Algorithmen und Datenstrukturen

Vorlesung #08 – Minimale Spannbäume

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

06 · Jun · 2023

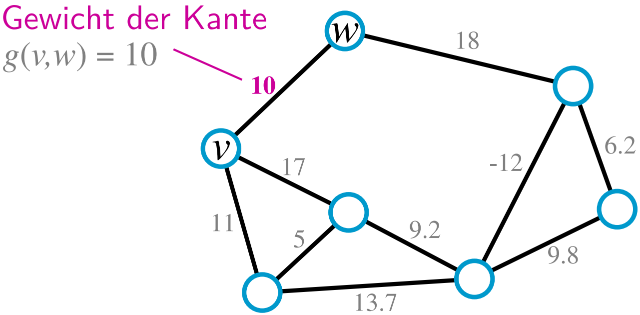
Themen der heutigen Vorlesung

- Gewichtete Graphen
- Bestimmung **minimaler Spannbäume** (*minimal spanning tree*, MST) in gewichteten ungerichteten Graphen
- Schnitte und kreuzende Kanten
- Ansätze zur Bestimmung von MST
  - Generischer Ansatz
  - Prims Algorithmus
  - *Reverse-delete* Algorithmus
  - Kruskals Algorithmus
- `UnionFind` Algorithmus, von einfach zu effizient zur Verwendung in Kruskal

Gewichtete Graphen

- Für die Modellierung vieler praktischer Probleme ist es wichtig, den Kanten eines Graphen Gewichte (oder Kosten) zuordnen zu können.
- Dies können z. B. die Längen der Streckenabschnitte oder die Fahrzeiten sein.
- Dafür benutzen wir **gewichtete Graphen** und in der nächsten Vorlesung gewichtete Digraphen.
- Jeder Kante des Graphen wird ein Gewicht zugeordnet. Man kann auch Knoten-gewichtete Graphen betrachten.

Kantengewichtete Graphen



- Formal schreiben wir $g(v, w)$ oder $weight(v,w)$ für das **Gewicht der Kante** (v, w).
- Es sind auch negative Gewichte erlaubt.

Änderung in der Implementation gegenüber ungewichteten Graphen:

- Klasse für gewichtete Kanten, die die beiden Knoten, sowie das Gewicht enthält.
- Statt Adjazenzlisten: Inzidenzlisten, mit inzidenten Kanten für jeden Knoten
- Jede Kante ist doppelt gespeichert, in den Inzidenzlisten beider Knoten.

API für einen kantengewichteten Graphen

API für eine gewichtete Kante

public class Edge implements Comparable<Edge>

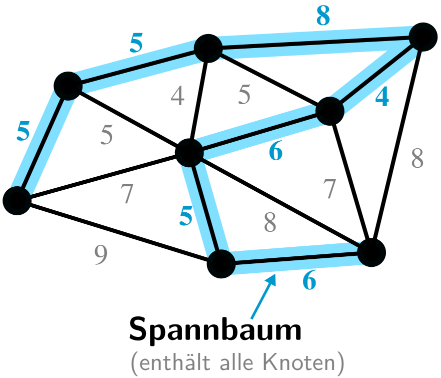
	`Edge(int v, int w, double weight)`	Kante v-w mit Gewicht weight
`double`	`weight()`	Gewicht der Kante
`int`	`either()`	Einer der beiden Knoten dieser Kante
`int`	`other(int v)`	Der Knoten, der nicht v ist
`int`	`compareTo(Edge that)`	Vergleich nach Gewicht

API für einen kantengewichteten Graphen

public class EdgeWeightedGraph

	`EdgeWeightedGraph(int V)`	Erzeugt leeren Graphen mit V Knoten
`int`	`V()`	Anzahl der Knoten
`void`	`addEdge(Edge e)`	Füge Kante e zum Graphen hinzu
`Iterable<Edge>`	`incident(int v)`	Kanten inzident zu v
`Iterable<Edge>`	`edges()`	Alle Kanten dieses Graphen

Gewichteter Graph



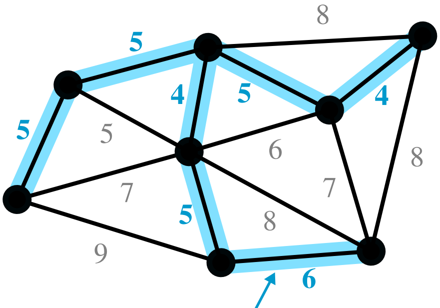
Gewicht des Spannbaums:

$5 + 5 + 8 + 4 + 6 + 5 + 6 = \mathbf{39}$

Ein Spannbaum hat immer V-1 Kanten.
Gibt es einen mit weniger Gewicht?

- Die Bestimmung minimaler Spannbäume (MST) ist für viele Anwendungen wichtig.
- Ein Graph mit V Knoten und allen möglichen Kanten hat V^{V-2} Spannbäume (Cayley, 1889). Wir brauchen also einen effizienten Algorithmus, kein *brute force*.

Gewichteter Graph



Spannbaum

(enthält alle Knoten)

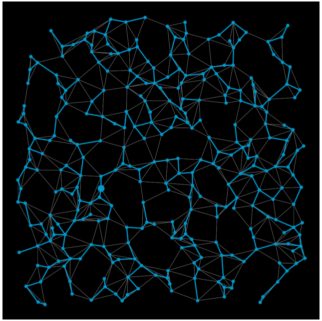
Gewicht des Spannbaums:

$5 + 5 + 4 + 4 + 5 + 5 + 6 = \mathbf{34}$

Minimaler Spannbaum?

- Die Bestimmung minimaler Spannbäume (MST) ist für viele Anwendungen wichtig.
- Ein Graph mit V Knoten und allen möglichen Kanten hat V^{V-2} Spannbäume (Cayley, 1889). Wir brauchen also einen effizienten Algorithmus, kein *brute force*.

