Algorithmen und Datenstrukturen

C4. Minimale Spannbäume

Gabriele Röger

Universität Basel

8./9. Mai 2019

Algorithmen und Datenstrukturen
8./9. Mai 2019 — C4. Minimale Spannbäume

C4.1 Minimale Spannbäume

C4.2 Generischer Algorithmus

C4.3 Graphenrepräsentation

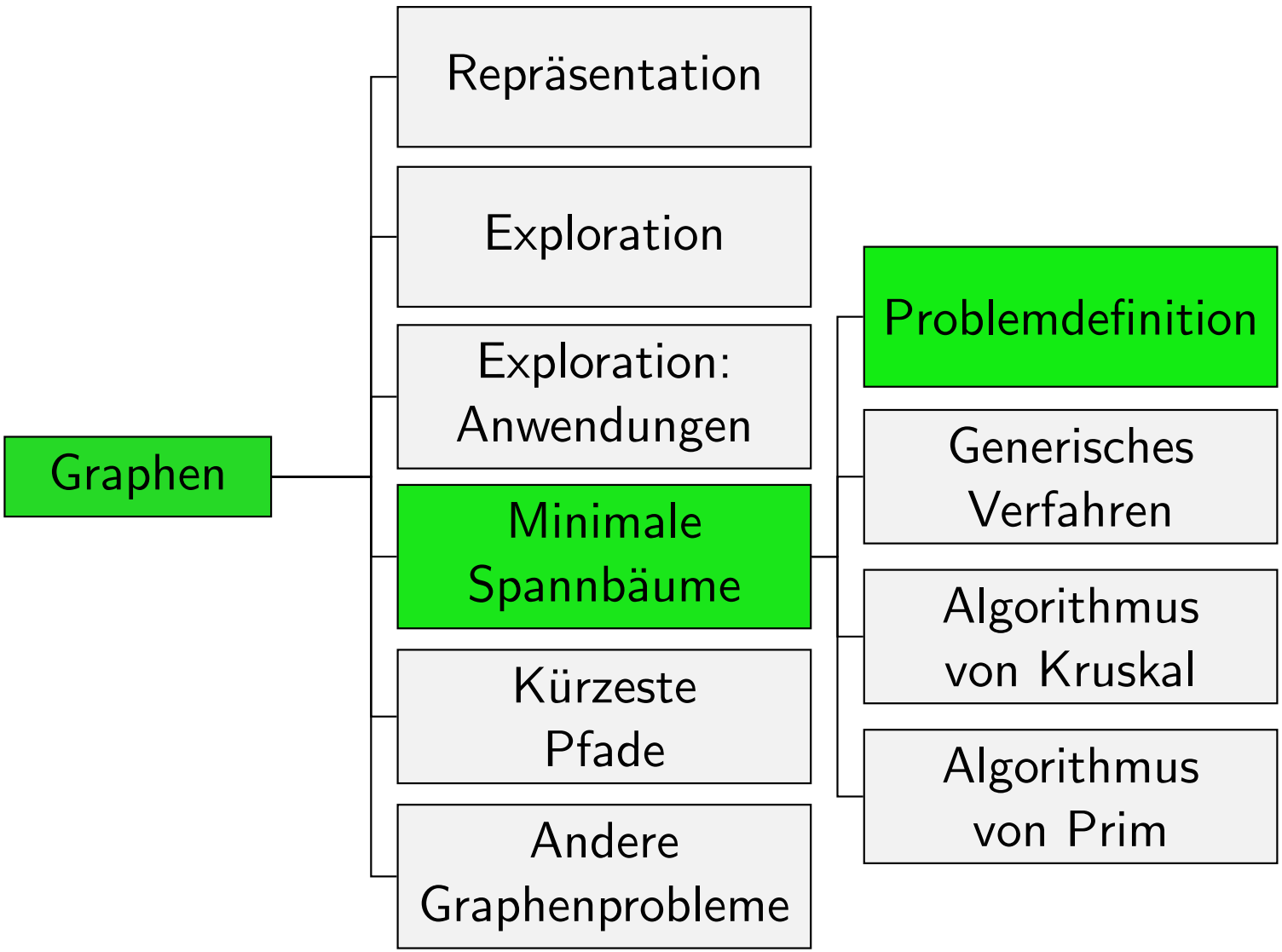
C4.4 Kruskals Algorithmus

C4.5 Prims Algorithmus

C4.6 Ausblick

C4.1 Minimale Spannbäume

Graphen: Übersicht



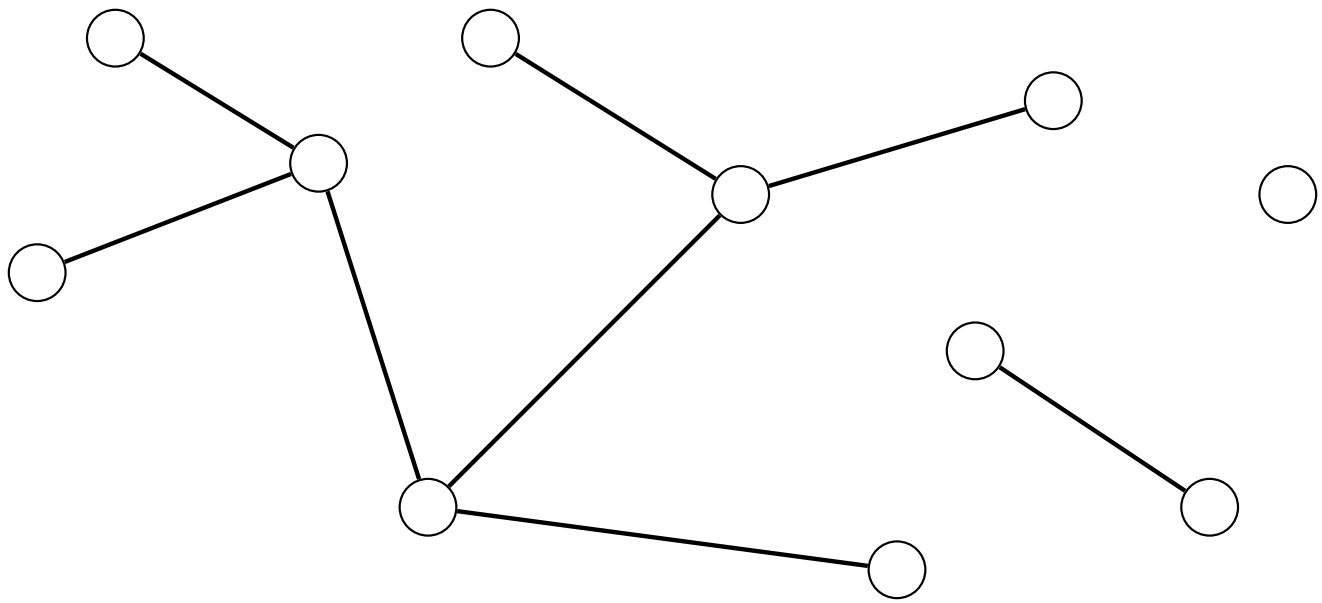
Ungerichtete Graphen

In Kapitel C4 betrachten wir nur ungerichtete Graphen.

