

Ein Graph $G = (V, E)$ wird durch die Knotenmenge V und die Kantenmenge E repräsentiert.

- G ist **ungerichtet**, wenn wir keinen Start- und Zielpunkt der Kanten auszeichnen. Wir stellen eine Kante als die **Menge** $\{u, v\}$ ihrer Endpunkte u und v dar.
- G ist **gerichtet**, wenn jede Kante einen Start- und Zielknoten besitzt. Wir stellen eine Kante als **geordnetes Paar** (u, v) ihres Startknotens u und ihres Zielknotens v dar.

Warum Graphen? Graphen modellieren

- das **World Wide Web**: Die Knoten entsprechen Webseiten, die (gerichteten) Kanten entsprechen Hyperlinks.
- **Rechnernetzwerke**: Die Knoten entsprechen Rechnern, die (gerichteten und/oder ungerichteten) Kanten entsprechen Direktverbindungen zwischen Rechnern.
- Das **Schiennetz der Deutschen Bahn**: Die Knoten entsprechen Bahnhöfen, die (ungerichteten) Kanten entsprechen Direktverbindungen zwischen Bahnhöfen.
- **Schaltungen**: die Knoten entsprechen Gattern, die (gerichteten) Kanten entsprechen Leiterbahnen zwischen Gattern.

Von jetzt an: $V = \{1, \dots, n\}$.

In der **Adjazenzlisten-Darstellung** wird G durch ein Array A_G von Listen dargestellt:

- Die Liste $A_G[v]$ führt alle Nachbarn von v auf, bzw. alle Nachfolger von v für gerichtete Graphen.
- Maßgeschneidert für die Operationen

Bestimme alle Nachfolger des Knotens v .

Diese Operation ist zentral für viele Algorithmen wie Tiefensuche und Breitensuche, Dijkstra's Algorithmus sowie die Algorithmen von Prim und Kruskal.

Der benötigte Speicherplatz passt sich dynamisch der Größe des Graphen an.

Tiefensuche

Tiefensuche: Die globale Struktur

Die Adjazenzliste besteht aus einem Array `A_Liste` von Zeigern auf die Struktur `Knoten` mit

```
struct Knoten { int name; Knoten *next; }
```

```
void Tiefensuche()  
{  
    for (int k = 0; k < n; k++) besucht[k] = 0;  
    for (k = 0; k < n; k++)  
        if (! besucht[k]) tsuche(k);  
}
```

Jeder Knoten wird besucht!

Tiefensuche: Die Rekursion

```
void tsuche(int v)
{
    Knoten *p ;
    besucht[v] = 1;
    for (p = A_Liste [v]; p !=0; p = p->next)
        if (!besucht [p->name]) tsuche(p->name);
}
```

- Die sofortige Markierung von v verhindert einen weiteren Besuch.
- Alle nicht markierten Nachbarn werden besucht.

Tiefensuche für ungerichtete Graphen

Sei $G = (V, E)$ ein ungerichteter Graph. W_G sei der Wald der Tiefensuche für G .

- (a) Wenn $\{u, v\} \in E$ eine Kante ist und wenn $\text{tsuche}(u)$ vor $\text{tsuche}(v)$ aufgerufen wird, dann ist v ein Nachfahre von u in W_G .

Die Bäume von W_G entsprechen genau den Zusammenhangskomponenten von G .

- (b) Tiefensuche besucht jeden Knoten von V genau einmal. Die Laufzeit von Tiefensuche ist linear, also durch

$$O(|V| + |E|)$$

beschränkt.

- (c) G besitzt nur Baum- und Rückwärtskanten.

Für einen ungerichteten Graphen $G = (V, E)$ überprüfe in Zeit $O(|V| + |E|)$, ob

- G ein Baum ist,
- G zusammenhängend ist oder ob
- G bipartit ist.

Warum kann man mit Tiefensuche schnell einen Weg aus einem ungerichteten Labyrinth finden?

Tiefensuche für gerichtete Graphen

Sei $G = (V, E)$ ein gerichteter Graph.

- (a) Für jeden Knoten v in V gilt: $tsuche(v)$ wird genau die Knoten besuchen, die auf einem unmarkierten Weg mit Anfangsknoten v liegen.

Ein Weg ist unmarkiert, wenn alle Knoten vor Beginn von $tsuche(v)$ unmarkiert sind.