Minimale Spannbäume in der Praxis



- Der erste MST-Algorithmus geht auf eine Arbeit des Tschechischen Mathematikers Borůvka von 1926 zurück.
- Entwickelt zur optimierten Planung des elektrischen Netzwerkes von Mähren (südöstlicher Teil der Tschechischen Republik).
- Heute werden minimale Spannbäume in vielen Bereichen eingesetzt, z.B. Clusteranalyse, Gesichtserkennung, Bildbearbeitung, allgemeines Netzwerkdesign.
- Die Algorithmen werden auch zur approximativen Lösung von NP-harten Problemen benutzt.

Spannendes über Spannbäume

Sei $\boldsymbol{T} = (\boldsymbol{V}, \boldsymbol{E}')$ ein Subgraph von $\boldsymbol{G}(\boldsymbol{V}, \boldsymbol{E})$. Dann ist $\boldsymbol{T}$ genau dann ein Spannbaum von $\boldsymbol{G}$, wenn eine (und damit alle) der folgenden, äquivalenten Bedingungen erfüllt ist:

- $\boldsymbol{T}$ ist azyklisch und zusammenhängend, d. h. T ist ein Baum..
- $\boldsymbol{T}$ hat $V - 1$ Kanten und ist zusammenhängend.
- $\boldsymbol{T}$ hat $V - 1$ Kanten und ist azyklisch.
- $\boldsymbol{T}$ ist azyklisch und das Hinzufügen einer beliebigen Kante erzeugt einen Zyklus.
- $\boldsymbol{T}$ ist zusammenhängend und das Entfernen einer beliebigen Kante macht ihn unzusammenhängend.
- Jedes Knotenpaar von $\boldsymbol{T}$ ist durch genau einen einfachen Pfad verbunden.

Allgemeines Vorgehen

- Das globale Optimierungsproblem kann durch iterierte Sicherstellung einer lokalen Eigenschaft mit einem *greedy* Ansatz gelöst werden.
- Wir werden zunächst einen allgemeinen Ansatz formulieren und dann zu konkreten Algorithmen für eine geeignete Kantenauswahl fortschreiten.
- Wir setzen im Folgenden voraus, dass der gegebene Graph zusammenhängend ist.
  - Andernfalls wendet man die Algorithmen auf seine Zusammenhangskomponenten an und erhält einen minimalen Spannwald.

Schnitte durch Graphen

- Ein **Schnitt** durch einen Graphen $G = (V, E)$ teilt seine Knoten in zwei nicht-leere Teilgraphen.

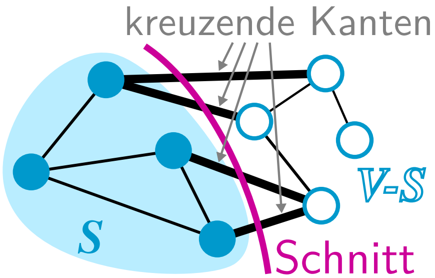
- Konkret beschreiben wir einen Schnitt durch eine Teilmenge $S \subseteq V$ mit der Eigenschaft, dass (S, E_S) und $(V - S, E_{V-S})$ nicht-leere Graphen sind.

- Dabei bezeichnet E_S die Menge aller derjenigen Kanten von G, die in S liegen:

$$E_S := \{(v, w) \in E \mid v \in S \;\&\; w \in S\}$$

- Kanten mit einem Knoten innerhalb und einem Knoten außerhalb von S heißen **kreuzende Kanten**.

- Diejenigen mit minimalem Gewicht, heißen **minimal kreuzende Kanten**.



Schnitteigenschaft

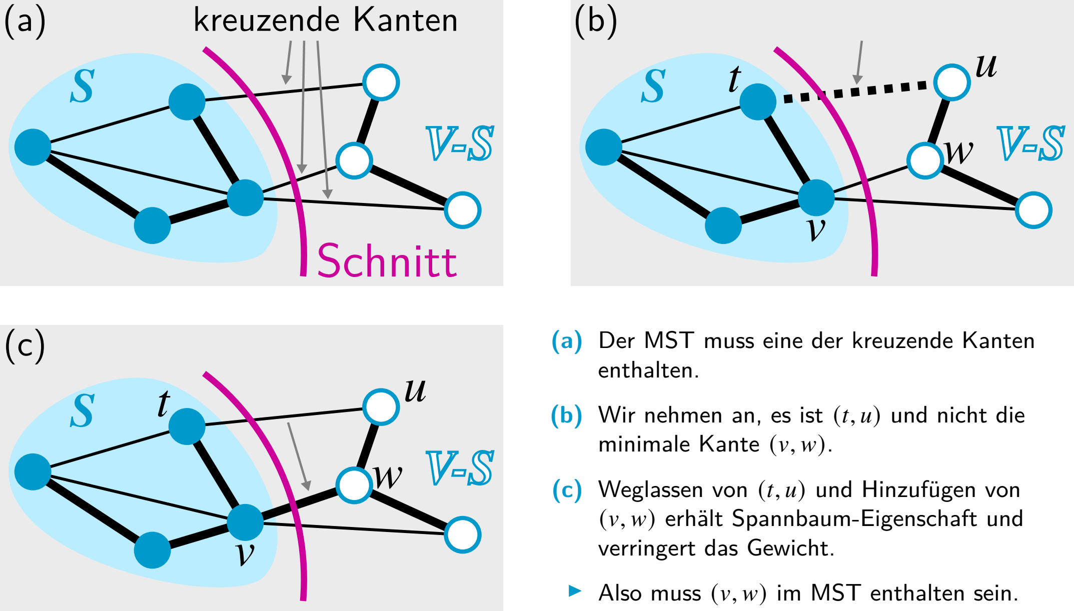
Schnitteigenschaft

Sei ein beliebiger Schnitt durch einen Graphen gegeben. Jeder minimale Spannbaum des Graphen muss eine der minimal kreuzenden Kanten enthalten.

Beweis.

- Wir nehmen an, dass es einen MST gibt, der zu dem gegebenen Schnitt keine der minimal kreuzenden Kanten enthält.
- Sei (v, w) eine minimal kreuzende Kante und $\boldsymbol{B}$ der MST plus Kante (v, w). Da der MST schon alle Knoten verbunden hat, entsteht so ein Zyklus und
- ... es muss eine weitere kreuzende Kante des MST geben, die an dem Zyklus beteiligt ist, sagen wir (t, u).
- Wenn (t, u) aus $\boldsymbol{B}$ entfernt wird, bleibt $\boldsymbol{B}$ zusammenhängend und ist also ein Spannbaum.
- Da (v, w) ein kleineres Gewicht als (t, u) hat, ist der neue Spannbaum leichter als der ursprüngliche, was der Annahme widerspricht, dass es ein MST war. $\square$

Schnitteigenschaft – Beweis in Bildern (etwas vereinfacht)



(a) Der MST muss eine der kreuzende Kanten enthalten.