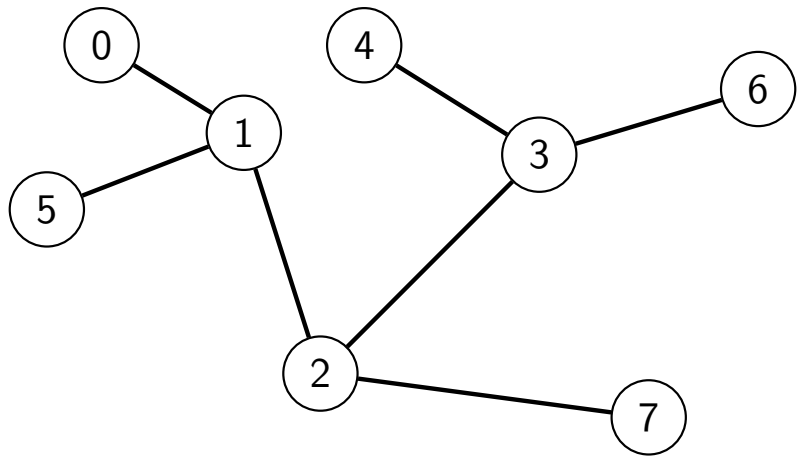
Bäume in ungerichteten Graphen

Definition

Ein Baum ist ein azyklischer, zusammenhängender Graph.
Eine disjunkte Menge von Bäumen wird Wald genannt.



Eigenschaften von Bäumen



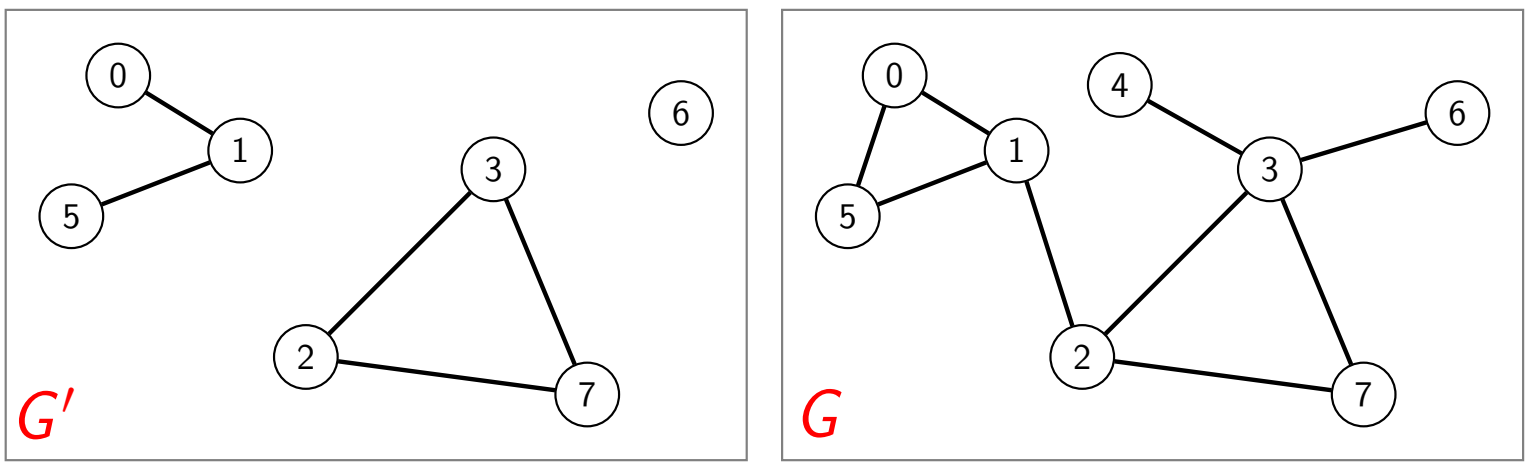
Für jeden Baum gilt:

- Jedes Knotenpaar ist durch genau einen einfachen Pfad verbunden (einfach = kein Knoten kommt zweimal vor).
- Entfernt man eine Kante, zerfällt er zu einem Graphen mit zwei Zusammenhangskomponenten.
- Fügt man eine Kante hinzu, erzeugt man einen Zyklus.

Teilgraph

Definition

Graph $G' = (V', E')$ ist ein Teilgraph von Graph $G = (V, E)$ falls $V' \subseteq V$ und $E' \subseteq E$.

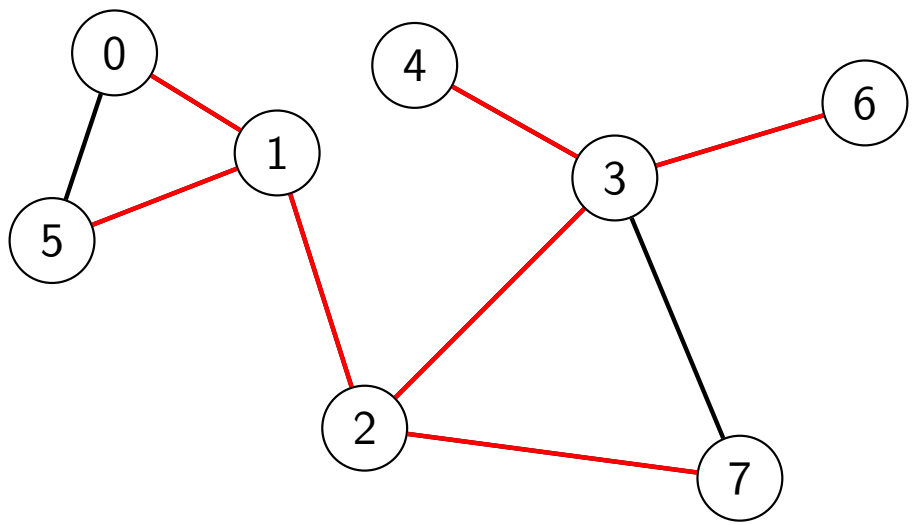


Spannbaum

Definition

Ein Spannbaum eines zusammenhängenden Graphen ist ein Teilgraph, der alle Knoten des Graphen enthält und ein Baum ist.

Ein Spannwald eines (nicht zusammenhängenden) Graphen ist die Vereinigung von je einem Spannbaum für jede Zusammenhangskomponente zu einem Graphen.



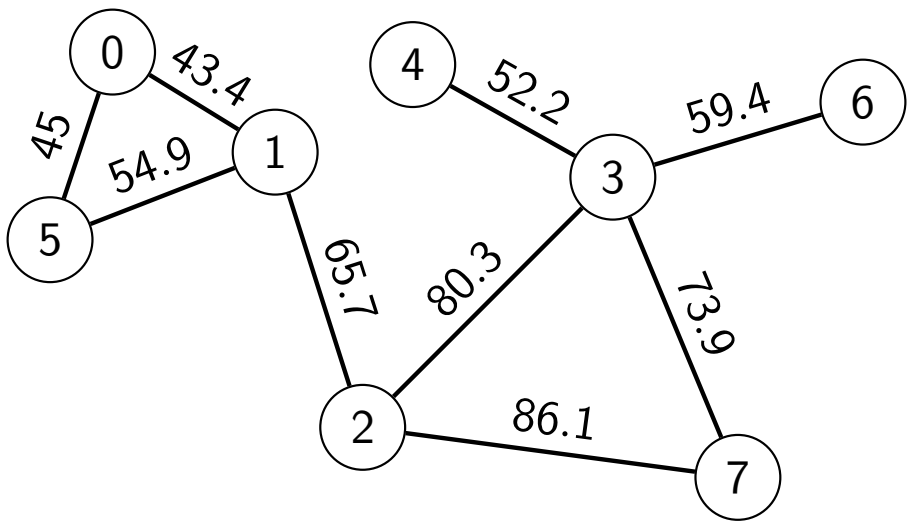
Wie viele Kanten hat ein Spannbaum?

