ETH zürich

Übung 11

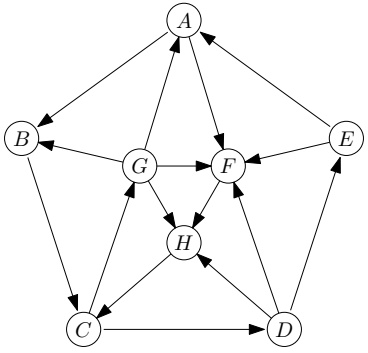
Datenstrukturen und Algorithmen, D-MATH, ETH Zurich

Heutiges Programm

- Feedback letzte Übung
- Wiederholung Vorlesung
  - All Pairs Shortest Paths
  - Kruskal
- Hinweise zu den Aufgaben
  - Closeness Centrality
  - TSP
- In-Class-Exercise praktisch
- In-Class-Exercise (theoretisch)

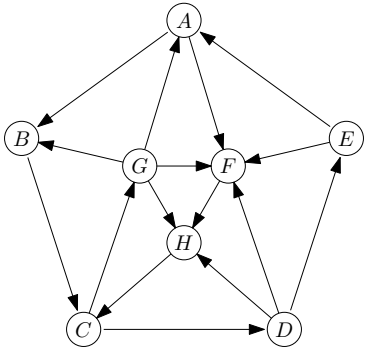
1. Feedback letzte Übung

Tiefen- und Breitensuche



Start bei A
DFS: A, B, C, D, E, F, H, G
BFS: A, B, F, C, H, D, G, E

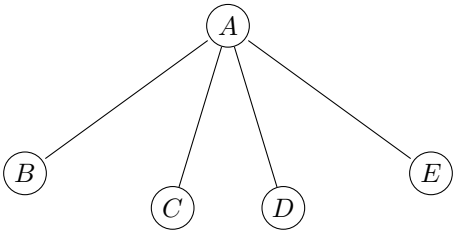
Tiefen- und Breitensuche



Start bei A
DFS: A, B, C, D, E, F, H, G
BFS: A, B, F, C, H, D, G, E
Es gibt keinen Startknoten, sodass die DFS-Ordnung der BFS-Ordnung entspricht.

Tiefen- und Breitensuche

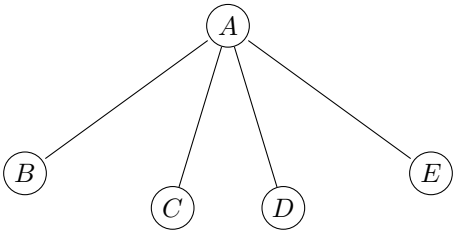
Stern: DFS-Ordnung entspricht BFS-Ordnung



Start bei A
DFS: A, B, C, D, E
BFS: A, B, C, D, E

Tiefen- und Breitensuche

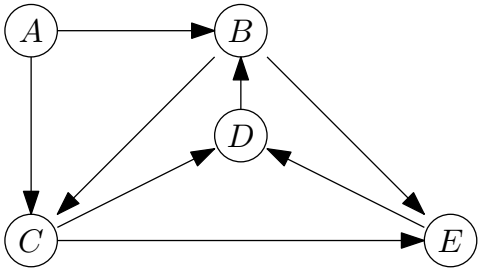
Stern: DFS-Ordnung entspricht BFS-Ordnung



Start bei A
DFS: A, B, C, D, E
BFS: A, B, C, D, E

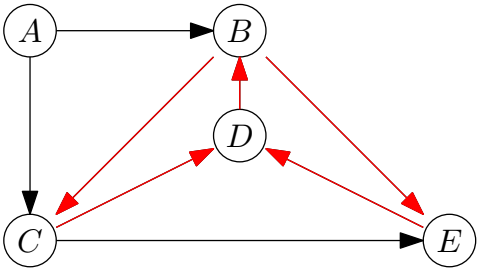
Start bei C
DFS: C, A, B, D, E
BFS: C, A, B, D, E