- (b) Die Laufzeit von $Tiefensuche()$ ist linear, also durch

$$O(|V| + |E|)$$

beschränkt.

- (c) G besitzt nur Baumkanten, Rückwärts- und Vorwärtskanten sowie Rechts-nach-Links Querkanten,
also Querkanten die von einem später zu einem früher besuchten Knoten führen.

Für einen gerichteten Graphen $G = (V, E)$ überprüfe in Zeit $O(|V| + |E|)$, ob

- es einen Weg von Knoten u nach Knoten v gibt: Das Labyrinth Problem für gerichtete Graphen,
- G kreisfrei ist oder
- ob G stark zusammenhängend ist.
 G ist stark zusammenhängend, wenn es für je zwei Knoten stets einen Weg vom ersten zum zweiten Knoten gibt.

Breitensuche

```
void Breitensuche(int v) {
    Knoten *p; int w; queue q; q.enqueue(v);
    for (int k =0; k < n ; k++) besucht[k] = 0;
    besucht[v] = 1;
    /* v wird in die Schlange eingefügt und markiert. */
    while (!q.empty ( )) {
        w = q. dequeue ( );
        for (p = A_List[w]; p != 0; p = p->next)
            if (!besucht[p->name]) {
                q.enqueue(p->name); besucht[p->name] = 1; }}}}
```

- Wenn ein Knoten u in die Schlange gestopft wird, dann wird u sofort markiert: u wird nur einmal „angefasst“.
- Jeder von v aus erreichbare Knoten wird besucht.

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph und v sein ein Knoten von G .

- (a) Breitensuche(v) besucht jeden von v aus erreichbaren Knoten genau einmal und sonst keine anderen Knoten.

Breitensuche(v) erzeugt einen Baum $T(v)$ kürzester Wege:
Wenn w von v aus erreichbar ist, dann ist der Weg in $T(v)$, von der Wurzel v zum Knoten w , ein **kürzester Weg**.

- (b) Breitensuche(v) besucht jeden von v aus erreichbaren Knoten genau einmal.

Die Laufzeit ist linear, also proportional in der Anzahl von v erreichbaren Knoten und der Gesamtzahl ihrer Kanten.

Dijkstra's Algorithmus

Kürzeste gewichtete Wege

Für einen gerichteten Graphen $G = (V, E)$, einen Knoten $s \in V$ und Kantenlängen

$$\text{länge} : E \rightarrow \mathbb{R}_{\geq 0},$$

bestimme kürzeste Wege von s zu allen anderen Knoten in V . (Web)

- Die Länge des Weges $p = (v_0, v_1, \dots, v_m)$ ist

$$\text{länge}(p) = \sum_{i=1}^m \text{länge}(v_{i-1}, v_i).$$

- Breitensuche funktioniert nur, falls $\text{länge}(e) = 1$ für alle Kanten.

Warum nicht einen kürzesten Weg von s zu **einem** Zielknoten t suchen? Alle bekannten Algorithmen finden gleichzeitig kürzeste Wege zu allen Knoten.

S-Wege

Dijkstra's Algorithmus: Die Idee

Sei $S \subseteq V$ eine Teilmenge von V mit $s \in S$. Ein **S-Weg** startet in s und durchläuft mit Ausnahme des letzten Knotens nur Knoten in S .

Angenommen,

für jeden Knoten $w \in V \setminus S$ gilt die Invariante

$\text{distanz}(w) =$ Länge eines kürzesten **S-Weges** von s nach w . ◀

Intuition: Wenn der Knoten $w \in V \setminus S$ einen **minimalen** Distanzwert unter allen Knoten in $V \setminus S$ besitzt, dann ist

ein kürzester **S-Weg** nach w auch ein kürzester Weg nach w .

Warum ist die Idee richtig?