(b) Wir nehmen an, es ist (t, u) und nicht die minimale Kante (v, w).

(c) Weglassen von (t, u) und Hinzufügen von (v, w) erhält Spannbaum-Eigenschaft und verringert das Gewicht.

- Also muss (v, w) im MST enthalten sein.

Allgemeiner Ansatz für minimal Spannbäume

Listing 1: **Generischer Algorithmus zur Bestimmung eines MST**

```
ME ← ∅
while ME ist kein Spannbaum
    wähle einen Schnitt S, der keine kreuzende Kante in ME hat
    füge eine minimal kreuzende Kante (v, w) zu ME hinzu
end
// ME ist ein minimaler Spannbaum
```

- Bei diesem allgemein Ansatz bleibt offen, wie der Schnitt S zu wählen ist.

Korrektheit des allgemeinen Ansatzes

Der MST Algorithmus in Listing 1 bestimmt einen minimalen Spannbaum

Beweis.

- Nach der Schnitteigenschaft (Seite 10) ist jede hinzugefügte Kante Teil des MST.
- Zu zeigen bleibt, dass alle Kanten des MST ausfindig gemacht werden.
- Wäre dies nicht der Fall, dann stellen die Kanten ME keinen Spannbaum von G dar.
- Es gibt also mindestens eine nicht leere Zusammenhangskomponente von ME, die nicht ganz G ist.
- Wir wählen eine dieser Zusammenhangskomponenten als S. Für den durch S definierten Schnitt sind keine der kreuzenden Kanten in ME. Also würde der Pseudocode eine weitere Kante auswählen. □

[Sedgewick & Wayne 2013, S. 650]

Konkrete Algorithmen für minimale Spannbäume

- **Prims Algorithmus**
  Bilde einen Baum ausgehend von einem Startknoten s. Füge iterativ eine der kreuzenden Kanten mit geringstem Gewicht hinzu.

- ***Reverse-Delete* Algorithmus**
  Durchlaufe die Kanten nach absteigendem Gewicht. Lösche eine Kante, falls sie den Graphen nicht unzusammenhängend macht.

- **Kruskals Algorithmus**
  Durchlaufe die Kanten nach aufsteigendem Gewicht. Füge eine Kante hinzu, wenn sie keinen Zyklus mit den bisher gewählten Kanten bildet.

- **Borůvkas Algorithmus** (wird hier nicht besprochen)
  Starte mit dem Wald der Einzelknoten und füge iterativ eine minimal kreuzende Kante jeder Zusammenhangskomponente hinzu.

Prims Algorithmus für minimale Spannbäume

- Markiere den zufällig gewählten Startknoten.
- Wähle eine minimal kreuzende Kante bezüglich des Schnittes, der durch den markierten Bereich definiert wird (Startknoten markiert, Endknoten nicht)
- Iteriere dies, bis $V - 1$ Kanten ausgewählt wurden.

Wie wird die Kante ausgewählt?

- Alle Kanten zu durchsuchen, hätte eine Laufzeit in $\boldsymbol{O}(E)$.
- Effizienter: kreuzende Kanten in eine Vorrangwarteschlange einzufügen, sobald sie entdeckt werden und Entnahme gemäß geringstem Gewicht:
  - Laufzeit in $\boldsymbol{O}(\log E)$ für einfügen und entnehmen.

Erste Implementation von Prims Algorithmus für MST

```
public class MSTPrim {
  protected double[] dist;
  protected PriorityQueue<Edge> pq;

  public MSTPrim(WeightedGraph G) {
    marked = new boolean[G.V()];
    parent = new int[G.V()];  // MST
    pq = new PriorityQueue<Edge>(G.E());

    visit(G, 0);              // Startknoten 0
    while (!pq.isEmpty()) {
      // hole minimal kreuzende Kante aus pq
      Edge e = pq.poll();
      int v = e.either();
      int w = e.other(v);
      // prüfe, ob w immer noch unmarkiert,
      // also e eine kreuzende Kante ist
      if (!marked[w]) {
        parent[w] = v;
        visit(G, w);
      } else if (!marked[v]) {  // dito für v
        parent[v] = w;
        visit(G, v);
      }
    }
  }
```

```
  public void visit(WeightedGraph G, int v)
  {
    marked[v] = true;
    for (Edge e : G.incident(v)) {
      int w = e.other(v);
      // nur kreuzende Kanten in pq speichern
      if (!marked[w])
        pq.add(e);
    }
  }
}
```

- Diese Implementation speichert den MST des Graphen G in dem Vorgängerarray `parent`.

Korrektheit des Prim Algorithmus

Korrektheit des Prim Algorithmus mit Priority Queue

Die Implementation des Prim Algorithmus mit Priority Queue bestimmt einen minimalen Spannbaum in einer Laufzeit in $\boldsymbol{O}(E \log E)$ und Speicherbedarf in $\boldsymbol{O}(V+E)$.

Beweis.

