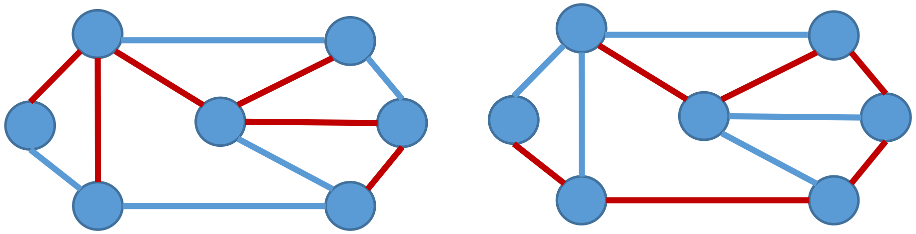
Algorithmen und Datenstrukturen

Weitere Algorithmen auf Graphen:

Minimaler Spannbaum ♦ Vollständige Pfade



Minimaler Spannbaum

Aufspannender Baum

- Sei $G = (V,E)$ ein beliebiger zusammenhängender Graph. Ein **aufspannender Baum** des Graphen G ist ein Teilgraph $T = (V,E')$ mit $E' \subseteq E$, der ein Baum ist.
  - existiert für *jeden* zusammenhängenden Graphen
  - sonst: aufspannender Wald

- Sei $G = (V,E,w)$ ein beliebiger zusammenhängender Graph mit Kantengewichten.
  Ein **aufspannender Baum *T* von** G ist *minimal*, wenn die Summe der Gewichte aller Kanten von T minimal ist.

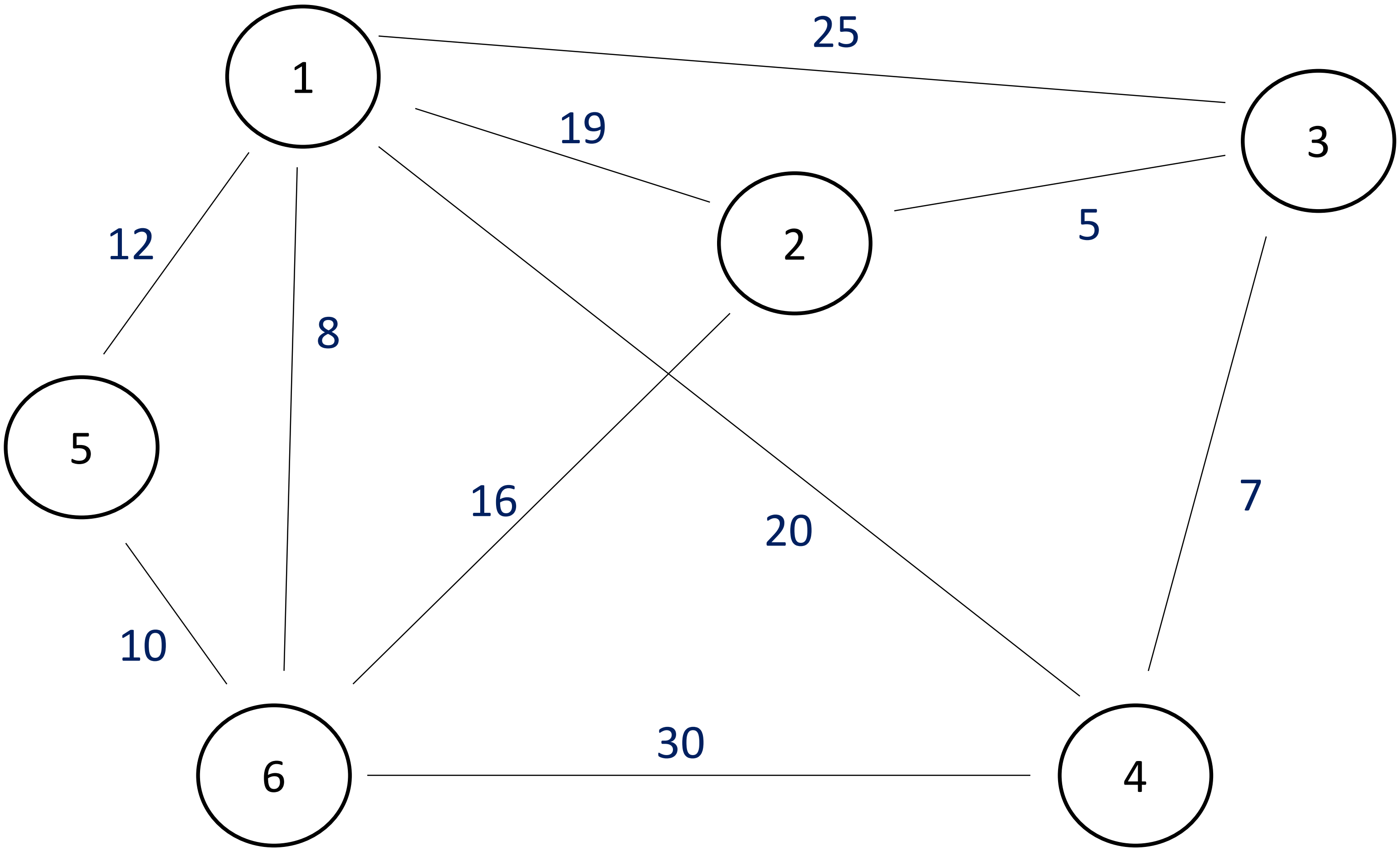
Anwendungsbeispiel

- *Aufgabe:* Minimierung der Erschließungskosten für eine neue Siedlung
- *Prämisse:* günstigste Kosten bei Verlegung der Leitungen entlang der Straßen
- *Kosten* wachsen mit der Straßenlänge
- Es genügt eine Leitung zu jedem Punkt (Kreisfreiheit).

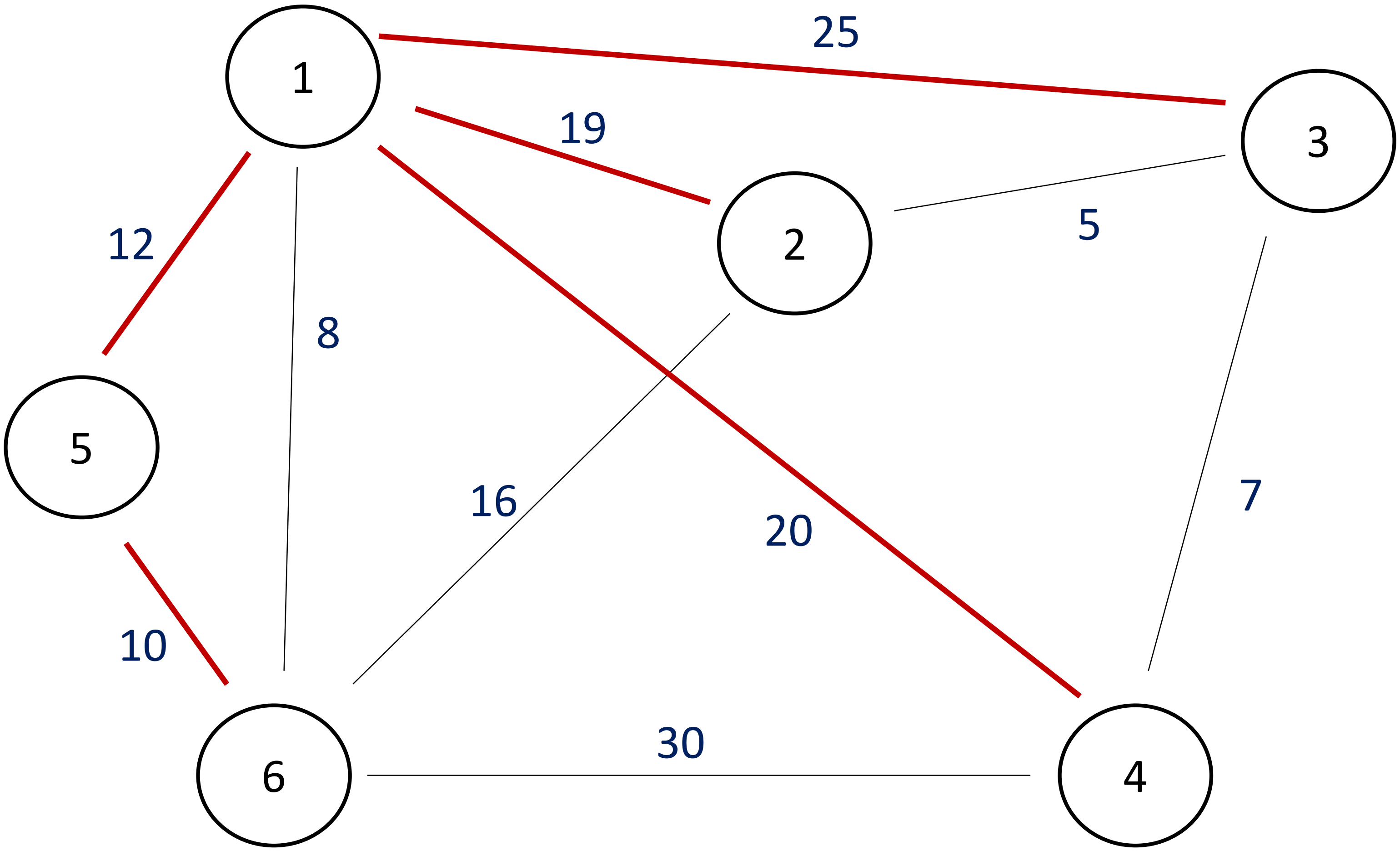
➢ Verlegte Leitungen bilden **Baum**.

➢ Optimum bei **minimal**er Leitungslänge (**Kantengewicht**)

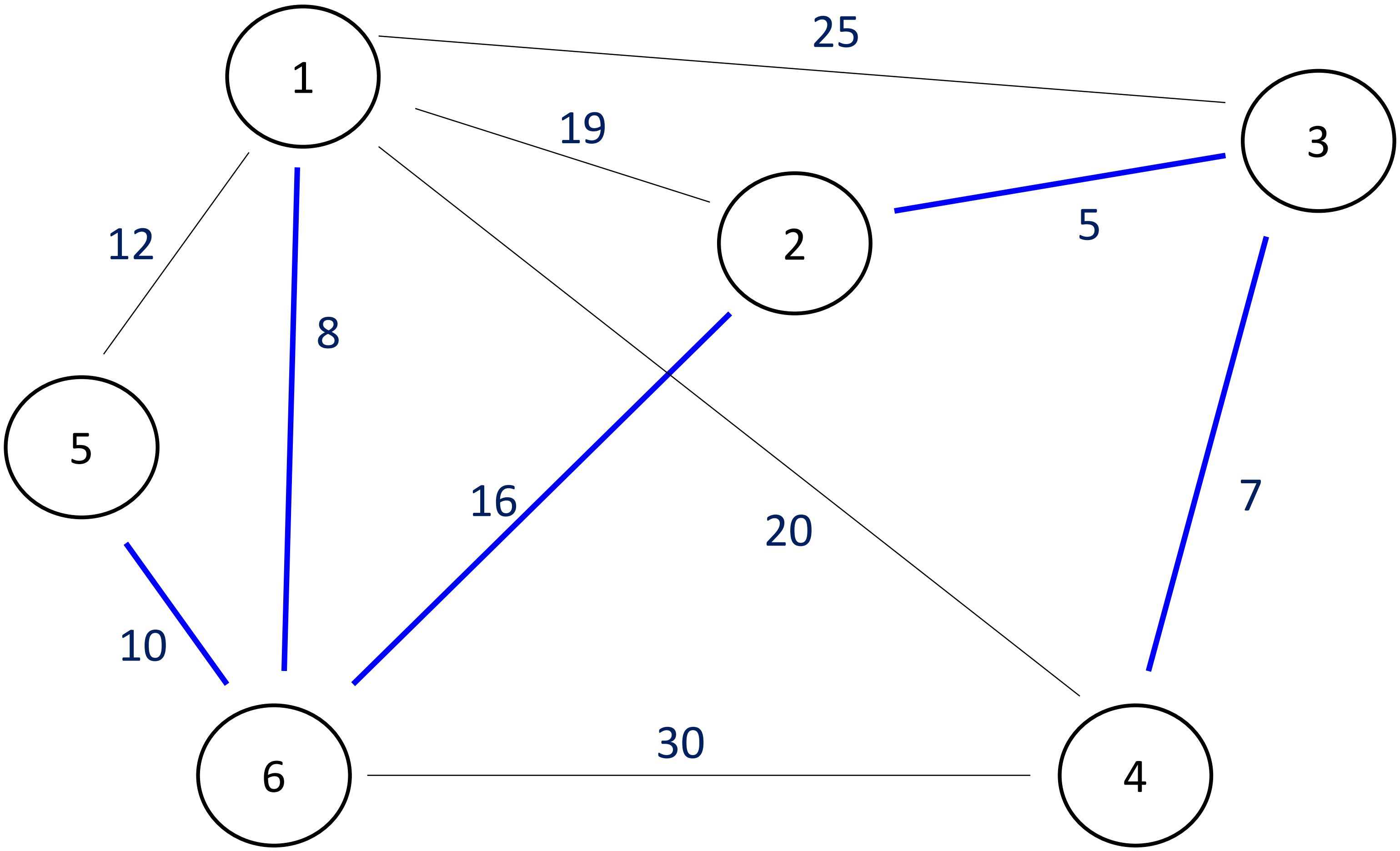
Siedlung als Graph mit Kantengewichten



Aufspannender Baum – Beispiel 1



Aufspannender Baum – Beispiel 2



Algorithmus von Kruskal

Eingabe: ungerichteter zusammenhängender Graph G mit Kantengewichten, $G = (V,E,w)$

Ausgabe: Teilgraph $T = (V,E',w)$ von G, so dass

T ein Baum und $\sum_{e \in E'} w(e)$ minimal ist.