Die Invariante gelte. Wenn $\text{distanz}[w] = \min_{u \in V \setminus S} \text{distanz}[u]$ für einen Knoten $w \in V \setminus S$ ist, dann folgt

$\text{distanz}[w] = \text{Länge eines kürzesten Weges von } s \text{ nach } w.$

- Angenommen, der Weg $p = (s, v_1, \dots, v_i, v_{i+1}, \dots, w)$ ist kürzer als $\text{distanz}[w]$.
 - ▶ Sei v_{i+1} der erste Knoten in p , der nicht zu S gehört.
 - ▶ $(s, v_1, \dots, v_i, v_{i+1})$ ist ein S -Weg nach v_{i+1} und

$$\text{distanz}[v_{i+1}] \leq \text{länge}(s, v_1, \dots, v_i, v_{i+1}) \leq \text{länge}(p) < \text{distanz}[w]$$

folgt aus der Invariante, da Kantenlängen nicht-negativ sind. ▶

- Aber

$$\text{distanz}[w] \leq \text{distanz}[v_{i+1}],$$

denn w hatte ja den kleinsten Distanz-Wert. **Widerspruch.**

Wie kann die Invariante aufrecht erhalten werden?

- Der Knoten w habe einen kleinsten Distanz-Wert.
- Angenommen w wird zur Menge S hinzugefügt.

Wie sieht ein kürzester S -Weg \mathcal{P} für $u \in V \setminus S$ aus?

- Wenn Knoten w nicht von \mathcal{P} durchlaufen wird, dann ist die Länge eines kürzesten S -Wegs unverändert.
- Sonst ist $\mathcal{P} = (s, \dots, w, v, \dots, u)$ ein kürzester S -Weg. Wenn $v \neq u$:
 - ▶ v wurde **vor** w in S eingefügt.
 - ▶ Ein kürzester S -Weg $p(v)$ von s nach v durchläuft w deshalb nicht.
 - ▶ Ersetze das Anfangsstück (s, \dots, w, v) von \mathcal{P} durch $p(v)$:
Der neue S -Weg nach u ist mindestens so kurz wie \mathcal{P} , enthält aber w nicht.

Die **neuen** kürzesten S -Wege sind von der Form (s, \dots, w, u) ◀.

Dijkstra's Algorithmus

(1) Setze $S = \{s\}$ und

$$\text{distanz}[v] = \begin{cases} \text{länge}(s, v) & \text{wenn } (s, v) \in E, \\ \infty & \text{sonst.} \end{cases}$$

(2) Solange $S \neq V$ wiederhole


(2a) wähle einen Knoten $w \in V \setminus S$ mit **kleinstem** Distanz-Wert.

(2b) Füge w in S ein.

(2c) Berechne den neuen Distanz-Wert für jeden Nachfolger $u \in V \setminus S$ von w :

$$\begin{aligned} c &= \text{distanz}[w] + \text{länge}(w, u); \\ \text{distanz}[u] &= (\text{distanz}[u] > c) ? c : \text{distanz}[u]; \end{aligned}$$

// Aktualisiere $\text{distanz}[u]$, wenn ein S -Weg über w nach u

// kürzer als der bisher kürzeste S -Weg nach u ist.  (Animation)

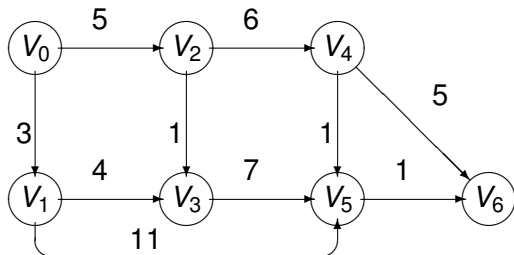
Und wo sind unsere kürzesten Wege?

Wenn wir gerade den Knoten w in die Menge S aufgenommen haben und $\text{distanz}[u]$ verringert sich, dann ...

- Vermerke, dass **gegenwärtig** ein kürzester S -Weg nach u den Knoten w besucht und dann zu u springt.
- Und wie sollen wir das vermerken?
 - ▶ Benutze einen Baum B , um kürzeste S -Wege zu speichern.
 - ▶ Implementiere B als Vater-Array:
Setze $\text{Vater}[u] = w$ genau dann, wenn sich der distanz -Wert von u verringert, wenn w zu S hinzugefügt wird.

Die Laufzeit steigt nur unmerklich an!