- Gemäß allgemeinem Ansatz (Seite 13) bleibt zu zeigen: es wird immer eine minimal kreuzende Kante auswählt.
- Die Menge der markierten Knoten definiert einen Schnitt.
- In die Warteschlange werden nur kreuzende Kanten eingefügt (Zeile 33-34), und von diesen wird per PQ diejenige mit geringstem Gewicht ausgewählt (Zeile 13).
- Bei der Auswahl werden diejenigen Kanten verworfen, deren zweiter Knoten mittlerweile markiert wurde (Zeilen 18 und 21).
- Daher wird immer eine minimal kreuzende Kante ausgewählt. □ (Korrektheit)

Laufzeit des Prim Algorithmus mit Priority Queue

- Die Initialisierung hat eine Laufzeit in $\boldsymbol{O}(V)$.
- `visit()` wird für jeden Knoten einmal aufgerufen.
  Denn der Aufruf geschieht nur für *unmarkierte* Knoten, der sodann markiert wird.
- Daher werden Zeilen 31-34 höchstens E-mal aufgerufen und jede Kante gelangt höchstens einmal in die Warteschlange.
- PQ-Methoden `isEmpty()`, `poll()`, `add()` werden also max. E-mal aufgerufen.
- Methoden `isEmpty()` und `poll()` haben konstante Laufzeit.
- Da höchstens E Elemente (Kanten) in die Warteschlange eingefügt werden, hat `add()` eine Laufzeit in $\boldsymbol{O}(\log E)$, siehe Vorlesung #3.
- Insgesamt ergibt sich eine Laufzeit in $\boldsymbol{O}(E \log E)$.
  Da wir den Graphen als zusammenhängend angenommen haben, gilt $E \geq V - 1$.

Verbesserte Implementierung mit IndexPQ

- Der Prim Algorithmus kann durch Verwendung einer indizierten Vorrangwarteschlange beschleunigt werden.
- Bei der bisherigen Implementierung gelangen *alle* kreuzenden Kanten in die Warteschlange und verbleiben dort, bis sie abgerufen werden.
- Dies ist ineffizient:
  - Die PQ enthält auch Kanten, die gar nicht mehr kreuzen.
  - Wir benötigen pro Knoten nur eine ihn verbindende Kante mit minimalem Gewicht, nicht alle.
- Daher reicht es, für jeden Knoten, die kürzeste Kante unter den bisher bekannten Kanten zu speichern, bzw. deren Gewicht: knoten-indiziertes Array `weightOfCE`.
- Dadurch wird auch der Speicherbedarf von $\boldsymbol{O}(E)$ auf $\boldsymbol{O}(V)$ reduziert.

Implementation von Prims Algorithmus mit IndexPQ

```java
public class MSTPrim
{ //Gewicht der min kreuzenden Kante zum Knoten
  protected double[] weightOfCE;
  protected IndexPQ<Double> pq;

  public MSTPrim(WeightedGraph G) {
    marked = new boolean[G.V()];
    parent = new int[G.V()];
    weightOfCE = new double[G.V()];
    for (int v = 0; v < G.V(); v++) {
      weightOfCE[v] = Double.POSITIVE_INFINITY;
    }
    weightOfCE[s] = 0;
    pq = new IndexPQ<Double>(G.V(), -1);
    pq.add(s, weightOfCE[s]);

    while (!pq.isEmpty()) {
      int v = pq.poll();
      visit(G, v);
    }
  }
```

```java
  public void visit(WeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e : G.incident(v)) {
      int w = e.other(v);
      if (marked[w])
        continue;
      // kreuzende Kante gefunden
      // ist es eine minimale?
      if (e.weight() < weightOfCE[w]) {
        parent[w] = v;
        weightOfCE[w] = e.weight();
        if (pq.contains(w)) {
          pq.change(w, weightOfCE[w]);
        } else {
          pq.add(w, weightOfCE[w]);
        }
      }
    }
  }
}
```


[Sedgewick & Wayne S. 664]

Analyse der Prim Implementation mit indizierter PQ

Korrektheit und Laufzeit des Prim Algorithmus mit IndexPQ

Diese Implementation bestimmt einen MST in einer Laufzeit in $\boldsymbol{O}(E \log V)$ und mit Speicherbedarf in $\boldsymbol{O}(V)$.

Beweis.

- Die Korrektheit ergibt sich wie zuvor. Laufzeit:
- Die Methode `visit()` wird für jeden Knoten maximal einmal aufgerufen.
- Die `for` Schleife in `visit()` wird also höchstens E-mal durchlaufen.
- Die Warteschlange enthält jeden Knoten höchstens einmal. Für schon vorhandene Knoten wird nur ggf. das Gewicht aktualisiert.
- Die Laufzeit der Aufrufe `change()` und `add()` ist also in $\boldsymbol{O}(\log V)$.
  Bei einer *indizierten* PQ hat `contains()` konstante Laufzeit.
- Insgesamt erhalten wir somit eine Laufzeit in $\boldsymbol{O}(E \log V)$.

Erinnerung an zwei Eigenschaften von Spannbäumen

Sei $T = (V, E')$ ein Subgraph von $G(V, E)$. Dann ist T genau dann ein Spannbaum von G, wenn eine (und damit alle) der folgenden, äquivalenten Bedingungen erfüllt ist:

- T ist zusammenhängend und das Entfernen einer beliebigen Kante macht ihn unzusammenhängend.
- T ist azyklisch und das Hinzufügen einer beliebigen Kante erzeugt einen Zyklus.

Der unbekanntere Kruskal Algorithmus: *Reverse-Delete*

Der Artikel [Kruskal 1956] enthält außer dem bekannten Kruskal Algorithmus, eine weitere Variante: *reverse-delete*. Sie funktioniert so:

- Beginne mit dem gegebenen Graphen.
- Durchlaufe die Kanten in der Reihenfolge nach absteigendem Gewicht.
- Falls der Graph durch Entfernen der Kante zusammenhängend bleibt, entferne sie.
- Andernfalls erhalte sie und fahre mit der nächsten Kante fort bis alle Kanten durchlaufen wurden.
- Eine effiziente Implementierung insbesondere des Tests auf Zusammenhang ist komplex, daher besprechen wir den Algorithmus nicht weiter.
- Nur als Hinweis: Mit [Thorup 2000] erreicht man eine Laufzeit in $\boldsymbol{O}(E \log V (\log\log V)^3)$.

[https://en.wikipedia.org/wiki/Reverse-delete_algorithm]

Kruskals Algorithmus für minimale Spannbäume

- Der prominente **Kruskal Algorithmus** geht umgekeht vor.
- Die Kanten werden in der Reihenfolge nach aufsteigendem Gewicht durchlaufen.
- In jedem Schritt wird die aktuelle Kante ggf. zu einer Menge MST hinzugefügt, die am Ende den minimalen Spannbaum bildet.
- Es wird geprüft, ob das Hinzufügen der Kante zu einem Zyklus in MST führt. Nur, wenn dies nicht der Fall ist, wird sie ausgewählt.
- Der Ablauf ist völlig anders, als bei den bisher besprochenen Algorithmen. Die ausgewählten Kanten bilden zunächst keinen Baum, der schrittweise vergrößert wird. Daher können die Kanten beim Ablauf nicht wie bisher in einem Knoten-indizierten `parent` Array gespeichert werden.