Strategie: Greedy:

Immer Kante mit nächst kleinstem Gewicht hinzufügen, falls Teilgraph Baum bleibt

Kruskal als kanonischer Greedy-Algorithmus

Eingabe: ungerichteter zusammenhängender Graph G
mit Kantengewichten, $G = (V,E,w)$

Ausgabe: minimaler Spannbaum von G

Ordne Kanten zu $e_1, e_2, \ldots, e_n$ so dass $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$

$T \longleftarrow \emptyset$

Für $i = 1$ bis n

Falls $T \cup \{e_i\}$ keinen Kreis bildet

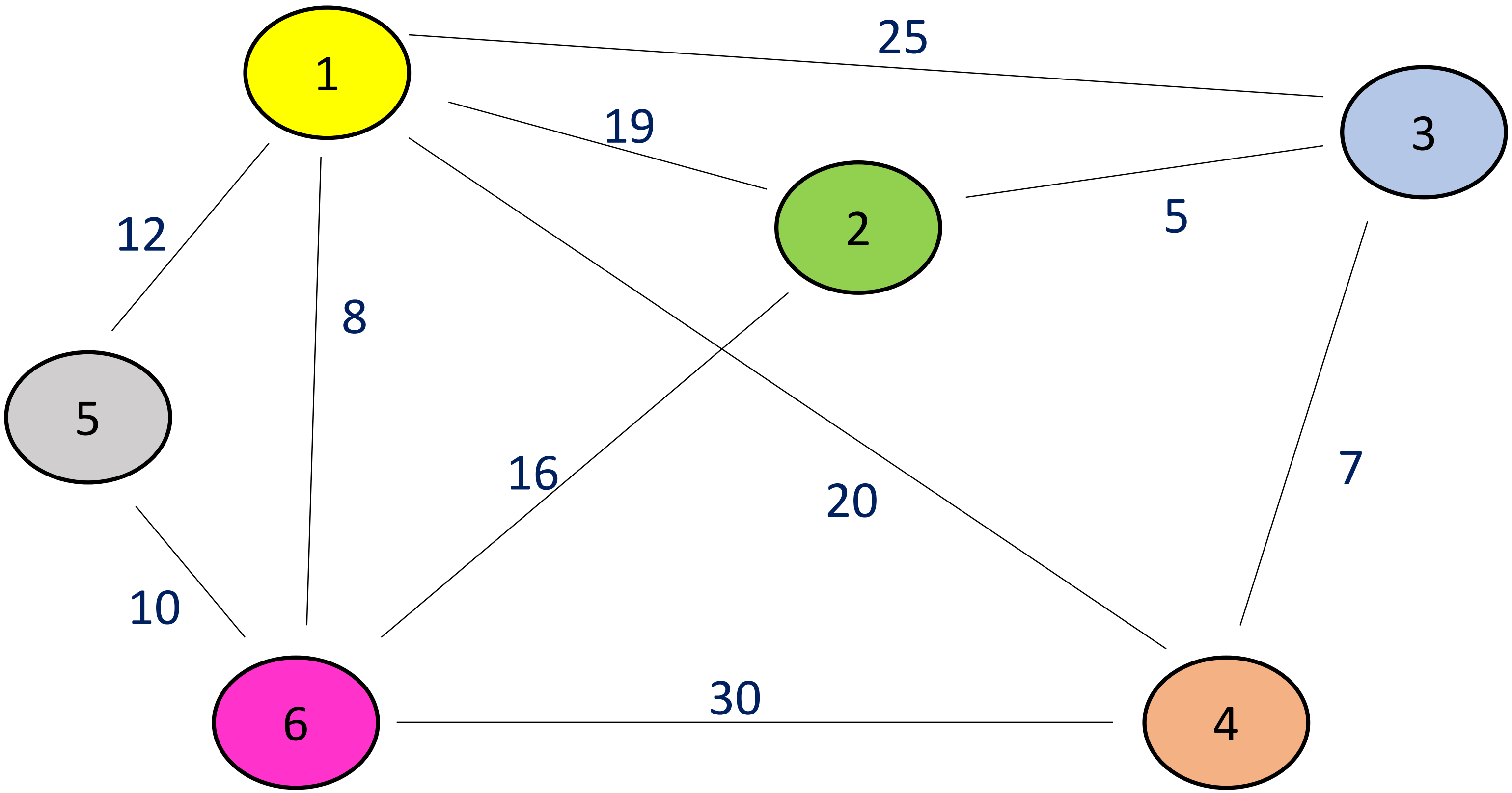
$T \longleftarrow T \cup \{e_i\}$

Kruskal-Algorithmus, Idee

1. Bilde Menge $\mathcal{B}$ von Bäumen (Wald).
   Initial: Menge der Knoten: $\mathcal{B} = \{ (\{v\},\emptyset) \mid v \in V \}$
2. Ordne Kanten aufsteigend nach ihrem Gewicht (→ Queue)
3. Solange $\mathcal{B}$ mehr als einen Baum enthält und die Queue nicht leer ist:
   1. Kante mit kleinstem Gewicht der Queue entnehmen
   2. Falls deren Knoten zu verschiedenen Bäumen in $\mathcal{B}$ gehören, dann Bäume über diese Kante vereinigen

