zeigen. Sei also $e_i \notin S$.
Da $(V, T \cup S)$ zusammenhängend, muss $T \cup S \cup \{e_i\}$ einen Kreis C enthalten. Es muss Kante $e_j \in S \cap C$ geben.
Behauptung $T' := T \cup (S \setminus \{e_j\}) \cup \{e_i\}$ ist MST.



Beobachtungen:

- (V, T') ist zusammenhängend.
- (V, T') ist kreisfrei.
- (V, T') ist also ein Spannbaum.

2. Fall: $T \cup \{e_i\}$ kreisfrei. e_i wird zu T hinzugefügt. Gilt $e_i \in S$, so ist nichts zu zeigen. Sei also $e_i \notin S$.
Da $(V, T \cup S)$ zusammenhängend, muss $T \cup S \cup \{e_i\}$ einen Kreis C enthalten. Es muss Kante $e_j \in S \cap C$ geben.
Behauptung $T' := T \cup (S \setminus \{e_j\}) \cup \{e_i\}$ ist MST.



Beobachtungen:

- (V, T') ist zusammenhängend.
- (V, T') ist kreisfrei.
- (V, T') ist also ein Spannbaum.

Wegen $i < j$ gilt $w(e_i) \leq w(e_j)$ und deshalb

$$w(T') = w(T \cup S) + w(e_i) - w(e_j) \leq w(T \cup S). \quad \square$$

5.2.2 Algorithmus von Kruskal

Theorem 5.15

Für einen ungerichteten zusammenhängenden Graphen $G = (V, E)$ benötigt der Algorithmus von Kruskal eine Laufzeit von $O(|E| \log |E|)$.

5.2.2 Algorithmus von Kruskal

Theorem 5.15

Für einen ungerichteten zusammenhängenden Graphen $G = (V, E)$ benötigt der Algorithmus von Kruskal eine Laufzeit von $O(|E| \log |E|)$.

Beweis:

- Laufzeit von **DFS**: $O(|V| + |E|)$.
- G ist zusammenhängend. Deshalb gilt $|E| \geq |V| - 1$, also $O(|V| + |E|) = O(|E|)$.

5.2.2 Algorithmus von Kruskal

Theorem 5.15

Für einen ungerichteten zusammenhängenden Graphen $G = (V, E)$ benötigt der Algorithmus von Kruskal eine Laufzeit von $O(|E| \log |E|)$.

Beweis:

- Laufzeit von **DFS**: $O(|V| + |E|)$.
- G ist zusammenhängend. Deshalb gilt $|E| \geq |V| - 1$, also $O(|V| + |E|) = O(|E|)$.
- Laufzeit für **Sortieren der Kanten**: $O(|E| \log |E|)$.

5.2.2 Algorithmus von Kruskal

Theorem 5.15

Für einen ungerichteten zusammenhängenden Graphen $G = (V, E)$ benötigt der Algorithmus von Kruskal eine Laufzeit von $O(|E| \log |E|)$.

Beweis:

- Laufzeit von **DFS**: $O(|V| + |E|)$.
- G ist zusammenhängend. Deshalb gilt $|E| \geq |V| - 1$, also $O(|V| + |E|) = O(|E|)$.
- Laufzeit für **Sortieren der Kanten**: $O(|E| \log |E|)$.
- Die **for-Schleife** wird $|E|$ mal durchlaufen. Abgesehen von UNION und FIND beträgt die Gesamtlaufzeit hierfür $O(|E|)$.

5.2.2 Algorithmus von Kruskal

Theorem 5.15

Für einen ungerichteten zusammenhängenden Graphen $G = (V, E)$ benötigt der Algorithmus von Kruskal eine Laufzeit von $O(|E| \log |E|)$.

Beweis:

- Laufzeit von **DFS**: $O(|V| + |E|)$.
- G ist zusammenhängend. Deshalb gilt $|E| \geq |V| - 1$, also $O(|V| + |E|) = O(|E|)$.
- Laufzeit für **Sortieren der Kanten**: $O(|E| \log |E|)$.
- Die **for-Schleife** wird $|E|$ mal durchlaufen. Abgesehen von UNION und FIND beträgt die Gesamtlaufzeit hierfür $O(|E|)$.
- Wir führen in der **Union-Find-Datenstruktur** $2|E|$ FIND-Operationen und $|V| - 1$ UNION-Operationen durch. Mit Lemma 5.13 ergibt sich demnach eine Laufzeit von $O(|V| \log |V| + 2|E|) = O(|E| \log |E|)$. □