Gewichtete Graphen

Definition

Ein (kanten-)gewichteter Graph ordnet jeder Kante $e \in E$ ein Gewicht (oder Kosten) $\mathit{weight}(e)$ aus den reellen Zahlen zu.

Das Gewicht des Graphen ist die Summe $\sum_{e \in E} \mathit{weight}(e)$ der Kantengewichte.



Minimale Spannbäume

Definition (Minimum-Spanning-Tree-Problem, MST-Problem)

Gegeben: Gewichteter, ungerichteter, zusammenhängender Graph

Gesucht: Spannbaum mit minimalem Gewicht
(es gibt keinen Spannbaum, bei dem die Summe der Kantengewichte geringer ist).

Anwendung: Clustering zur Tumorerkennung

- Analysis of soft tissue tumors by an attributed minimum spanning tree.

Kayser K[1], Sandau K, Böhm G, Kunze KD, Paul J

Analytical and Quantitative Cytology and Histology [01 Oct 1991, 13(5):329-334]

Abstract

Histologic slides of 22 soft tissue tumors (9 malignant fibrous histiocytoma, 8 fibrosarcoma, 2 rhabdomyosarcoma, 2 osteosarcoma, 1 Askin tumor) were Feulgen stained. Using an automated image analyzing system (Cambridge 570) at low magnification (25x), the tumor cell nuclei were segmented. The geometrical center of the nuclei was considered the vertex. A basic graph was constructed according to the neighborhood condition of O'Callaghan. Neighboring tumor cell nuclei were visualized by connecting edges. Several features of tumor cell nuclei were measured, including area, surface, major and minor axis of best fitting ellipsis and extinction (DNA content). Nuclear features are attributed to the vertices. The differences, or "distances," between features of connected vertices are attributed to the corresponding edges, which are dependent on the attributes. Thus, different minimum spanning trees (MST) result. Each MST can be decomposed into clusters using a suitable decomposition function on the edges, which rejects an edge if its attributes differ from the mean of the attributed values of surrounding edges more than a neighbor dependent bound (lower limit). Taking into account the length and other attributes of edges (e.g., differences in orientation of the major axis), clusters of different nuclear orientation can be detected. A cluster tree can be constructed by defining the geometric center of a cluster as a new vertex, and by computing the neighborhood of the cluster vertices. The result is an attributed MST containing characteristic structural properties of the image (in cases of sarcomatous tumors, local orientation of tumor cell nuclei and local DNA abnormalities).

Anwendung: Identitätsverifikation

ELSEVIER

Neurocomputing

Volume 72, Issues 7–9, March 2009, Pages 1859-1869

Minimum spanning tree based one-class classifier

Piotr Juszczak [a], David M.J. Tax [a], Elżbieta Pe˛kalska [b], Robert P.W. Duin [a]

Show more

https://doi.org/10.1016/j.neucom.2008.05.003

Get rights and content

Abstract

In the problem of one-class classification one of the classes, called the target class, has to be distinguished from all other possible objects. These are considered as non-targets. The need for solving such a task arises in many practical applications, e.g. in machine fault detection, face recognition, authorship verification, fraud recognition or person identification based on biometric data.

This paper proposes a new one-class classifier, the minimum spanning tree class descriptor (MST_CD). This classifier builds on the structure of the minimum spanning tree constructed on the target training set only. The classification of test objects relies on their distances to the closest edge of that tree, hence the proposed method is an example of a distance-based one-class classifier. Our experiments show that the MST_CD performs especially well in case of small sample size problems and in high-dimensional spaces.

Anwendung: Zellsegmentierung in Mikroskopiebildern

Optimal cut in minimum spanning trees for 3-D cell nuclei segmentation

7 Author(s) | A. Abreu ; F.-X. Frenois ; S. Valitutti ; P. Brousset ; P. Denèfle ; B. Naegel ; C. Wemmert View All Authors

Abstract | Authors | Figures | References | Citations | Keywords | Metrics | Media

Abstract:
In biology and pathology immunofluorescence microscopy approaches are leading techniques for deciphering of the molecular mechanisms of cell activation and disease progression. Although several commercial softwares for image analysis are presently in the market, available solutions do not allow a totally non subjective image analysis. There is therefore strong need for new methods that could allow a completely non-subjective image analysis procedure including for thresholding and for choice of the objects of interest. To address this need, we describe a fully automatic segmentation of cell nuclei in 3-D confocal immunofluorescence images. The method merges segments of the image to fit with a nuclei model learned by a trained random forest classifier. The merging procedure explores efficiently the fusion configurations space of an over-segmented image by using minimum spanning trees of its region adjacency graph.

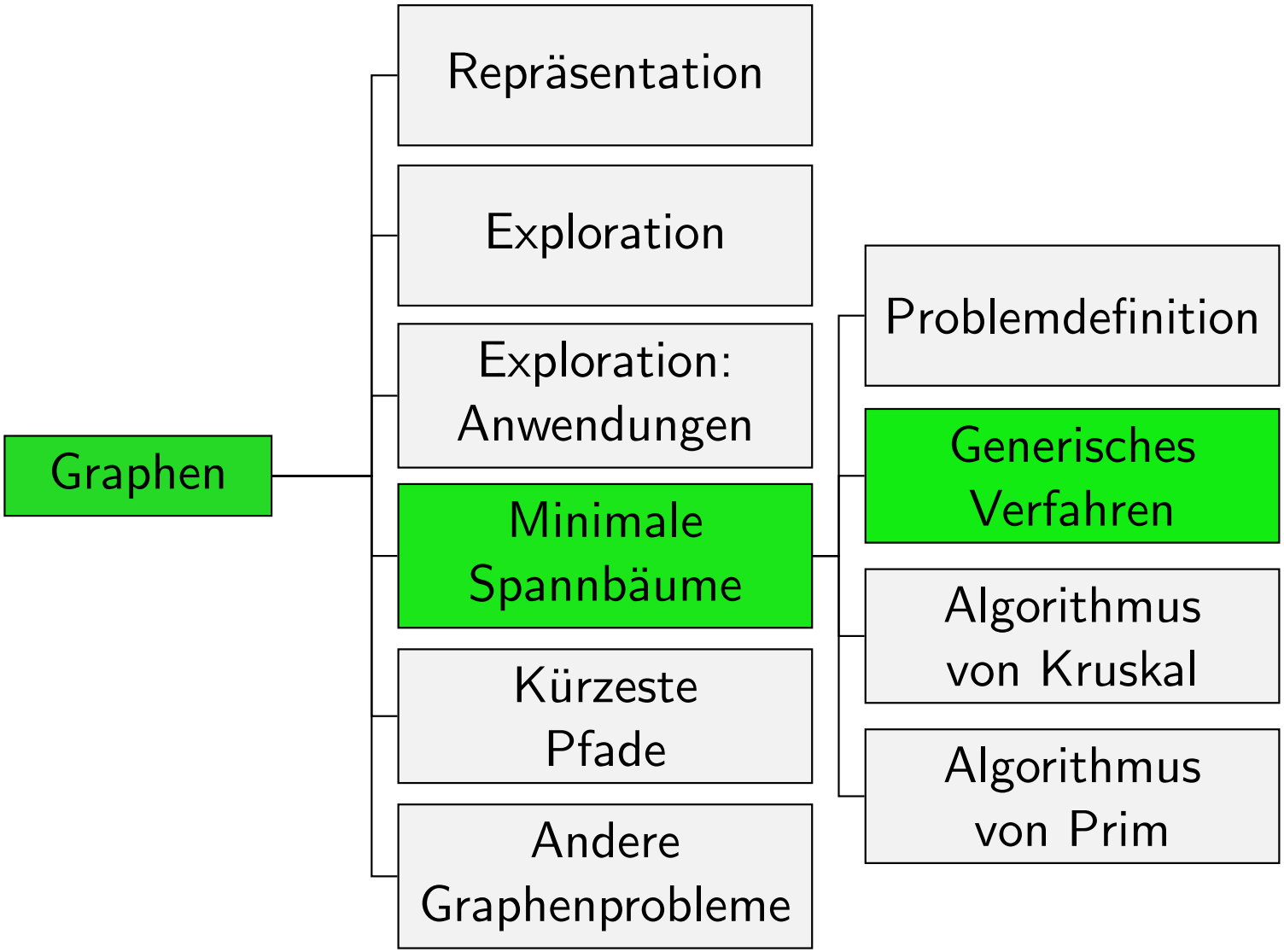
Published in: Image and Signal Processing and Analysis (ISPA), 2017 10th International Symposium on