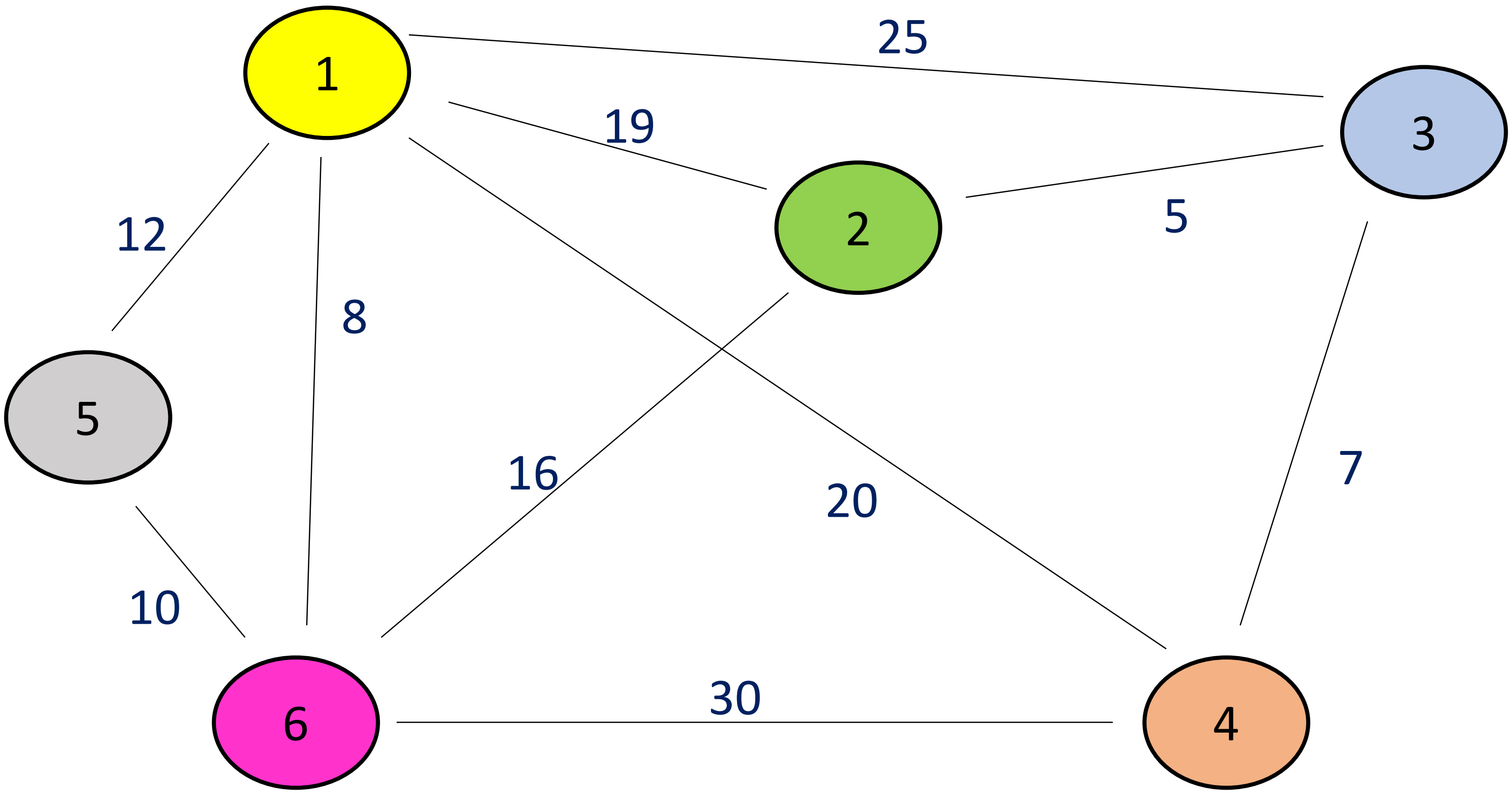
Algorithmus am Beispiel

1. Initialisierung



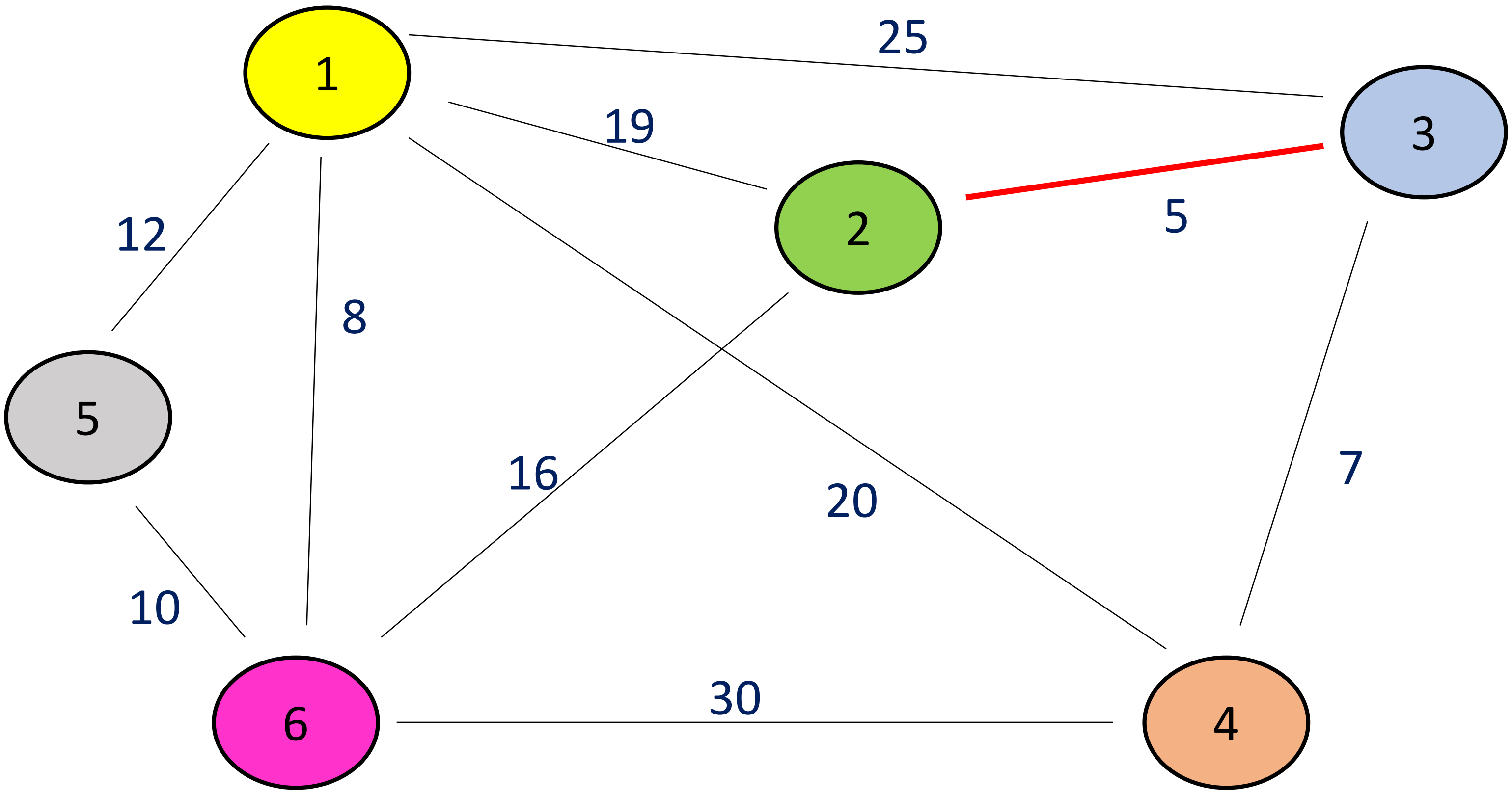
Algorithmus am Beispiel

2. Gewichte, geordnet: 5, 7, 8, 10, 12, 16, 19, 20, 25, 30



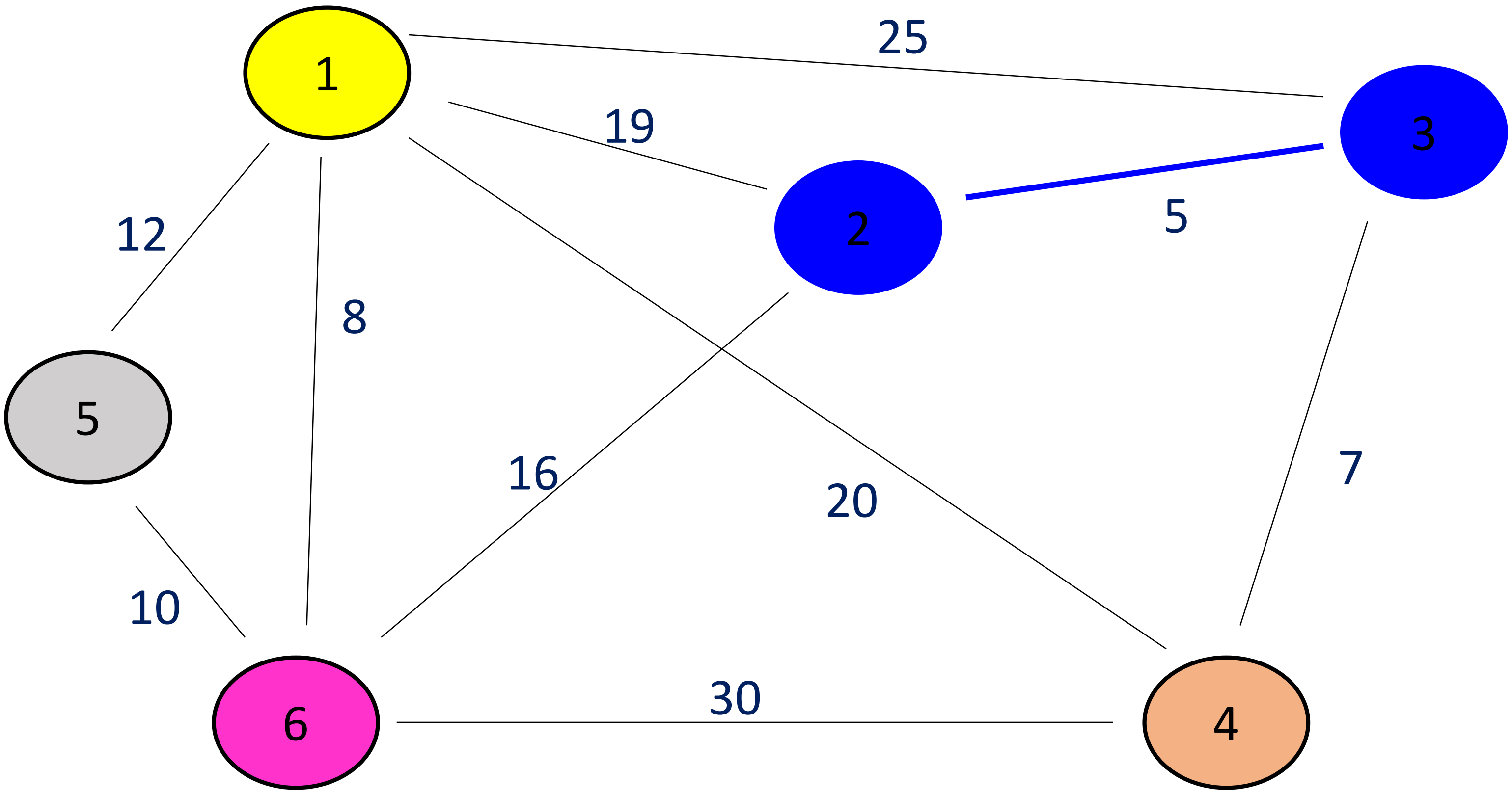
Algorithmus am Beispiel

3. Schleife 5, 7, 8, 10, 12, 16, 19, 20, 25, 30



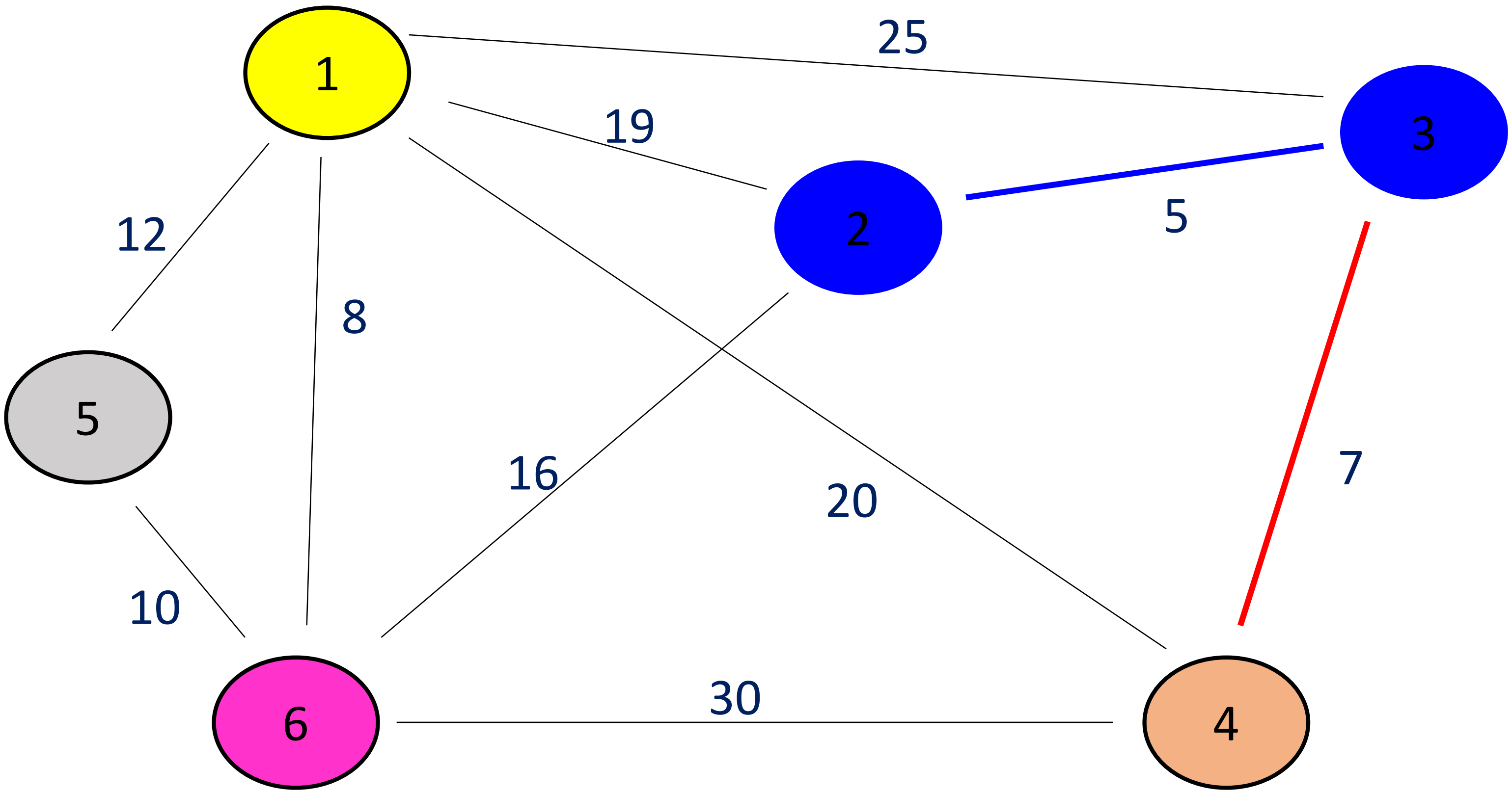
Algorithmus am Beispiel

3. Schleife 7, 8, 10, 12, 16, 19, 20, 25, 30



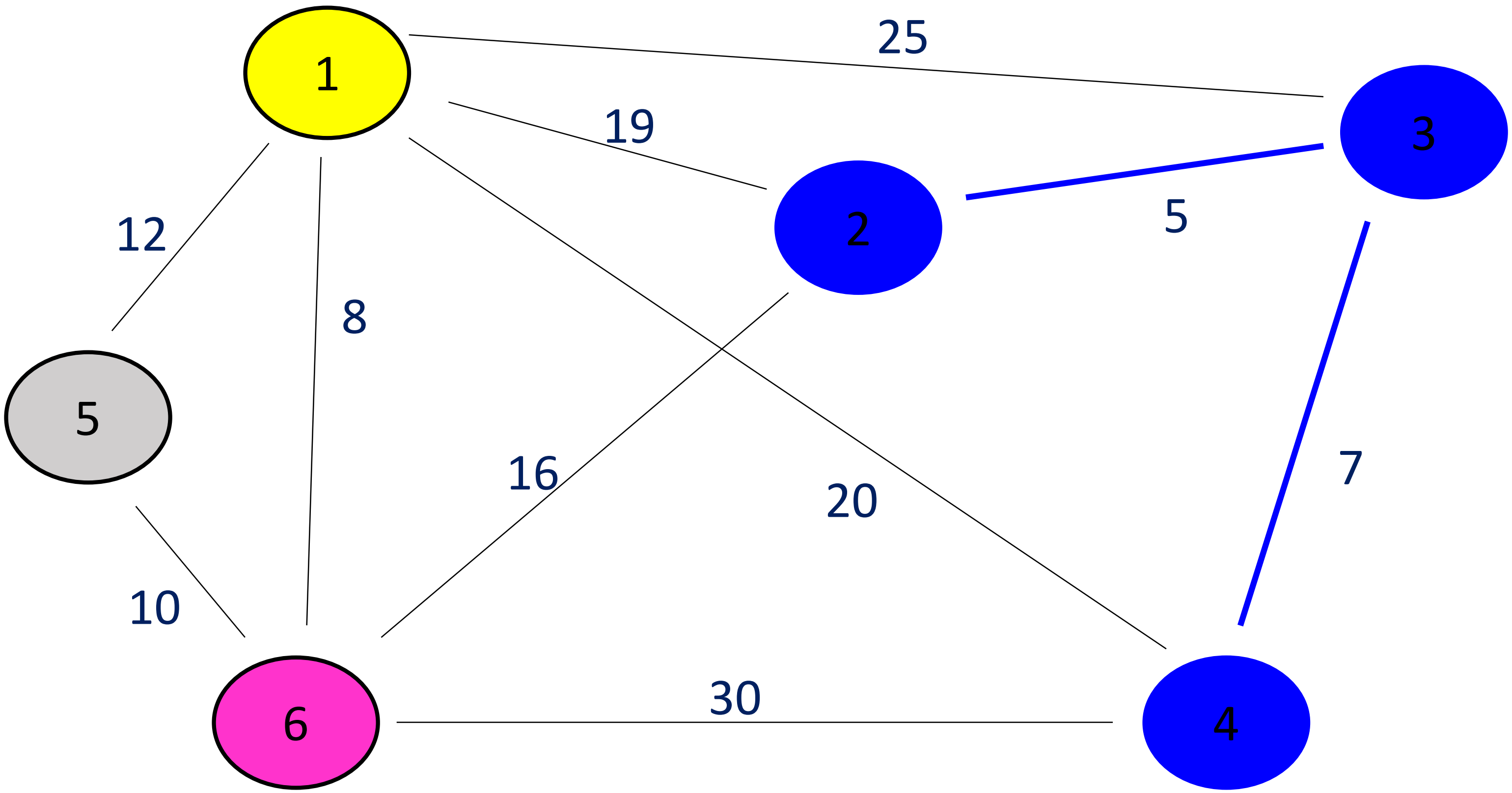
Algorithmus am Beispiel

3. Schleife 7, 8, 10, 12, 16, 19, 20, 25, 30



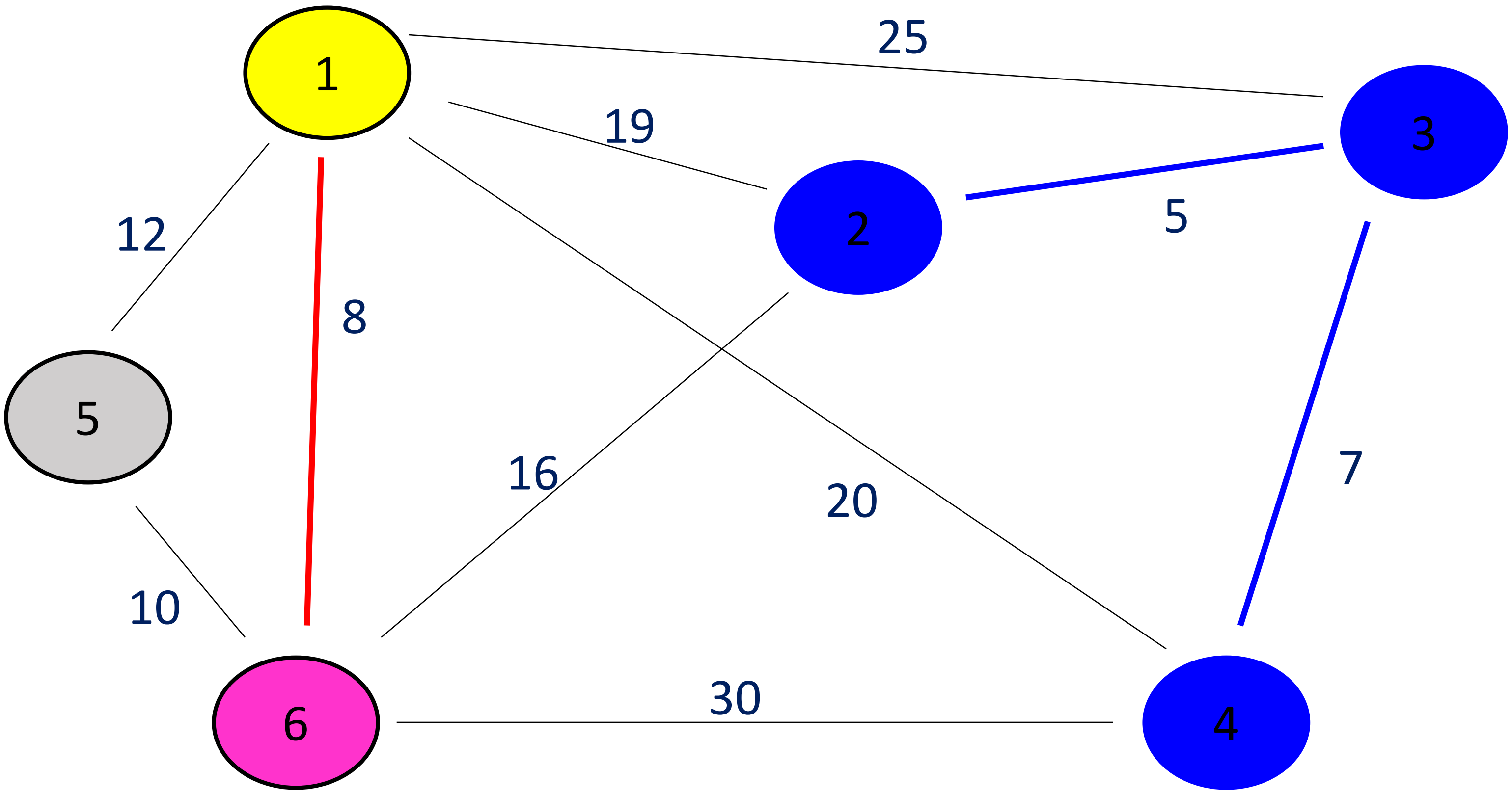
Algorithmus am Beispiel

3. Schleife 8, 10, 12, 16, 19, 20, 25, 30



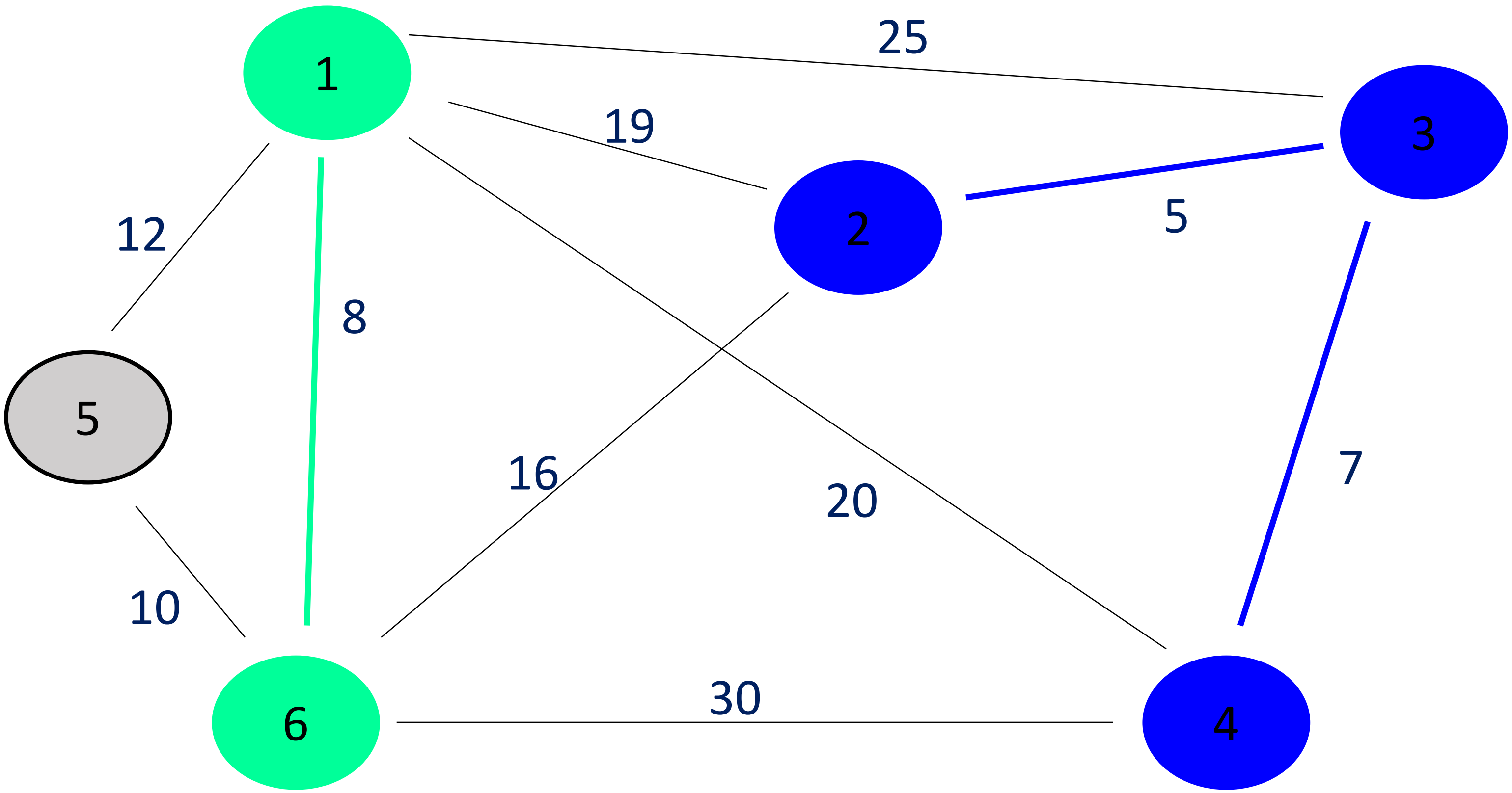
Algorithmus am Beispiel

3. Schleife 8, 10, 12, 16, 19, 20, 25, 30



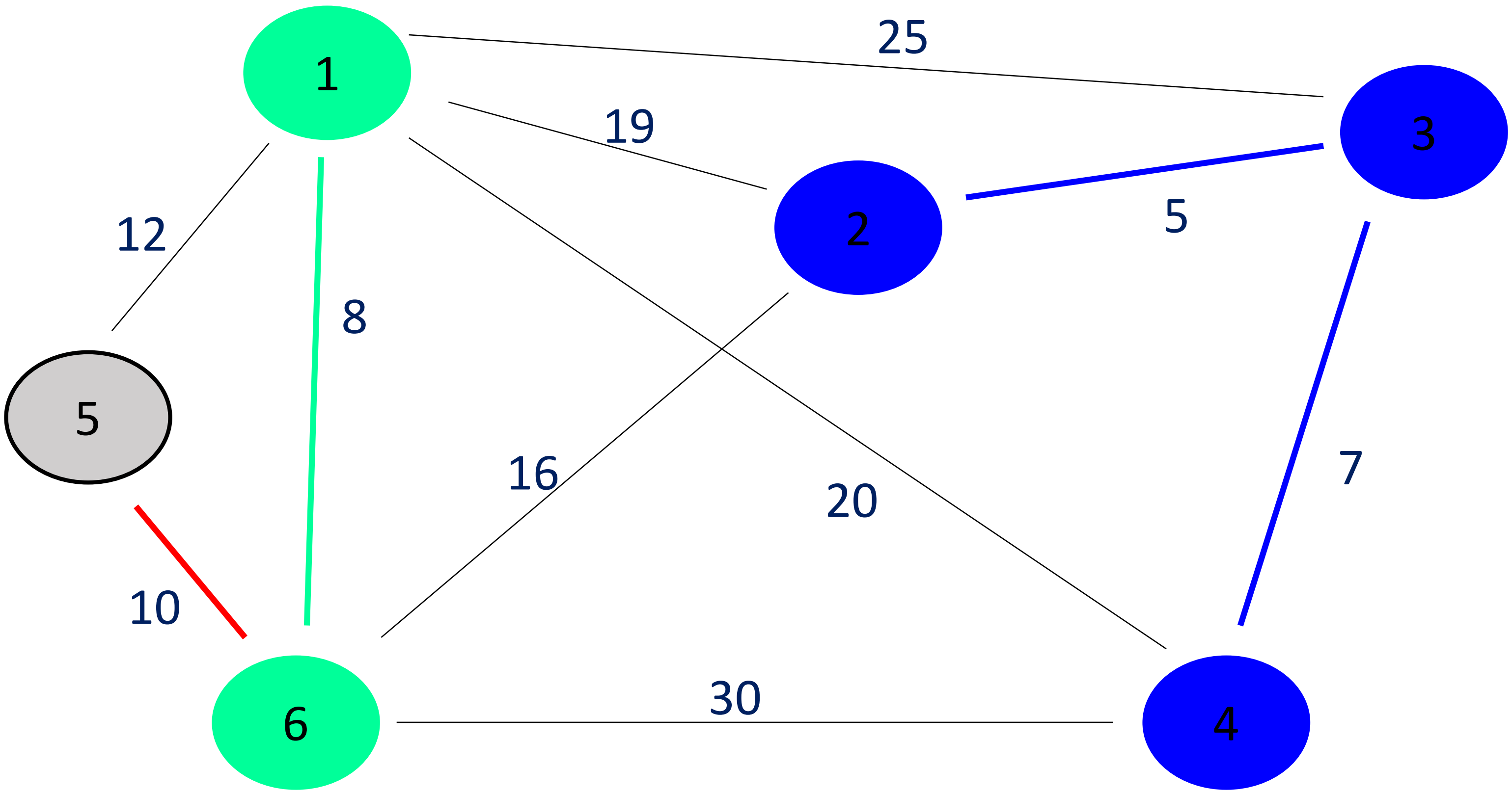
Algorithmus am Beispiel

3. Schleife 10, 12, 16, 19, 20, 25, 30



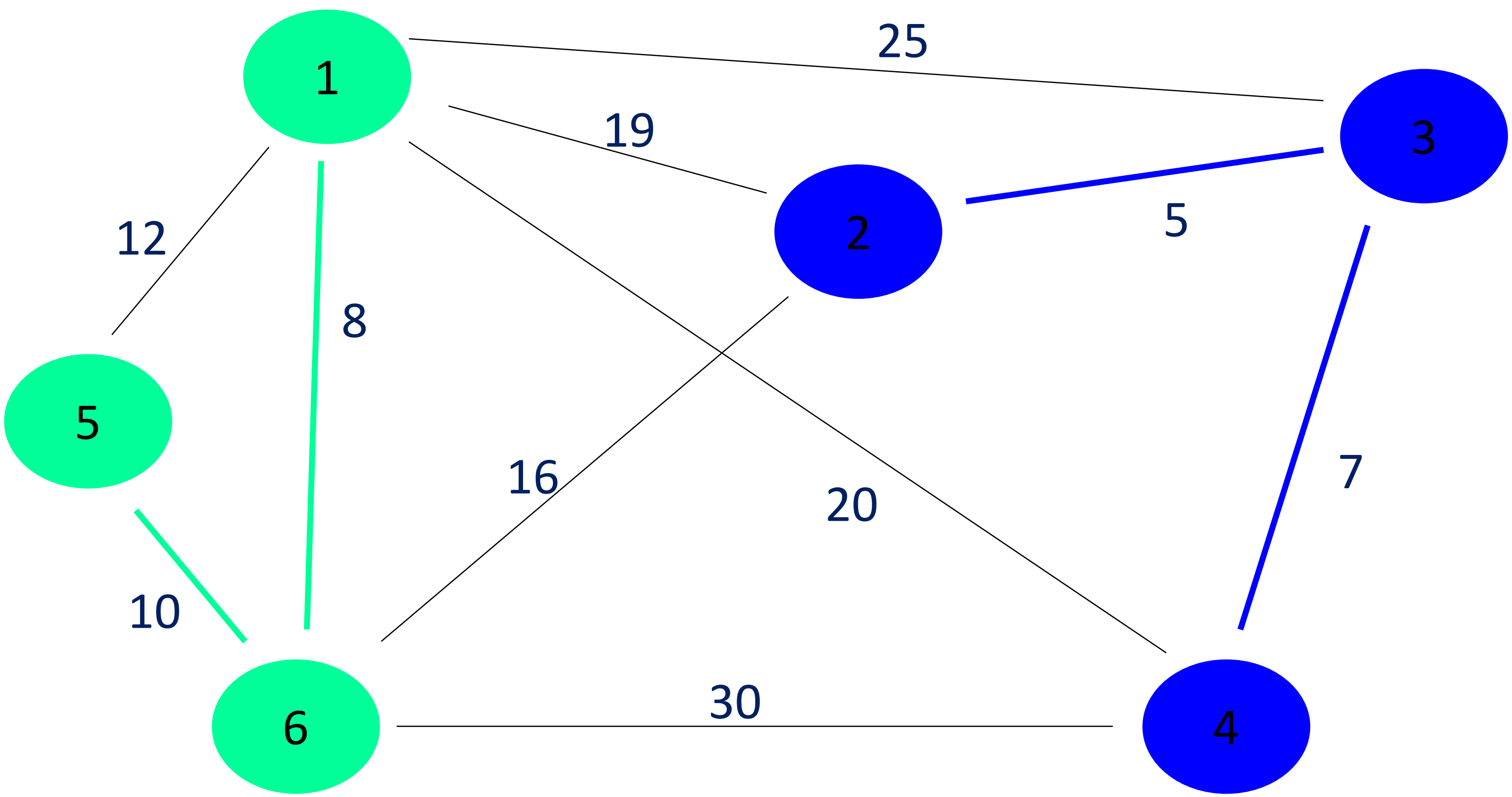
Algorithmus am Beispiel

3. Schleife   10, 12, 16, 19, 20, 25, 30



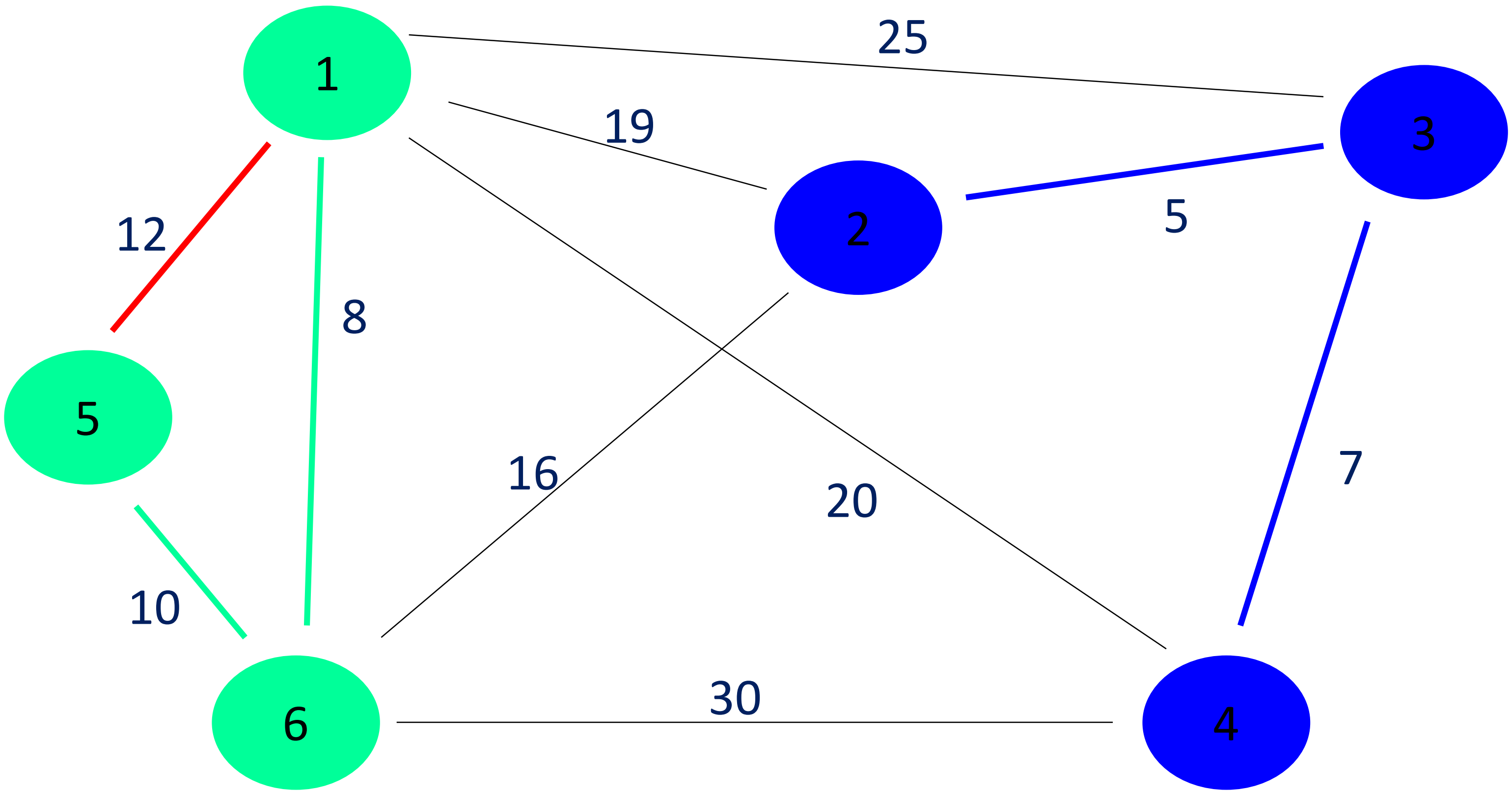
