Datenstrukturen und Algorithmen

Vorlesung 16: Minimale Spannbäume

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

http://www-i2.rwth-aachen.de/i2/dsal10/

29. Juni 2010

RWTH AACHEN UNIVERSITY

Übersicht

Übersicht

Probleme auf kanten-gewichtete Graphen

Betrachte einen gewichteten Graphen, wobei den *Kanten* ein Gewicht zugeordnet ist.

Beispiel (Optimierungsprobleme auf Graphen)

- Finde den minimalen Spannbaum (minimal spanning tree) in einem ungerichteten Graphen.
- Finde den kürzesten Weg (shortest path) in einem gerichteten oder ungerichteten Graphen.

„Minimal“ und „kürzester“ beziehen sich hierbei auf die besuchten Gewichte. Die Gewichte können als Kosten für die Benutzung der Kante aufgefasst werden.

Diese Probleme können durch greedy Algorithmen gelöst werden.

Greedy Algorithmen

Eine Lösungstechnik:

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 kann beliebig schlecht werden:

- Traveling Salesman Problem (TSP)

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

Was ist ein minimaler Spannbaum?

Spannbaum

Ein Spannbaum eines ungerichteten, zusammenhängenden Graphen G ist

- ein Teilgraph von G, der ein *ungerichteter Baum* ist und alle Knoten von G enthält.

Gewicht eines Graphen

Das Gewicht $W(G')$ eines Teilgraphen $G' = (V', E')$ vom gewichteten Graph G ist:

- $W(G') = \sum_{(u,v)\in E'} W(u, v)$.

Minimaler Spannbaum

Ein Spannbaum mit minimalem Gewicht heißt Minimaler Spannbaum (minimum spanning tree), MST.

Anwendungen

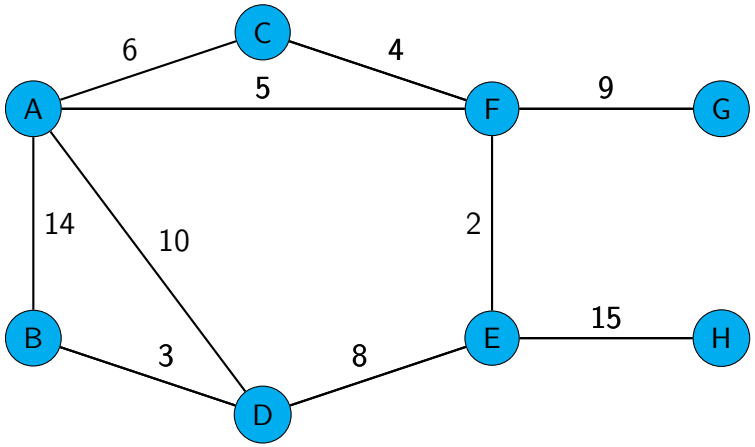
Problem

Finde einen MST eines gewichteten, ungerichteten, zusammenhängenden Graphen.

Beispiel

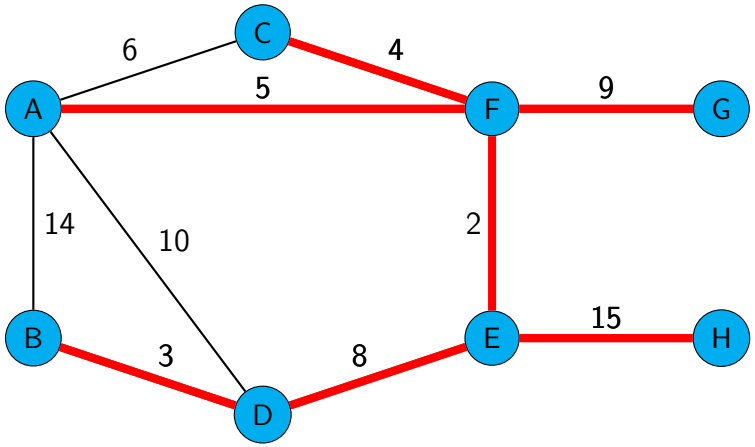
- Finde den kostengünstigsten Weg, um eine Menge von Flughafenterminals, Städten, ... zu verbinden.
- Grundlage für viele andere Probleme, etwa Routing-Probleme („Wegfindung“).
- Bestandteil von Approximationsalgorithmen für das TSP Problem.
- Verdrahtung von Schaltungen mit geringsten Energieverbrauch.

