Datenstrukturen und Algorithmen

Vorlesung 16: Minimale Spannbäume (K23)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

https://moves.rwth-aachen.de/teaching/ss-18/dsal/

22. Juni 2018

Übersicht

1956

Jarník, 1926

Übersicht

Spannbaum Probleme

Spannbaum

Ein Spannbaum eines ungerichteten, zusammenhängenden Graphen G ist ein Teilgraph von G, der ein ungerichteter Baum ist und alle Knoten von G enthält.

Spannbaum Probleme

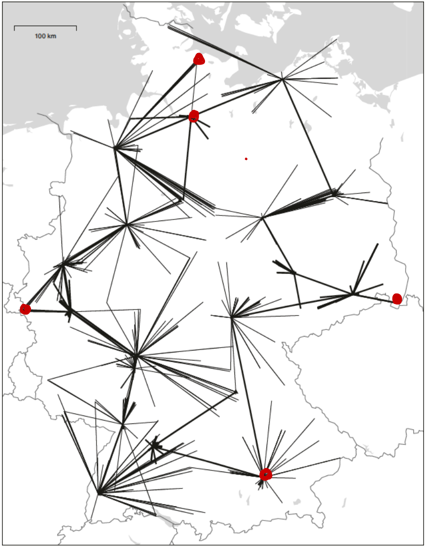
Spannbaum

Ein Spannbaum eines ungerichteten, zusammenhängenden Graphen G ist ein Teilgraph von G, der ein ungerichteter Baum ist und alle Knoten von G enthält.

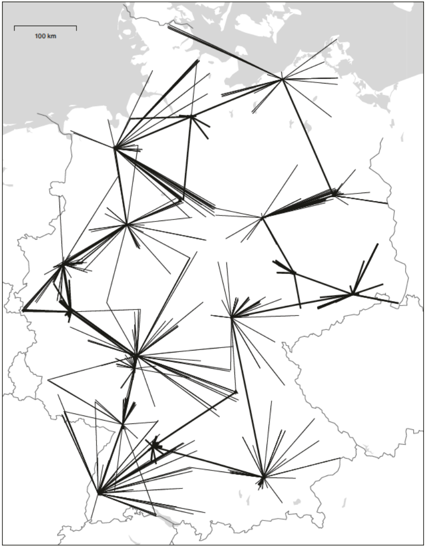
Beispiel (Spannbaum Probleme)

- Verbinde alle Kunden durch Glasfaberkabeln
- Computernetzwerke verkabeln
- Verdrahtung von Schaltungen beim Chipdesign
- Erneuere die Straßenbeläge zwischen alle Flughafenterminals
- ......

Glasfabernetz

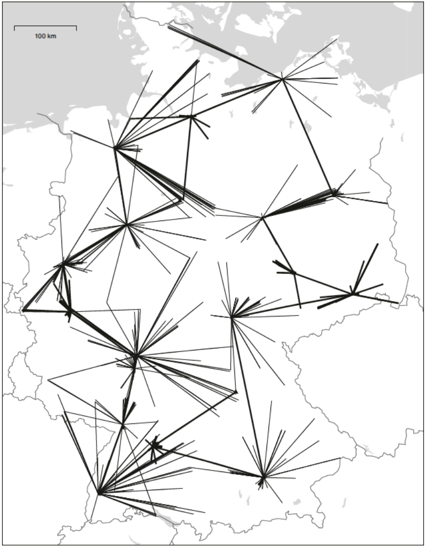


Glasfabernetz



- Müssen alle Leitungen erneuert werden?
- Wie wählt man die zu erneuernden Abschnitte aus?
- Gibt es eine eindeutige Lösung?

Glasfabernetz

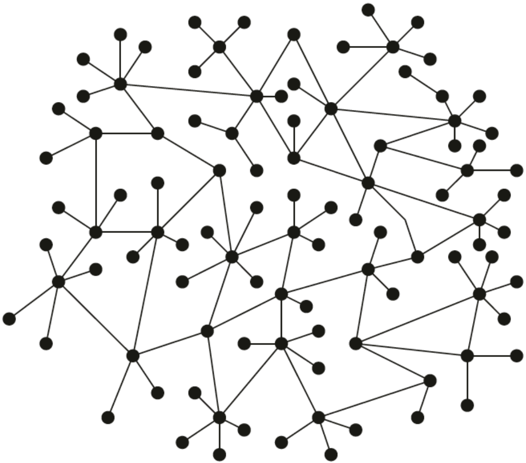


- Müssen alle Leitungen erneuert werden?
- Wie wählt man die zu erneuernden Abschnitte aus?
- Gibt es eine eindeutige Lösung?

so daß

- Alle Kunden profitieren
- Jeder kann mit jedem über Glasfaber kommunizieren
- Sparsamkeit: keine doppelte Verbindungen

Telefonleitungsnetz



Zwei Abdeckungen

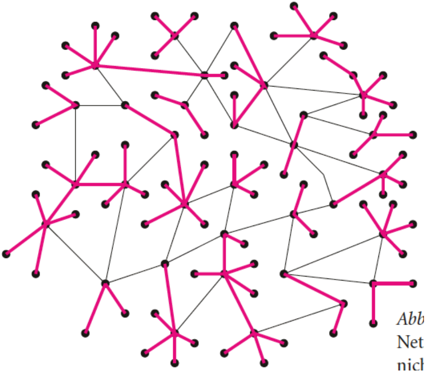
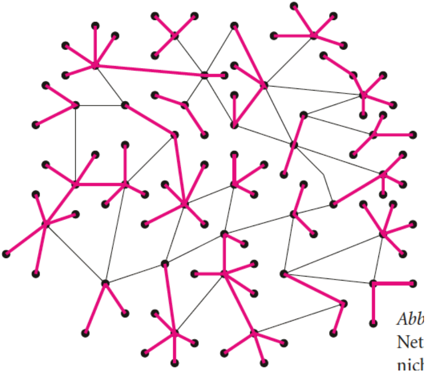


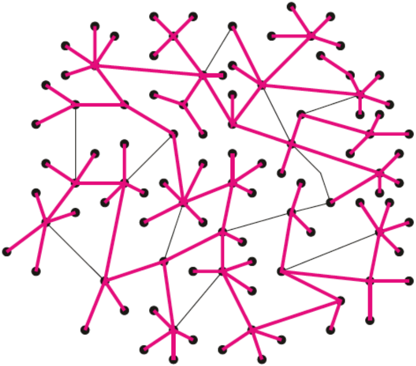
Abb
Net
nich

Alle Kunden sind angeschlossen

Zwei Abdeckungen



Alle Kunden sind angeschlossen



... und sind miteinander verbunden

Einige Fakten bzgl. Bäume

Satz

In einem Baum sind je zwei Knoten durch genau einen Weg verbunden.

Einige Fakten bzgl. Bäume

Satz

In einem Baum sind je zwei Knoten durch genau einen Weg verbunden.

Satz

Ein Baum ist minimal zusammenhängend und maximal kreisfrei.

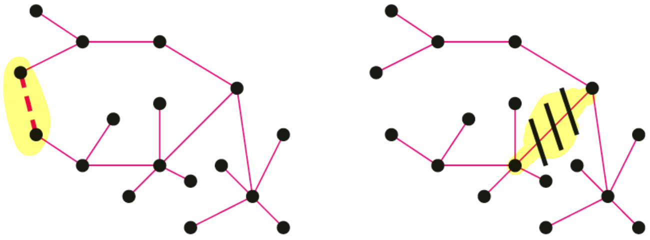
Einige Fakten bzgl. Bäume

Satz

In einem Baum sind je zwei Knoten durch genau einen Weg verbunden.

Satz

Ein Baum ist minimal zusammenhängend und maximal kreisfrei.



Fügt man eine Kante in einem Baum hinzu, entsteht ein Kreis.

Löscht man eine Kante, erhält man zwei Zusammenhangskomponenten.

Einige Fakten bzgl. Bäume

Satz

Ein Baum ist minimal zusammenhängend und maximal kreisfrei.

Beweis.

In jedem Baum sind je 2 Knoten durch einen eindeutigen Weg verbunden.

Einige Fakten bzgl. Bäume

Satz

Ein Baum ist minimal zusammenhängend und maximal kreisfrei.

Beweis.

In jedem Baum sind je 2 Knoten durch einen eindeutigen Weg verbunden. Löscht man eine Kante, wird mindestens einer dieser Wege unterbrochen.

Einige Fakten bzgl. Bäume

Satz

Ein Baum ist minimal zusammenhängend und maximal kreisfrei.

Beweis.

In jedem Baum sind je 2 Knoten durch einen eindeutigen Weg verbunden. Löscht man eine Kante, wird mindestens einer dieser Wege unterbrochen. Damit ist der Restgraph nicht zusammenhängend.

Einige Fakten bzgl. Bäume

Satz

Ein Baum ist minimal zusammenhängend und maximal kreisfrei.

Beweis.

In jedem Baum sind je 2 Knoten durch einen eindeutigen Weg verbunden. Löscht man eine Kante, wird mindestens einer dieser Wege unterbrochen. Damit ist der Restgraph nicht zusammenhängend. Fügt man eine Kante (u, v) hinzu, dann bekommt man zusätzlich zu den eindeutigen Weg im Baum zwischen u und v einen weiteren Weg zwischen u und v. Das ergibt ein Kreis. $\square$

Einige Fakten bzgl. Bäume

Satz

Ein Baum ist minimal zusammenhängend und maximal kreisfrei.

Beweis.

In jedem Baum sind je 2 Knoten durch einen eindeutigen Weg verbunden. Löscht man eine Kante, wird mindestens einer dieser Wege unterbrochen. Damit ist der Restgraph nicht zusammenhängend. Fügt man eine Kante (u, v) hinzu, dann bekommt man zusätzlich zu den eindeutigen Weg im Baum zwischen u und v einen weiteren Weg zwischen u und v. Das ergibt ein Kreis. □

1. Jeder Baum (mit mindestens 2 Knoten) besitzt mindestens 2 Blätter
2. Ein Baum mit n Knoten hat $n-1$ Kanten.

Beweis.

Hausaufgabe. Hinweis für 2: plücken Sie Blatt um Blatt ab. □

Anzahl der Spannbäume

Spannbaum

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

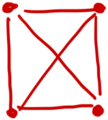
Anzahl der Spannbäume

Spannbaum

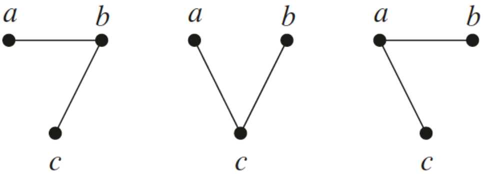
Ein Spannbaum eines ungerichteten, zusammenhängenden Graphen G ist ein Teilgraph von G, der ein Baum ist und alle Knoten von G enthält.

Die Cayley Formel

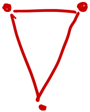
Den vollständigen Graphen mit n Knoten hat n^{n-2} Spannbäume.