Algorithmus am Beispiel

3. Schleife 12, 16, 19, 20, 25, 30



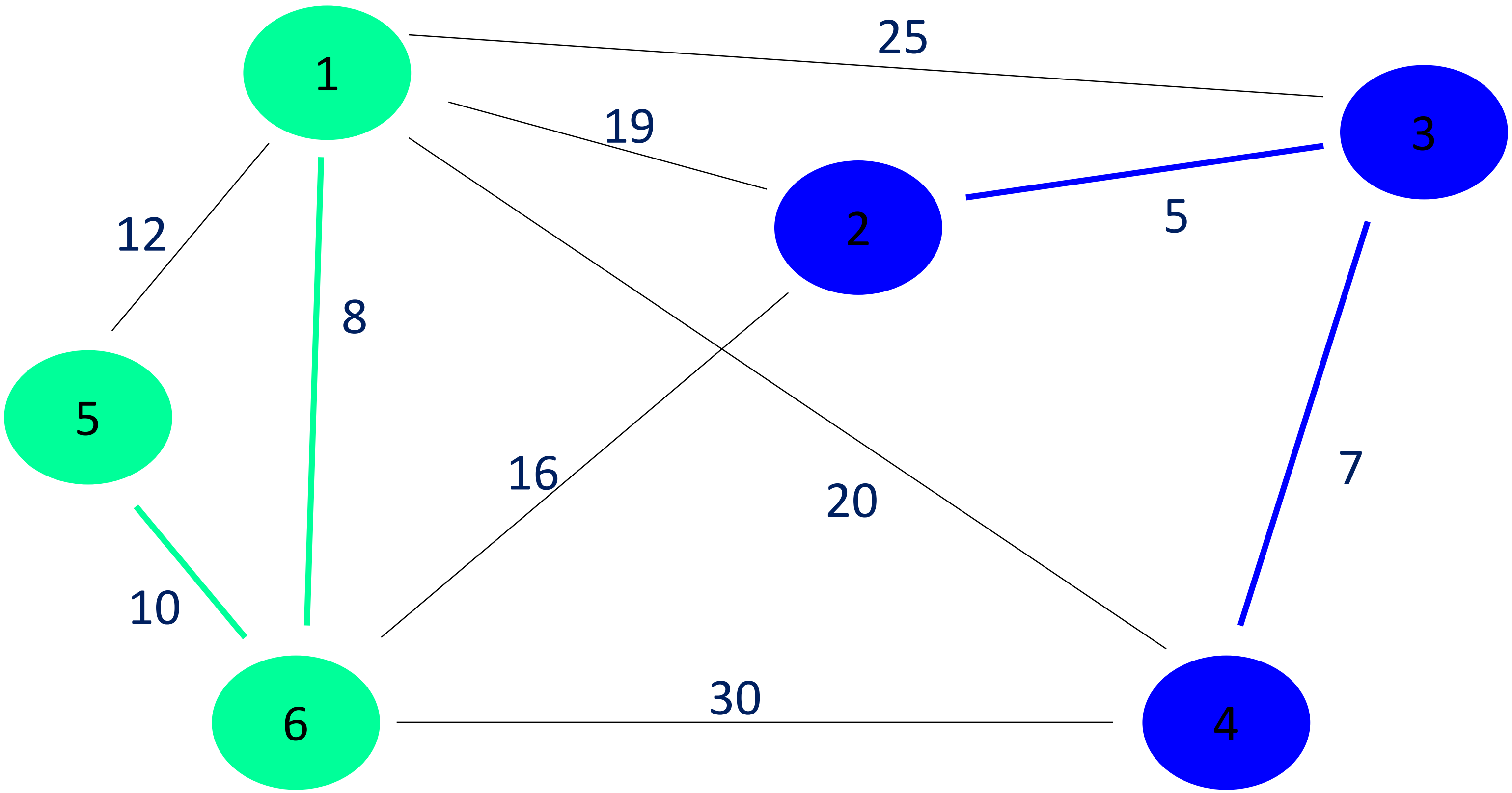
Algorithmus am Beispiel

3. Schleife 12, 16, 19, 20, 25, 30



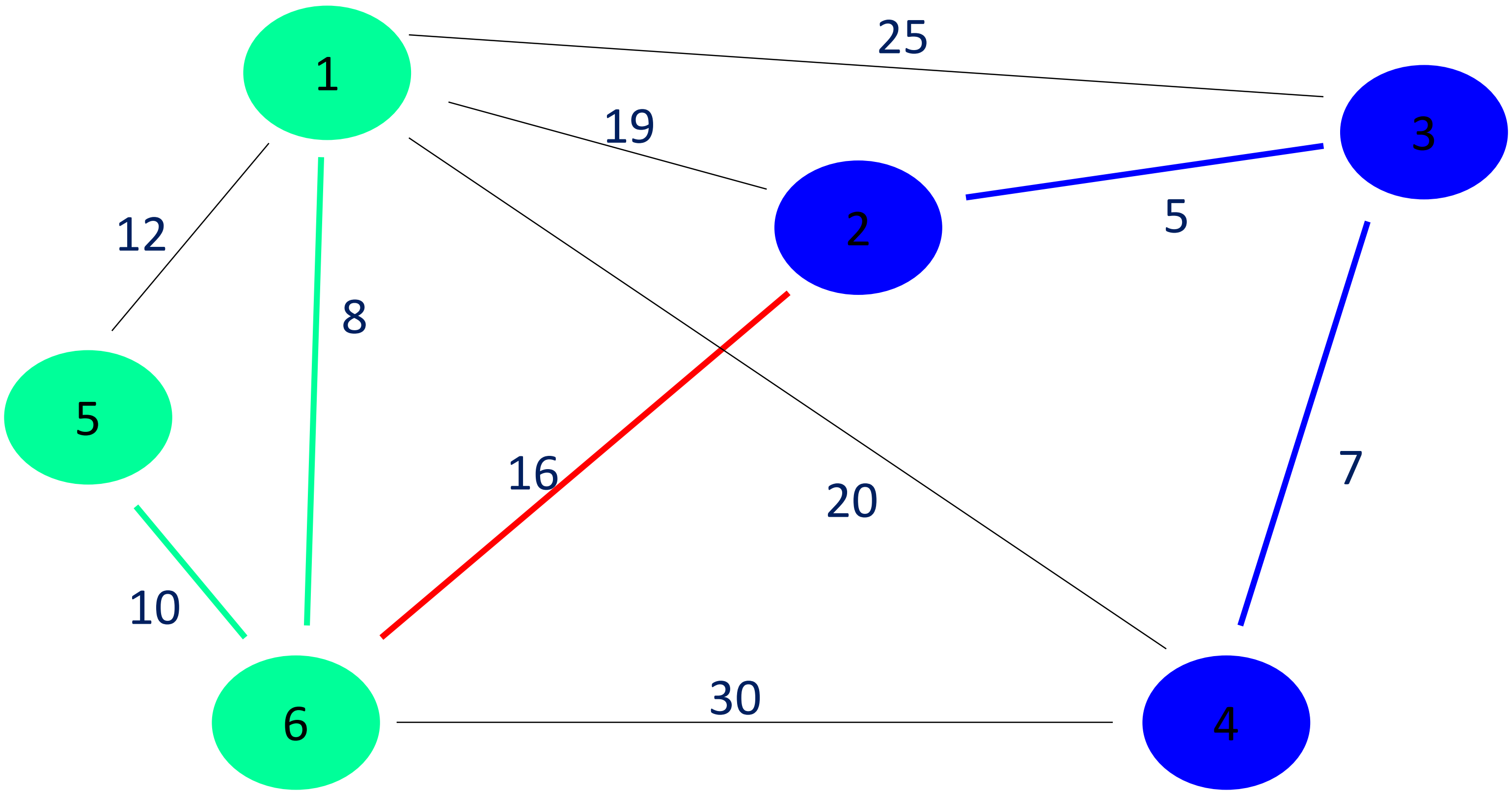
Algorithmus am Beispiel

3. Schleife 16, 19, 20, 25, 30



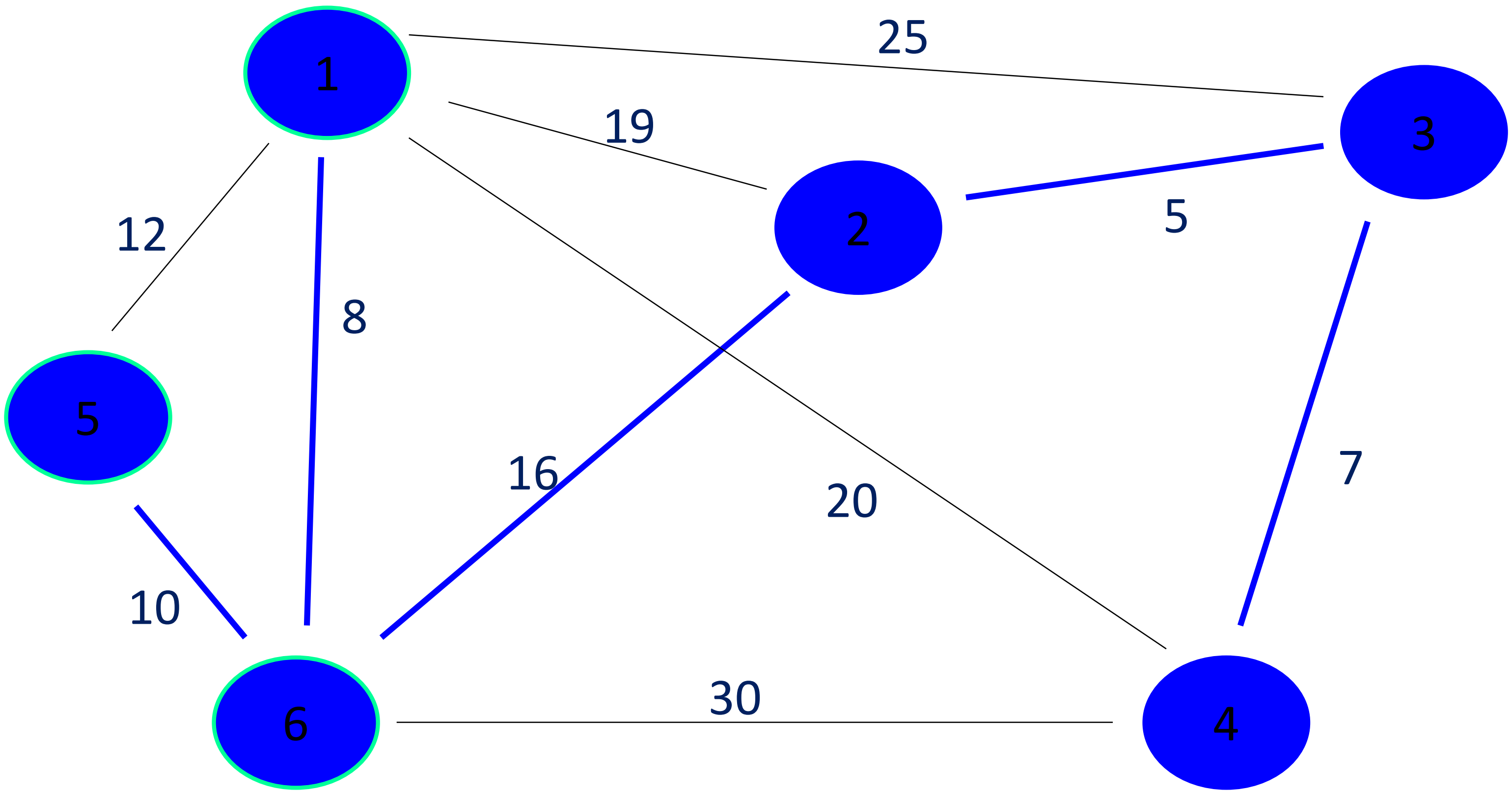
Algorithmus am Beispiel

3. Schleife 16, 19, 20, 25, 30



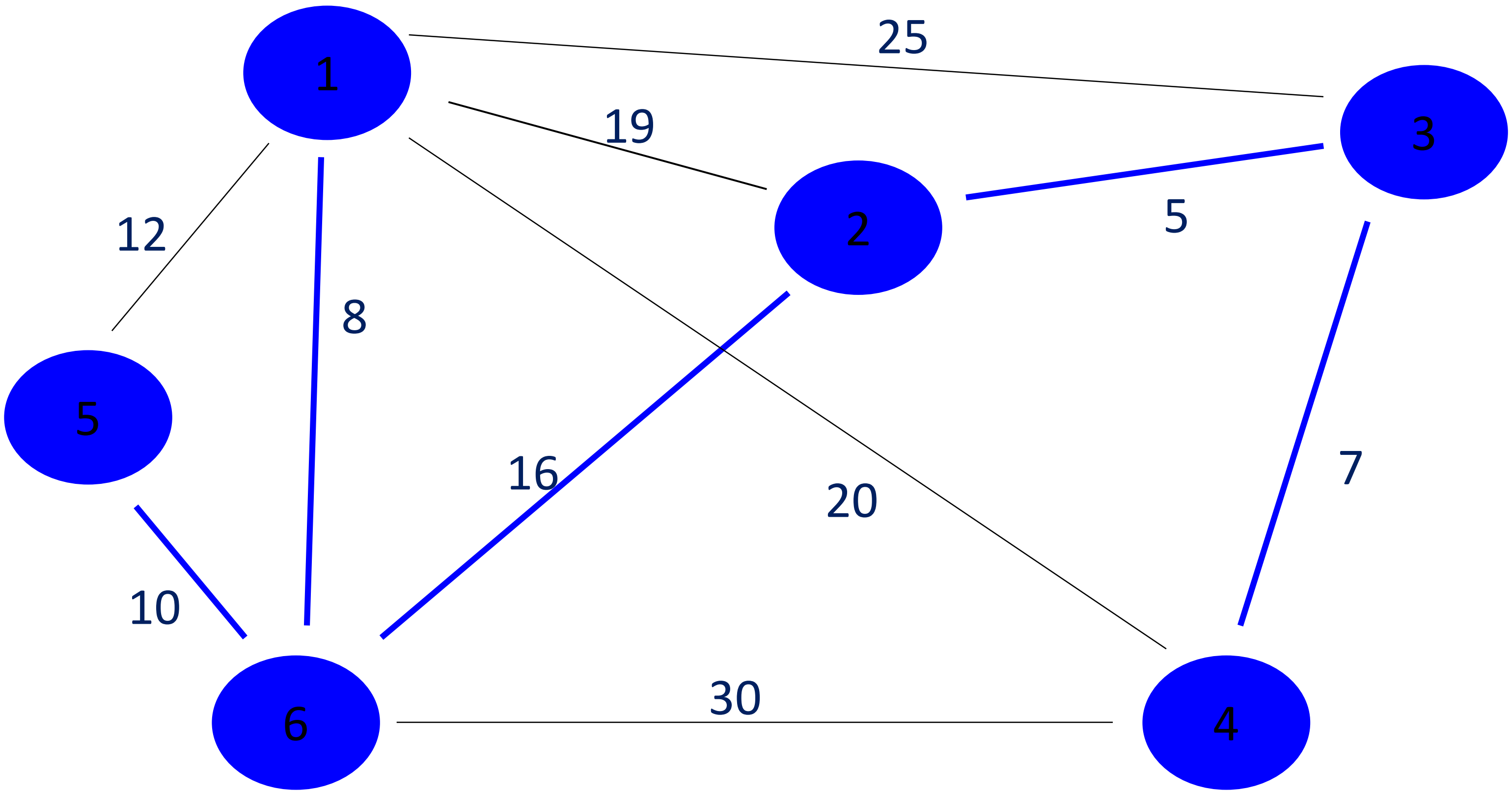
Algorithmus am Beispiel

3. Schleife 19, 20, 25, 30



Algorithmus am Beispiel

3. Schleife terminiert: nur noch ein Baum



Implementierung: Herausforderungen

- effiziente Verwaltung der Menge der Bäume
  - Vereinigungen
  - Prüfen, ob zwei gegebene Knoten zum selben Baum gehören

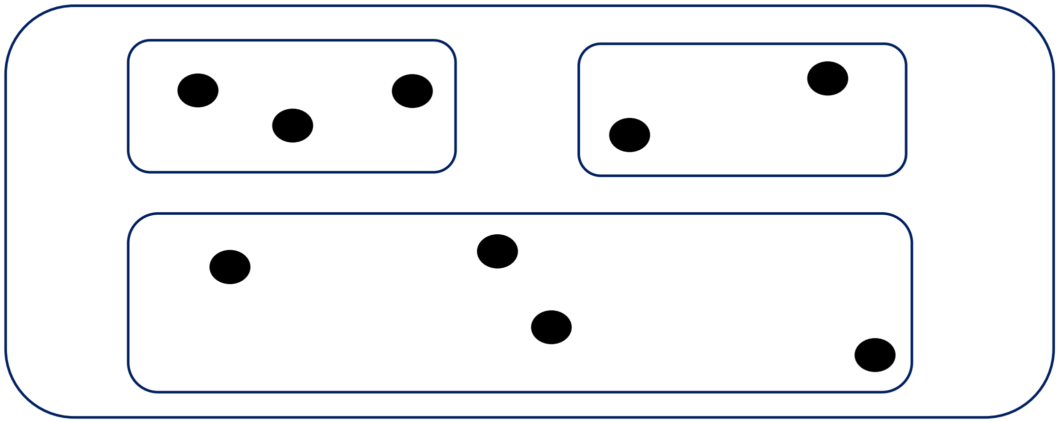
➢ Zuweisung einer „ID“ an jeden Knoten von *G*
(im Beispiel repräsentiert als Farbe)

➢ bei Kante, deren Knoten verschiedene ID haben:
Vereinigung der Bäume durch Zuweisung
einer gemeinsamen ID an alle Knoten
der zu vereinigenden Bäume

➢ Realisierung mit Datenstruktur **Union-Find**

Union-Find