Ein Beispiel



Das Distanzarray:

	V ₁	V ₂	V ₃	V ₄	V ₅	V ₆
$S = \{V_0\}$	3	5	∞	∞	∞	∞
$S = \{V_0, V_1\}$	–	5	7	∞	14	∞
$S = \{V_0, V_1, V_2\}$	–	–	6	11	14	∞
$S = \{V_0, V_1, V_2, V_3\}$	–	–	–	11	13	∞
$S = \{V_0, V_1, V_2, V_3, V_4\}$	–	–	–	–	12	16
$S = \{V_0, V_1, V_2, V_3, V_4, V_5\}$	–	–	–	–	–	13

- Wir benutzen einen Min-Heap, der die Knoten in $V \setminus S$ nach ihrem distanz-Wert verwaltet.
- Die Laufzeit:
 - ▶ $|V| - 1$ mal wird nach einem Knoten in $V \setminus S$ mit kleinstem distanz-Wert gesucht.
Es gibt höchstens $|V| - 1$ Delete_Min Operationen.
 - ▶ Eine Decrease_Key Operation kann nur durch eine Kante (w, u) hervorgerufen werden.
Also gibt es höchstens $|E|$ Decrease_Key Operationen.

Die Laufzeit ist durch $O((|V| + |E|) \cdot \log_2 |V|)$ beschränkt, denn jede Heap-Operation läuft in Zeit $O(\log_2 |V|)$.

Die Laufzeit von Dijkstra's Algorithmus

Dijkstra's Algorithmus löst das „Single-Source-Shortest-Path“ Problem für gerichtete Graphen mit n Knoten, e Kanten und nicht-negativen Gewichten in Zeit

$$O((n + e) \cdot \log_2 n).$$

- Was passiert, wenn Kantengewichte negativ sind?
Wenn alle Kantengewichte mit -1 übereinstimmen, dann sind kürzeste, mit -1 gewichtete Wege längste (ungewichtete) Wege.
- Kürzeste (ungewichtete) Wege können wir blitzschnell mit Breitensuche bestimmen, **längste** (ungewichtete) Wege lassen sich hingegen **nicht effizient** bestimmen!
Warum? Das Problem, längste Wege zu bestimmen, führt auf ein NP-vollständiges Problem.

Minimale Spannbäume

Minimale Spannäume

- Sei $G = (V, E)$ ein ungerichteter, **zusammenhängender** Graph. Ein Baum $T = (V', E')$ heißt ein **Spannbaum** für G , falls $V' = V$ und $E' \subseteq E$.

Spannbäume sind die kleinsten zusammenhängenden Teilgraphen, die alle Knoten von G beinhalten.

- Die Funktion

$$\text{länge} : E \rightarrow \mathbb{R}$$

weist Kanten diesmal beliebige, auch negative Längen zu.

- ▶ Für einen Spannbaum $T = (V, E')$ definieren wir

$$\text{länge}(T) = \sum_{e \in E'} \text{länge}(e)$$

als die Länge von T .

- ▶ Berechne einen **minimalen Spannbaum** T , also einen Spannbaum minimaler Länge unter allen Spannbäumen von G .

Ein Kommunikationsnetzwerk zwischen einer Menge von Zentralen ist zu errichten.

- Für die Verbindungen zwischen den Zentralen müssen Kommunikationsleitungen gekauft werden, deren Preis von der Distanz zwischen den zu verbindenden Zentralen abhängt.
 - ▶ Einzige Bedingung: Je zwei Zentralen müssen indirekt (über andere Zentralen) miteinander kommunizieren können.
- Wie sieht ein billigstes Kommunikationsnetzwerk aus?
 - ▶ Repräsentiere jede Zentrale durch einen eigenen Knoten.
 - ▶ Zwischen je zwei Knoten setzen wir eine Kante ein und markieren die Kante mit der Distanz der entsprechenden Zentralen.
 - ▶ Wenn G der resultierende Graph ist, dann müssen wir einen billigsten zusammenhängenden Teilgraphen, also einen **minimalen Spannbaum** für G suchen. (Web)

Das **Traveling Salesman Problem** (TSP):

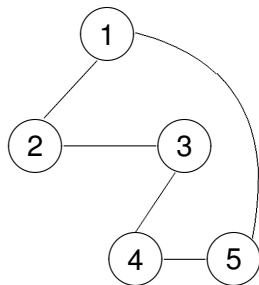
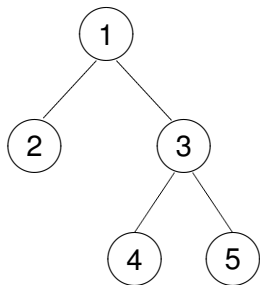
- Ein Handlungsreisender muss, von seinem Ausgangspunkt aus, eine Menge von Städten in einer Rundreise besuchen.
- **Das Ziel:** Bestimme eine Rundreise kürzester Länge. (Web)

Die **Heuristik des minimalen Spannbaums**:

- Jeder Stadt wird ein Knoten zugeordnet.
- Für jedes Paar (s_1, s_2) von Städten wird eine Kante eingesetzt und mit der Distanz zwischen s_1 und s_2 beschriftet.