G_4

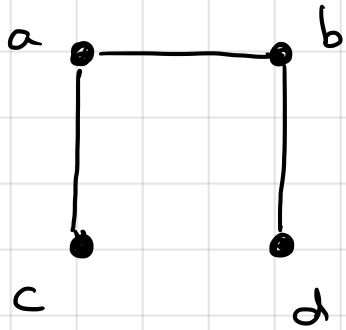


3 Spannbäume für n=3 Knoten



G_3

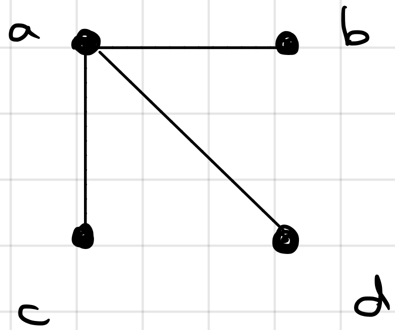
Spannbäume für vollständige Graphen $n=4$



„Kette"

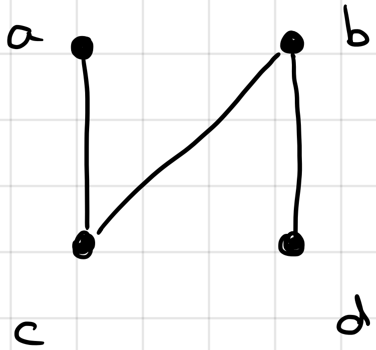
+ 3 Varianten

(keine Kante zwischen a und b statt c und d) usw.

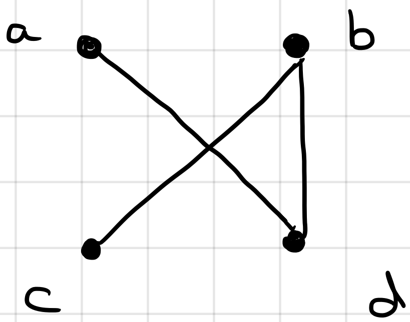


+ 3 Varianten „Stern"

(Wurzel b statt a usw.)



+ 3 Varianten „Zorro"



+ 3 Varianten „Stuhl"

$\Longrightarrow 4^{4-2} = 16$ Spannbäume

Der vollständige Graph G_n hat n^{n-2} Spannbäume (für $n \geq 2$)

<u>Bew:</u> zeige Gleichmächtigkeit zwei Menge durch eine Bijektion.

Menge 1 = Menge der Spannbäume von G_n

Menge 2 = Menge der $(n-2)$-Tupel mit Einträgen aus $\{1, \dots, n\}$

Offensichtlich $|\text{Menge } 2| = n^{n-2}$

Bijektion: Spannbaum B von G_n $\longleftrightarrow$ $(n-2)$ Tupel

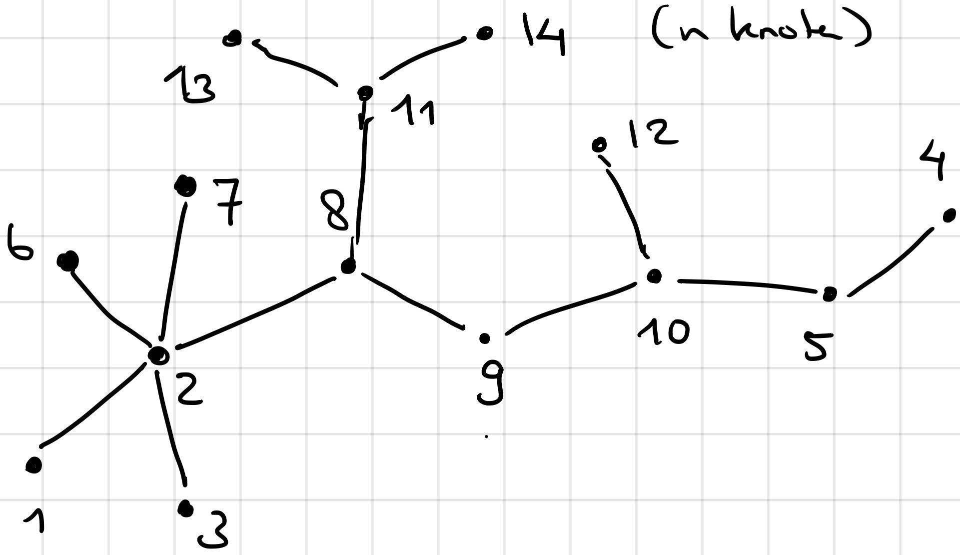
"$\longrightarrow$": 1. nummeriere die Knoten von B durch

2. nimm das Blatt in B mit dem kleinsten Nummer. Die Nummer <u>seines</u> <u>Nachbars</u> wird das nächste Element des $(n-2)$-Tupels

3 entferne dieses Blatt aus B

4. wiederhole 2+3 bis 2 Knoten (mit einer verbindende Kante) übrig sind.

Beispiel "→" Baum B ⟼ (n-2) Tupel
(n Knoten)

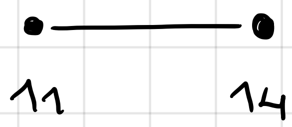


n = 14

lösche Blatt	1	3	4	5	6	7	2	12	10	9	8	13
Code	2	2	5	10	2	2	8	10	9	8	11	14

$\underbrace{\qquad\qquad\qquad\qquad}_{12}$

übrigen Graph



"$\longleftarrow$": $(n-2)$ Tupel $\longmapsto$ Spannbaum B (für n)

Schreibe die zwei Zahlenfolgen auf

$1 \quad 2 \quad \ldots \ldots \quad n$ Knoten

$k_1 \quad k_2 \quad k_3 \quad \ldots \quad k_{n-2}$ $(n-2)$ Tupel

1. Fange an mit einem leeren (kantenlose) Graphen mit n durchnummerierte Knoten
2. Suche die kleinste nicht im Code enthaltene
   a. Zahl und streiche die (oberste Zeile)
   b. Streiche die erste Zahl k_1 aus der Code
   c. Füge eine Kante hinzu zwischen diese beide Zahlen
3. Wiederhole 2. bis $n-2$ Kanten gezeichnet sind

Die letzte Kante ergibt sich aus den beiden in der oberen Zeile übriggebliebene Zahlen.

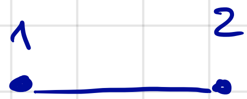
Zu zeigen: wir erhalten so ein Baum

$(n-2)$ Tupel $\longrightarrow$ Baum B_n

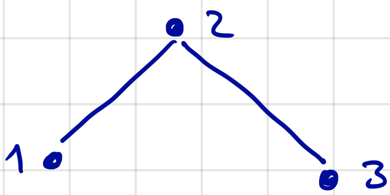
" ⟵ " $(n-2)$ Tupel $\longmapsto$ Baum B_n

~~1~~	2	~~3~~	~~4~~	~~5~~	~~6~~	7	8	9	10	11	12	13	14
~~2~~	~~2~~	~~8~~	~~10~~	~~2~~	2	8	10	9	8	11	14		

a.



b.



c.

etcetera.

Anzahl der Spannbäume

Die Cayley Formel

Den vollständigen Graphen mit n Knoten hat n^{n-2} Spannbäume.

Beweis.

In der Vorlesung. $\square$

Übersicht

Kosten

Kosten

Beispiel

- Finde den kostengünstigsten Weg, um eine Menge von Flughafenterminals, Städten, ... zu verbinden

Minimale Spannbäume

Kosten

Beispiel

- Finde den kostengünstigsten Weg, um eine Menge von Flughafenterminals, Städten, ... zu verbinden
- Verdrahtung von Schaltungen mit geringstem Energieverbrauch
- Verbinde alle Kunden kostengünstig durch Glasfaberkabeln
- Computernetzwerke verkabeln

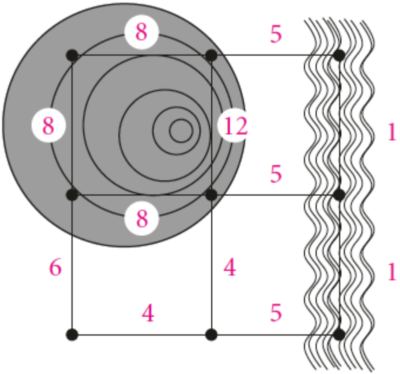
Kosten

Beispiel

- Finde den kostengünstigsten Weg, um eine Menge von Flughafenterminals, Städten, ... zu verbinden
- Verdrahtung von Schaltungen mit geringstem Energieverbrauch
- Verbinde alle Kunden kostengünstig durch Glasfaberkabeln
- Computernetzwerke verkabeln

Das ist die Begründung für (kosten-) minimale Spannbäume!

Leitungskosten



Unterschiedliche Umgebungen verursachen unterschiedliche Kosten für die Verlegung von Kabeln.

Was ist ein minimaler Spannbaum?

Kantengewichteter ungerichteter Graph

Ein (kanten-)gewichteter Graph G ist ein Tripel (V, E, W), wobei:

- (V, E) ein ungerichteter Graph ist, und
- $W : E \longrightarrow \mathbb{R}$ Gewichtsfunktion. $W(e)$ ist das Gewicht der Kante e.

Was ist ein minimaler Spannbaum?

Kantengewichteter ungerichteter Graph

Ein (kanten-)gewichteter Graph G ist ein Tripel (V, E, W), wobei:

- (V, E) ein ungerichteter Graph ist, und
- $W : E \longrightarrow \mathbb{R}$ Gewichtsfunktion. $W(e)$ ist das Gewicht der Kante e.

Gewicht eines Graphen

Das Gewicht $W(G')$ des Teilgraphen $G' = (V', E')$ vom gewichteten Graph G ist: $W(G') = \sum\limits_{e \in E'} W(e)$.

Was ist ein minimaler Spannbaum?

Kantengewichteter ungerichteter Graph

Ein (kanten-)gewichteter Graph G ist ein Tripel (V, E, W), wobei:

- (V, E) ein ungerichteter Graph ist, und
- $W : E \longrightarrow \mathbb{R}$ Gewichtsfunktion. $W(e)$ ist das Gewicht der Kante e.

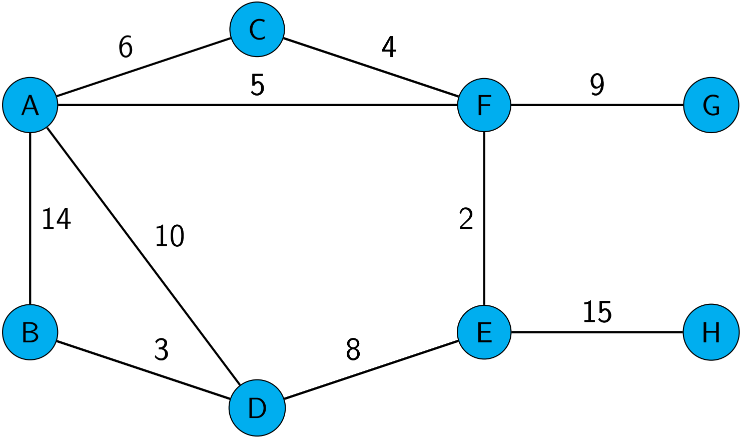
Gewicht eines Graphen

Das Gewicht $W(G')$ des Teilgraphen $G' = (V', E')$ vom gewichteten Graph G ist: $W(G') = \sum\limits_{e \in E'} W(e)$.