Überlegungen zur Implementierung von Kruskals Algorithmus

- Obwohl der Algorithmus einfach zu implementieren aussieht, gibt es einen Haken für eine halbwegs effiziente Lösung.
- Das Problem ist der Test, ob eine Kante (v, w) einen Zyklus erzeugen würde, der in jedem Schritt durchgeführt werden muss.
- Die Prüfung auf Zyklen mit Tiefensuche hat eine Laufzeit in $\boldsymbol{O}(V+E)$.
- Hier ginge der Test einfacher, da wir nur mit Tiefensuchen prüfen müssen, ob w von v aus erreichbar ist. Dies hat eine Laufzeit in $\boldsymbol{O}(V)$.
- Da bei Kruskal dieser Test für jede Kante, also E-mal, durchgeführt werden muss, ergibt sich eine Gesamtlaufzeit von $\boldsymbol{O}(VE)$: zu langsam!
- Der Graph, der während des Kruskal Algorithmus auf Zyklen geprüft wird, ändert sich bei jedem Schritt nur um eine Kante.
- Anstatt die Zyklenprüfung jedes Mal naiv neu zu starten, können wir schrittweise Strukturen aufbauen, die eine schnelle Prüfung ermöglichen.

Dynamisch Prüfung auf Zyklen mit `UnionFind`

- Die Frage, ob durch Hinzufügung der Kante (v, w) ein Zyklus entsteht, ist äquivalent zu der Frage, ob v und w bereits in derselben Zusammenhangskomponente liegen.
- Bei Benutzung der Identifikation von Zusammenhangskomponenten mit Tiefensuche (Vorlesung #07) wäre in Bezug auf Laufzeit nichts gewonnen.
- Wir führen daher die Datenstruktur `UnionFind` ein, die für die dynamische Verwaltung von Zusammenhangskomponenten bestens geeignet ist:

API für Union-Find (dynamische Verwaltung von Zusammenhangskomponenten)

public class UnionFind

	`UnionFind(int V)`	Initialisiert V Komponenten, jeweils mit einem Knoten `0`,...,`V-1`
`void`	`union(int v, int w)`	Verbindet die Komponenten, die v und w enthalten
`int`	`find(int v)`	ID der Komponente, die v enthält
`boolean`	`connected(int v, int w)`	Sind v und w in derselben Komponente?
`int`	`count()`	Anzahl der Komponenten

[Sedgewick & Wayne S. 241]

Kruskals Algorithmus als Pseudocode

- Die Implementierung von `UnionFind` folgt im letzten Teil der Vorlesung.
- Wir setzen diese nun voraus und fahren mit Kruskal fort.

Listing 2: **Algorithmus von Kruskal mit UnionFind**

```
UF : UnionFind(V)     // Initialisierung mit einzelnen Komponenten für jeden Knoten
MST ← ∅               // Kanten des minimalen Spannbaums (z.B. als Queue)
sortiere Kanten E aufsteigend nach Gewicht
for (v,w) ∈ E in dieser Reihenfolge
   if UF.find(v) ≠ UF.find(w) then
      UF.union(v, w)
      add (v,w) to MST
   end
end
```

- Das Durchlaufen der Kanten aufsteigend nach Gewicht kann z.B. über eine minPQ implementiert werden.

Korrektheit von Kruskals Algorithmus

Korrektheit des Kruskal Algorithmus

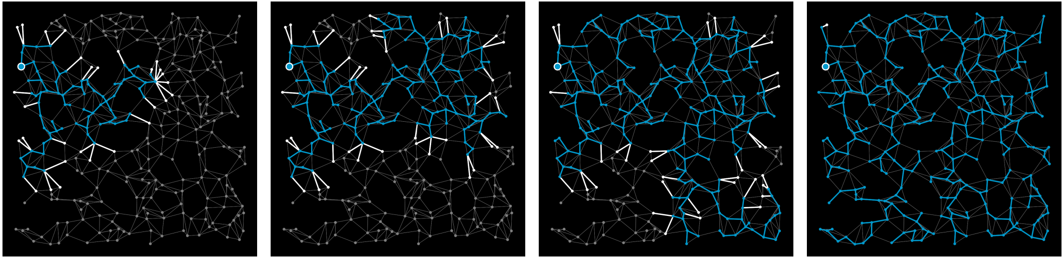
Der Algorithmus von Kruskal in Listing 2 bestimmt den minimalen Spannbaum.

Beweis.

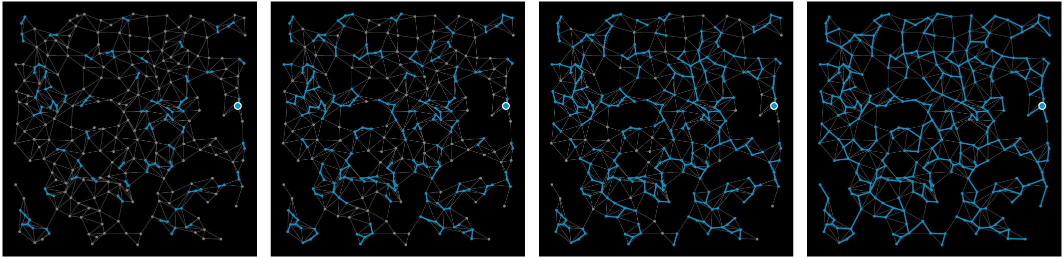
- Die Korrektheit folgt aus dem Beweis zum allgemeinen Ansatz, S. 13:
- Jede Kante (v, w), die hinzugefügt wird, ist eine kreuzende Kante des Schnittes, der durch alle mit v verbundenen Knoten definiert wird, also die entsprechende Zusammenhangskomponente von Union-Find.
- Diese Kante (v, w) ist tatsächliche eine kreuzende Kante, da v zu der Menge gehört und w nicht, denn $UF.find(v) \neq UF.find(w)$.
- Unter den kreuzenden Kanten hat (v, w) minimales Gewicht, da alle anderen kreuzenden Kanten später in der Sortierung vorkommen.