- Datenstruktur zum Verwalten einer dynamischen Familie von disjunkten Mengen
- Sei X eine endliche Menge.
- **Partition** von X:



Union-Find

- Datenstruktur zum Verwalten einer dynamischen Familie von disjunkten Mengen
- Sei X eine endliche Menge.
- **Partition** von X: $\{X_1, X_2, \ldots, X_k\}$ mit $X_i \subseteq X$ und
  - $X_1 \cup X_2 \cup \ldots \cup X_k = X$
  - $X_i \cap X_j = \emptyset$ falls $i \neq j$
  - $X_i \neq \emptyset$ für $1 \leq i \leq k$
- Wählen für jede Teilmenge X_i einen **Repräsentanten $\boldsymbol{x_i}$** und schreiben X_i als **$[\boldsymbol{x_i}]$** .

ADT Union-Find

<u>type</u> Union-Find =

<u>sorts</u> T, class

<u>functions</u>

make-set: $T \rightarrow$ class

find: $T \rightarrow T$

union : $T \times T \rightarrow$ class

<u>end</u>.

Interpretation Union-Find

- **T**: eine endliche Menge X
- **class**: Klasse $[x]$ einer Partition von X

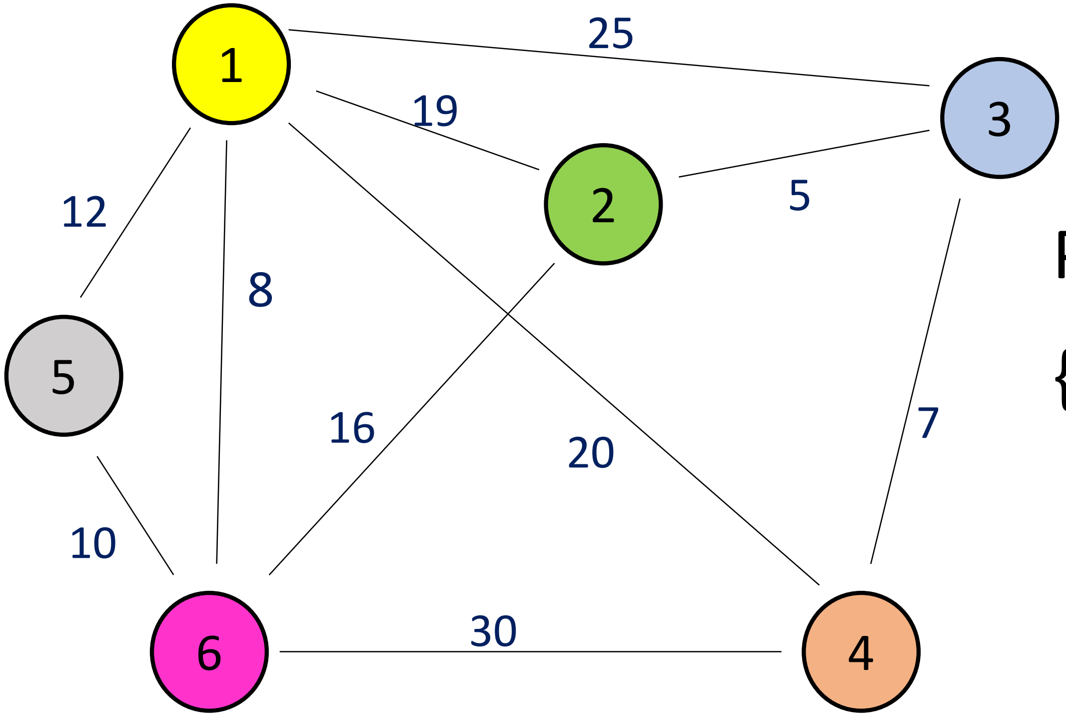
- **make-set:** T $\rightarrow$ class *vermöge* $x \mapsto [x] = \{x\}$
- **find:** T $\rightarrow$ T *vermöge* $x \mapsto y$ mit $x \in [y]$