Bestimme einen minimalen Spannbaum T und durchlaufe T in Präorder-Reihenfolge. Wie lang ist die erhaltene Rundreise?

Eine Rundreise in Präorder-Reihenfolge



Wie lang ist unsere Rundreise **höchstens**?

$$d_{1,2} + (d_{2,1} + d_{1,3}) + d_{3,4} + (d_{4,3} + d_{3,5}) + (d_{5,3} + d_{3,1}).$$

Alle Kanten des Spannbaums treten genau zweimal auf. Es ist

$$\text{Länge}(T) \leq \text{minimale Länge einer Rundreise} \leq 2 \cdot \text{Länge}(T)$$

Unsere Rundreise ist höchstens doppelt so lang wie eine kürzeste Rundreise!

Kreuzende Kanten

Kreuzende Kanten

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. Die Kante $e = \{u, v\}$ kreuzt die Knotenmenge $S \subseteq V$, wenn genau ein Endpunkt von e in S liegt.

Kürzeste kreuzende Kanten

Der Wald $W = (V, E')$ sei in einem minimalen Spannbaum T für G enthalten. Keine Kante von W möge die Menge S kreuzen. Wenn e eine **kürzeste S -kreuzende Kante** ist, dann ist auch $W' = (V, E' \cup \{e\})$ in einem minimalen Spannbaum für G enthalten.

Warum? Wenn T die Kante e enthält, dann ist W' weiterhin in T enthalten. Ansonsten:

- Füge e zu T hinzu. Ein Kreis wird geschlossen, der mindestens eine weitere S -kreuzende Kante e^* besitzt.
- $\text{Länge}(e) \leq \text{Länge}(e^*)$. Ersetze e^* in T durch e und wir erhalten einen Spannbaum, der W' enthält und nicht länger als T ist.

Minimale Spannäume: Die Idee

Baue einen minimalen Spannbaum, indem **kürzeste** kreuzende Kanten für *geeignete* Knotenmengen $S \subseteq V$ eingesetzt werden.

Welche Knotenmengen S sind „geeignet“? Wir verfolgen zwei Ansätze.

- **Der Algorithmus von Prim** lässt einen einzigen Baum „Kante für Kante“ wachsen.
 - ▶ Wenn der bisherige Baum die Knotenmenge S hat, dann wird eine kürzeste S -kreuzende Kante hinzugefügt.
- **Der Algorithmus von Kruskal** beginnt mit einem Wald aus Einzelknoten.
 - ▶ Die Kanten werden nach aufsteigender Länge sortiert und in dieser Reihenfolge abgearbeitet.
 - ▶ Füge Kante $e = \{u, v\}$ in den Wald ein, wenn kein Kreis geschlossen wird.
 - ▶ e kreuzt z.B. die Knotenmenge des Baums, der u enthält. ◀ (Web)

Der Algorithmus von Prim

Der Algorithmus von Prim

(1) Setze $S = \{0\}$ und $E' = \emptyset$.

// Wir beginnen mit einem Baum, der nur aus dem Knoten 0
// besteht.

(2) Solange $S \neq V$, wiederhole:

(2a) Bestimme eine kürzeste S -kreuzende Kante $e = \{u, v\} \in E$ mit
 $u \in S$ and $v \in V \setminus S$. Es gilt also

$$\text{länge}(e) = \min\{\text{länge}(e') \mid e' \in E \text{ und } e' \text{ kreuzt } S\}.$$

(2b) Setze $S = S \cup \{v\}$ und $E' = E' \cup \{e\}$.

// Der bisherige Baum wird um die kürzeste kreuzende Kante
// erweitert. (Animation)

Der Algorithmus von Prim: Die Datenstruktur

Angenommen, wir kennen für jeden Knoten $v \in V \setminus S$ die Länge $e(v)$ einer kürzesten kreuzende Kante mit Endpunkt v .