Minimaler Spannbaum – Beispiel



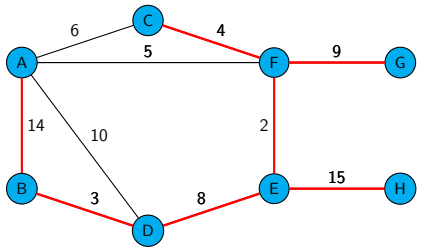
Was ist ein minimale Spannbaum?

Minimaler Spannbaum – Beispiel

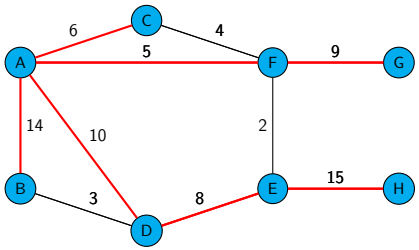


Das ist *ein* minimale Spannbaum (mit Gesamtgewicht 46).
In diesem Fall ist es auch der einzige.

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 MSTs.

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:

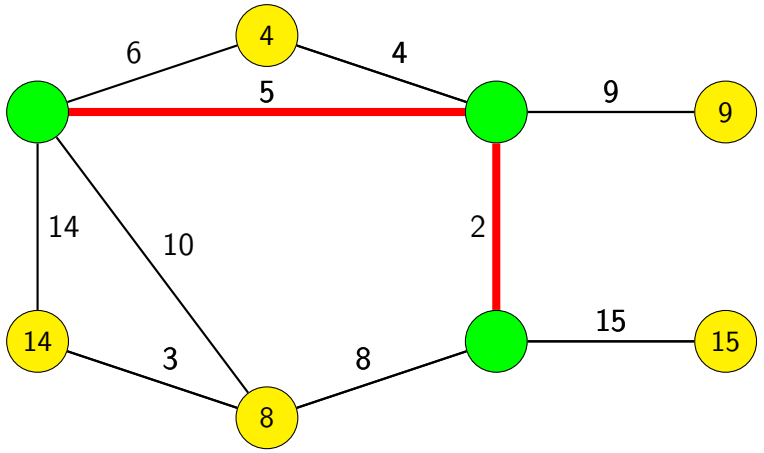
- Fange mit einen Baum aus nur einem Knoten an, indem ein beliebiger Knoten des Graphens ausgewählt wird.
- Finde die günstigste Kante (d. h. mit minimalem Gewicht), die den bisherigen Baum verlässt.
- Füge den über diese Kante erreichten (Rand-)Knoten dem Baum hinzu, zusammen mit der Kante.
- Fahre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

Ist das korrekt? Und wenn, was ist die Komplexität?

Prim's Algorithmus – Grundgerüst

```
1  // ungerichteter Graph G mit n Knoten
2  void primMST(Graph G, int n) {
3    initialisiere alle Knoten als ungesehen (WHITE);
4    wähle irgendeinen Knoten s und markiere ihn mit Baum (BLACK);
5    reklassifiziere alle zu s adjazenten Knoten als Rand (GRAY);
6    while (es gibt Randknoten) {
7      wähle von allen Kante zwischen einem Baumknoten t und
8        einem Randknoten v die billigste;
9      reklassifiziere v als Baum (BLACK);
10     füge Kante tv zum Baum hinzu;
11     reklassifiziere alle zu v adjazenten ungesehenen Knoten
12       als Rand (GRAY);
13   }
14 }
```