Topologische Sortierung



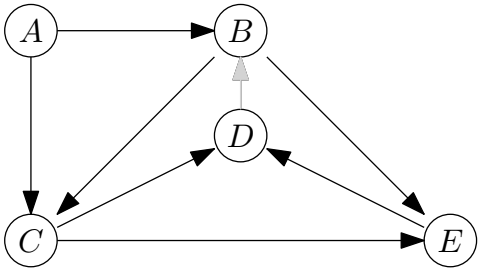
- der Graph hat Kreise

Topologische Sortierung



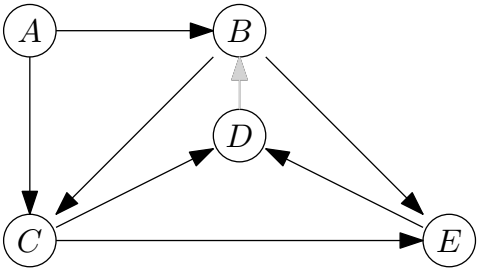
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante

Topologische Sortierung



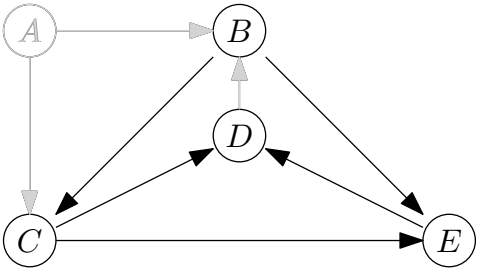
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei

Topologische Sortierung



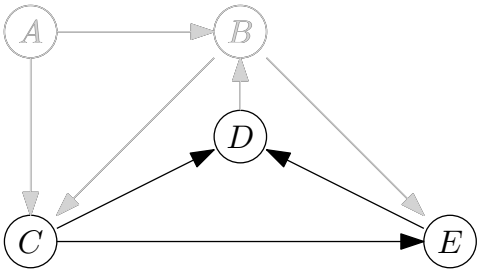
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



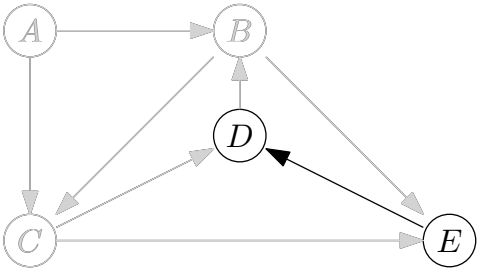
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



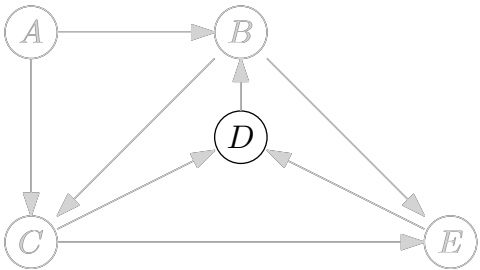
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

2. Wiederholung Vorlesung

DP-Algorithmus Floyd-Warshall(G)

Input: Azyklischer Graph $G = (V, E, c)$
Output: Minimale Gewichte aller Pfade d
$d^0 \leftarrow c$
for $k \leftarrow 1$ **to** $|V|$ **do**
  for $i \leftarrow 1$ **to** $|V|$ **do**
    for $j \leftarrow 1$ **to** $|V|$ **do**
      $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$

Laufzeit: $\Theta(|V|^3)$

Bemerkung: Der Algorithmus kann auf einer einzigen Matrix d (in place) ausgeführt werden.

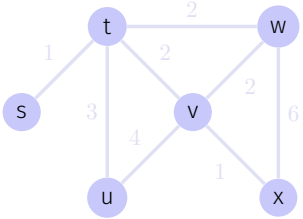
Vergleich der Verfahren

Algorithmus			Laufzeit
Dijkstra (Heap)	$c_v \geq 0$	1:n	$\mathcal{O}(\lvert E\rvert \log \lvert V\rvert)$
Dijkstra (Fibonacci-Heap)	$c_v \geq 0$	1:n	$\mathcal{O}(\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert)$ *
Bellman-Ford		1:n	$\mathcal{O}(\lvert E\rvert \cdot \lvert V\rvert)$
Floyd-Warshall		n:n	$\Theta(\lvert V\rvert^3)$
Johnson		n:n	$\mathcal{O}(\lvert V\rvert \cdot \lvert E\rvert \cdot \log \lvert V\rvert)$
Johnson (Fibonacci-Heap)		n:n	$\mathcal{O}(\lvert V\rvert^2 \log \lvert V\rvert + \lvert V\rvert \cdot \lvert E\rvert)$ *

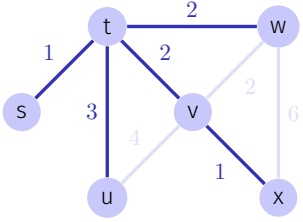
* amortisiert

Johnson (dieses Jahr nicht erklärt) ist besser als Floyd-Warshall nur für dünn besetzte Graphen ($|E| \approx \Theta(|V|)$).