Anwendungen

- Netzwerkdesign
  - z.B. Kommunikationsnetze, Stromnetze, hydraulische Netze
- Segmentierung
  - z.B. von Zellkernen in Mikroskopiebildern
- Cluster-Analyse
  - z.B. von Zellkernen zur Krebsdiagnose
- Approximation schwieriger Graphenprobleme
  - Steiner-Bäume, Traveling Salesperson
- Viele indirekte Anwendungen
  - LDPC fehlerkorrigierende Codes
  - Features für Gesichtsverifikation etc.
  - Ethernetprotokoll zum Vermeiden von Zykeln beim Broadcasting
  - Partikelinteraktion in turbulenten Flüssigkeitsströmungen

C4.2 Generischer Algorithmus

Graphen: Übersicht

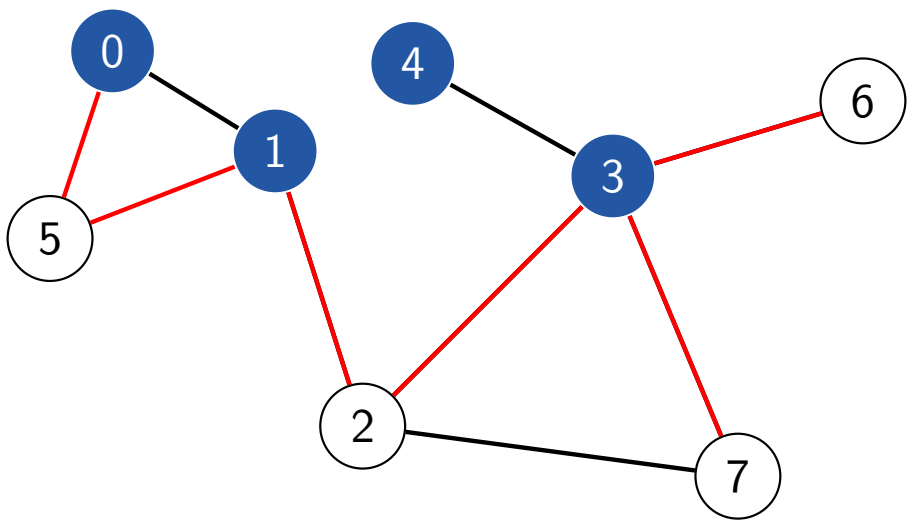


Schnitte in Graphen

Definition

Sei $G = (V, E)$ ein ungerichteter Graph und $V' \subseteq V$.

Der von V' induzierte Schnitt $S_{V'}$ besteht aus allen Kanten aus E, bei denen genau ein Endpunkt in V' liegt.



Generische Schritte

- Drei Zustände von Kanten
  - unbearbeitet
  - akzeptiert
  - abgelehnt
- Akzeptanzschritt:
  - Wähle einen Schnitt S, der keine akzeptierte Kante enthält.
  - Akzeptiere eine unbearbeitete Kante in S mit minimalem Gewicht.
- Ablehnungsschritt:
  - Wähle einen Zyklus Z, der keine abgelehnte Kante enthält.
  - Lehne eine unbearbeitete Kante in Z mit maximalem Gewicht ab.

Generischer Algorithmus

Eingabe: Zusammenhängender, ungerichteter Graph $G = (V, E)$

1. Setze alle Kanten auf unbearbeitet
2. Solange noch Kanten unbearbeitet sind:
   - Wende nicht-deterministisch einen Akzeptanz- oder Ablehnungsschritt an.
3. Die akzeptierten Kanten bilden einen MST.

Greedy-Verfahren: Trifft lokal optimale Entscheidungen
Hier ist das immer auch eine global optimale Entscheidung

Generischer Algorithmus: Vollständigkeit

Theorem

Jede Instanziierung des generischen Algorithmus terminiert.

Beweisskizze

- Knoten bilden mit den akzeptierten Kanten einen Wald W.
- Betrachte unbearbeitete Kante $e = \{v, v'\}$
  - Fall 1: Hinzufügen von e zu W führt zu Zyklus
    $\rightarrow$ Ablehnungsschritt mit e möglich
    (e ist einzige unbearbeitete Kante in Zyklus)
  - Fall 2: Hinzufügen von e zu W führt nicht zu Zyklus
    $\rightarrow$ Die Endpunkte von e liegen nicht in gleicher Zusammenhangskomponente von W.
    $\rightarrow$ Betrachte Knotenmenge V', die v und alle in W mit v verbundenen Knoten enthält.
    $\rightarrow$ Akzeptanzschritt mit von V' induziertem Schnitt möglich (von unbearbeiteter Kante mit minimalem Gewicht)

Generischer Algorithmus: Korrektheit

Theorem

Nach der Terminierung bilden die akzeptierten Kanten einen MST.

Beweis

Induktion über die Anzahl der Schritte.

Induktionshypothese: Es gibt einen MST B, der alle akzeptierten Kanten und keine abgelehnte Karte enthält.

Induktionsanfang: Keine Kanten akzeptiert oder abgelehnt, daher erfüllt jeder MST die Bedingung. ...

Generischer Algorithmus: Korrektheit

Beweis (Fortsetzung).

Induktionsschritt:

Fall 1: Akzeptanzschritt

- Sei S der betrachtete Schnitt und e die akzeptierte Kante.
- Falls e in B, ist Ind.hypothese für B weiterhin erfüllt.
- Sonst erzeugt Hinzufügen von e zu B Zyklus Z, der eine weitere Kante e' aus S enthält.
- Kante e' ist unbearbeitet: nicht abgelehnt, da in B; nicht akzeptiert, da in S
- $\mathit{weight}(e) \leq \mathit{weight}(e')$, da e akzeptiert wurde
- Erzeuge B' aus B durch Entfernen von Kante e' und Hinzufügen von Kante e.
- B' ist MST und erfüllt Ind.hypothese.