Prim's Algorithmus – Beispiel

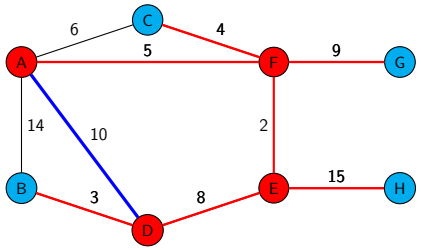


Die MST-Eigenschaft

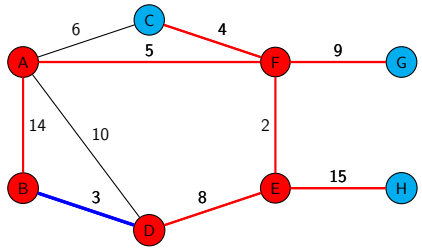
MST-Eigenschaft auf G

Ein Spannbaum T hat die minimale-Spannbaum-Eigenschaft auf G, wenn

1. jede Kante $(u, v) \in G - T$ einen Zyklus in T erzeugen würde, und
2. in diesem Zyklus die Kante mit maximalem Gewicht ist.



Spannbaum, der die MST-Eigenschaft hat



Spannbaum, der die MST-Eigenschaft verletzt

Spannbäume mit gleichem Gewicht

Lemma

Wenn zwei Spannbäume T_1 und T_2 die MST-Eigenschaft auf G haben, dann ist $W(T_1) = W(T_2)$.

Beweis.

Induktion über die Anzahl k an Kanten in $T_1 - T_2$.

Induktionsanfang:

$k = 0$, T_1 und T_2 sind gleich, daher ist $W(T_1) = W(T_2)$.

Induktionsschritt:

$k > 0$, Angenommen das Lemma gilt für Spannbäume, die sich nur in $j < k$ Kanten unterscheiden.

- T_1 und T_2 unterscheiden sich nun um k Kanten.
- Betrachte die günstigste Kante (u, v) aus $T_2 - T_1$.
- Der Pfad von u nach v in T_1 (mit Länge $\geqslant 2$) muss eine Kante $(w, x) \notin T_2$ enthalten. Es gilt $W(w, x) = W(u, v)$, denn:

→

Spannbäume mit gleichem Gewicht – Beweis

Beweis. (Forts.)

- Der Pfad von u nach v in T_1 (mit Länge $\geqslant 2$) muss eine Kante $(w, x) \notin T_2$ enthalten. Es gilt $W(w, x) = W(u, v)$, denn:
  - Wegen der MST-Eigenschaft von T_1 gilt $W(w, x) \leqslant W(u, v)$.
  - Wegen der MST-Eigenschaft von T_2 gilt $W(u, v) \leqslant W(w, x)$.
  - $\Rightarrow$ $W(w, x) = W(u, v)$.
- Füge (u, v) zu T_1 hinzu und entferne (w, x); Wir erhalten T_1' mit $W(T_1) = W(T_1')$.
- Bemerke, daß T_1' die MST-Eigenschaft hat, da T_1 die hatte.
- Da T_1' die MST-Eigenschaft hat und T_1' und T_2 sich nur noch um $k-1$ Kanten unterscheiden:

$\Rightarrow$ mit Induktionsannahme folgt, dass $W(T_1') = W(T_2)$ und damit $W(T_1) = W(T_2)$. □

Theorem

Theorem

Ein Baum ist ein minimaler Spannbaum gdw. er die MST-Eigenschaft hat.

Beweis.

($\Rightarrow$) Durch Widerspruch. Sei T ein MST von G. Nehme an, daß T die MST-Eigenschaft verletzt, d. h., das Hinzufügen von der Kante $(u, v) \notin T$ zu T erzeugt einen Zyklus, so dass für $(x, y) \in T$ aus dem Zyklus $W(u, v) < W(x, y)$. Das Ersetzen von (x, y) durch (u, v) in T liefert den Spannbaum T' mit $W(T') < W(T)$. Also kann T kein MST gewesen sein. Widerspruch.