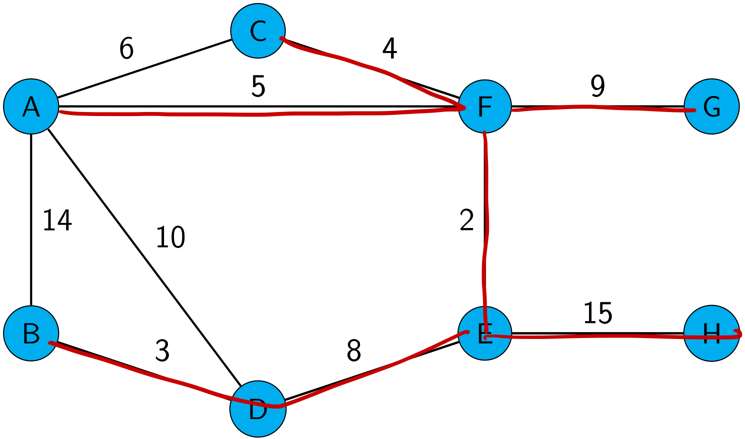
Minimaler Spannbaum

Ein Spannbaum vom (ungerichteter, gewichteter, zusammenhängen) Graphen G mit minimalem Gewicht heißt Minimaler Spannbaum (minimum spanning tree, MST) von G.

Minimaler Spannbaum – Beispiel

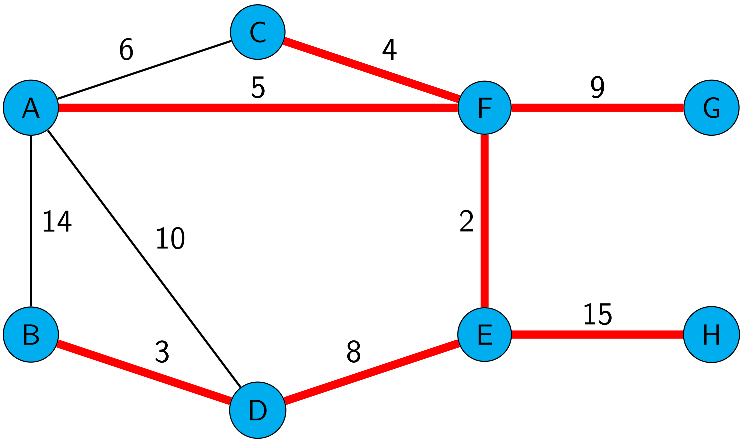


Minimaler Spannbaum – Beispiel



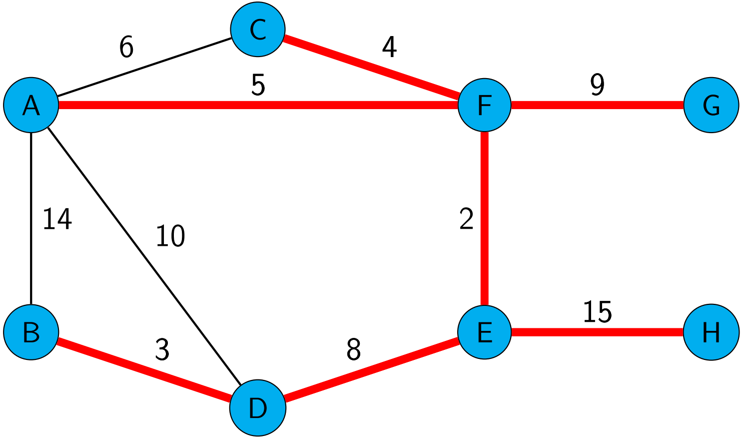
Was ist ein minimaler Spannbaum?

Minimaler Spannbaum – Beispiel



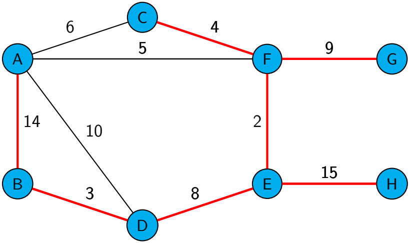
Das ist *ein* minimaler Spannbaum (mit Gesamtgewicht 46).

Minimaler Spannbaum – Beispiel

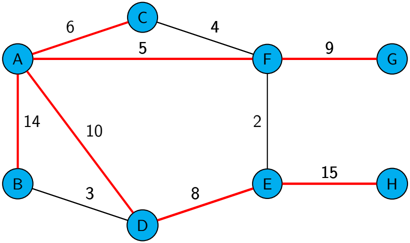


Das ist *ein* minimaler Spannbaum (mit Gesamtgewicht 46).
In diesem Fall ist es auch der einzige.

Tiefen- oder Breitensuche?



Tiefensuchbaum (von A gestartet)
Gesamtgewicht: 55



Breitensuchbaum (von A gestartet)
Gesamtgewicht: 67

Der Tiefensuchbaum und der Breitensuchbaum sind zwar Spannbäume, aber nicht notwendigerweise minimale Spannbäume.

Übersicht

Greedy Algorithmen

Wir werden zwei Greedy-Algorithmen für MST präsentieren.

Greedy-Algorithmen („gierig“)

Greedy Algorithmen

Wir werden zwei Greedy-Algorithmen für MST präsentieren.

Greedy-Algorithmen („gierig“)

- Treffe in jedem Schritt eine Entscheidung, die bezüglich eines „kurzfristigen“ Kriteriums optimal ist.

Greedy Algorithmen

Wir werden zwei Greedy-Algorithmen für MST präsentieren.

Greedy-Algorithmen („gierig“)

- Treffe in jedem Schritt eine Entscheidung, die bezüglich eines „kurzfristigen“ Kriteriums optimal ist.
- Dieses Kriterium sollte günstig ($\rightarrow$ Komplexität) auswertbar sein.

Greedy Algorithmen

Wir werden zwei Greedy-Algorithmen für MST präsentieren.

Greedy-Algorithmen („gierig“)

- Treffe in jedem Schritt eine Entscheidung, die bezüglich eines „kurzfristigen“ Kriteriums optimal ist.
- Dieses Kriterium sollte günstig ($\rightarrow$ Komplexität) auswertbar sein.
- Nachdem eine Wahl getroffen wurde, kann sie nicht mehr rückgängig gemacht werden.

Greedy Algorithmen

Wir werden zwei Greedy-Algorithmen für MST präsentieren.

Greedy-Algorithmen („gierig“)

- Treffe in jedem Schritt eine Entscheidung, die bezüglich eines „kurzfristigen“ Kriteriums optimal ist.
- Dieses Kriterium sollte günstig ($\rightarrow$ Komplexität) auswertbar sein.
- Nachdem eine Wahl getroffen wurde, kann sie nicht mehr rückgängig gemacht werden.

Mit Greedy-Methoden ist nicht garantiert, dass immer die beste Lösung gefunden wird, denn

Greedy Algorithmen

Wir werden zwei Greedy-Algorithmen für MST präsentieren.

Greedy-Algorithmen („gierig“)

- Treffe in jedem Schritt eine Entscheidung, die bezüglich eines „kurzfristigen“ Kriteriums optimal ist.
- Dieses Kriterium sollte günstig ($\rightarrow$ Komplexität) auswertbar sein.
- Nachdem eine Wahl getroffen wurde, kann sie nicht mehr rückgängig gemacht werden.

Mit Greedy-Methoden ist nicht garantiert, dass immer die beste Lösung gefunden wird, denn

- immer das lokale Optimum zu nehmen, führt nicht automatisch auch zum globalen Optimum.

Greedy Algorithmen

Wir werden zwei Greedy-Algorithmen für MST präsentieren.

Greedy-Algorithmen („gierig“)

- Treffe in jedem Schritt eine Entscheidung, die bezüglich eines „kurzfristigen“ Kriteriums optimal ist.
- Dieses Kriterium sollte günstig ($\rightarrow$ Komplexität) auswertbar sein.
- Nachdem eine Wahl getroffen wurde, kann sie nicht mehr rückgängig gemacht werden.

Mit Greedy-Methoden ist nicht garantiert, dass immer die beste Lösung gefunden wird, denn

- immer das lokale Optimum zu nehmen, führt nicht automatisch auch zum globalen Optimum.
- In einigen Fällen, wie dem minimalen Spannbaum und dem Kürzesten-Wege-Problem, wird aber *immer* die optimale Lösung gefunden.

Greedy?

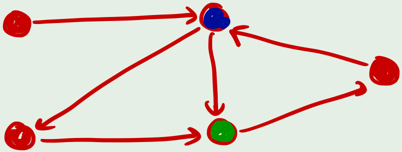
Beispiel

Greedy?

Beispiel

Greedy kann beliebig schlecht werden:

- Knotenfärbungsproblem für Graphen



Greedy?

Beispiel

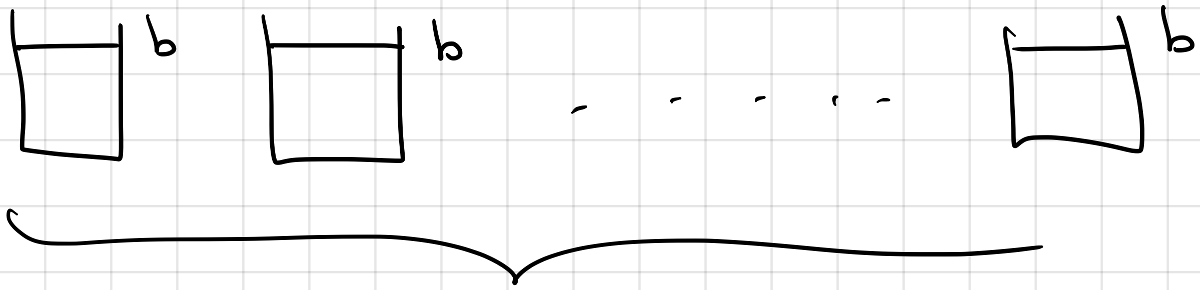
Greedy kann beliebig schlecht werden:

- Knotenfärbungsproblem für Graphen

Greedy kann gut sein:

- Bin Packing ($\leqslant$ 2x Optimum)

Behälterproblem



k Behälter jeder mit Kapazität b

n Objekte mit Größen $a_1, a_2, \ldots, a_n \leq b$

$\exists f: \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$ so dass

$$\forall\, 0 < j \leq k. \quad \sum_{f(i)=j} a_i \leq b$$

Greedy-Strategie: $\longrightarrow$ nicht optimal

1. sortiere die Objekte nach <u>absteigendem</u> Gewicht
2. füge die Objekte der Reihe nach ein

1,5 <u>optimal</u> $\longrightarrow$ max. die Hälfte der Behälter mehr als das Optimal

Greedy?

Beispiel

Greedy kann beliebig schlecht werden:

- Knotenfärbungsproblem für Graphen

Greedy kann gut sein:

- Bin Packing ($\leqslant$ 2x Optimum)

Greedy kann optimal sein:

- Minimaler Spannbaum, Kürzester-Weg-Problem.