...

Generischer Algorithmus: Korrektheit

Beweis (Fortsetzung).

Fall 2: Ablehnungsschritt

- Sei Z der betrachtete Zyklus und e die abgelehnte Kante.
- Falls e nicht in B, ist Ind.hypothese für B weiterhin erfüllt.
- Sonst zerfällt B durch Entfernen von e in zwei Zusammenhangskomponenten.
- Betrachte Schnitt S zwischen den Komponenten.
- S enthält eine weitere Kante e' aus Z.
- Kante e' ist unbearbeitet:
  nicht abgelehnt, da in Z; nicht akzeptiert, da nicht in B
- $\mathit{weight}(e) \geq \mathit{weight}(e')$, da e abgelehnt wurde.
- Erzeuge B' aus B durch Entfernen von Kante e und Hinzufügen von Kante e': MST und erfüllt Ind.hypothese

Generischer Algorithmus

Input: Zusammenhängender, ungerichteter Graph $G = (V, E)$

1. Setze alle Kanten auf unbearbeitet
2. Solange noch Kanten unbearbeitet sind:
   - Wende nicht-deterministisch einen Akzeptanz- oder Ablehnungsschritt an.
3. Die akzeptierten Kanten bilden einen MST.

Beobachtung

Wir können nach $|V| - 1$ akzeptierten Kanten abbrechen.

Warum?

Offene Fragen

- Wie wählen wir geschickt die nächste Kante zum Akzeptieren oder Ablehnen?
  - Algorithmus von Kruskal
  - Algorithmus von Prim
- Vorher: Wie repräsentieren wir den gewichteten Graphen?

C4.3 Graphenrepräsentation

Repräsentation gewichteter Kanten

Erweiterung bisheriger Repräsentationen möglich

- Adjazenzmatrix: Gewicht statt binärerer Einträge
  - Können wir parallele Kanten unterstützen?
- Adjazenzliste: Paare von Nachfolger und Gewicht in Liste

Aber

- Generischer Algorithmus konzentriert sich auf Kanten
- Daher: Repräsentiere Kanten als Objekte

API für gewichtete Kante

```
1  class Edge:
2    # Kante zwischen n1 und n2 mit Gewicht w
3    def __init__(n1: int, n2: int, w: float) -> None
4
5    # Gewicht der Kante
6    def weight() -> float
7
8    # Einer der beiden Knoten
9    def either_node() -> int
10
11   # Der andere Knoten (nicht n)
12   def other_node(int n) -> int
```

Gewichtete Kante: Mögliche Implementierung

```
1  class Edge:
2      def __init__(self, n1, n2, weight):
3          self.n1 = n1
4          self.n2 = n2
5          self.edge_weight = weight
6
7      def weight(self):
8          return self.edge_weight
9
10     def either_node(self):
11         return self.n1
12
13     def other_node(self, n):
14         if self.n1 == n:
15             return self.n2
16         return self.n1
```

Repräsentation gewichteter Graphen

Graphenrepräsentation

- Wir wollen weiterhin schnell die an einem Knoten anliegenden Kanten bestimmen können.
- Speichere für jeden Knoten Referenzen auf die anliegenden Kanten.
- Benötigen für jede Kante ein Objekt und zwei Referenzen darauf.

API für gewichtete Graphen

```
1  class EdgeWeightedGraph:
2    # Graph mit no_nodes Knoten und keinen Kanten
3    def __init__(no_nodes: int) -> None
4
5    # Füge gewichtete Kante hinzu
6    def add_edge(e: Edge) -> None
7
8    # Anzahl der Knoten
9    def no_nodes() -> int
10
11   # Anzahl der Kanten
12   def no_edges() -> int
13
14   # Alle an Knoten n anliegenden Kanten
15   def adjacent_edges(n: int) -> Generator[Edge]
16
17   # Alle Kanten
18   def all_edges() -> Generator[Edge]
```

Gewichteter Graph: Mögliche Implementierung

```
1  class EdgeWeightedGraph:
2      def __init__(self, no_nodes):
3          self.nodes = no_nodes
4          self.edges = 0
5          self.adjacent= [[] for l in range(no_nodes)]
6
7      def add_edge(self, edge):
8          either = edge.either_node()
9          other = edge.other_node(either)
10         self.adjacent[either].append(edge)
11         self.adjacent[other].append(edge)
12         self.edges += 1
13
14     def no_nodes(self):
15         return self.nodes
16
17     def no_edges(self):
18         return self.edges
```

Gewichteter Graph: Mögliche Implementierung (Forts.)

```
19
20      def adjacent_edges(self, node):
21          for edge in self.adjacent_edges[node]:
22              yield edge
23
24      def all_edges(self):
25          for node in range(self.nodes):
26              for edge in self.adjacent_edges[node]:
27                  if edge.other_node(node) > node:
28                      yield edge
```