- Benutze einen Min-Heap, um die Knoten $v \in V \setminus S$ gemäß ihren Prioritäten $e(v)$ zu verwalten: Berechne eine kürzeste kreuzende Kante mit Hilfe der Deletemin-Operation.
- Wenn jetzt $w \in V \setminus S$ in die Menge S eingefügt wird,
 - ▶ können nur die Nachbarn w' von w eine neue kürzeste kreuzende Kante, nämlich die Kante $\{w, w'\}$, erhalten.
 - ▶ Für jeden Nachbarn w' von w setze deshalb

$$e(w') = (e(w') > \text{länge}(\{w, w'\})) ? \text{länge}(\{w, w'\}) : e(w');$$

wobei anfänglich

$$e(w) = \begin{cases} \text{länge}(\{0, w\}) & \{0, w\} \text{ ist eine Kante,} \\ \infty & \text{sonst.} \end{cases}$$

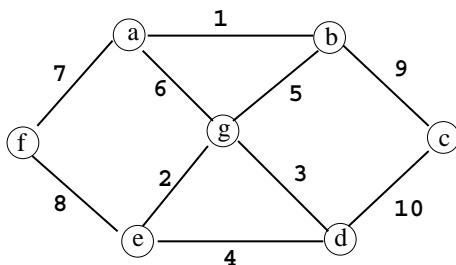
Der Algorithmus von Prim: Zusammenfassung

- Ein Spannbaum für einen Graphen mit n Knoten besteht aus $n - 1$ Kanten.
 - ▶ Insgesamt sind somit genau $n - 1$ Deletemin-Operationen notwendig.
- Aktualisierungen der Kantenlängen $e(v)$ werden durch die mit v inzidenten Kanten erzwungen.
 - ▶ Also gibt es höchstens $e = |E|$ Aktualisierungen.
- Jede Heap-Operation, ob ein Deletemin oder eine Aktualisierung, läuft in Zeit $O(\log_2 n)$.

Der Algorithmus von Prim: Die Laufzeit

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum für einen Graphen mit n Knoten und e Kanten in Zeit $O((n + e) \cdot \log_2 n)$.

Der Algorithmus von Prim: Ein Beispiel



Das Array e für den Startknoten g :

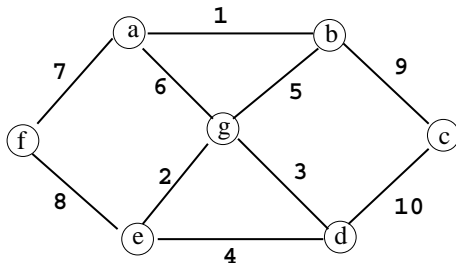
	a	b	c	d	e	f
$S = \{g\}$	6	5	∞	3	2	∞
$S = \{e, g\}$	6	5	∞	3	—	8
$S = \{d, e, g\}$	6	5	10	—	—	8
$S = \{b, d, e, g\}$	1	—	9	—	—	8
$S = \{a, b, d, e, g\}$	—	—	9	—	—	7
$S = \{a, b, d, e, f, g\}$	—	—	9	—	—	—

Der Algorithmus von Kruskal

Der Algorithmus von Kruskal

- (1) Sortiere die Kanten gemäß aufsteigender Länge.
Sei $W = (V, E')$ der leere Wald, also $E' = \emptyset$.
// Kruskal beginnt mit dem leeren Wald W , der nur aus
// Einzelknoten besteht.
- (2) Solange W kein Spannbaum ist, wiederhole
 - (2a) Nimm die gegenwärtige kürzeste Kante e und entferne sie aus der sortierten Folge.
 - (2b) Verwerfe e , wenn e einen Kreis in W schließt.
// Die Kante e verbindet zwei Knoten desselben Baums von W .
 - (2c) Ansonsten akzeptiere e und setze $E' = E' \cup \{e\}$.
// Die Kante e verbindet zwei Bäume T_1, T_2 von W .
// Sie wird zu W hinzugefügt. Wenn S die Knotenmenge von T_1 ist,
// dann ist e eine kürzeste S -kreuzende Kante.