- **union:** T $\times$ T $\rightarrow$ class *vermöge* $(r,s) \mapsto [\mathrm{find}(r)] \cup [\mathrm{find}(s)]$
  wobei find(r) als Repräsentant von $[\mathrm{find}(r)] \cup [\mathrm{find}(s)]$ gewählt wird

Implementierung Union-Find

- als Array von Listen, in denen jedes Element einen Zeiger auf das Head-Element (Repräsentant) hat *oder*
- als Menge von Bäumen (Wald):
  - **make-set(*x*)**: Hinzufügen des einelementigen Baums mit Wurzel *x* zum Wald
  - **find(*x*)**: Suchen der Wurzel des Baums, der *x* enthält
  - **union(*r*,*s*)**: Anfügen des Baums mit *s* als neuen Teilbaum der Wurzel des Baums mit *r*
- *Optimierung union(r,s):*
  - Füge Baum geringerer Tiefe als Teilbaum der Wurzel des Baums größerer Tiefe an
    → kürzere Laufzeit für find(*x*)

Union-Find für die Knotenmenge

Initialisierung: sechs mal **make-set** aufrufen

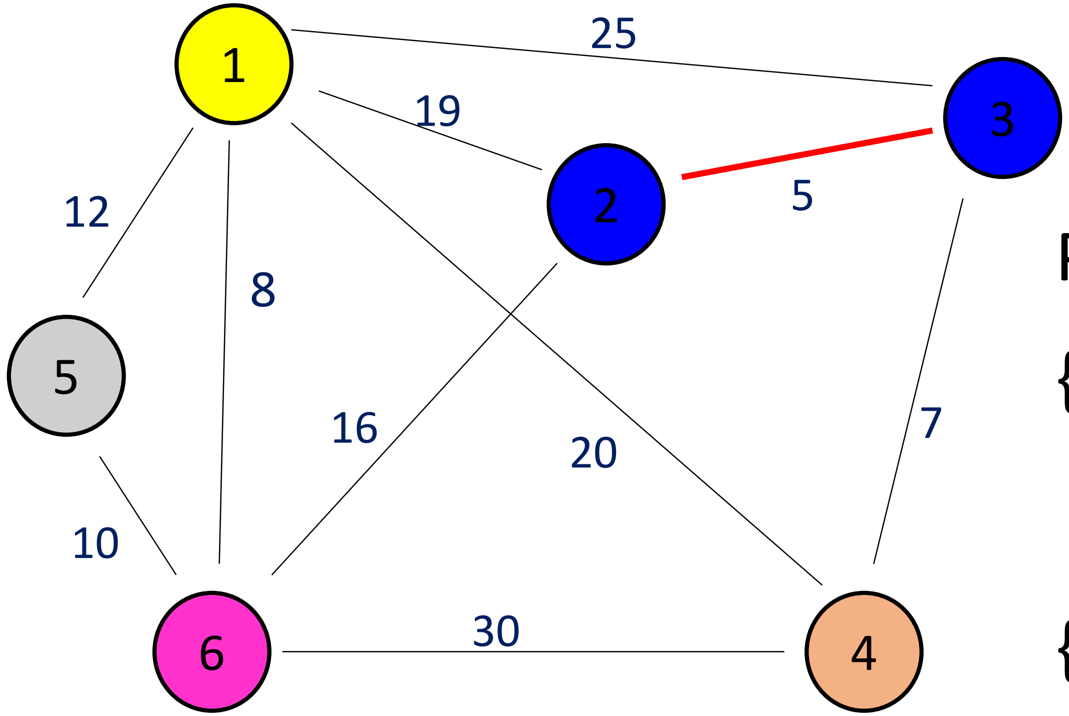


Partition:

{{1}, {2}, {3}, {4}, {5}, {6}}

Union-Find für die Knotenmenge

union(2,3) = [find(2)] ∪ [find(3)] = [2] ∪ [3]