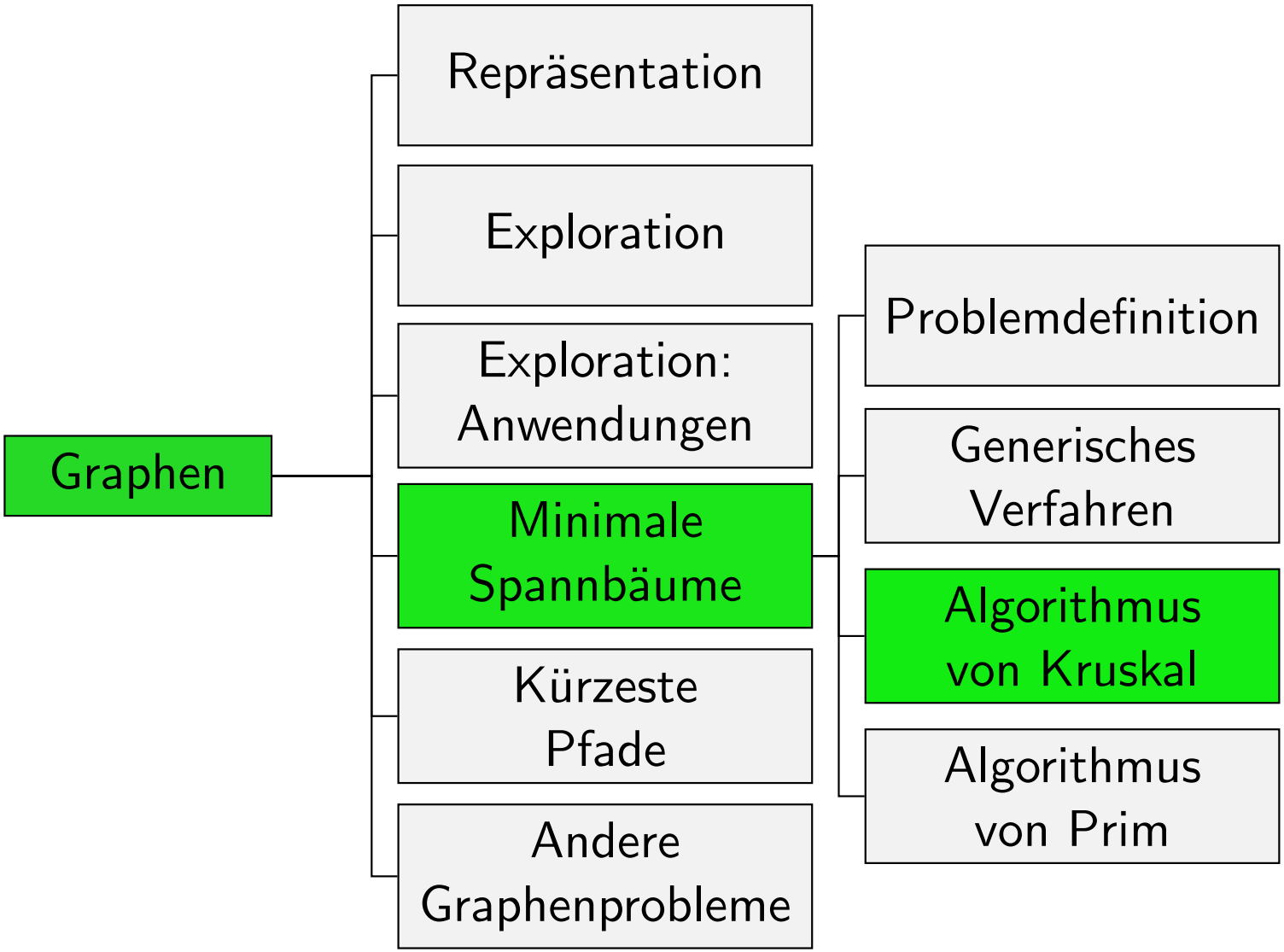
API für MST-Implementierungen

Die Algorithmen für minimale Spannbäume sollen folgendes Interface implementieren:

```
1  class MST:
2    # Konstruktor
3    def __init__(graph: EdgeWeightedGraph) -> None
4
5    # Alle Kanten eines minimalen Spannbaums
6    def edges() -> Generator[Edge]
7
8    # Gewicht des minimalen Spannbaums
9    def weight() -> int
```

C4.4 Kruskals Algorithmus

Graphen: Übersicht



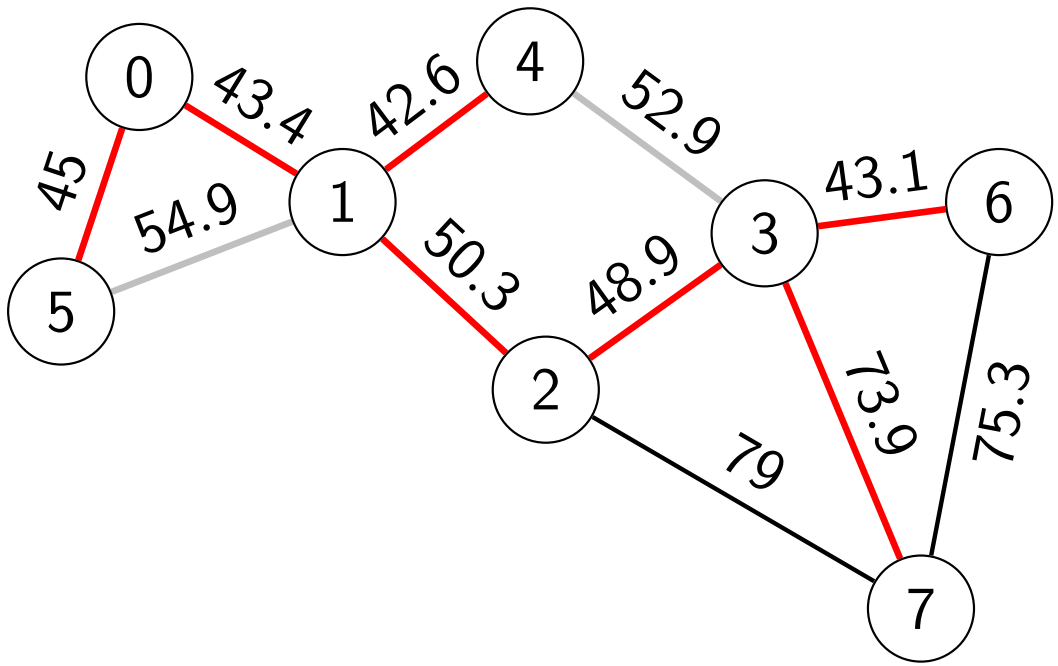
High-Level-Perspektive

Algorithmus von Kruskal

- Verarbeite Kanten in aufsteigender Reihenfolge ihrer Gewichte.
- Akzeptiere Kante, wenn sie mit bereits akzeptierten Kanten keinen Zyklus bildet. Sonst lehne sie ab.
- Nach $|V| - 1$ akzeptierten Kanten fertig

Wieso ist das eine Instanziierung des generischen Algorithmus?

Illustration



rot: akzeptiert

grau: abgelehnt

Algorithmus von Kruskal konzeptionell

Konzeptionelles Vorgehen

- Beginne mit Wald von $|V|$ Bäumen,
  die jeweils nur aus einem Knoten bestehen.
- Jeder Akzeptanzschritt verbindet zwei Bäume zu einem.
- Nach $|V| - 1$ Schritten besteht der Wald aus einem Baum.

Fragen

- Wie können wir feststellen, ob eine Kante
  zwei Bäume miteinander verbindet oder
  ob beide Endknoten im gleichen Baum liegen?
- Müssen wir die einzelnen Bäume vollständig repräsentieren?

$\rightarrow$ Uns interessieren nur die Zusammenhangskomponenten

$\rightarrow$ Union-Find zur Hilfe!

Algorithmus von Kruskal: Implementierung

```
1  class MSTKruskal:
2      def __init__(self, graph):
3          self.included_edges = []
4          self.total_weight = 0
5          candidates = minPQ()  # priority queue
6          for edge in graph.all_edges():
7              candidates.insert(edge)
8          uf = UnionFind(graph.no_nodes())
9
10         while (not candidates.empty() and
11                len(self.included_edges) < graph.no_nodes() - 1):
12             edge = candidates.del_min()
13             v = edge.either_node()
14             w = edge.other_node(v)
15             if uf.connected(v, w):
16                 continue
17             uf.union(v,w)
18             self.included_edges.append(edge)
19             self.total_weight += edge.weight()
```

Wie sehen Methoden edges() und weight() aus?

