

# Graph Cheatsheet für Algorithmen und Datenstrukturen

Kenji Nakano, HS23, Stand 17.12.2023

Keine Garantie für Vollständigkeit oder Korrektheit

## Definition Graph

Ein **Graph** ist ein Tupel  $\mathbf{G} = (\mathbf{V}, \mathbf{E})$  wobei

- $V :=$  Knotenmenge (vertices)
- $E :=$  Kantenmenge (edges)

jede Kante ist ein ungeordnetes Paar zweier Knoten  $u \neq v$ ,  $e = \{u, v\} \in E$  (Kurzform:  $uv$ )

## Weg, Pfad, Zyklus

- **Weg:** Folge von benachbarten Knoten (engl. walk)
- **Pfad:** Weg ohne wiederholte Knoten
- **Zyklus:** Weg mit  $v_0 = v_l$ ,  $l \geq 2$  (engl. closed walk)

Die Länge eines Wegs bzw. Pfads ist die Anzahl an Kanten, nicht die Anzahl an Knoten

## Begriffe

- $u, v$  adjazent/benachbart  $\Leftrightarrow e = \{u, v\} \in E$
- $e \in E$  inzident/anliegend zu  $v \Leftrightarrow \exists u \in V$ , so dass  $e = \{u, v\}$
- $\deg(u) =$  Knotengrad von  $u$  (Anzahl Nachbarn)
- $u$  erreicht  $v \Leftrightarrow \exists$  Weg zwischen  $u$  und  $v$  (engl. reachable)  
Äquivalenzrelation (symmetrisch, reflexiv, transitiv)
- Zusammenhangskomponente (ZHK): Äquivalenzklasse der "erreichbar"-relation (engl. connected component)
- Graph ist zusammenhängend  $\Leftrightarrow$  es gibt genau eine ZHK

$$\text{Handschlag Lemma: } \sum_{v \in V} \deg(v) = 2 \cdot |E|$$

## Eulerweg, Hamiltonpfad, Eulerzyklus

- **Eulerweg:** Weg welcher jede Kante **genau** einmal enthält (engl. Eulerian walk)
- **Hamiltonpfad:** Pfad der jeden Knoten **genau** einmal enthält
- **Eulerzyklus:** Zyklus welcher jede Kante **genau** einmal enthält

$\exists$  Eulerzyklus  $\Leftrightarrow$  alle Knotengrade gerade und alle Kanten in einer ZHK

## Algorithmus Eulertour / Eulerwalk

### **Euler( $G$ ):**

- Input: Graph  $G = (V, E)$
- Output: Liste  $Z$  mit Eulerzyklus, falls existiert.
- Laufzeit:  $\mathcal{O}(m)$

### **EulerWalk( $u$ ):**

- Input: Knoten  $u \in V$
- Output: Keiner
- Laufzeit:  $\mathcal{O}(m)$

---

### **Algorithm 1** Euler( $G$ )

**Require:** Alle Kanten unmarkiert

- 1:  $Z \leftarrow$  Leere Liste
  - 2: EulerWalk( $u_0$ )  $\triangleright$  für  $u_0 \in V$  beliebig
  - 3: **return**  $Z$
- 

---

### **Algorithm 2** EulerWalk( $u$ )

- 1: **for**  $uv \in E$ , nicht markiert **do**
  - 2:     markiere Kante  $uv$
  - 3:     EulerWalk( $v$ )
  - 4:  $Z \leftarrow Z \cup \{u\}$
-

## Begriffe gerichtete Graphen

Der Graph  $G = (V, E)$  ist definiert wie beim ungerichteten **ausser** Kanten sind geordnete Paare  $e = (u, v) \in E$

- $v$  ist Nachfolger von  $u$
- $u$  ist Vorgänger von  $v$
- $\deg_{in}(u)$  = Eingangsgrad
- $\deg_{out}(u)$  = Ausgangsgrad
- Quelle  $u$ :  $\deg_{in}(u) = 0$
- Senke  $v$ :  $\deg_{out}(v) = 0$

## DFS (Depth-First-Search)

**DFS( $G$ ) (und Visit( $u$ )):**

- Input: Graph  $G = (V, E)$
- Output: Implementationsabhängig (kann für sehr viel verwendet werden)
- Laufzeit:  $\mathcal{O}(n + m)$  (für Adjazenzlisten)

---

### Algorithm 3 Visit( $u$ )

---

```
1: pre[u]  $\leftarrow$  T; T  $\leftarrow$  T + 1
2: markiere u
3: for Nachfolger v von u, unmarkiert do
4:   Visit(v)
5: post[u]  $\leftarrow$  T; T  $\leftarrow$  T + 1
```

