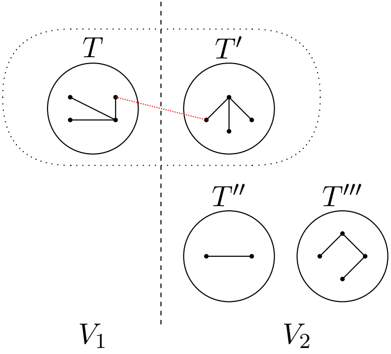
Generischer MST-Algorithmus



Wald F

3.3 Kruskal's Algorithmus

algorithm Kruskal $(G, w) :=$

  sortiere die Kanten nach aufsteigendem Gewicht in eine Liste L
  initialisiere Wald $F = \{T_i, i = 1, \ldots, n\}$, mit $T_i = \{v_i\}$
  MSB:=$\emptyset$
  for $i := 1$ **to** length(L) **do**
    $\{v, w\} := L_i$
    $x :=$ Baum $\in F$, der v enthält; **co** $x :=$*Find*(v) **oc**
    $y :=$ Baum $\in F$, der w enthält; **co** $y :=$*Find*(w) **oc**
    if $x \neq y$ **then**
      MSB:=MSB $\cup \{\{v, w\}\}$
      Union(x, y) **co** gewichtete Vereinigung **oc**
    fi
  od

Korrektheit: Falls die Gewichte eindeutig sind ($w(\cdot)$ injektiv), folgt die Korrektheit direkt mit Hilfe der "blauen" und "roten" Regel. Ansonsten Induktion über die Anzahl $|V|$ der Knoten:
Ind. Anfang: $|V|$ klein: $\surd$

Sei $r \in \mathbb{R}$, $E_r := \{e \in E; w(e) < r\}$.
Es genügt zu zeigen:
Sei $T_1, \ldots, T_k$ ein minimaler Spannwald für $G_r := \{V, E_r\}$ (d.h., wir betrachten nur Kanten mit Gewicht $< r$). Sei weiter T ein MSB von G, dann gilt die

Hilfsbehauptung: Die Knotenmenge eines jeden T_i induziert in T einen zusammenhängenden Teilbaum, dessen Kanten alle Gewicht $< r$ haben.

Beweis der Hilfsbehauptung:

Sei $T_i =: (V_i, E_i)$. Wir müssen zeigen, dass V_i in T einen zusammenhängenden Teilbaum induziert. Seien $u, v \in V_i$ zwei Knoten, die in T_i durch eine Kante e verbunden sind. Falls der Pfad in T zwischen u und v auch Knoten $\notin V_i$ enthält (also der von V_i induzierte Teilgraph von T nicht zusammenhängend ist), dann enthält der in T durch Hinzufügen der Kante e entstehende Fundamentalkreis notwendigerweise auch Kanten aus $E \setminus E_r$ und ist damit gemäß der "roten" Regel nicht minimal! Da T_i zusammenhängend ist, folgt damit, dass je zwei Knoten aus V_i in T immer durch einen Pfad verbunden sind, der nur Kanten aus E_r enthält.

Zeitkomplexität: (mit $n = |V|, m = |E|$)

Sortieren	$m \log m = \mathcal{O}(m \log n)$
$\mathcal{O}(m)$ *Find*-Operationen	$\mathcal{O}(m)$
$n-1$ *Unions*	$\mathcal{O}(n \log n)$

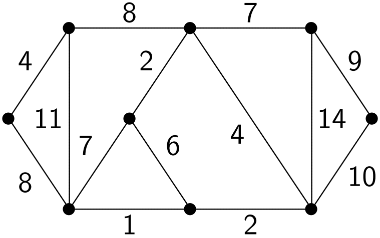
Satz 103

Kruskal's MSB-Algorithmus hat die Zeitkomplexität $\mathcal{O}((m+n)\log n)$.