Reihenfolge der Knotenauswahl bei Prim und Kruskal

Prims Algorithmus (Kanten in der Warteschlange sind weiß dargestellt)

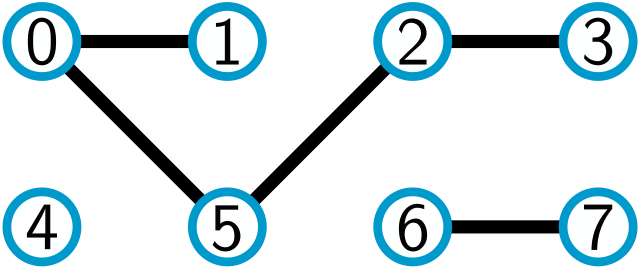


Kruskals Algorithmus



Union-Find: Dynamische Zusammenhangskomponenten

- Union-Find startet mit N separaten Knoten, jeder in einer eigenen Zusammenhangskomponente.
- Mit `union()` werden zwei Komponenten verbunden.
- Mit `connected()` wird geprüft, ob zwei Knoten zusammenhängen.

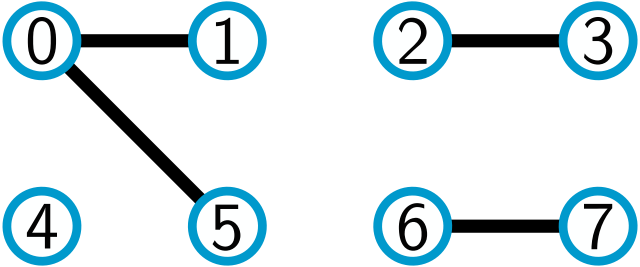



```
union(0,5)
union(2,3)
union(6,7)
union(0,1)
connected(1,5)? true
connected(0,3)? false
union(5,2)
connected(0,3)? true
```

- Wir werden unterschiedliche Implementierungsansätze besprechen.

Implementierung optimiert für schnelles Finden (*quick-find*)

- Erste Variante: `find()`, bzw. `connected()` möglichst schnell
- Knoten-indiziertes Array `id`, wie bei Zusammenhangskomponenten
- Eine `connected(v,w)` Anfrage entspricht dem Vergleich `id[v] == id[w]`.



v	0	1	2	3	4	5	6	7
`id[v]`	0	0	2	2	4	0	6	6

Implementation von Union-Find mit quick-find

```java
public class UnionFind { // quick-find
  private int[] id;
  private int count;

  public UnionFind(int V) {
    count = V;
    id = new int[V];
    for (int v = 0; v < V; v++) {
      id[v] = v;
    }
  }

  public int count()
  { return count; }

  public int find(int v)
  { return id[v]; }

  public boolean connected(int v, int w)
  { return id[v] == id[w]; }
```

```java
  public void union(int v, int w) {
    if (connected(v, w))
      return;
    int wid = id[w];
    for (int u = 0; u < id.length; u++) {
      if (id[u] == wid) {
        id[u] = id[v];
      }
    }
    count--;
  }
}
```

- Die Methoden `find()` und `connected()` sind sehr schnell, $\boldsymbol{O}(1)$.
- Aber `union()` muss das ganze Feld durchsuchen, also $\boldsymbol{O}(V)$.
- Daher ist dieser Ansatz nicht geeignet.

[Sedgewick & Wayne, S. 244]

Implementierung optimiert für schnelles Zusammenlegen (*quick-union*)

- Schnelles `union()`: nutze Baumstruktur für jede Komponente
- Das Array verweist für jeden Knoten auf den Elternknoten.
  Daher nennen wir die Variable jetzt `parent`.
- Ein Knoten jeder Komponente verweist per `parent` auf sich selbst: **Wurzel**
- `union(v,w)`: Wurzel der `w`-Komponente verweist auf die Wurzel der `v`-Komponente
- Man könnte auch die Wurzel der `w`-Komponente auf `v` verweisen lassen, aber das führt zu hohen Bäumen und damit zu einer schlechteren Laufzeit.
- Nun `find()` aufwendiger: Suche Wurzel entlang des `parent`-Array

Implementation von Union-Find mit quick-union (*lazy union-find*)

```
1 // UnionFind mit lazy quick-union
2 //   gebenüber der quick-find Version
3 //   sind nur folgende Methoden anders:
4   public int find(int v)
5   {
6     while (v != parent[v])
7       v = parent[v];
8     return v;
9   }
10
11  public void union(int v, int w)
12  {
13    int vRoot = find(v);
14    int wRoot = find(w);
15    if (vRoot == wRoot)
16      return;
17
18    parent[wRoot] = vRoot;
19    count--;
20  }
```

- Die Methode `find()` hat im schlechtesten Fall eine Laufzeit in $\boldsymbol{O}(V)$.
- Auch wenn das eigentliche Zusammenlegen zweier Komponenten schnell geht (Zeile 18), bringt es im *worst case* keinen Vorteil, da zunächst per langsamen `find()` die Wurzeln der Komponenten gesucht werden müssen.
- Also geht die Suche nach einem geeigneten Ansatz weiter.

Union-Find mit flacheren Bäumen (*eager union-find*)

- Schlechte Laufzeit bei *quick-union* wird durch hohe Bäume bedingt.
  (längere Suche nach Wurzel in `while`-Schleife von `find()`)
- Diese entstehen, wenn größere Komponenten an kleinere angehängt werden, weil dabei der entstehende Baum höher wird.
- Um die Bäume flach zu halten, hänge immer den kleineren an den größeren Baum.
- Dazu speichern wir die Höhe jedes Baums in dem Knoten-indizierten Array `ht`.