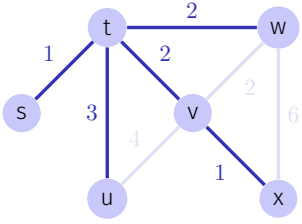
Minimale Spannbäume

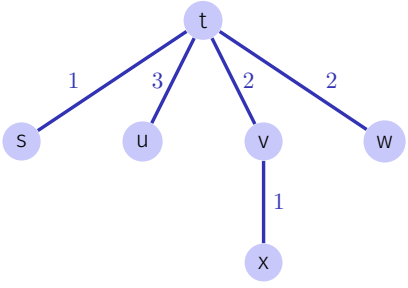


Minimale Spannbäume

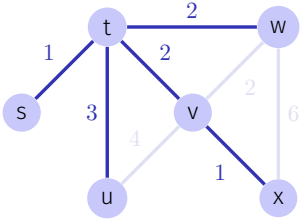


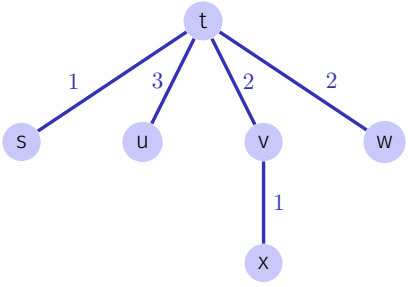
Minimale Spannbäume





Minimale Spannbäume





(Lösung ist nicht eindeutig.)

MakeSet, Union, und Find

- Make-Set(i): Hinzufügen einer neuen Menge i.
- Find(e): Name i der Menge, welche e enthält.
- Union(i, j): Vereinigung der Mengen mit Namen i und j.

MakeSet, Union, und Find

- Make-Set(i): Hinzufügen einer neuen Menge i.
- Find(e): Name i der Menge, welche e enthält.
- Union(i, j): Vereinigung der Mengen mit Namen i und j.

In MST-Kruskal:

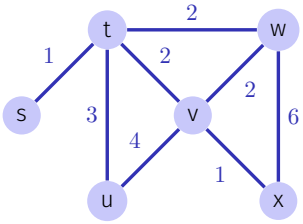
- Make-Set(i): Neuer Baums mit Wurzel i .
- Find(e): Finde Wurzel von e
- Union(i, j): Vereinigung der Bäume i und j.

Algorithmus MST-Kruskal(G)

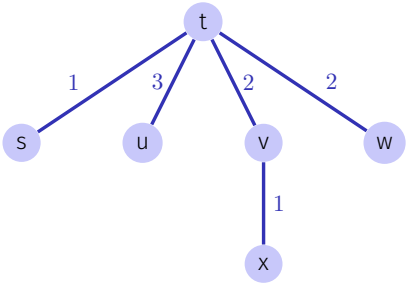
Input: Gewichteter Graph $G = (V, E, c)$
Output: Minimaler Spannbaum mit Kanten A.

Sortiere Kanten nach Gewicht $c(e_1) \leq ... \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ **to** $|V|$ **do**
  MakeSet(k)
for $k = 1$ **to** m **do**
  $(u, v) \leftarrow e_k$
  if Find(u) $\neq$ Find(v) **then**
    Union(Find(u), Find(v))
    $A \leftarrow A \cup e_k$
return (V, A, c)

Repräsentation als Array



Index s t w v u x



Index	s	t	u	v	w	x
Parent	t	t	t	t	t	v

Andere Verbesserung

Bei jedem Find alle Knoten direkt an den Wurzelknoten hängen.

Find(i):

$j \leftarrow i$
while $(p[i] \neq i)$ **do** $i \leftarrow p[i]$
while $(j \neq i)$ **do**
    $t \leftarrow j$
    $j \leftarrow p[j]$
    $p[t] \leftarrow i$
return i

Laufzeit: amortisiert *fast* konstant (Inverse der Ackermannfunktion).[1]

[1] Wird hier nicht vertieft.