Greedy?

Beispiel

Greedy kann beliebig schlecht werden:

- Knotenfärbungsproblem für Graphen

Greedy kann gut sein:

- Bin Packing ($\leqslant$ 2x Optimum)

Greedy kann optimal sein:

- Minimaler Spannbaum, Kürzester-Weg-Problem.

Greedy?

Beispiel

Greedy kann beliebig schlecht werden:

- Knotenfärbungsproblem für Graphen

Greedy kann gut sein:

- Bin Packing ($\leqslant$ 2x Optimum)

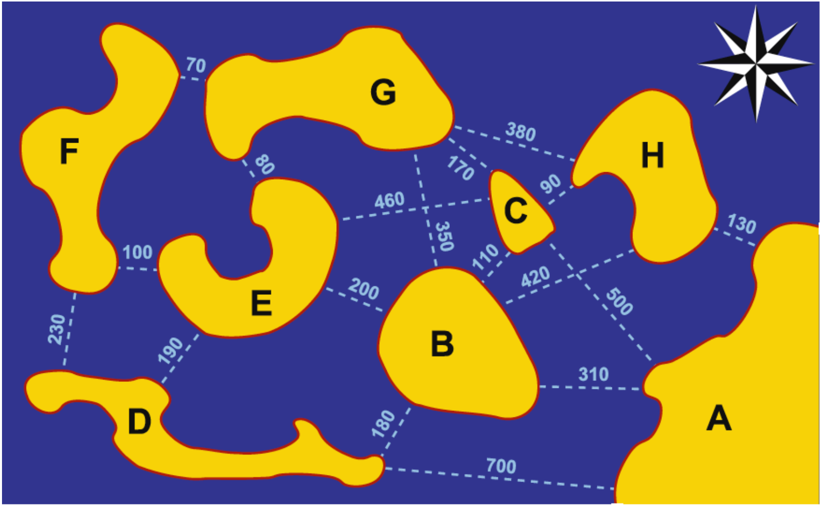
Greedy kann optimal sein:

- Minimaler Spannbaum, Kürzester-Weg-Problem.

Wann ist eine greedy Lösungsstrategie optimal?

- Optimale Lösung setzt sich aus optimalen Teilproblemen zusammen
- Unabhängigkeit von anderen Teillösungen

Das Inselreich der Stamm der Algolaner



Was ist die optimale Strategie die Fährverbindungen durch Brücken zu ersetzen?

Übersicht

Zwei Greedy minimaler Spannbaumalgorithmen

Eingabe: ein gewichteter zusammenhängender Graph G mit n Knoten

Ausgabe: ein minimaler Spannbaum von G

Zwei Greedy minimaler Spannbaumalgorithmen

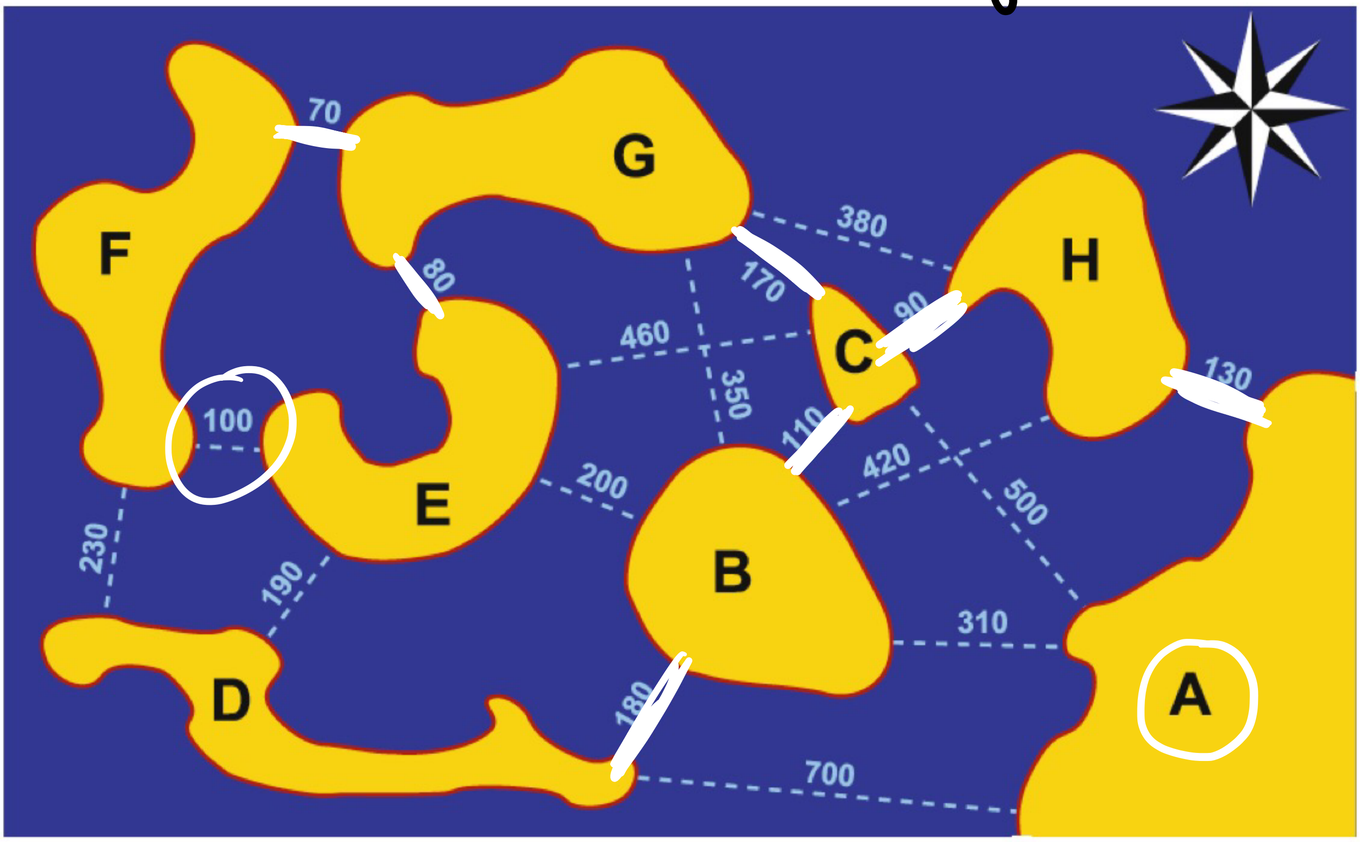
Eingabe: ein gewichteter zusammenhängender Graph G mit n Knoten

Ausgabe: ein minimaler Spannbaum von G

Prim's Strategie

1. Wähle einen Startknoten.
2. Markiere die "billigste" vom bereits konstruierten Baum ausgehende Kante, falls sie keinen Kreis schließt
3. Wiederhole Schritt 2., so lange noch keine $n-1$ Kanten markiert sind.

Prim's Strategy



Zwei Greedy minimaler Spannbaumalgorithmen

Eingabe: ein gewichteter zusammenhängender Graph G mit n Knoten

Ausgabe: ein minimaler Spannbaum von G

Prim's Strategie

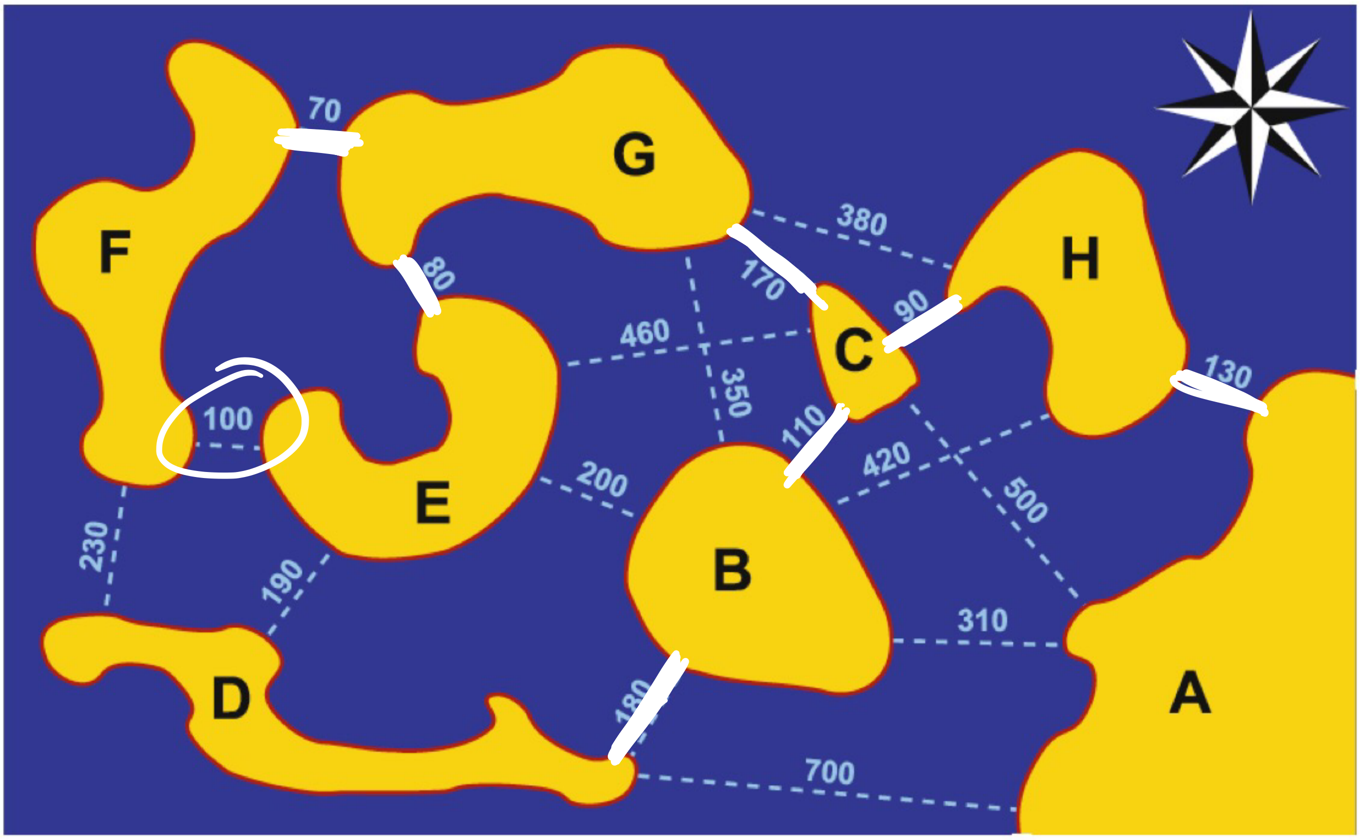
1. Wähle einen Startknoten.
2. Markiere die "billigste" vom bereits konstruierten Baum ausgehende Kante, falls sie keinen Kreis schließt
3. Wiederhole Schritt 2., so lange noch keine $n-1$ Kanten markiert sind.