Implementation von *eager* Union-Find

```java
public class UnionFind { // flache Bäume
  private int[] parent;
  private int[] ht;
  private int count;

  public UnionFind(int V) {
    count = V;
    parent = new int[V];
    ht = new int[V];
    for (int v = 0; v < V; v++) {
      parent[v] = v;
      ht[v] = 0;
    }
  }

// count, connected und find wie oben
```

```java
  public void union(int v, int w)
  {
    int vRoot = find(v);
    int wRoot = find(w);
    if (vRoot == wRoot)
      return;

    if (ht[wRoot] < ht[vRoot]) {
      parent[wRoot] = vRoot;
    } else {
      parent[vRoot] = wRoot;
      if (ht(vRoot) == ht[wRoot]) {
        ht[wRoot]++;
      }
    }
    count--;
  }
}
```

Laufzeit von Union-Find mit flachen Bäumen

Maximale Höhe der Bäume in *eager* Union-Find

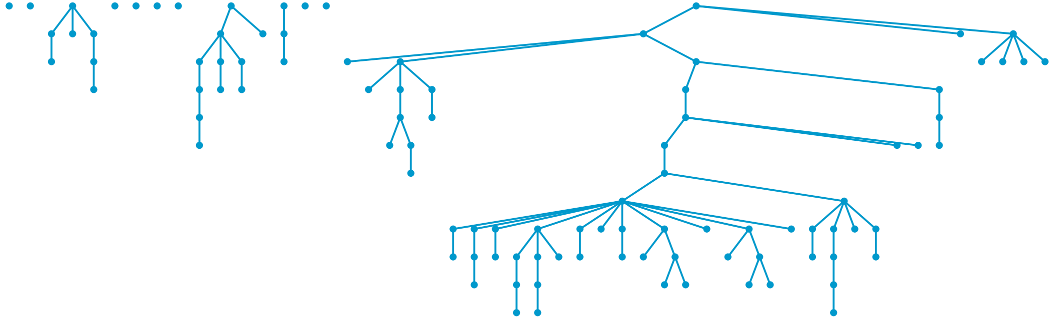
Die Bäume in *eager* Union-Find (Seite 37) haben maximal die Höhe $\lg V$.
($\lg$ bezeichnet den Logarithmus zur Basis 2.)

Beweis. Wir zeigen durch Induktion nach n, dass ein Baum mit n Knoten in *eager* Union-Find höchstens die Höhe $\lg n$ hat.

- Werden zwei Bäume mit unterschiedlichen Höhen zusammengelegt, so hat der entstehende Baum dieselbe Höhe, wie der größere der beiden ursprünglichen Bäume. Daher folgt die IB direkt aus der IV.
- Betrachten wir also den Fall, dass zwei Bäume mit k_0 und k_1 Knoten und Höhe h zu einem Baum mit $k_0 + k_1$ Knoten vereint werden.
- Nach IV gilt $h \leq \lg k$ für $k = \min(k_0, k_1)$. Durch das Anhängen des einen Baumes an die Wurzel des anderen Baumes, steigt die Höhe um eins.
- Also ist für den neuen Baum die Höhe maximal $1 + h \leq 1 + \lg k = \lg(2k) \leq \lg(k_0 + k_1)$. $\square$

Lazy Union Find

mittlere Distanz zur Wurzel: 5,11



Eager Union Find

mittlere Distanz zur Wurzel: 1,52

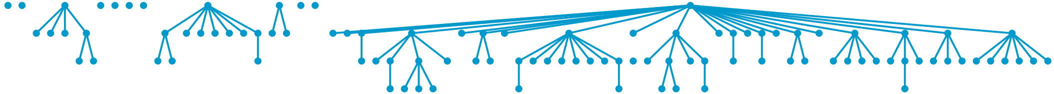


Bild erstellt nach Vorlage [Segdewick & Wayne, S. 254]

Weitere Verbesserung

- *Eager* Union-Find hat Laufzeit in $O(\lg V)$ auch im *worst case*.
- Diese Laufzeitgarantie ist für die Praxis ausreichend, aber es geht noch besser!
- Man kann bei der `find()` Operation die Baumstruktur nebenbei verbessern.
- 'Verbessern' heißt in unserem Fall, die Bäume möglichst flach zu halten.
- **Pfadkompression**: Hänge bei der Iteration in `find()` alle durchlaufenen Knoten näher an die Wurzel:
  - Zweistufig: Bestimme im ersten Durchlauf die Wurzel wie bisher und hänge im zweiten Durchlauf alle Knoten direkt an die Wurzel.
  - Einstufig: Hänge jeden Knoten in der Iteration an seinen Vorvorgänger (Halbierung der Pfadtiefe).

Implementation von Union-Find mit Pfadkompression

```java
public class UnionFind //Pfadkompression
{
  // ... wie eager UnionFind
  // es ändert sich nur find()

  public int find(int v) {
    while (v != parent[v]) {
      parent[v] = parent[parent[v]];
      v = parent[v];
    }
    return v;
  }
}
```

- Die Laufzeitanalyse dieser Variante ist sehr komplex. Daher benügen wir uns in dieser Vorlesung mit dem Ergebnis.

[Sedgewick, *AlgoDat COS 226, 2007*]

Laufzeiten der Union-Find Operationen

Wachstumsordnung für V Knoten (*worst case*)			
Algorithmus	**Init**	**union**	**find**
Quick-find	$\boldsymbol{O}(V)$	$\boldsymbol{O}(V)$	$\boldsymbol{O}(1)$
Lazy quick-union	$\boldsymbol{O}(V)$	$\boldsymbol{O}(V)$	$\boldsymbol{O}(V)$
Eager quick-union	$\boldsymbol{O}(V)$	$\boldsymbol{O}(\lg V)$	$\boldsymbol{O}(\lg V)$
Union-Find mit Pfadkompression	$\boldsymbol{O}(V)$	$\boldsymbol{O}(\lg^* V)$	$\boldsymbol{O}(\lg^* V)$

V	$\lg^* V$
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

- $\lg^*(V) < 5$ (für $V <$ # Atome im Universum)
- $\lg^*(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \lg^*(\lg(n)) & \text{otherwise} \end{cases}$
- Für Union-Find mit Pfadkompression gibt es eine noch schärfere Abschätzung mit der inversen Ackermann Funktion.