Laufzeit des Kruskal Algorithmus

- Sortieren der Kanten: $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$. [2]
- Initialisieren der Union-Find Datenstruktur $\Theta(|V|)$
- $|E|\times$ Union(Find(x),Find(y)): $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$.

Insgesamt $\Theta(|E| \log |V|)$.

[2] da G zusammenhängend: $|V| \leq |E| \leq |V|^2$

3. Hinweise zu den Aufgaben

Closeness Centrality, TSP

Closeness Centrality

- Gegeben: Eine Adjazenzmatrix für einen *ungerichteten* Graphen auf n Knoten.
- Aufgabe: für jeden Knoten v die *Closeness Centrality* $C(v)$ von v.

$$C(v) = \sum_{u \in V \setminus \{v\}} d(v, u)$$

Closeness Centrality

- Gegeben: Eine Adjazenzmatrix für einen *ungerichteten* Graphen auf n Knoten.
- Aufgabe: für jeden Knoten v die *Closeness Centrality* $C(v)$ von v.

$$C(v) = \sum_{u \in V \setminus \{v\}} d(v, u)$$

- Intuitiv: Wenn viele verbundene Knoten nahe bei v liegen, dann ist $C(v)$ klein.
- „Wie zentral ist ein Knoten in seiner Zusammenhangskomponente?“

Alle kürzesten Pfade

- Wir brauchen $d(u, v)$ für alle Knotenpaare (u, v).
- $\Longrightarrow$ berechne alle kürzesten Pfade mit Floyd-Warshall. (`APSH.h`)

```
template<typename Matrix>
void allPairsShortestPaths(unsigned n, Matrix& m)
{
  // your code here
}
```

- Das Feld `m` soll mit den Distanzen überschrieben werden.
- Achtung: anfangs bedeutet `0` „keine Kante“.
- Ungerichteter Graph: `m[i][j] == m[j][i]`

Closeness Centrality

Centrality.h

```
void printCentrality(unsigned n, vector<vector<unsigned>>
   adjacencies, vector<string> names)
{
  for(unsigned i = 0; i < n; ++i)
  {
    cout << names[i] << ": ";
    unsigned centrality = 0;
    // TODO: compute centrality of vertex i here
    cout << centrality << endl;
  }
}
```

Closeness Centrality: Eingabedaten

- Der Eingabegraph beschreibt die Zusammenarbeit von gewissen Autoren an wissenschaftlichen Publikationen.
- Die Knoten des Graphen stehen für die Co-Autoren des Mathematikers Paul Erdős.
- Wenn sie zusammen eine Arbeit veröffentlicht haben, sind sie durch eine Kante verbunden.
- Quelle: `https://oakland.edu/enp/thedata/`

Closeness Centrality: Output

```
vertices: 511
ABBOTT, HARVEY LESLIE                : 1625
ACZEL, JANOS D.                      : 1681
AGOH, TAKASHI                        : 2132
AHARONI, RON                         : 1578
AIGNER, MARTIN S.                    : 1589
AJTAI, MIKLOS                        : 1492
ALAOGLU, LEONIDAS*                   : 0
ALAVI, YOUSEF                        : 1561
...
```

Wo kommt die 0 her?

Travelling Salesperson Problem

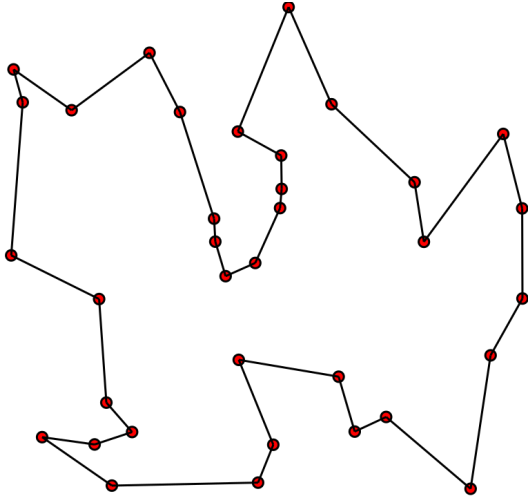
Problem

Gegeben ist eine Karte und eine Liste von Städten. Welches ist die kürzeste Route, die jede Stadt einmal besucht und in die ursprüngliche Stadt zurückkehrt?