Kruskal's Strategie

So lange noch keine $n-1$ Kanten markiert (d.h. selektiert) sind:

1. Wähle eine "billigste" noch unmarkierte Kante
2. Markiere sie, falls sie keinen Kreis mit anderen markierten Kanten schließt

kruskal's strategy



Korrektheit des Algorithmus von Kruskal

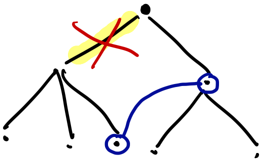
Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Die Beweisidee beruht auf den Kantentausch für Kreise: man kann aus einem Spannbaum B einen anderen Spannbaum B' konstruieren, indem man eine neue Kante hinzufügt, die—da B ein Baum ist—einen Kreis schließt, und anschließend eine andere Kante aus diesem Kreis löscht.

$\mathcal{B}$



Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

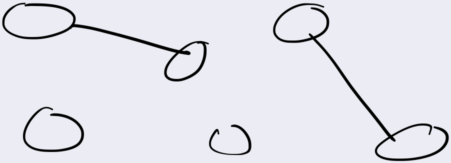
Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen.



Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

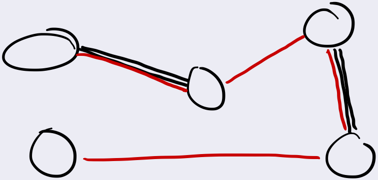
Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden. Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\underline{\{e_1, \ldots, e_k\}}$ enthält.



$k=2$

B_k

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden. Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\{e_1, \ldots, e_k\}$ enthält. Betrachte die $(k+1)$. Kante e_{k+1}.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

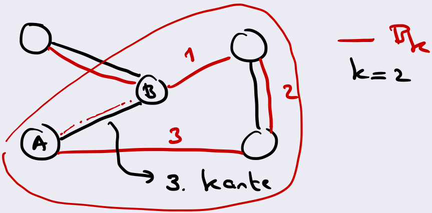
Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden. Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\{e_1, \ldots, e_k\}$ enthält. Betrachte die $(k+1)$. Kante e_{k+1}. Falls $e_{k+1} \in B_k$, folgt die Behauptung.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden. Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\{e_1, \ldots, e_k\}$ enthält. Betrachte die $(k+1)$. Kante e_{k+1}. Falls $e_{k+1} \in B_k$, folgt die Behauptung. Sei $\underline{e_{k+1} \notin B_k}$.



Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden.
Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\{e_1, \ldots, e_k\}$ enthält. Betrachte die $(k+1)$. Kante e_{k+1}. Falls $e_{k+1} \in B_k$, folgt die Behauptung. Sei $e_{k+1} \notin B_k$. Das Hinzufügen von e_{k+1} zu B_k führt zu einem Kreis, da B_k ein Baum ist.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden. Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\{e_1, \ldots, e_k\}$ enthält. Betrachte die $(k+1)$. Kante e_{k+1}. Falls $e_{k+1} \in B_k$, folgt die Behauptung. Sei $e_{k+1} \notin B_k$. Das Hinzufügen von e_{k+1} zu B_k führt zu einem Kreis, da B_k ein Baum ist. Der (eindeutige!) Kreis in B_k enthält mindestens eine Kante $e \notin B$, da B ein Baum ist.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden. Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\{e_1, \ldots, e_k\}$ enthält. Betrachte die $(k+1)$. Kante e_{k+1}. Falls $e_{k+1} \in B_k$, folgt die Behauptung. Sei $e_{k+1} \notin B_k$. Das Hinzufügen von e_{k+1} zu B_k führt zu einem Kreis, da B_k ein Baum ist. Der (eindeutige!) Kreis in B_k enthält mindestens eine Kante $e \notin B$, da B ein Baum ist. Die Entfernung von e aus B_k liefert wieder ein Baum.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden. Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\{e_1, \ldots, e_k\}$ enthält. Betrachte die $(k{+}1)$. Kante e_{k+1}. Falls $e_{k+1} \in B_k$, folgt die Behauptung. Sei $e_{k+1} \notin B_k$. Das Hinzufügen von e_{k+1} zu B_k führt zu einem Kreis, da B_k ein Baum ist. Der (eindeutige!) Kreis in B_k enthält mindestens eine Kante $e \notin B$, da B ein Baum ist. Die Entfernung von e aus B_k liefert wieder ein Baum. Es folgt $\underline{W(e)} \geqslant \underline{W(e_{k+1})}$, ja sonst wäre e vom Algorithmus vor e_{k+1} gewählt worden.

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden. Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\{e_1, \ldots, e_k\}$ enthält. Betrachte die $(k{+}1)$. Kante e_{k+1}. Falls $e_{k+1} \in B_k$, folgt die Behauptung. Sei $e_{k+1} \notin B_k$. Das Hinzufügen von e_{k+1} zu B_k führt zu einem Kreis, da B_k ein Baum ist. Der (eindeutige!) Kreis in B_k enthält mindestens eine Kante $e \notin B$, da B ein Baum ist. Die Entfernung von e aus B_k liefert wieder ein Baum. Es folgt $W(e) \geqslant W(e_{k+1})$, ja sonst wäre e vom Algorithmus vor e_{k+1} gewählt worden. Damit folgt: $W(B_{k+1}) \leqslant W(B_k)$.

$B_k + e_{k+1}$

Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden. Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\{e_1, \ldots, e_k\}$ enthält. Betrachte die $(k+1)$. Kante e_{k+1}. Falls $e_{k+1} \in B_k$, folgt die Behauptung. Sei $e_{k+1} \notin B_k$. Das Hinzufügen von e_{k+1} zu B_k führt zu einem Kreis, da B_k ein Baum ist. Der (eindeutige!) Kreis in B_k enthält mindestens eine Kante $e \notin B$, da B ein Baum ist. Die Entfernung von e aus B_k liefert wieder ein Baum. Es folgt $W(e) \geqslant W(e_{k+1})$, ja sonst wäre e vom Algorithmus vor e_{k+1} gewählt worden. Damit folgt: $W(B_{k+1}) \leqslant W(B_k)$. Da B_k minimal ist (I.V.), ist B_{k+1} minimal.

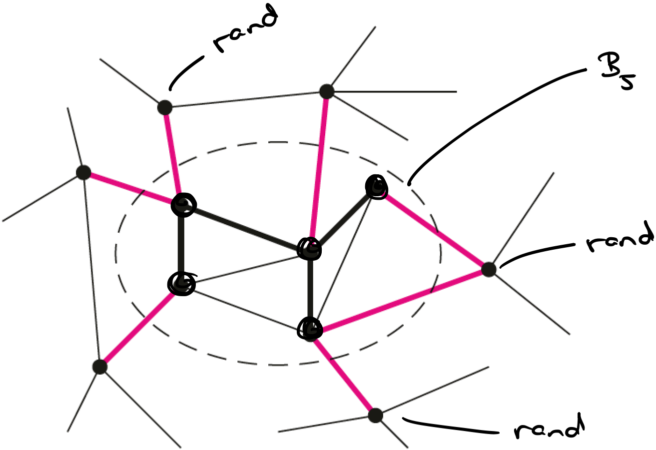
Korrektheit des Algorithmus von Kruskal

Der Algorithmus von Kruskal bestimmt einen minimalen Spannbaum.

Beweis.

Der Algorithmus liefert ein Baum B. Zu beweisen: B ist minimal. Mit Induktion. Idee: die Mengen von Kanten die nach k Iterationen vorliegt, ist zu einem MST von G zu ergänzen. Am Ende gilt dann die Behauptung. Basis: die erste markierte Kante hat ein minimales Gewicht. Sie kann zu einem MST ergänzt werden. Induktionsschritt: sei B_k eine MST von G, die die erste k gewählte Kanten $\{e_1, \ldots, e_k\}$ enthält. Betrachte die $(k+1)$. Kante e_{k+1}. Falls $e_{k+1} \in B_k$, folgt die Behauptung. Sei $e_{k+1} \notin B_k$. Das Hinzufügen von e_{k+1} zu B_k führt zu einem Kreis, da B_k ein Baum ist. Der (eindeutige!) Kreis in B_k enthält mindestens eine Kante $e \notin B$, da B ein Baum ist. Die Entfernung von e aus B_k liefert wieder ein Baum. Es folgt $W(e) \geqslant W(e_{k+1})$, ja sonst wäre e vom Algorithmus vor e_{k+1} gewählt worden. Damit folgt: $W(B_{k+1}) \leqslant W(B_k)$. Da B_k minimal ist (I.V.), ist B_{k+1} minimal. Somit ist B_{k+1} ein MST der die Kanten $\{e_1, \ldots, e_{k+1}\}$ enthält. □

Korrektheit Prim's Algorithmus: Was ist ein Schnitt?



Der Schnitt (rote Kanten) für den bereits konstruierten Teilbaum T (dicke Kanten) besteht aus allen Kanten (u, v) mit $u \in T$ und $v \notin T$. v wird auch Randknote genannt.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

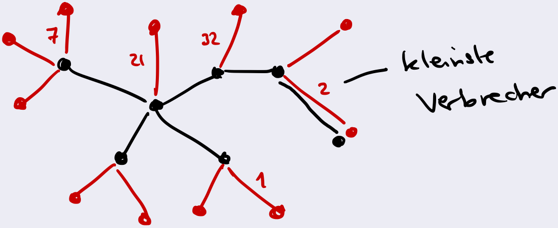
Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde.



Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.)