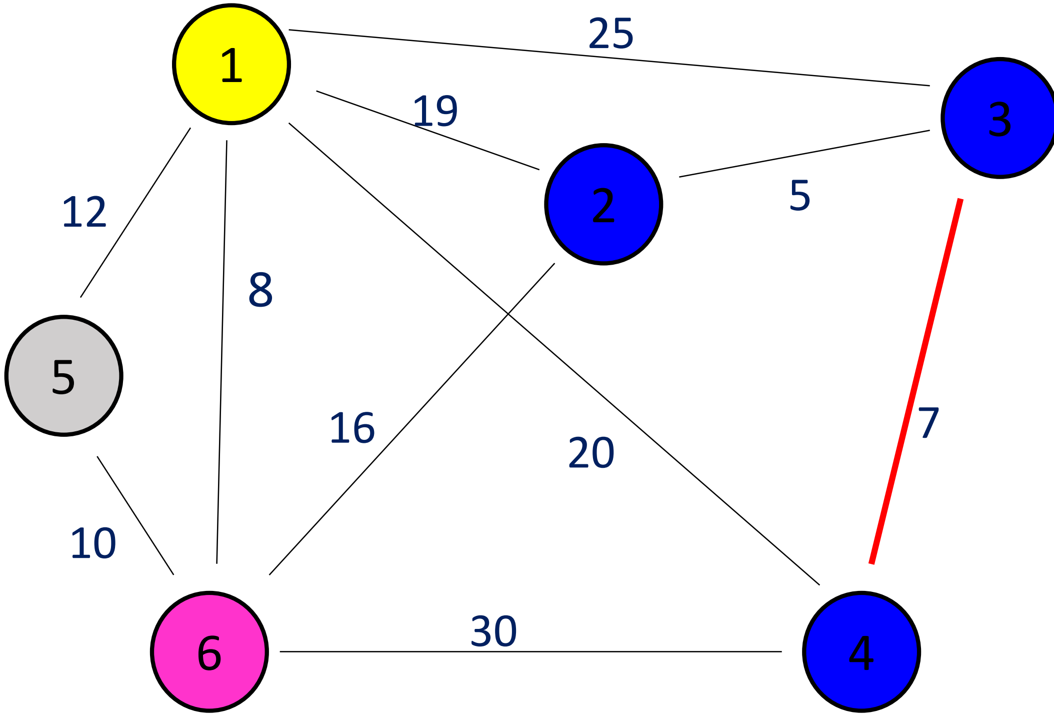
Partition:

{{1}, {2}, {3}, {4}, {5}, {6}}

{{1}, {2,3}, {4}, {5}, {6}}

Union-Find für die Knotenmenge

union(3,4) = [find(3)] ∪ [find(4)] = [2] ∪ [4]



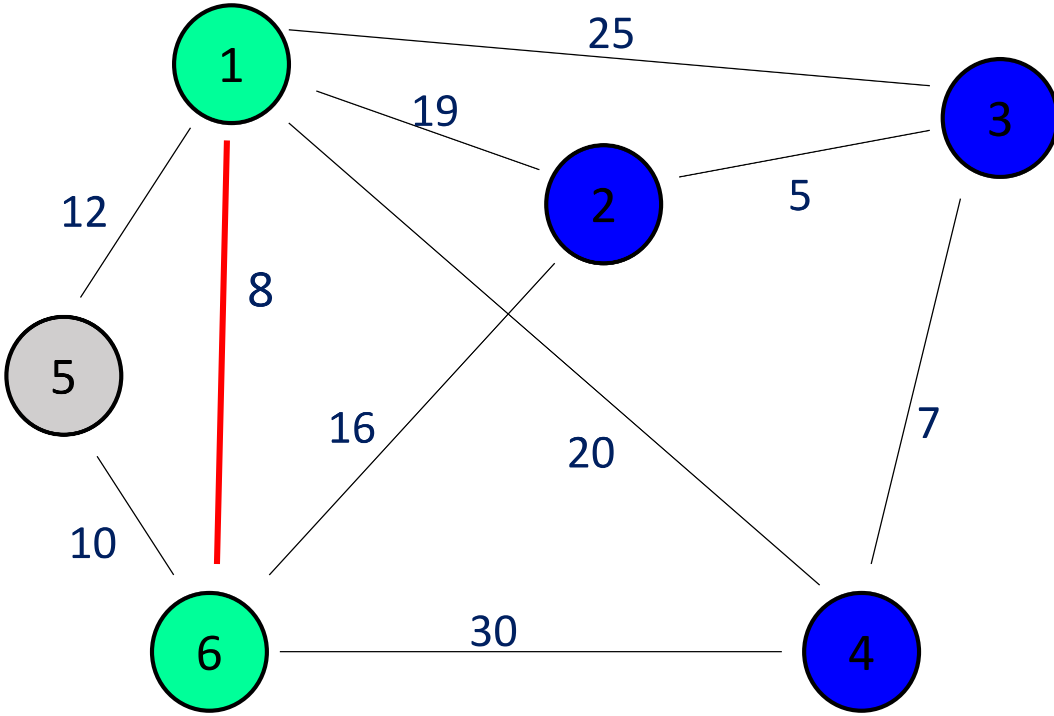
Partition:

{{1}, {2,3}, {4}, {5}, {6}}

{{1}, {2,3,4}, {5}, {6}}

Union-Find für die Knotenmenge

union(1,6) = [find(1)] ∪ [find(6)] = [1] ∪ [6]



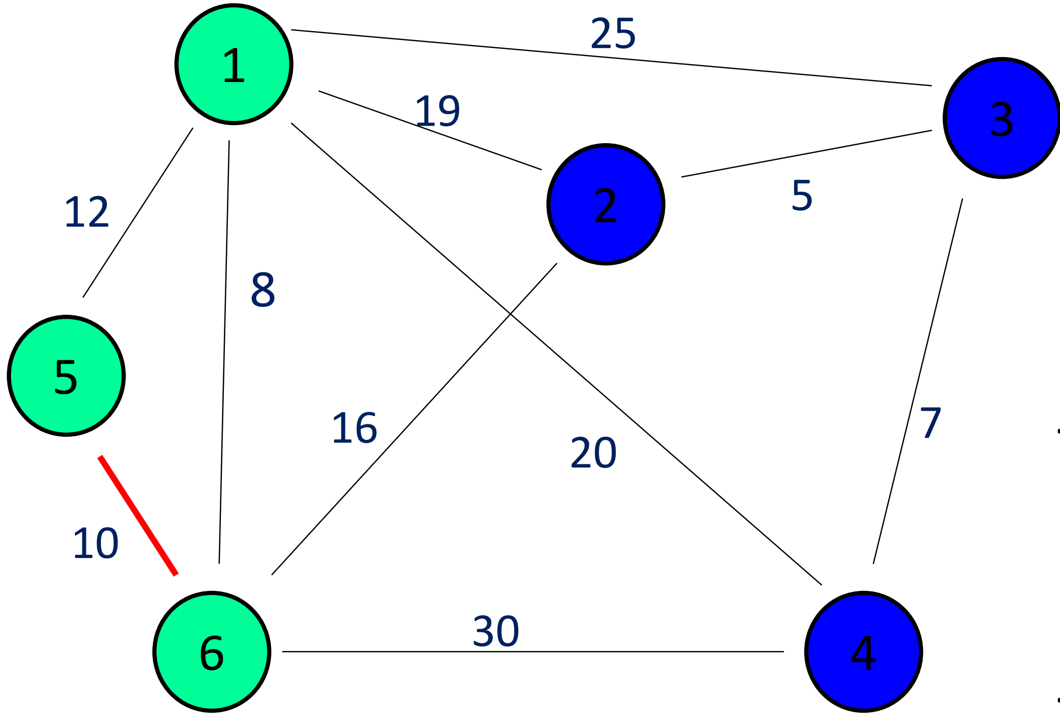
Partition:

{{1}, {2,3,4}, {5}, {6}}

{{1,6}, {2,3,4}, {5}}

Union-Find für die Knotenmenge

union(5,6) = [find(5)] ∪ [find(6)] = [5] ∪ [1]



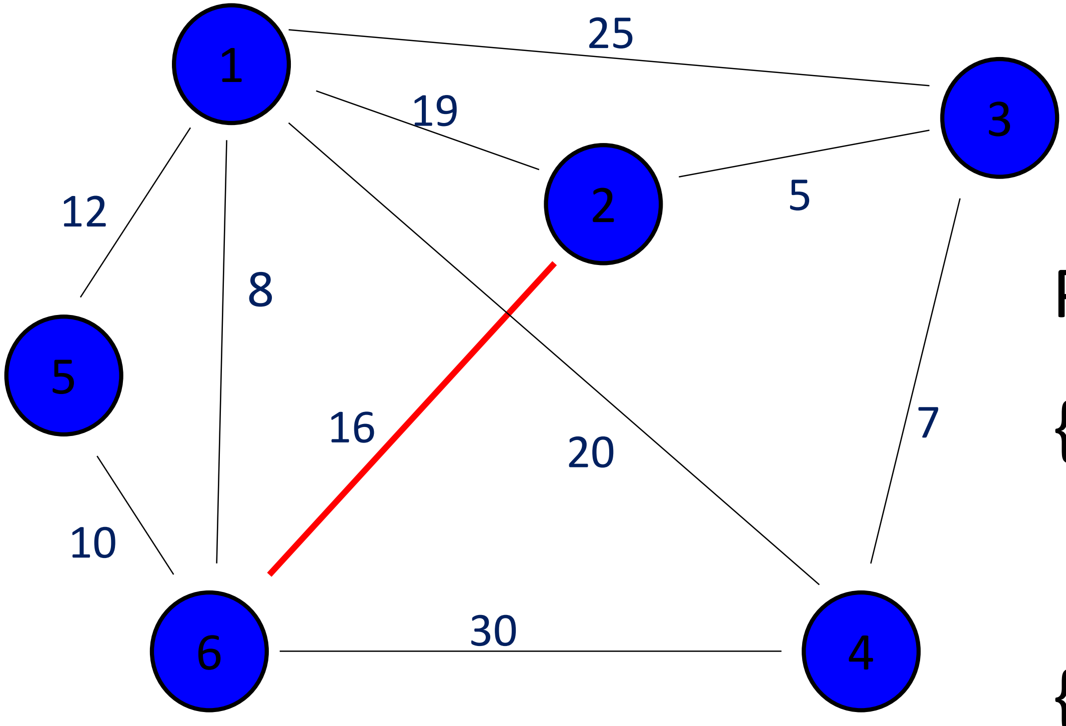
Partition:

{{1,6}, {2,3,4}, {5}}

{{1,5,6}, {2,3,4}}

Union-Find für die Knotenmenge

union(2,6) = [find(2)] ∪ [find(6)] = [2] ∪ [1]



Partition:

{{1,5,6}, {2,3,4}}

{{1,2,3,4,5,6}}

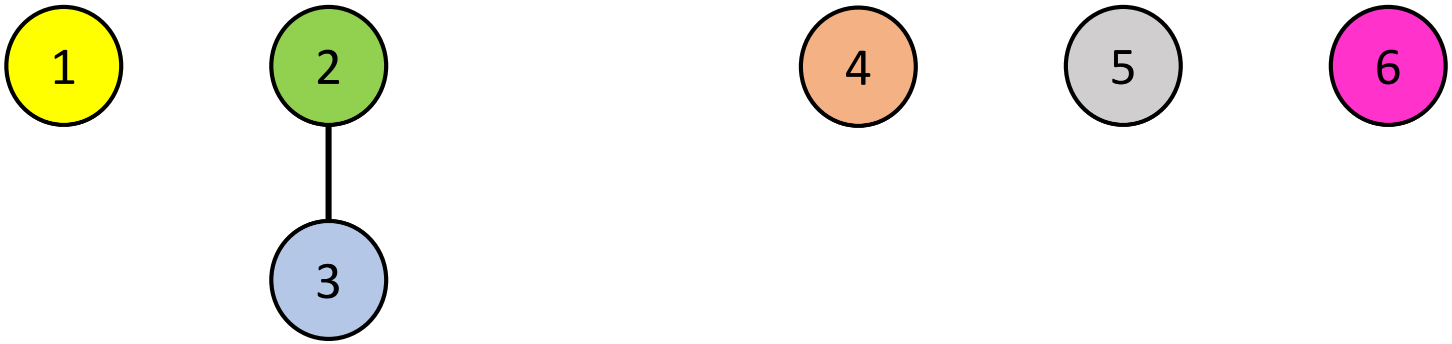
... und als Wald

sechs mal make-set:

... dann union(2,3)

... und als Wald

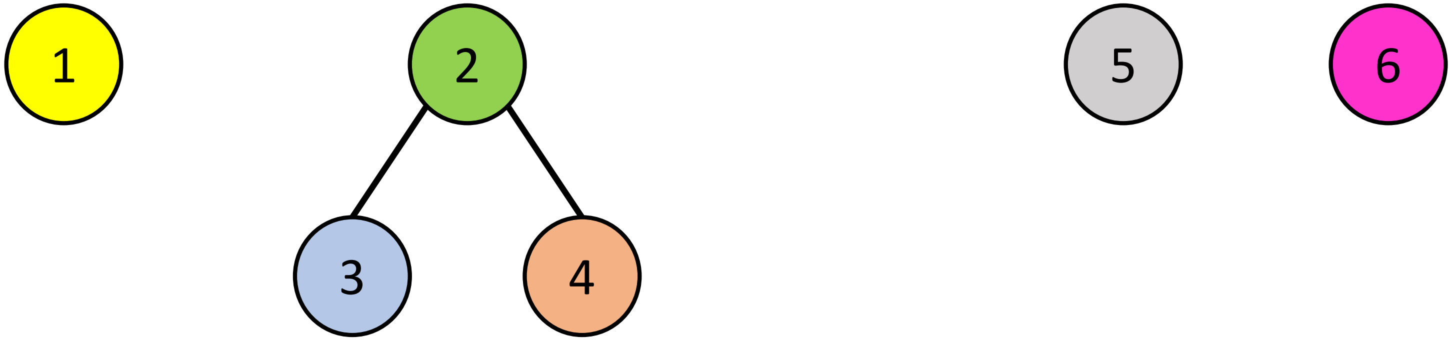
union(2,3):