($\Leftarrow$) Angenommen T hat die MST-Eigenschaft. Sei T' ein MST von G. Wegen $\Rightarrow$ hat dann T' die MST-Eigenschaft. Mit dem vorigen Lemma haben Spannbäume mit MST-Eigenschaft das selbe Gewicht, also: $W(T) = W(T')$. Also ist auch T ein MST. □

Korrektheit von Prim's Algorithmus

Theorem

Der vom Prim's Algorithmus erzeugte Spannbaum T_k mit $k > 0$ Knoten ($k = 1, \ldots, n$) hat die MST-Eigenschaft auf dem durch die Knoten in T_k induzierten Teilgraph G_k (d. h. (u, v) ist eine Kante in G_k wenn (u, v) eine Kante in G ist, und u und v sind in T_k).

Beweis.

Induktion nach k.

Induktionsanfang:

$k = 1$, T_1 und G_1 enthalten nur Knoten und keine Kanten. T_1 hat damit die MST-Eigenschaft in G_1.

→

Korrektheit von Prim's Algorithmus – Beweis

Beweis. (Forts.)

Induktionsschritt:

$k > 1$. Angenommen T_j hat die MST-Eigenschaft auf G_j für $j < k$.

- Sei $v \in T_k - T_{k-1}$ die k-te Knote die hinzugefügt wurde und
- $(u_1, v), \ldots, (u_m, v)$ die Kanten zwischen Knoten in T_{k-1} und v.
- Sei (u_1, v) die günstigste dieser Kanten die in T_k gewählt wurde.
- Betrachte die Kante $(x, y) \in G_k - T_k$.
  1. Sei $x \neq v$ und $y \neq v$. Dann $(x, y) \in G_{k-1} - T_{k-1}$. Hinzufügen von (x, y) zu T_{k-} liefert einen Zyklus, mit (nach Ind. Annahme) (x, y) maximalem Gewicht auf dem Zyklus. Dies ist jedoch der Zykel den es auch in T_k gibt. Deswegen hat T_k die MST-Eigenschaft auf G_k.
  2. siehe nächste Folie.

→

Korrektheit von Prim's Algorithmus – Beweis

Beweis. (Forts.)

Induktionsschritt:

2. $(x, y) \in \{(u_2, v), \ldots, (u_m, v)\}$ mit $m > 2$. Betrachte den Pfad von v nach u_i $(1 < i \leqslant m)$ in T_k: $v \underbrace{w_1}_{=u_1} w_2 \ldots \underbrace{w_\ell}_{=u_i}$. Nimm an, daß (w_j, w_{j+1}) die erste Kante auf diesem Pfad ist mit $W(w_i, w_{i+1}) > W(u_i, v)$. Sei (w_{p-1}, w_p) die letzte Kante auf dem Pfad mit $W(w_{p-1}, w_p) > W(u_i, v)$. (Möglicherweise gilt $p = j+1$.) Wir zeigen, dass w_j und w_p nicht in T_{k-1} existieren können. (Siehe Vorlesung.) Damit hat keine Kante auf dem Pfad $w_1 \ldots w_\ell$ ein größeres Gewicht als $W(u_i, v)$, und damit auch nicht größer als $W(u_1, v)$. Also, hat T_k die MST-Eigenschaft.

□

Korrektheit

Korrektheit

Der Algorithmus von Prim bestimmt einen minimalen Spannbaum.

Beweis.

Sei $G = (V, E)$ und $|V| = n$ Anzahl der Knoten.

- Sei T_n der durch Prim berechnete Baum. Dann ist $G_n = G$, und es folgt – siehe letztes Theorem – dass T_n die MST-Eigenschaft hat.
- Da T_n die MST-Eigenschaft hat gdw. es ein MST ist, folgt: T_n ist ein MST.

□

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 für die Randknoten.

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:

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

- Welche Implementierung der Prioritätswarteschlange ist dafür gut geeignet?