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.) Der Schnitt von T enthält alle Kanten $(u, v) \in G$ mit $u \in T$ and $v \notin T$.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.) Der Schnitt von T enthält alle Kanten $(u, v) \in G$ mit $u \in T$ and $v \notin T$. Die Kante e gehört zum Schnitt.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.) Der Schnitt von T enthält alle Kanten $(u, v) \in G$ mit $u \in T$ and $v \notin T$. Die Kante e gehört zum Schnitt. Der MST B_T die T enthält, enthält eine Kante e_{B_T} aus dem Schnitt, sonst ist sie nicht zusammenhängend.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.) Der Schnitt von T enthält alle Kanten $(u, v) \in G$ mit $u \in T$ and $v \notin T$. Die Kante e gehört zum Schnitt. Der MST B_T die T enthält, enthält eine Kante e_{B_T} aus dem Schnitt, sonst ist sie nicht zusammenhängend. Es gilt $\underline{W(e_{B_T})} \geqslant \underline{W(e)}$ da sonst der Algorithmus e nicht gewählt hätte.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.) Der Schnitt von T enthält alle Kanten $(u, v) \in G$ mit $u \in T$ and $v \notin T$. Die Kante e gehört zum Schnitt. Der MST B_T die T enthält, enthält eine Kante e_{B_T} aus dem Schnitt, sonst ist sie nicht zusammenhängend. Es gilt $W(e_{B_T}) \geqslant W(e)$ da sonst der Algorithmus e nicht gewählt hätte. Anderseits, $W(e_{B_T}) \not> W(e)$, denn sonst könnte man e durch e_{B_T} ersetzen und eine MST konstruieren die günstiger ist als B_T.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.) Der Schnitt von T enthält alle Kanten $(u, v) \in G$ mit $u \in T$ and $v \notin T$. Die Kante e gehört zum Schnitt. Der MST B_T die T enthält, enthält eine Kante e_{B_T} aus dem Schnitt, sonst ist sie nicht zusammenhängend. Es gilt $W(e_{B_T}) \geqslant W(e)$ da sonst der Algorithmus e nicht gewählt hätte. Anderseits, $W(e_{B_T}) \not> W(e)$, denn sonst könnte man e durch e_{B_T} ersetzen und eine MST konstruieren die günstiger ist als B_T. Da B_T ein MST ist, kann das nicht sein.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.) Der Schnitt von T enthält alle Kanten $(u, v) \in G$ mit $u \in T$ and $v \notin T$. Die Kante e gehört zum Schnitt. Der MST B_T die T enthält, enthält eine Kante e_{B_T} aus dem Schnitt, sonst ist sie nicht zusammenhängend. Es gilt $W(e_{B_T}) \geqslant W(e)$ da sonst der Algorithmus e nicht gewählt hätte. Anderseits, $W(e_{B_T}) \not> W(e)$, denn sonst könnte man e durch e_{B_T} ersetzen und eine MST konstruieren die günstiger ist als B_T. Da B_T ein MST ist, kann das nicht sein. Damit gilt $W(e_{B_T}) = W(e)$.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.) Der Schnitt von T enthält alle Kanten $(u, v) \in G$ mit $u \in T$ and $v \notin T$. Die Kante e gehört zum Schnitt. Der MST B_T die T enthält, enthält eine Kante e_{B_T} aus dem Schnitt, sonst ist sie nicht zusammenhängend. Es gilt $W(e_{B_T}) \geqslant W(e)$ da sonst der Algorithmus e nicht gewählt hätte. Anderseits, $W(e_{B_T}) \not> W(e)$, denn sonst könnte man e durch e_{B_T} ersetzen und eine MST konstruieren die günstiger ist als B_T. Da B_T ein MST ist, kann das nicht sein. Damit gilt $W(e_{B_T}) = W(e)$. Da e_{B_T} und e gleich teuer sind, liefert den Austausch von e_{B_T} durch e ein MST.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.) Der Schnitt von T enthält alle Kanten $(u, v) \in G$ mit $u \in T$ and $v \notin T$. Die Kante e gehört zum Schnitt. Der MST B_T die T enthält, enthält eine Kante e_{B_T} aus dem Schnitt, sonst ist sie nicht zusammenhängend. Es gilt $W(e_{B_T}) \geqslant W(e)$ da sonst der Algorithmus e nicht gewählt hätte. Anderseits, $W(e_{B_T}) \not> W(e)$, denn sonst könnte man e durch e_{B_T} ersetzen und eine MST konstruieren die günstiger ist als B_T. Da B_T ein MST ist, kann das nicht sein. Damit gilt $W(e_{B_T}) = W(e)$. Da e_{B_T} und e gleich teuer sind, liefert den Austausch von e_{B_T} durch e ein MST. Also war der Wahl des "kleinsten Verbrechers" e nicht falsch.

Korrektheit des Algorithmus von Prim

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Widerspruchsbeweis. Sei G einen Graph wofür den Algorithmus einen nicht-minimalen Spannbaum konstruiert. Betrachte die Iteration wobei zuerst eine zu teure Kante e, der "kleinster Verbrecher", gewählt wurde. Betrachte der zuvor konstruierten Teilbaum T. (Also $e \notin T$.) Der Schnitt von T enthält alle Kanten $(u, v) \in G$ mit $u \in T$ and $v \notin T$. Die Kante e gehört zum Schnitt. Der MST B_T die T enthält, enthält eine Kante e_{B_T} aus dem Schnitt, sonst ist sie nicht zusammenhängend. Es gilt $W(e_{B_T}) \geqslant W(e)$ da sonst der Algorithmus e nicht gewählt hätte. Anderseits, $W(e_{B_T}) \not> W(e)$, denn sonst könnte man e durch e_{B_T} ersetzen und eine MST konstruieren die günstiger ist als B_T. Da B_T ein MST ist, kann das nicht sein. Damit gilt $W(e_{B_T}) = W(e)$. Da e_{B_T} und e gleich teuer sind, liefert den Austausch von e_{B_T} durch e ein MST. Also war der Wahl des "kleinsten Verbrechers" e nicht falsch. Widerspruch. $\square$

Übersicht

Der Algorithmus von Prim – Übersicht

Wir ordnen die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

Baum-knoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Der Algorithmus von Prim – Übersicht

Wir ordnen die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

Baum-knoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

Der Algorithmus von Prim – Übersicht

Wir ordnen die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

Baum-knoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

- Fange mit einen Baum aus nur einem Knoten an, indem ein beliebiger Knoten des Graphens ausgewählt wird.

Der Algorithmus von Prim – Übersicht

Wir ordnen die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

Baum-knoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

- Fange mit einen Baum aus nur einem Knoten an, indem ein beliebiger Knoten des Graphens ausgewählt wird.
- Finde die günstigste Kante (d. h. mit minimalem Gewicht), die den bisherigen Baum verlässt.

Der Algorithmus von Prim – Übersicht

Wir ordnen die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

Baum-knoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

- Fange mit einen Baum aus nur einem Knoten an, indem ein beliebiger Knoten des Graphens ausgewählt wird.
- Finde die günstigste Kante (d. h. mit minimalem Gewicht), die den bisherigen Baum verlässt.
- Füge den über diese Kante erreichten (Rand-)Knoten dem Baum hinzu, zusammen mit der Kante.

Der Algorithmus von Prim – Übersicht

Wir ordnen die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

Baum-knoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

- Fange mit einen Baum aus nur einem Knoten an, indem ein beliebiger Knoten des Graphens ausgewählt wird.
- Finde die günstigste Kante (d. h. mit minimalem Gewicht), die den bisherigen Baum verlässt.
- Füge den über diese Kante erreichten (Rand-)Knoten dem Baum hinzu, zusammen mit der Kante.
- Fahre fort, bis keine weiteren Randknoten mehr vorhanden sind.

Der Algorithmus von Prim – Übersicht

Wir ordnen die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

Baum-knoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

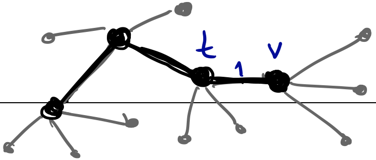
Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

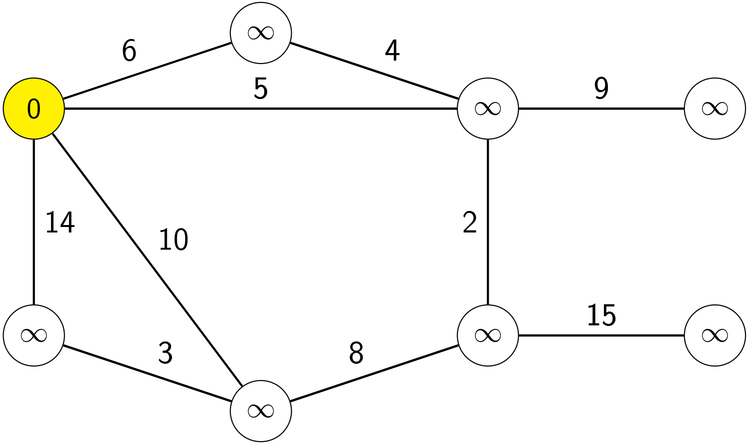
- Fange mit einen Baum aus nur einem Knoten an, indem ein beliebiger Knoten des Graphens ausgewählt wird.
- Finde die günstigste Kante (d. h. mit minimalem Gewicht), die den bisherigen Baum verlässt.
- Füge den über diese Kante erreichten (Rand-)Knoten dem Baum hinzu, zusammen mit der Kante.
- Fahre fort, bis keine weiteren Randknoten mehr vorhanden sind.

Prim's Algorithmus – Grundgerüst

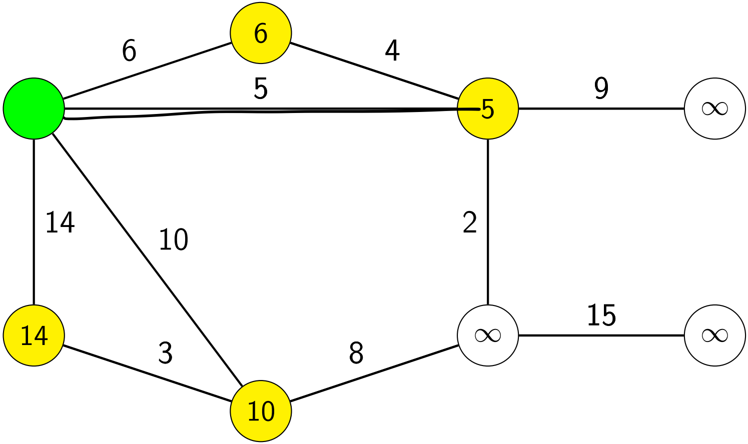
```
// ungerichteter Graph G mit n Knoten
void primMST(Graph G, int n) {
  initialisiere alle Knoten als ungesehen (WHITE);
  wähle irgendeinen Knoten s und markiere ihn mit Baum (BLACK);
  reklassifiziere alle zu s adjazenten Knoten als Rand (GRAY);
  while (es gibt Randknoten) {
    wähle von allen Kanten zwischen einem Baumknoten t und
      einem Randknoten v die billigste;
    reklassifiziere v als Baum (BLACK);
    füge Kante (t, v) zum Baum hinzu;
    reklassifiziere alle zu v adjazenten ungesehenen Knoten
      als Rand (GRAY);
  }
}
```