... dann union(3,4)

... und als Wald

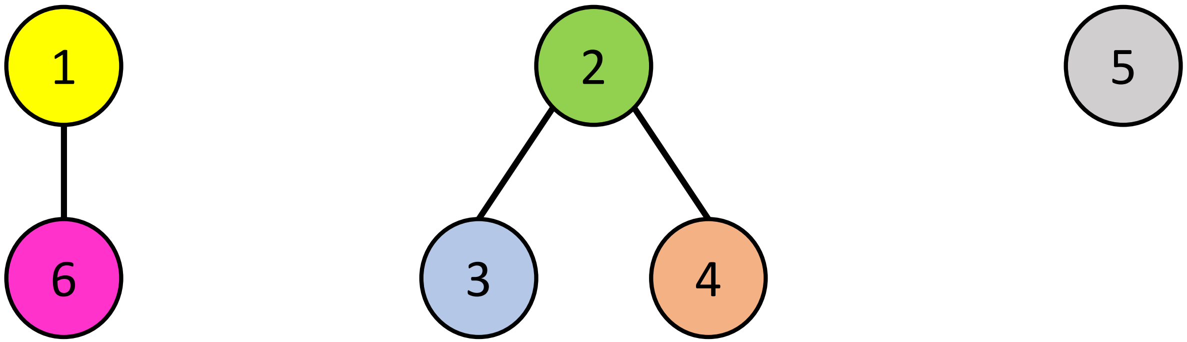
union(3,4):



... dann union(1,6)

... und als Wald

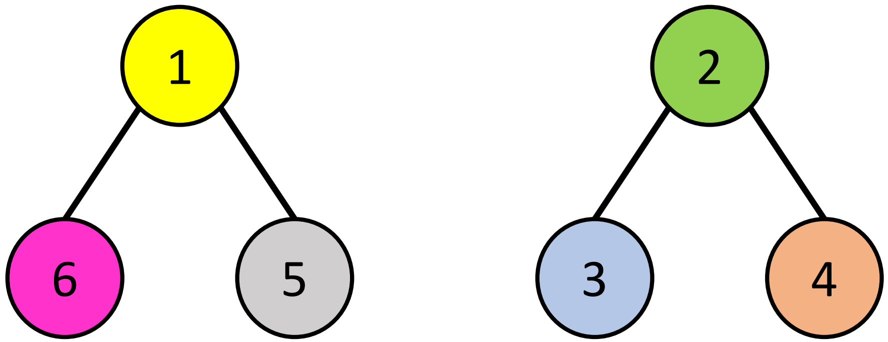
union(1,6):



... dann union(5,6)

... und als Wald

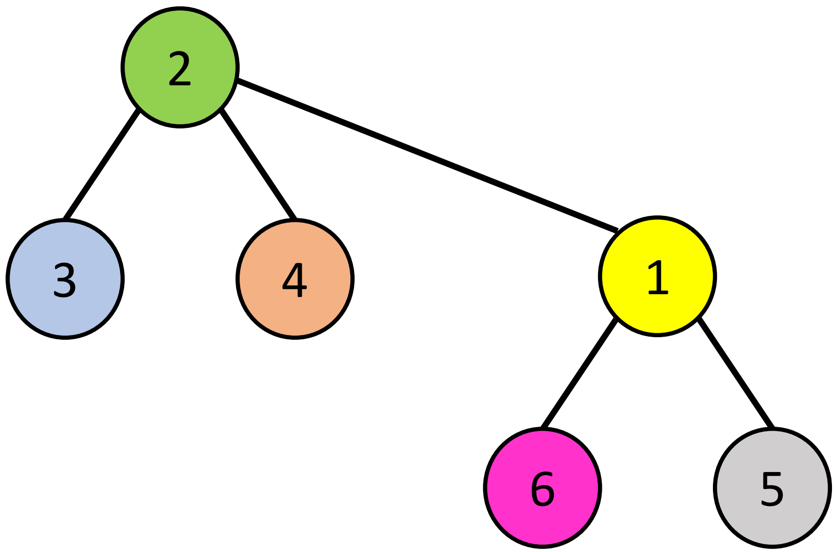
union(5,6):



... dann union(2,6)

... und als Wald

union(2,6):



Bei Kruskal muss zusätzlich Buch über die Kanten geführt werden.

Pseudocode Kruskal-Algorithmus

Eingabe: ungerichteter zusammenhängender Graph G
mit Kantengewichten, $G = (V,E,w)$

Ausgabe: minimaler Spannbaum von G

Ordne Kanten zu $e_1, e_2, \ldots, e_n$ so dass $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$

Für alle $v \in V$

    `make-set`(v)

$E' \longleftarrow \{\}$

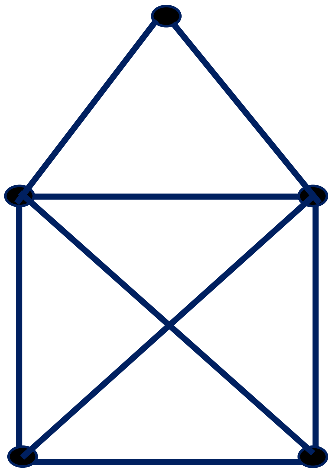
Für $i = 1$ bis n

    Falls $e_i = \{u,v\}$ und **`find`**$(u) \neq$ **`find`**(v)

        `union`(u,v)

        $E' \longleftarrow E' \cup \{\{u,v\}\}$

Gib (V,E') aus



Vollständige Pfade

Euler- und Hamilton-Pfade

- $G = (V,E)$ (gerichteter oder ungerichteter) Graph
- **Euler-Pfad:** Pfad, der jede Kante des Graphen genau einmal verwendet
  - Pfad $v_0, v_1, \ldots, v_k$ mit $\{ (v_i,v_{i+1}) \mid 0 \leq i < \mathrm{k} \} = E$ und falls $i \neq j$, dann $(v_i,v_{i+1}) \neq (v_j,v_{j+1})$.
- **Hamilton-Pfad:** Pfad, der jeden Knoten des Graphen genau einmal enthält
- **Euler-Kreis (Hamilton-Kreis):** Euler-Pfad (Hamilton-Pfad), der ein Kreis ist.

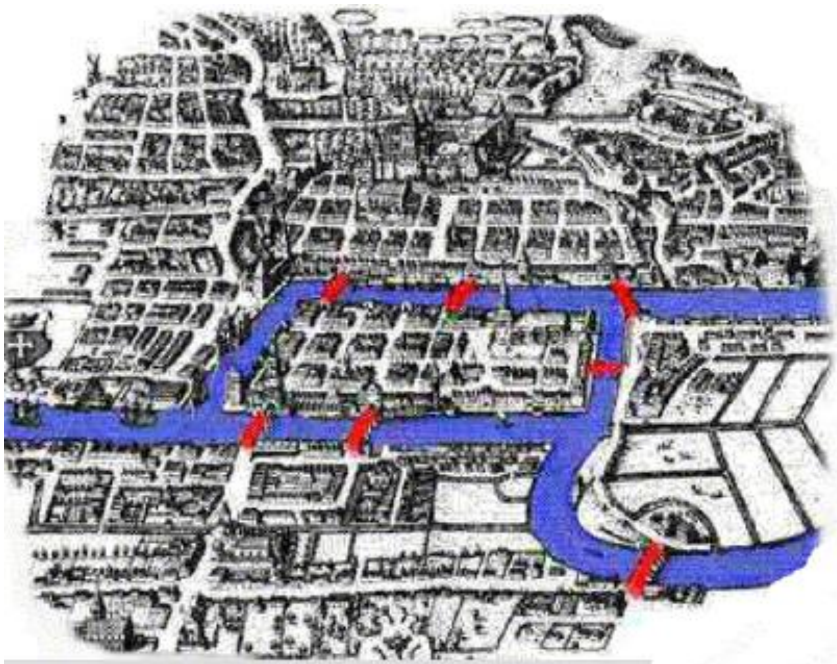
Suche nach Euler-Kreis

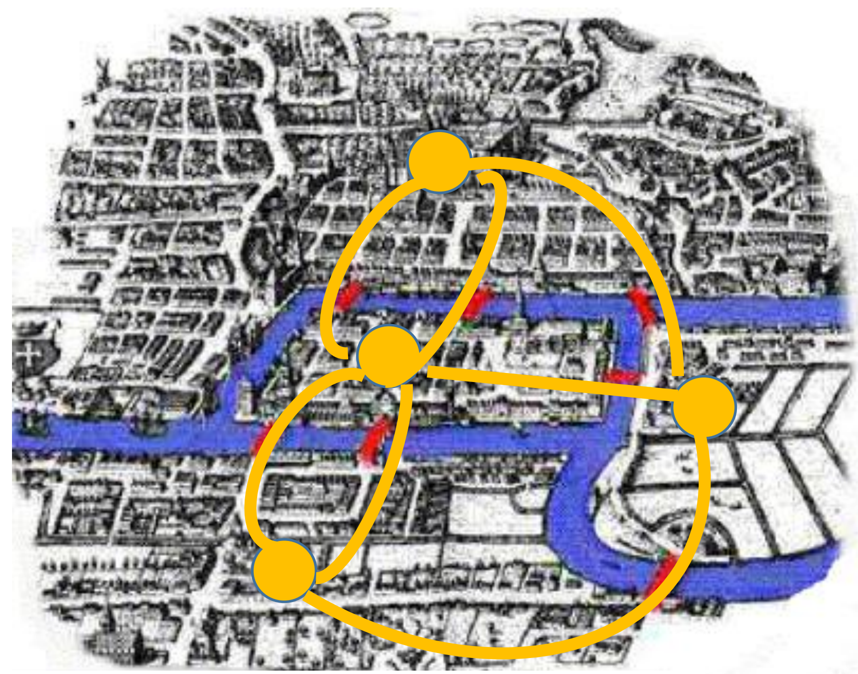
- *Ursprungsfrage:* **Königsberger Brückenproblem**

  Der Fluss Pregel durchfließt Kaliningrad (früher Königsberg). Zur Zeit Eulers teilte der Fluss die Stadt in vier Gebiete, die durch sieben Brücken miteinander verbunden waren. *Gibt es einen Spaziergang, der über jede der Brücken genau einmal führt?*

- Gelöst von Leonard Euler (1707-1783)
  → Gibt es einen Euler-Kreis im zugehörigen Graphen?

Eulers Modellierung





Existenz von Euler-Pfaden und -Kreisen

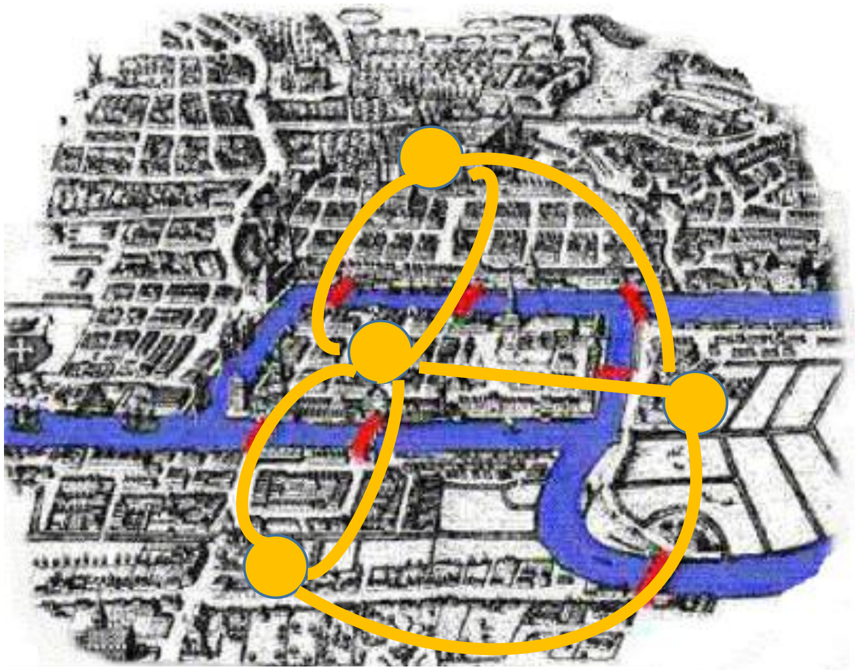
Satz: Ein (ungerichteter) zusammenhängender Graph enthält genau dann einen Euler-Pfad, wenn zwei oder keiner seiner Knoten von ungeradem Grad ist;
hat kein Knoten ungeraden Grad, handelt es sich bei dem Eulerpfad um einen Eulerkreis.

Beweisidee: Für jeden Knoten v mit geradem Grad gilt: Wenn v über eine Kante betreten wird, kann er über eine andere Kante verlassen werden.

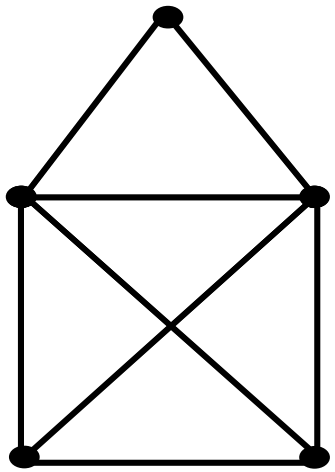
Bei zwei Knoten ungeraden Grads werden diese als erster und letzter Knoten des Pfads gewählt (kein Kreis).

Beispiele

weder Eulerkreis noch -pfad:



Eulerpfad
aber kein -kreis:



Entscheidung in $O(|V|^2)$.

Schnellere Algorithmen existieren.

Suche nach Hamilton-Pfad

- Graphentheoretische Charakterisierungen: z.B. jeder vollständige Graph besitzt einen HK.
- *Für den allgemeinen Fall:* Jeder bekannte Algorithmus ist im Grunde Brute-Force, *z.B.*
  - Bilde alle Anordnungen aller Knoten ($|V|!$ viele).
  - Prüfe jeweils, ob es ein Hamilton-Pfad ist.
- Gibt es <u>wesentlich</u> bessere Algorithmen?
  - ➢ Wird nicht erwartet, da das Problem **NP-vollständig** ist.