Mathematical model

Auf einem ungerichteten, gewichteten Graph G ist nach dem Kreis gesucht, welcher jede der Knoten von G genau einmal enthält und die kleinste Kantensumme aufweist.

Travelling Salesperson Problem



Travelling Salesperson Problem

- Es ist kein Polynomialzeitalgorithmus zum Lösen des Problems bekannt.
- Es gibt verschiedene heuristische Algorithmen. Diese liefern oft nicht die optimale Lösung.

Travelling Salesperson Problem

- Der heuristische Algorithmus, den Sie auf CodeExpert implementieren sollen (*The Travelling Student*) benutzt einen Minimalen Spannbaum:
  1. Berechne den Minimalen Spannbaum M
  2. Mache eine Tiefensuche auf M
- Der Algorithmus ist eine 2-Approximation. Das bedeutet, dass er eine Lösung liefert, die maximal 2 mal die Kosten einer optimalen Lösung aufweist.
- Der Algorithmus geht von einem vollständigen Graphen $G = (V, E, c)$ aus, auf dem die Dreiechsungleichung gilt:
  $c(v, w) \leq c(v, x) + c(x, w) \; \forall \; v, w, x \in V$

4. In-Class-Exercise praktisch

Union-Find Experiments (Code-Expert)

5. In-Class-Exercise (theoretisch)

In-Class-Exercises: Längster Pfad in DAGs

Das Kürzeste-Pfad-Problem hat einfache Lösungen (BFS, Dijkstra, Bellman-Ford). Das Längste-Pfad-Problem hingegen ist sehr schwierig! Für gerichtete Graphen gibt es vermutlich keinen schnellen Algorithmus, um Pfade der Länge $\gg \log^2 n$ zu finden.

In-Class-Exercises: Längster Pfad in DAGs

Das Kürzeste-Pfad-Problem hat einfache Lösungen (BFS, Dijkstra, Bellman-Ford). Das Längste-Pfad-Problem hingegen ist sehr schwierig! Für gerichtete Graphen gibt es vermutlich keinen schnellen Algorithmus, um Pfade der Länge $\gg \log^2 n$ zu finden.

Aufgabe:

Gegeben sei ein gerichteter, **kreisfreier** Graph (DAG) $G = (V, E)$.

Entwerfen Sie einen $\mathcal{O}(|V| + |E|)$-Laufzeit Algorithmus, um den *längsten Pfad* zu finden.

In-Class-Exercises: Längster Pfad in DAGs

Das Kürzeste-Pfad-Problem hat einfache Lösungen (BFS, Dijkstra, Bellman-Ford). Das Längste-Pfad-Problem hingegen ist sehr schwierig! Für gerichtete Graphen gibt es vermutlich keinen schnellen Algorithmus, um Pfade der Länge $\gg \log^2 n$ zu finden.

Aufgabe:

Gegeben sei ein gerichteter, **kreisfreier** Graph (DAG) $G = (V, E)$.

Entwerfen Sie einen $\mathcal{O}(|V| + |E|)$-Laufzeit Algorithmus, um den *längsten Pfad* zu finden.

Tipp: G ist kreisfrei, Sie können also zuerst topologisch sortieren.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.
3. Besuche jeden Knoten v in Reihenfolge der topologischen Sortierung und betrachte die Eingangs-Kanten: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.
3. Besuche jeden Knoten v in Reihenfolge der topologischen Sortierung und betrachte die Eingangs-Kanten: $\mathcal{O}(|V| + |E|)$.

$$\texttt{dist}[v] = \begin{cases} 0 & \text{keine Kanten,} \\ \max\limits_{(u,v)\in E} \{\texttt{dist}[u] + c(u,v)\} & \text{sonst.} \end{cases}$$

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.
3. Besuche jeden Knoten v in Reihenfolge der topologischen Sortierung und betrachte die Eingangs-Kanten: $\mathcal{O}(|V| + |E|)$.

$$\texttt{dist}[v] = \begin{cases} 0 & \text{keine Kanten,} \\ \max\limits_{(u,v)\in E} \{\texttt{dist}[u] + c(u,v)\} & \text{sonst.} \end{cases}$$

   Vorgänger merken!