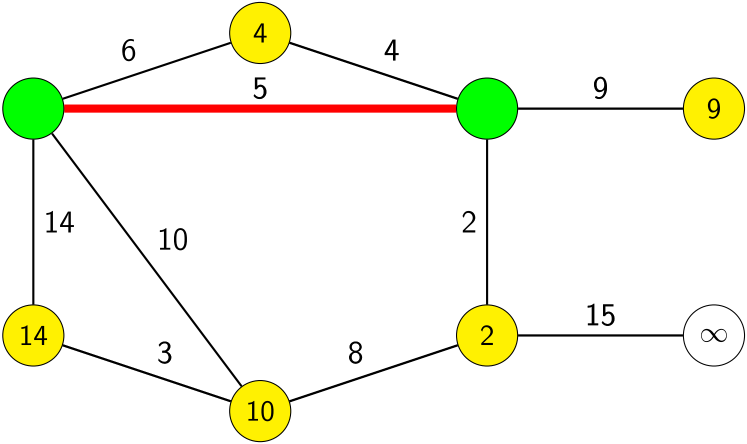
Prim's Algorithmus – Beispiel



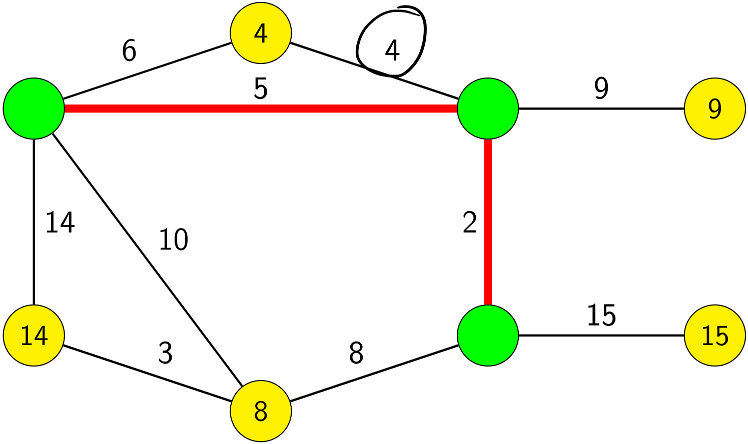
Prim's Algorithmus – Beispiel



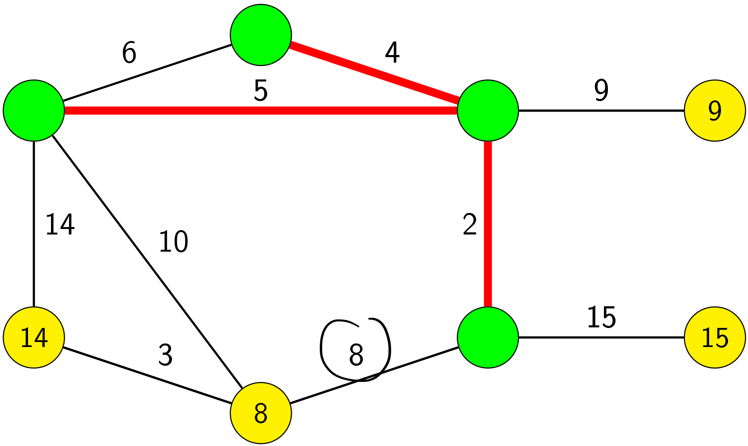
Prim's Algorithmus – Beispiel



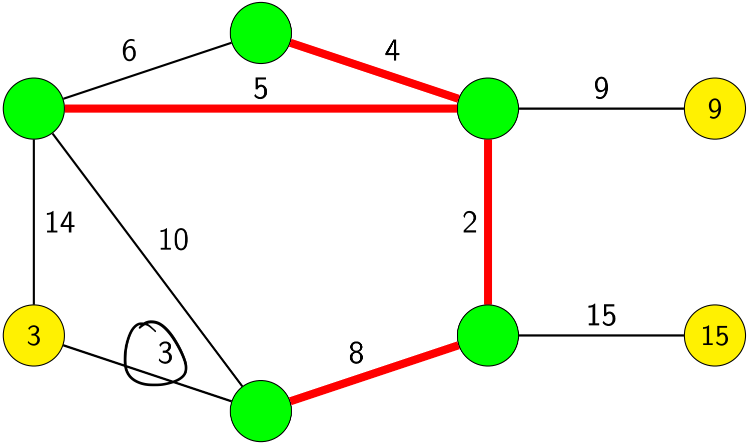
Prim's Algorithmus – Beispiel



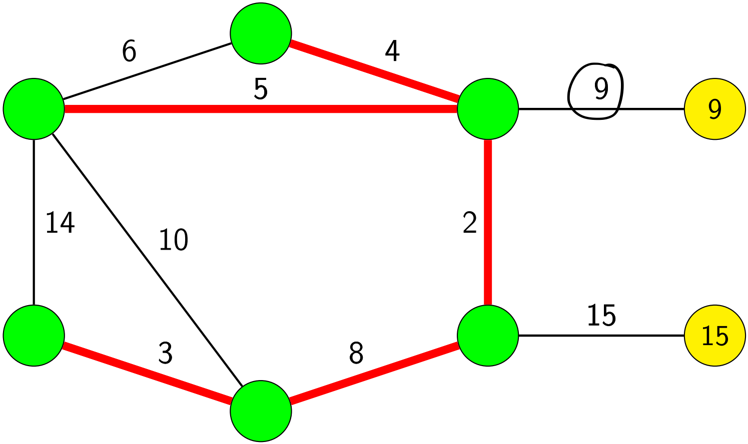
Prim's Algorithmus – Beispiel



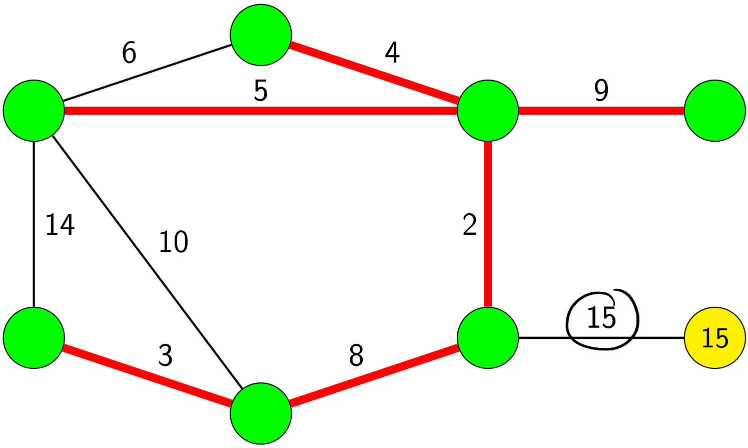
Prim's Algorithmus – Beispiel



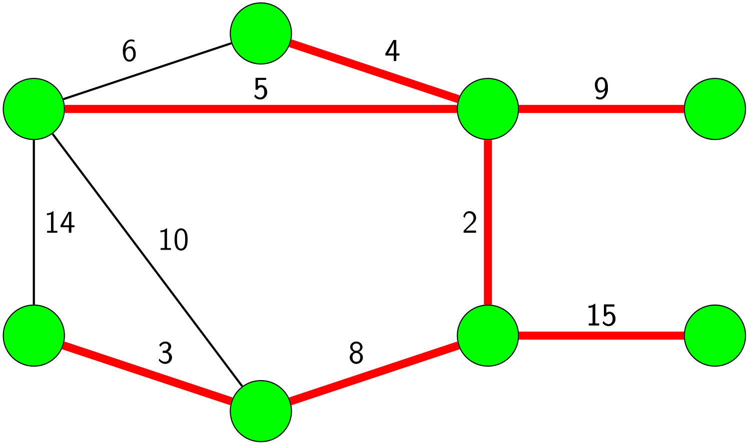
Prim's Algorithmus – Beispiel



Prim's Algorithmus – Beispiel



Prim's Algorithmus – Beispiel



ADT zum Vorhalten der Randknoten (I)

Die benötigten Operationen für den Algorithmus von Prim sind:

ADT zum Vorhalten der Randknoten (I)

Die benötigten Operationen für den Algorithmus von Prim sind:

- Wähle eine billigste Kante zu einem Randknoten (Kantenkandidat).

ADT zum Vorhalten der Randknoten (I)

Die benötigten Operationen für den Algorithmus von Prim sind:

- Wähle eine billigste Kante zu einem Randknoten (Kantenkandidat).
- Reklassifiziere einen Randknoten als Baumknoten (füge den Kantenkandidat zum Baum hinzu).

ADT zum Vorhalten der Randknoten (I)

Die benötigten Operationen für den Algorithmus von Prim sind:

- Wähle eine billigste Kante zu einem Randknoten (Kantenkandidat).
- Reklassifiziere einen Randknoten als Baumknoten (füge den Kantenkandidat zum Baum hinzu).
- Ändere die Kosten (Randgewicht) eines Randknotens, wenn ein günstigerer Kantenkandidat gefunden wird.

ADT zum Vorhalten der Randknoten (I)

Die benötigten Operationen für den Algorithmus von Prim sind:

- Wähle eine billigste Kante zu einem Randknoten (Kantenkandidat).
- Reklassifiziere einen Randknoten als Baumknoten (füge den Kantenkandidat zum Baum hinzu).
- Ändere die Kosten (Randgewicht) eines Randknotens, wenn ein günstigerer Kantenkandidat gefunden wird.

Idee: *Ordne die Randknoten nach ihrer Priorität (= Randgewicht).*

ADT zum Vorhalten der Randknoten (I)

Die benötigten Operationen für den Algorithmus von Prim sind:

- Wähle eine billigste Kante zu einem Randknoten (Kantenkandidat).
- Reklassifiziere einen Randknoten als Baumknoten (füge den Kantenkandidat zum Baum hinzu).
- Ändere die Kosten (Randgewicht) eines Randknotens, wenn ein günstigerer Kantenkandidat gefunden wird.

Idee: *Ordne die Randknoten nach ihrer Priorität (= Randgewicht).*

Prioritätswarteschlange (priority queue)

- `PriorityQueue pq;`
- `pq.insert(int e, int k)`, `int pq.getMin()`, `pq.delMin()`
- `void pq.decrKey(int e, int k)` setzt den Schlüssel von Element `e` auf `k`; `k` muss kleiner als der bisherige Schlüssel von `e` sein.

ADT zum Vorhalten der Randknoten (I)

Die benötigten Operationen für den Algorithmus von Prim sind:

- Wähle eine billigste Kante zu einem Randknoten (Kantenkandidat).
- Reklassifiziere einen Randknoten als Baumknoten (füge den Kantenkandidat zum Baum hinzu).
- Ändere die Kosten (Randgewicht) eines Randknotens, wenn ein günstigerer Kantenkandidat gefunden wird.

Idee: *Ordne die Randknoten nach ihrer Priorität (= Randgewicht).*

Prioritätswarteschlange (priority queue)

- `PriorityQueue pq;`
- `pq.insert(int e, int k)`, `int pq.getMin()`, `pq.delMin()`
- `void pq.decrKey(int e, int k)` setzt den Schlüssel von Element `e` auf `k`; `k` muss kleiner als der bisherige Schlüssel von `e` sein.

$\Rightarrow$ Wir entscheiden uns für die Prioritätswarteschlange als Datenstruktur

Vorläufige Komplexitätsanalyse

Im Worst-Case:

Vorläufige Komplexitätsanalyse

Im Worst-Case:

- Jeder Knoten muss zur Prioritätswarteschlange hinzugefügt werden.

Vorläufige Komplexitätsanalyse

Im Worst-Case:

- Jeder Knoten muss zur Prioritätswarteschlange hinzugefügt werden.
- Auf jeden Knoten muss auch wieder zugegriffen werden und er muss gelöscht werden.

Vorläufige Komplexitätsanalyse

Im Worst-Case:

- Jeder Knoten muss zur Prioritätswarteschlange hinzugefügt werden.
- Auf jeden Knoten muss auch wieder zugegriffen werden und er muss gelöscht werden.
- Die Priorität eines Randknotens muss nach jeder gefundenen Kante angepasst werden.

Vorläufige Komplexitätsanalyse

Im Worst-Case:

- Jeder Knoten muss zur Prioritätswarteschlange hinzugefügt werden.
- Auf jeden Knoten muss auch wieder zugegriffen werden und er muss gelöscht werden.
- Die Priorität eines Randknotens muss nach jeder gefundenen Kante angepasst werden.

Bei einem Graph mit n Knoten und m Kanten ergibt sich:

$$T(n,m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

wobei $T(n,m)$ die Zeitkomplexität von Prim's Algoritmus ist

Vorläufige Komplexitätsanalyse

Im Worst-Case:

- Jeder Knoten muss zur Prioritätswarteschlange hinzugefügt werden.
- Auf jeden Knoten muss auch wieder zugegriffen werden und er muss gelöscht werden.
- Die Priorität eines Randknotens muss nach jeder gefundenen Kante angepasst werden.