---

---

### Algorithm 4 DFS( $G$ )

---

```
1: T  $\leftarrow$  1
2: alle Knoten unmarkiert
3: for  $u_0 \in V$ , unmarkiert do
4:   Visit( $u_0$ )
```

---

## BFS (Breadth-First-Search)

**BFS( $s$ ):**

- Input: Knoten  $s \in V$
- Output: Implementationsabhängig (kann für sehr viel verwendet werden, ähnlich wie DFS)
- Laufzeit:  $\mathcal{O}(n + m)$

---

### Algorithm 5 BFS( $s$ )

---

```
1:  $Q \leftarrow \{s\}$  ▷  $Q$  ist eine Queue
2:  $\text{enter}[s] \leftarrow 0$ ;  $T \leftarrow 1$ 
3: while  $Q \neq \emptyset$  do
4:    $u \leftarrow \text{dequeue}(Q)$ 
5:    $\text{leave}[u] \leftarrow T$ ;  $T \leftarrow T + 1$ 
6:   for  $(u, v) \in E$ ,  $\text{enter}[v]$  nicht zugewiesen do
7:      $\text{enqueue}(Q, v)$ 
8:      $\text{enter}[v] \leftarrow T$ ;  $T \leftarrow T + 1$ 
```

---

## Shortest-Path Algorithmen

### Dijkstra( $s$ ):

- Input: Knoten  $s \in V$ 
  - Graph darf nur nicht-negative Kantenkosten haben
- Output:  $dist(s, v)$  für alle  $v \in V$
- Laufzeit:  $\mathcal{O}((m + n) \cdot \log(n))$

---

### Algorithm 6 Dijkstra( $s$ )

---

```
1:  $d[s] \leftarrow 0; \quad d[v] \leftarrow \infty \quad \forall v \in V \setminus \{s\}$ 
2:  $S \leftarrow \emptyset$ 
3:  $H \leftarrow \text{make-heap}(V); \text{decrease-key}(H, s, 0)$ 
4: while  $S \neq V$  do
5:    $v^* \leftarrow \text{extract-min}(H)$ 
6:    $S \leftarrow S \cup \{v^*\}$ 
7:   for  $(v^*, v) \in E, v \notin S$  do
8:      $d[v] \leftarrow \min\{d[v], d[v^*] + c(v^*, v)\}$ 
9:      $\text{decrease-key}(H, v, d[v])$ 
```

---

### Bellman-Ford( $s$ ):

- Input: Knoten  $s \in V$ 
  - Graph darf auch negative Kanten (und negative Zyklen) enthalten.
- Output:  $dist(s, v)$  für alle  $v \in V$
- Laufzeit:  $\mathcal{O}(m \cdot n)$
- Wenn man nach der  $(n - 1)$ -ten Iteration nochmals eine weitere Iteration durchführt und die Werte sich nochmals verändern existiert ein gerichteter Zyklus mit negativem Totalgewicht

---

### Algorithm 7 Bellman-Ford( $s$ )

---

```
1:  $d[s] \leftarrow 0; \quad d[v] \leftarrow \infty \quad \forall v \in V \setminus \{s\}$  ▷ 0-gute Schranken
2: for  $i \in \{1, \dots, n - 1\}$  do ▷ Verbessere Schranken  $(n - 1)$ -mal
3:   for  $(u, v) \in E$  do
4:      $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$ 
```

---

## MST Algorithmen

### Boruvka( $G$ ):

- Input: Graph  $G = (V, E)$
- Output: Minimaler Spannbaum  $F$
- Laufzeit:  $\mathcal{O}((m + n) \cdot \log(n))$  (wie Dijkstra)

---

### Algorithm 8 Boruvka( $G$ )

---

```
1:  $F \leftarrow \emptyset$  ▷ sichere Kanten
2: while  $F$  nicht Spannbaum do ▷  $\leq \log(n)$  Iterationen,  $\mathcal{O}(m + n)$  pro Iteration
3:    $(S_1, \dots, S_k) \leftarrow \text{ZHKs von } F$ 
4:    $(e_1, \dots, e_k) \leftarrow \text{minimale Kanten an } S_1, \dots, S_k$ 
5:    $F \leftarrow F \cup \{e_1, \dots, e_k\}$ 
```

---

### Idee: Konzentrieren uns auf eine ZHK

### Prim( $G, s$ ):

- Input: Graph  $G = (V, E)$  und  $s \in V$
- Output: Minimaler Spannbaum  $F$
- Laufzeit:  $\mathcal{O}((m + n) \cdot \log(n))$  (wie Dijkstra und Boruvka)