Laufzeiten der Union-Find Varianten

Laufzeiten für N `union/find` Operationen bei V Knoten (*worst case*)

Algorithmus	Laufzeit
Quick-find	$\boldsymbol{O}(NV)$
Lazy quick-union	$\boldsymbol{O}(NV)$
Eager quick-union	$\boldsymbol{O}(V + N \lg V)$
Union-Find mit Pfadkompression	$\boldsymbol{O}((V + N) \lg^* V)$

- Beispiel (passt gerade in heutige Speicher):
  $V = 10^9$ Knoten und $N = 10^9$ `union/find` Operationen:
- **Reduktion** der Laufzeit von 30 Jahren auf **6 Sekunden**.
- Es kann keinen Algorithmus mit linearer Laufzeit geben [Fredman & Saks 1989].

Laufzeit und Speicherbedarf von Kruskals MST Algorithmus

Laufzeit des Kruskal Algorithmus

Der Algorithmus von Kruskal in Listing 2 bestimmt den minimalen Spannbaum in einer *worst-case* Laufzeit in $\boldsymbol{O}(E \log E)$ und einem Speicherbedarf in $\boldsymbol{O}(E)$.

Wir gehen von einer Implementation mittels einer Vorrangewarteschlange (minPQ) für das nach Gewicht sortierte Durchlaufen der Kanten aus.

- Die Initialisierung der Union-Find Struktur in Zeile 1 benötigt $\boldsymbol{O}(V)$ und das Einfügen der Kanten in eine PQ benötigt $\boldsymbol{O}(E)$.
- Die *for*-Schleife wird für jede Kante einmal durchlaufen. Also benötigen die `poll()` Operationen der minPQ insgesamt $\boldsymbol{O}(E \log E)$.
- Durch die effiziente Union-Find Implementation fallen die `union()` und `find()` Operationen nicht ins Gewicht.
- Der Speicherbedarf ist proportional zu V für Union-Find und zum Speichern des Spannbaums in MST, und für die PQ ist er proportional zu E.

Bemerkung zur Historie

- Der Algorithmus von Borůvka (hier nicht besprochen) ist leicht zu implementieren, parallelisierbar und hat eine *worst-case* Laufzeit in $\mathcal{O}(E \log V)$. In der Praxis ist er meist deutlich schneller. Bei einer großen Klasse von Graphen inklusive aller planaren Graphen ist die Laufzeit in $\mathcal{O}(E)$ [Erickson 2019].
- Borůvka hat in seinem Aufsatz bereits 1929 auch wesentliche Ideen für die Algorithmen von Prim und Kruskal in seinem Artikel erwähnt.
- Weitere Grundlagen für den Prim Algorithmus wurden von Jarnik 1939 und von Kruskal 1956 veröffentlicht. Prims Veröffentlichung ist von 1961.
- Bei der Einschätzung der frühen Ansätze ist zu berücksichtigen, dass damals viele der effizienten Datentypen wie Vorrangwarteschlagen noch nicht bekannt waren.

Effizientere Weiterentwicklungen

- Fredman-Tarjan 1984: Prim mit Fibonacci Heap: $\boldsymbol{O}(E + V \log V)$
- Chazelle 1997, 2000: Laufzeit sehr nah an $\boldsymbol{O}(E)$, aber für die Praxis zu kompliziert.
- Karger-Klein-Tarjan 1995: Randomisierter Algorithmus mit linearer Laufzeit (im Erwartungswert).

Literatur I

Generell:

- Segdewick R & Wayne K, *Algorithmen: Algorithmen und Datenstrukturen*, Pearson Studium, 4. Auflage, 2014. ISBN: 978-3868941845; in Teilen auch auf `http://www.cs.princeton.edu/IntroAlgsDS`
- TH Cormen, CE Leiserson, R Rivest, C Stein, *Algorithmen - Eine Einführung* . De Gruyter Oldenbourg, 4. Auflage; 2013. ISBN: 978-3486748611
- Erickson J, *Algorithms*, 1. Auflage 2019. `http://algorithms.wtf`, ISBN: 978-1-792-64483-2.

Originalveröffentlichungen Minimale Spannbäume:

- Nešetřil J, Milková E, Nešetřilová H. *Otakar Borůvka on minimum spanning tree problem translation of both the 1926 papers*, comments, history. Discrete mathematics. 2001 Apr 28;233(1-3):3-6.

Literatur II

- Kruskal JB. *On the shortest spanning subtree of a graph and the traveling salesman problem*. Proceedings of the American Mathematical society. 1956;7(1):48-50.
- Prim RC. *Shortest connection networks and some generalizations*. Bell Labs Technical Journal. 1957 Nov 1;36(6):1389-401.
- Thorup M. *Near-optimal fully-dynamic graph connectivity*. In: Proceedings of the thirty-second annual ACM symposium on Theory of computing 2000 May 1 (pp. 343-350). ACM.
- Dijkstra EW. *A note on two problems in connexion with graphs*. Numerische mathematik. 1959 Dec 1;1(1):269-71.
- Fredman ML, Tarjan RE. *Fibonacci heaps and their uses in improved network optimization algorithms*. Journal of the ACM (JACM). 1987 Jul 1;34(3):596-615.
- Karger DR, Klein PN, Tarjan RE. *A randomized linear-time algorithm to find minimum spanning trees*. Journal of the ACM (JACM). 1995 Mar 1;42(2):321-8.

Literatur III

- Chazelle B. *A minimum spanning tree algorithm with inverse-Ackermann type complexity*. Journal of the ACM (JACM). 2000 Nov 1;47(6):1028-47.
- Pettie S, Ramachandran V. *An optimal minimum spanning tree algorithm*. Journal of the ACM (JACM). 2002 Jan 1;49(1):16-34.

Originalveröffentlichungen Union-Find:

- Tarjan RE. *Efficiency of a good but not linear set union algorithm*. Journal of the ACM (JACM). 1975 Apr 1;22(2):215-25.
- Fredman M, Saks M. *The cell probe complexity of dynamic data structures*. In: Proceedings of the twenty-first annual ACM symposium on Theory of computing 1989 Feb 1 (pp. 345-354). ACM.

Bildreferenzen I

- Die Abbildung auf Seite 39 ist aus dem Buch [Sedgewick & Wayne 2014, S. 254].

Index