Bei einem Graph mit n Knoten und m Kanten ergibt sich:

$$T(n, m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

wobei $T(n, m)$ die Zeitkomplexität von Prim's Algoritmus ist

Nächster Schritt:

Wähle eine geeignete Implementierung der Prioritätswarteschlange

Drei Prioritätswarteschlangenimplementierungen

$$T(n,m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

	Implementierung		
Operation	unsortiertes Array	sortiertes Array	Heap
`pq.isEmpty()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`pq.insert(e,k)`	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
`pq.getMin()`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
`pq.delMin()`	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$
`pq.getElt(k)`	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
`pq.decrKey(e,k)`	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
Prim	$O(n^2 + m))$	$O(n^2 + m \cdot n)$	$O(n \log n + m \log n)$

n^2

$n^2 \log n$

Drei Prioritätswarteschlangenimplementierungen

$$T(n,m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

	Implementierung		
Operation	unsortiertes Array	sortiertes Array	Heap
`pq.isEmpty()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`pq.insert(e,k)`	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
`pq.getMin()`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
`pq.delMin()`	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$
`pq.getElt(k)`	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
`pq.decrKey(e,k)`	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
Prim	$O(n^2 + m))$	$O(n^2 + m \cdot n)$	$O(n \log n + m \log n)$

Drei Prioritätswarteschlangenimplementierungen

$$T(n,m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

	Implementierung		
Operation	unsortiertes Array	sortiertes Array	Heap
`pq.isEmpty()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`pq.insert(e,k)`	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
`pq.getMin()`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
`pq.delMin()`	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$
`pq.getElt(k)`	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
`pq.decrKey(e,k)`	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
Prim	$O(n^2 + m))$	$O(n^2 + m \cdot n)$	$O(n \log n + m \log n)$

- Wir ergänzen außerdem noch zwei Operationen:

Eine leicht angepasste Prioritätswarteschlange

Prioritätswarteschlange

- `bool pq.isEmpty()` $\Theta(n)$

Eine leicht angepasste Prioritätswarteschlange

Prioritätswarteschlange

- `bool pq.isEmpty()` $\Theta(n)$
- `void pq.insert(int elem, VertexState &key)` $\Theta(1)$

Eine leicht angepasste Prioritätswarteschlange

Prioritätswarteschlange

- `bool pq.isEmpty()` $\Theta(n)$
- `void pq.insert(int elem, VertexState &key)` $\Theta(1)$
- `float pq.getMin()` $\Theta(n)$

Eine leicht angepasste Prioritätswarteschlange

Prioritätswarteschlange

- `bool pq.isEmpty()` $\Theta(n)$
- `void pq.insert(int elem, VertexState &key)` $\Theta(1)$
- `float pq.getMin()` $\Theta(n)$
- `void pq.delMin()` $\Theta(n)$

Eine leicht angepasste Prioritätswarteschlange

Prioritätswarteschlange

- `bool pq.isEmpty()` $\Theta(n)$
- `void pq.insert(int elem, VertexState &key)` $\Theta(1)$
- `float pq.getMin()` $\Theta(n)$
- `void pq.delMin()` $\Theta(n)$
- `void pq.decrKey(int elem, VertexState &newkey)` setzt den Schlüssel von `elem` auf `newkey`; `newkey.curWeight` muss kleiner als beim bisherigen Schlüssel von `elem` sein. $\Theta(1)$

Eine leicht angepasste Prioritätswarteschlange

Prioritätswarteschlange

- `bool pq.isEmpty()` $\Theta(n)$
- `void pq.insert(int elem, VertexState &key)` $\Theta(1)$
- `float pq.getMin()` $\Theta(n)$
- `void pq.delMin()` $\Theta(n)$
- `void pq.decrKey(int elem, VertexState &newkey)` setzt den Schlüssel von `elem` auf `newkey`; `newkey.curWeight` muss kleiner als beim bisherigen Schlüssel von `elem` sein. $\Theta(1)$
- `int pq.getColor(int elem)` gibt `color` von `elem` zurück. `elem` muss dazu *nicht* in der Warteschlange sein. $\Theta(1)$

Eine leicht angepasste Prioritätswarteschlange

Prioritätswarteschlange

- `bool pq.isEmpty()` $\Theta(n)$
- `void pq.insert(int elem, VertexState &key)` $\Theta(1)$
- `float pq.getMin()` $\Theta(n)$
- `void pq.delMin()` $\Theta(n)$
- `void pq.decrKey(int elem, VertexState &newkey)` setzt den Schlüssel von `elem` auf `newkey`; `newkey.curWeight` muss kleiner als beim bisherigen Schlüssel von `elem` sein. $\Theta(1)$
- `int pq.getColor(int elem)` gibt `color` von `elem` zurück. `elem` muss dazu *nicht* in der Warteschlange sein. $\Theta(1)$
- `float pq.getWeight(int elem)` gibt `curWeight` von `elem` zurück. `elem` muss dazu *nicht* in der Warteschlange sein. $\Theta(1)$

Der Algorithmus von Prim – Implementierung

```
// Ergebnis als Vorgängerbaum in .parent:
// ∀v ∈ V : (x, v) ∈ MST(V, E) gdw. x = state[v].parent, x ≠ −1
VertexState[n] primMST(List adjLst[n], int n, int start) {
  VertexState state[n] = // (eigentlich im Konstruktor von pq)
      { color: WHITE, parent: -1, curWeight: +inf };
  PriorityQueue pq = VS_PriorityQueue<&VS.curWeight>(&state);

  pq.insert(start, {parent: -1, curWeight: 0});
  while (!pq.isEmpty()) { // solange es Randknoten gibt
    int v = pq.getMin(); // günstigste Kante, bzw. Randknoten
    pq.delMin(); // setzt auch Farbe auf BLACK
    updateFringe(pq, adjList, v); // update den Rand
  }
  return state;
}
```

Der Algorithmus von Prim – Implementierung

```
void updateFringe(PriorityQueue &pq, List adjLst[], int v) {
  foreach (edge in adjLst[v]) {
    // berechnet MST.
    float newWeight = edge.weight;

    if (pq.getColor(edge.w) == WHITE) { // -> GRAY
      pq.insert(edge.w, {parent: v, curWeight: newWeight});
    } else if (pq.getColor(edge.w) == GRAY) {
      if (newWeight < pq.getWeight(edge.w)) {
        // Randknoten-update: Kante von v aus ist besser
        pq.decrKey(edge.w, {parent: v, curWeight: newWeight});
      }
    }
  }
}
```

Komplexitätsanalyse

$$T(n, m) \in O\left(n \cdot T(\mathtt{insert}) + n \cdot T(\mathtt{getMin}) + n \cdot T(\mathtt{delMin}) + m \cdot T(\mathtt{decrKey})\right)$$

Komplexitätsanalyse

$$T(n, m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

- Die Schleife in `primMST` wird n mal ausgeführt.

Komplexitätsanalyse

$$T(n, m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

- Die Schleife in `primMST` wird n mal ausgeführt.
  - $\Rightarrow$ `isEmpty`, `getMin`, `delMin` und `updateFringe` wird n mal ausgeführt.
  - Beachte, dass `getMin` eine Komplexität von $\Theta(n)$ hat.

Komplexitätsanalyse

$$T(n, m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

- Die Schleife in `primMST` wird n mal ausgeführt.
  - $\Rightarrow$ `isEmpty`, `getMin`, `delMin` und `updateFringe` wird n mal ausgeführt.
  - Beachte, dass `getMin` eine Komplexität von $\Theta(n)$ hat.
- Die Schleife in `updateFringe` wird etwa $2m$ mal durchlaufen.

Komplexitätsanalyse

$$T(n, m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

- Die Schleife in `primMST` wird n mal ausgeführt.
  - $\Rightarrow$ `isEmpty`, `getMin`, `delMin` und `updateFringe` wird n mal ausgeführt.
  - Beachte, dass `getMin` eine Komplexität von $\Theta(n)$ hat.
- Die Schleife in `updateFringe` wird etwa $2m$ mal durchlaufen.
  - $\Rightarrow$ `insert`, `getColor`, `getWeight` und `decrKey` werden m mal ausgeführt und sind $\Theta(1)$.

Komplexitätsanalyse

$$T(n, m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

- Die Schleife in `primMST` wird n mal ausgeführt.
  - $\Rightarrow$ `isEmpty`, `getMin`, `delMin` und `updateFringe` wird n mal ausgeführt.
  - Beachte, dass `getMin` eine Komplexität von $\Theta(n)$ hat.
- Die Schleife in `updateFringe` wird etwa $2m$ mal durchlaufen.
  - $\Rightarrow$ `insert`, `getColor`, `getWeight` und `decrKey` werden m mal ausgeführt und sind $\Theta(1)$.
- Der zusätzliche Speicherbedarf ist $\Theta(n)$.

Komplexitätsanalyse

$$T(n, m) \in O\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$$

- Die Schleife in `primMST` wird n mal ausgeführt.
  - $\Rightarrow$ `isEmpty`, `getMin`, `delMin` und `updateFringe` wird n mal ausgeführt.
    - Beachte, dass `getMin` eine Komplexität von $\Theta(n)$ hat.
- Die Schleife in `updateFringe` wird etwa $2m$ mal durchlaufen.
  - $\Rightarrow$ `insert`, `getColor`, `getWeight` und `decrKey` werden m mal ausgeführt und sind $\Theta(1)$.
- Der zusätzliche Speicherbedarf ist $\Theta(n)$.
- Die untere Schranke der Zeitkomplexität ist $\Omega(m)$, da jede Kante des Graphen untersucht werden muss, um einen MST zu konstruieren.

Komplexitätsanalyse

$$T(n,m) \in O\left(n\cdot T(\texttt{insert}) + n\cdot T(\texttt{getMin}) + n\cdot T(\texttt{delMin}) + m\cdot T(\texttt{decrKey})\right)$$

- Die Schleife in `primMST` wird n mal ausgeführt.
  - $\Rightarrow$ `isEmpty`, `getMin`, `delMin` und `updateFringe` wird n mal ausgeführt.
    - Beachte, dass `getMin` eine Komplexität von $\Theta(n)$ hat.
- Die Schleife in `updateFringe` wird etwa $2m$ mal durchlaufen.
  - $\Rightarrow$ `insert`, `getColor`, `getWeight` und `decrKey` werden m mal ausgeführt und sind $\Theta(1)$.
- Der zusätzliche Speicherbedarf ist $\Theta(n)$.
- Die untere Schranke der Zeitkomplexität ist $\Omega(m)$, da jede Kante des Graphen untersucht werden muss, um einen MST zu konstruieren.

$\Rightarrow$ Insgesamt: Worst-Case-Komplexität $O(n^2 + m) = O(n^2)$.

DUNKLE STUNDEN

Nächste Vorlesung

Nächste Vorlesung

Montag 25. Juni, 08:30 (Hörsaal H01). Bis dann!