---

### Algorithm 9 Prim( $G, s$ ) (allgemeine Form)

---

```
1:  $F \leftarrow \emptyset$ 
2:  $S \leftarrow \{s\}$  ▷ ZHK von  $s$  in  $F$ 
3: while  $F$  nicht Spannbaum do
4:    $u^*v^* \leftarrow \text{minimale Kante an } S \quad (u^* \in S, v^* \notin S)$ 
5:    $F \leftarrow F \cup \{u^*v^*\}$ 
6:    $S \leftarrow S \cup \{v^*\}$ 
```

---

---

### Algorithm 10 Prim( $G, s$ ) (mit min-heap)

---

```
1:  $H \leftarrow \text{make-heap}(V, \infty)$ ,  $S \leftarrow \emptyset$ 
2:  $d[s] \leftarrow 0$ ;  $d[v] \leftarrow \infty \quad \forall v \in V \setminus \{s\}$ 
3:  $\text{decrease-key}(H, s, 0)$ 
4: while  $H \neq \emptyset$  do
5:    $v^* \leftarrow \text{extract-min}(H)$ 
6:    $S \leftarrow S \cup \{v^*\}$ 
7:   for  $v^*v \in E, v \notin S$  do
8:      $d[v] \leftarrow \min\{d[v], c(v^*, v)\}$  ▷ Unterschied zu Dijkstra
9:      $\text{decrease-key}(H, v, d[v])$ 
```

---

## Union-Find Datenstruktur

Idee: verwalte alle Knoten einer ZHK als Liste

---

### Algorithm 11 Union-Find( $G$ )

---

```
1: Implementierung:
2: MAKE( $V$ ):  $\text{rep}[v] \leftarrow v \quad \forall v \in V$   $\triangleright \mathcal{O}(n)$ 
3:
4: SAME( $u, v$ ): teste ob  $\text{rep}[u] = \text{rep}[v]$   $\triangleright \mathcal{O}(1)$ 
5:
6: UNION( $u, v$ ):  $\triangleright \mathcal{O}(|\text{ZHK}(u)|)$ 
7: for  $x \in \text{members}[\text{rep}[u]]$  do
8:    $\text{rep}[x] \leftarrow \text{rep}[v]$ 
9:    $\text{members}[\text{rep}[v]] \leftarrow \text{members}[\text{rep}[v]] \cup \{x\}$ 
```

---

Idee: sichere Kanten sortiert nach Gewicht

Kruskal( $G$ ):

- Input: Graph  $G = (V, E)$  und  $s \in V$
- Output: Minimaler Spannbaum  $F$
- Brauchen eine Union-Find Datenstruktur für effiziente Laufzeit
- Laufzeit:  $\underbrace{\mathcal{O}(m \cdot \log(m))}_{\text{Sortieren}} + \underbrace{n \cdot \log(n)}_{\text{Union-Find}}$

---

### Algorithm 12 Kruskal( $G$ ) (mit UF-Datenstruktur)

---

```
1:  $F \leftarrow \emptyset$ 
2:  $UF \leftarrow \text{MAKE}(V)$   $\triangleright$  UF-Datenstruktur initialisieren
3:  $\text{SORT}(E)$   $\triangleright$  Sortiere Kanten nach Gewicht
4: for  $uv \in E$ , aufsteigend sortiert do
5:   if SAME( $u, v$ ) = false then  $\triangleright u, v$  in verschiedenen ZHKs von  $F$ 
6:      $F \leftarrow F \cup \{uv\}$ 
7:     UNION( $u, v$ )
```

---

## All pairs Shortest Path

### Floyd-Warshall( $G$ ):

- Input: Graph  $G = (V, E)$
- Output: Länge von beliebigem kürzestem Pfad.
- Laufzeit:  $\mathcal{O}(n^3)$
- Teilproblem:

$d_{uv}^i$  = Länge von kürzestem  $u - v$ -Weg, der nur Zwischenknoten aus  $\{1, \dots, i\}$  benutzen darf

---

### Algorithm 13 Floyd-Warshall( $G$ )

---

```
1: for  $u \in V$  do
2:    $d_{uu}^0 \leftarrow 0$  ▷ falls keine negativen Schleifen
3:   for  $v \in V \setminus \{u\}$  do
4:     if  $(u, v) \in E$  then
5:        $d_{uv}^0 \leftarrow c(u, v)$ 
6:     else
7:        $d_{uv}^0 \leftarrow \infty$ 
8:   for  $i = 1 \dots n$  do
9:     for  $u = 1 \dots n$  do
10:      for  $v = 1 \dots n$  do
11:         $d_{uv}^i \leftarrow \min\{d_{uv}^{i-1}, d_{ui}^{i-1} + d_{iv}^{i-1}\}$ 
return  $d^n$  ▷  $n \times n$  Resultatmatrix
```

---