Drei Prioritätswarteschlangenimplementierungen

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

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

- Leider kann $m \in \Theta(n^2)$ sein, wodurch Prim schlechter als $O(n^2)$ wird.
- Können wir vielleicht `decrKey` schneller machen? – Ja.

Prioritätswarteschlange, die Vierte (I)

Wie erhalten wir $\Theta(1)$ für `decrKey`?

$\Rightarrow$ Indem wir die Priorität direkt bei den Knoten speichern.

- Wir kennen das bereits von `color` bei DFS und BFS.
- Gleichzeitig können wir so direkt die verwendete Kante ($\rightarrow$ Ergebnis) speichern (als Vorgänger*baum*, vgl. Kritische-Pfad-Analyse):

```
1 struct VertexState {
2   int color;
3   int parent;
4   float curWeight;
5 }

7 VertexState state[n]; // enthält color[n]
```

- Das entspricht der Prioritätswarteschlangenimplementierung auf unsortierten Arrays, allerdings mit „Löchern“.
- Nur die Einträge mit `color == GRAY` (Randknoten) sind in der Warteschlange gesetzt und zwar mit Priorität `curWeight`.

Prioritätswarteschlange, die Vierte (II)

- Die Implementierung operiert direkt auf `state`.

```
1 // man könnte das etwa so schreiben:
2 VertexState state[n];
3 PriorityQueue pq = VS_PriorityQueue<&VS.curWeight>(&state);
```

- In `pq.decrKey(int elem, VertexState &newkey)` muss nur `state[elem]` mit `newkey` überschrieben werden (außer `color`). $\Rightarrow \Theta(1)$
- Zum Einfügen (`pq.insert(int elem, VertexState &key)`) wird `color = GRAY` gesetzt und der Rest von `key` übernommen. $\Theta(1)$
- Löschen (`pq.delMin()`) setzt einen Knoten auf `color = BLACK`.
- Als Elemente halten wir also die Nummer des entsprechenden Randknotens (`int`); als Schlüssel sozusagen `VertexState`, wobei `curWeight` daraus als Priorität verwendet wird.
- Wir ergänzen außerdem noch zwei Operationen:

Prioritätswarteschlange, die Vierte (III)

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 (I)

```
1 // Ergebnis als Vorgängerbaum in .parent:
2 // ∀v ∈ V : (x, v) ∈ MST(V, E) gdw. x = state[v].parent, x ≠ −1
3 VertexState[n] primMST(List adjLst[n], int n, int start) {
4   VertexState state[n] = // (eigentlich im Konstruktor von pq)
5       { color: WHITE, parent: -1, curWeight: +inf };
6   PriorityQueue pq = VS_PriorityQueue<&VS.curWeight>(&state);
7
8   pq.insert(start, {parent: -1, curWeight: 0});
9   while (!pq.isEmpty()) { // solange es Randknoten gibt
10    int v = pq.getMin(); // günstigste Kante, bzw. Randknoten
11    pq.delMin(); // setzt auch Farbe auf BLACK
12    updateFringe(pq, adjList, v); // update den Rand
13  }
14  return state;
15 }
```


Der Algorithmus von Prim – Implementierung (II)

```
1 void updateFringe(PriorityQueue &pq, List adjLst[], int v) {
2   foreach (edge in adjLst[v]) {
3     // berechnet MST.
4     float newWeight = edge.weight;
5
6     if (pq.getColor(edge.w) == WHITE) { // -> GRAY
7       pq.insert(edge.w, {parent: v, curWeight: newWeight});
8     } else if (pq.getColor(edge.w) == GRAY) {
9       if (newWeight < pq.getWeight(edge.w)) {
10        // Randknoten-update: Kante von v aus ist besser
11        pq.decrKey(edge.w, {parent: v, curWeight: newWeight});
12      }
13    }
14  }
15 }
```


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 insgesamt 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 $O(n^2 + m) = O(n^2)$.