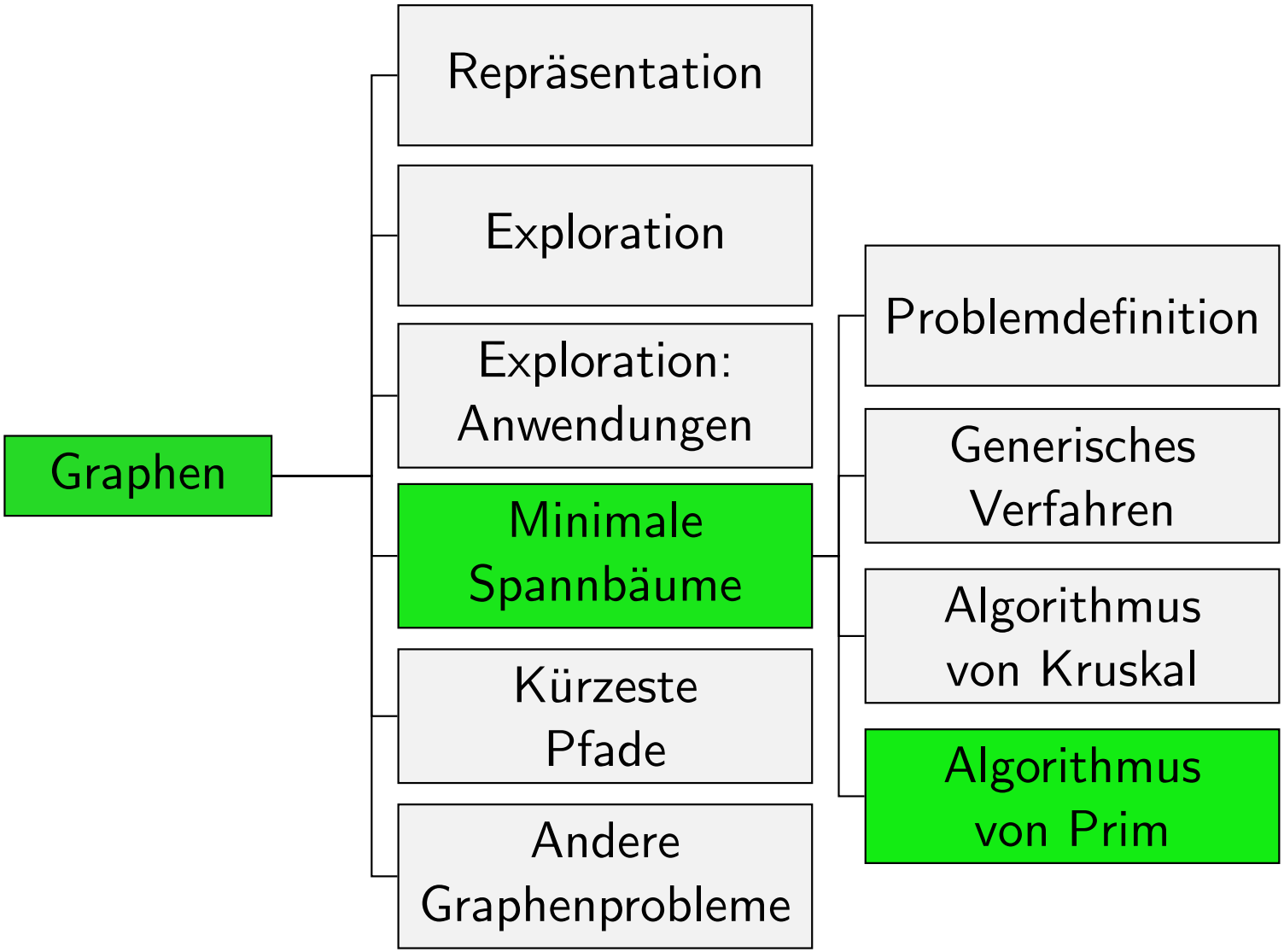
Algorithmus von Kruskal: Laufzeit

- Annahme: Heap-Implementierung der Priority-Queue
- Initialisierung Priority-Queue mit allen Kanten: $|E|$ Vergleiche
- Nie mehr als $|E|$ Kanten in Priority-Queue
  - Kosten pro Operation in $O(\log_2 |E|)$
  - Insgesamt Kosten für Priority-Queue-Operationen in $O(|E| \log_2 |E|)$
- Dominiert Kosten für Union-Find-Struktur

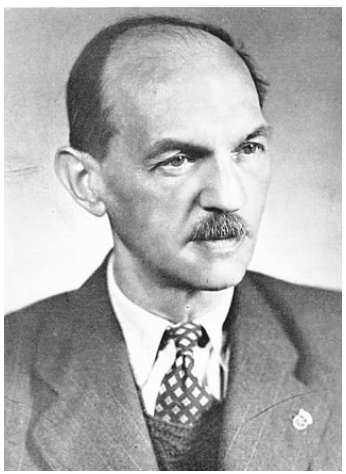
Insgesamt: Laufzeit in $O(|E| \log_2 |E|)$, Speicherbedarf in $O(|E|)$

C4.5 Prims Algorithmus

Graphen: Übersicht



Mathematiker des Tages: Vojtěch Jarník



Vojtěch Jarník

- Tschechischer Mathematiker, 1897–1970
- Beiträge zur Zahlentheorie und Analysis
- 1930: Algorithmus für minimale Spannbäume
  - 1959: nochmal entdeckt durch Robert Prim
  - 1961: nochmal entdeckt durch Edsger Dijkstra
  - Verschiedene Namen:
    - Algorithmus von Prim
    - Algorithmus von Jarník, Prim und Dijkstra
    - Prim-Dijkstra-Algorithmus
    - DJP algorithm

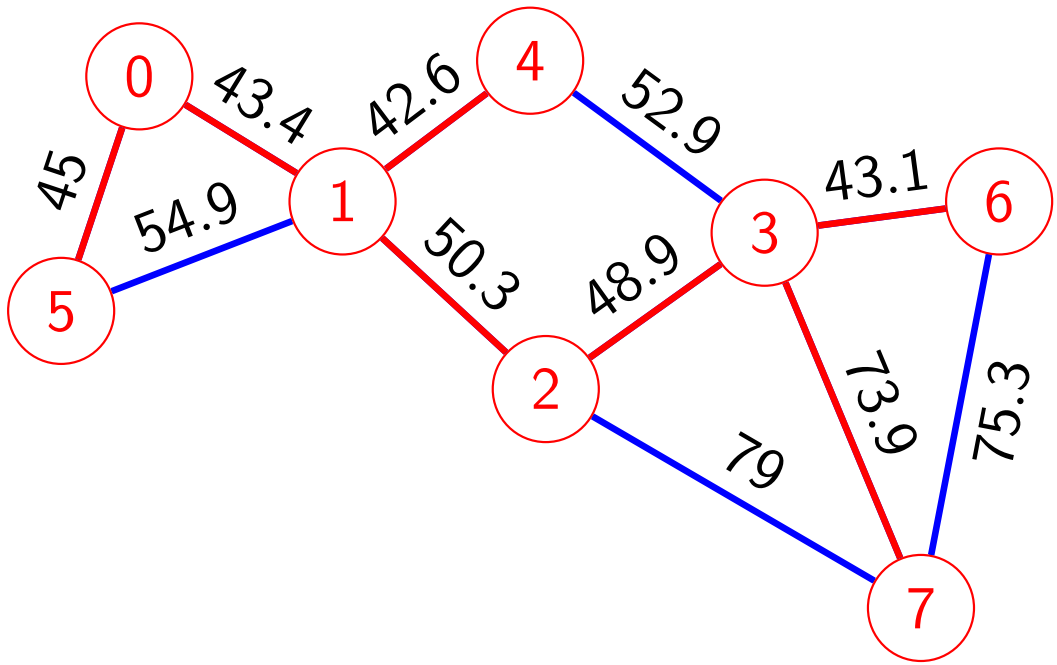
High-Level-Perspektive

Algorithmus von Prim

- Wähle einen zufälligen Knoten als initialen Baum.
- Lasse Baum schrittweise um eine weitere Kante wachsen
- Füge jeweils Kante mit minimalem Gewicht hinzu, die genau einen Endknoten im Baum hat.
  $\rightarrow$ Akzeptanzschritt
- Fertig, wenn $|V| - 1$ Kanten hinzugefügt.

Illustration

Mit Startknoten 0



rot: akzeptiert

blau: potentielle nächste Kante

Implementierung

Schwierigkeit

Finde die Kante mit minimalem Gewicht, die genau einen Endpunkt im Baum hat.