Der Algorithmus von Kruskal: Ein Beispiel



Die Wälder in Kruskal's Algorithmus:

- Zu Anfang: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}$.
- $\{a, b\}$ mit dem neuen Wald $\{a, b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}$
- $\{e, g\}$ mit dem neuen Wald $\{a, b\}, \{c\}, \{d\}, \{e, g\}, \{f\}$
- $\{d, g\}$ mit dem neuen Wald $\{a, b\}, \{c\}, \{d, e, g\}, \{f\}$
- $\{b, g\}$ mit dem neuen Wald $\{a, b, d, e, g\}, \{c\}, \{f\}$
- $\{a, f\}$ mit dem neuen Wald $\{a, b, d, e, f, g\}, \{c\}$
- $\{b, c\}$ mit dem neuen Wald $\{a, b, c, d, e, f, g\}$

Der Algorithmus von Kruskal: Die Datenstruktur

- Wir überprüfen, ob die Kante $e = \{u, v\}$ einen Kreis schließt.
- Deshalb erfinden wir die **union-find Datenstruktur**, die die Operationen `union(i, j)` und `find(u)` unterstützt:
 - ▶ `find(u)` bestimmt die Wurzel des Baums, der u als Knoten besitzt.
 - ▶ `union(i, j)` vereinigt die Knotenmenge des Baums mit Wurzel i mit der Knotenmenge des Baums mit Wurzel j .

$e = \{u, v\}$ schließt genau dann keinen Kreis, wenn

$$\text{find}(u) \neq \text{find}(v).$$

Ist dies der Fall, dann wende die union-Operation an:

Vereinige die Knotenmenge des Baums von u mit der Knotenmenge des Baums von v .

Die Grundidee: Implementiere

- die union-Operation durch ein Anhängen der Wurzel des einen Baums unter die Wurzel des anderen Baums und
- die find-Operation durch ein Hochklettern im Baum.

Aber wir können doch nicht einfach einen Baum an einen anderen anhängen: Der entstehende Wald entspricht nicht mehr dem von Kruskal berechneten Baum!

Wir benötigen zwei Datenstrukturen:

- Zuerst stellen wir genau den von Kruskal berechneten Baum dar.
- Die zweite Datenstruktur ist für die Unterstützung der union- und find-Operationen zuständig: Nur die Knotenmengen der einzelnen Bäume werden dargestellt.

- Wir stellen die Knotenmengen der einzelnen Bäume mit einem Vater-Array dar: Anfänglich ist

$$\text{Vater}[u] = u \quad \text{für alle Knoten } u \in V.$$

- Generell: u ist genau dann eine Wurzel, wenn $\text{Vater}[u] = u$.

- Einen find-Schritt führen wir aus,
 - ▶ indem wir den Baum mit Hilfe des Vater-Arrays hochklettern bis die Wurzel gefunden ist.
 - ▶ Die Zeit ist höchstens proportional zur Tiefe des Baums.
- Wie garantieren wir, dass die Bäume nicht zu tief werden?
 - ▶ Hänge in einem Union-Schritt stets die Wurzel des **kleineren** Baumes **unter** die Wurzel des **größeren** Baumes.

Die Analyse eines Union-Schritts

Wir beobachten einen beliebigen Knoten v , während Kruskal einen minimalen Spannbaum aus einem Wald von Einzelbäumen baut.

- v beginnt mit Tiefe 0, denn v ist Wurzel seines eigenen „Einzelbäumchens“.
- Seine Tiefe vergrößert sich nur dann um 1, wenn v dem kleineren Baum in einer union-Operation angehört. Also:
die Tiefe von v vergrößert sich nur dann um 1, wenn sich der Baum von v in der Größe mindestens verdoppelt.

Das Fazit

- Die Tiefe aller Bäume ist durch $\log_2(|V|)$ beschränkt.
- Eine union-Operation benötigt nur konstante Zeit, während ein find-Schritt höchstens logarithmische Zeit benötigt.

Der Algorithmus von Kruskal: Zusammenfassung

Es gibt höchstens zwei find-Operationen und eine union-Operation pro Kante, nämlich

- um herauszufinden, ob ein Kreis geschlossen wird (**find**),
- und um Bäume zu vereinigen (**union**), wenn kein Kreis geschlossen wird.

Der Algorithmus von Kruskal: Die Laufzeit

- Es gibt höchstens $|E|$ viele find-Operationen und $n - 1$ union-Operationen.
- Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum für einen Graphen mit n Knoten und e Kanten in Zeit $O(n + e \cdot \log_2 n)$.