- Priority Queue `candidates`, die Kanten nach Gewicht ordnet.
- Zwei Versionen:
  - eager: nur Kanten, die exakt einen Endpunkt im Baum haben
  - lazy: Kanten, die mindestens einen Endpunkt im Baum haben

Hauptschleife Lazy-Version

Invariante

Priority-Queue `candidate`

- enthält alle Kanten mit genau einem Endpunkt im Baum
- und möglicherweise Kanten mit beiden Endpunkten im Baum.

Solange noch nicht $|V| - 1$ Kanten hinzugefügt wurden:

- Nimm Kante e mit minimalen Kosten aus Priority-Queue
- Verwirf e, falls beide Endpunkte im Baum.
- Sonst sei v Endpunkt, der nicht im Baum ist
  - Füge alle an v anliegenden Kanten, deren anderer Endpunkt nicht im Baum ist, zu `candidates` hinzu.
  - Füge e und v zum Baum hinzu.

Lazy Prim-Algorithmus

```
1  class LazyPrim:
2      def __init__(self, graph):
3          self.included_edges = []
4          self.total_weight = 0
5
6          # node-indexed list: True if node already in tree
7          included_nodes = [False] * graph.no_nodes()
8          candidates = minPQ()
9
10         # include an arbitrary node (we use 0) in tree
11         included_nodes[0] = True
12         for edge in graph.adjacent_edges(0):
13             candidates.insert(edge)
```

Lazy Prim-Algorithmus (Forts.)

```
14
15          while (not candidates.empty() and
16                 len(self.included_edges) < graph.no_nodes() - 1):
17              edge = candidates.del_min()
18              v = edge.either_node()
19              w = edge.other_node(v)
20              if included_nodes[v] and included_nodes[w]:
21                  continue
22              if included_nodes[w]:
23                  v, w = w, v
24              # v is in tree, w is not
25              included_nodes[w] = True
26              self.included_edges.append(edge)
27              self.total_weight += edge.weight()
28              for adjacent in graph.adjacent_edges(w):
29                  if not included_nodes[adjacent.other_node(w)]:
30                      candidates.insert(adjacent)
```

Laufzeit und Speicherbedarf

- Engpass ist Anzahl der Vergleiche von Kantengewichten in Methoden `insert` und `del_min` der Priority-Queue.
- Höchstens $|E|$ Kanten in Priority-Queue
- Einfügen und Entfernen des Minimums jeweils in $O(\log |E|)$
- Höchstens $|E|$ Einfüge- und $|E|$ Lösch-Operationen
  $\rightarrow$ Laufzeit $O(|E| \log |E|)$
- Speicherbedarf $O(|E|)$

Eager-Version

Überlegungen

- Wir könnten Kanten, die bereits beide Endpunkte im Baum haben, aus der Priority-Queue entfernen.
- Gibt es mehrere Kanten, die einen noch nicht enthaltenen Knoten mit dem Baum verbinden, können nur die mit minimalem Gewicht gewählt werden.
- Es reicht, jeweils nur eine solche Kante zu betrachten.
- Idee: Merke dir eine solche Kante für jeden Knoten
- Priority-Queue enthält Knoten, wobei die Priorität das Gewicht der gespeicherten Kante ist.

Problem: Wie können wir günstig die Priority-Queue updaten?

Exkurs: Indizierte Vorrangwarteschlange

```
1   class IndexMinPQ:
2     # Fügt key mit Priorität val ein
3     def insert(entry: Object, val: int) -> None
4
5     # Entfernt Eintrag mit kleinster Priorität
6     # und liefert ihn zurück
7     def del_min() -> Object
8
9     # Ist die Priority-Queue leer?
10    def empty() -> bool
11
12    # Ist Eintrag enthalten?
13    def contains(entry: Object) -> bool
14
15    # Ändert Priorität von entry auf val
16    def change(entry: Object, val: int) -> None
17
18    ...
```

Exkurs: Indizierte Vorrangwarteschlange

Priority-Queue-Implementierung kann leicht erweitert werden.

Mit der heap-basierten Implementierung erhält man dabei Laufzeit

- $O(\log n)$ für insert, change und del_min
- $O(1)$ für contains und empty

Eager Prim-Algorithmus: Datenstrukturen

Verwende nicht (indizierte) Priority-Queue von Kanten, sondern

- `edge_to`: knotenindiziertes Array, das an Stelle v die Kante (`Edge`) enthält, die v (in Richtung des gewählten Startknotens) mit dem Baum verbindet bzw. das am günstigsten könnte.
- `dist_to`: Array, das an Stelle v das Gewicht von Kante `edge_to[v]` enthält.
- `pq`: indizierte Priority-Queue von Knoten
  - Knoten noch nicht im Baum
  - Können aber mit einer Kante mit dem bestehenden Baum verbunden werden
  - Sortiert nach Gewicht der günstigsten solchen Kante

Eager Prim-Algorithmus

```
1  class EagerPrim:
2      def __init__(self, graph):
3          self.edge_to = [None] * graph.no_nodes()
4          self.total_weight = 0
5          self.dist_to = [float('inf')] * graph.no_nodes()
6          self.included_nodes = [False] * graph.no_nodes()
7
8          self.pq = IndexMinPQ()
9
10         self.dist_to[0] = 0
11         self.pq.insert(0, 0)
12         while not self.pq.empty():
13             self.visit(graph, self.pq.del_min())
```

Eager Prim-Algorithmus (Forts.)

```
14
15      def visit(self, graph, v):
16          self.included_nodes[v] = True
17          for edge in graph.adjacent_edges(v):
18              w = edge.other_node(v)
19              if self.included_nodes[w]:
20                  continue
21              if edge.weight() < self.dist_to[w]:
22                  # update cheapest edge between tree and w
23                  self.edge_to[w] = edge
24                  self.dist_to[w] = edge.weight()
25                  if self.pq.contains(w):
26                      self.pq.change(w, edge.weight())
27                  else:
28                      self.pq.insert(w, edge.weight())
```

Laufzeit und Speicherbedarf

- Drei knotenindizierte Arrays
- Höchstens $|V|$ Knoten in Priority-Queue
- Speicherbedarf $O(|V|)$
- Priority-Queue: Benötigen $|V|$ Einfügeoperationen, $|V|$ Operationen zum Entfernen des Minimums und höchstens $|E|$ Prioritätsänderungen
- Jeweils in Zeit $O(\log |V|)$ möglich
- Laufzeit $O(|E| \log |V|)$

C4.6 Ausblick

Gibt es einen MST-Algorithmus mit linearer Laufzeit?

Algorithmus	Speicher	Zeit
Kruskal	$\|E\|$	$\|E\| \log \|E\|$
Lazy-Prim	$\|E\|$	$\|E\| \log \|E\|$
Eager-Prim	$\|V\|$	$\|E\| \log \|V\|$
Fredman-Tarjan	$\|V\|$	$\|E\| + \|V\| \log \|V\|$
Chazelle	$\|V\|$	$\|E\|\alpha(\|V\|)$ (beinahe $\|E\|$)
unmöglich?	$\|V\|$	$\|E\|$?

Es gibt randomisiertes Verfahren mit linearen Zeitbedarf (Erwartungswert) [Karger, Klein, Tarjan, 1995].