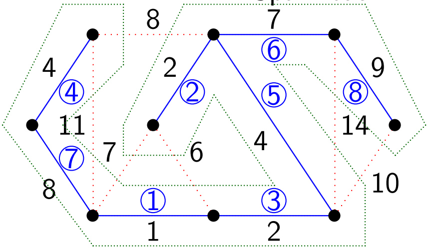
Beweis:

s.o. □

Beispiel 104 (Kruskals Algorithmus)



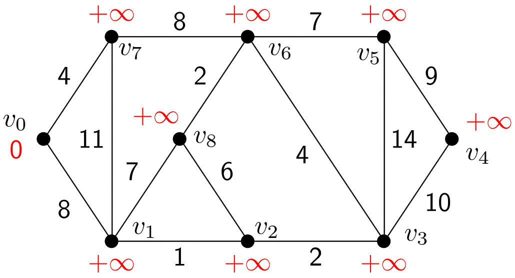
$\Downarrow$ minimaler (nicht eindeutiger) Spannbaum



3.4 Prim's Algorithmus

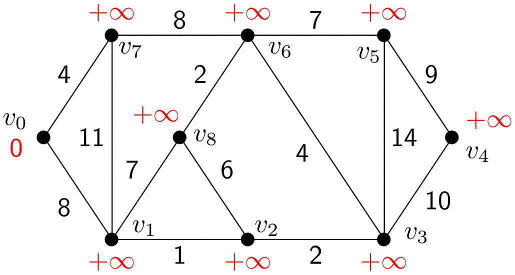
algorithm PRIM-MSB $(G, w) :=$
  initialisiere Priority Queue PQ mit Knotenmenge V und
    Schlüssel $+\infty$, $\forall v \in V$
  wähle Knoten r als Wurzel (beliebig)
  Schlüssel $k[r] := 0$
  Vorgänger$[r]$:= **nil**
  while $PQ \neq \emptyset$ **do**
    $u :=$ *ExtractMin*(PQ)
    for alle Knoten v, die in G zu u benachbart sind **do**
      if $v \in PQ$ **and** $w(\{u, v\}) < k[v]$ **then**
        Vorgänger$[v] := u$
        $k[v] := w(\{u, v\})$
      fi
    od
  od

Beispiel 105 (Prim's Algorithmus)

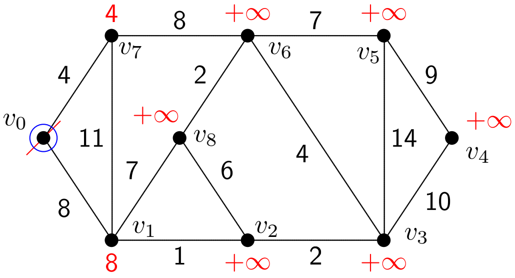


Ausgangszustand:
alle Schlüssel $= +\infty$
aktueller Knoten u: ⦿
Startknoten: r $(= v_0)$

Beispiel 105 (Prim's Algorithmus)

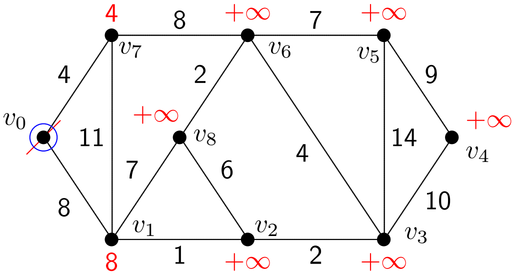


Ausgangszustand:
alle Schlüssel $= +\infty$
aktueller Knoten u: ⦿
Startknoten: r $(= v_0)$

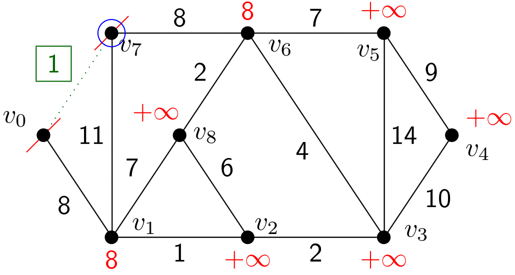


suche u := FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\})$ < Schlüssel$[v]$:
$(v_1 = 8, v_7 = 4)$

Beispiel 105 (Prim's Algorithmus)

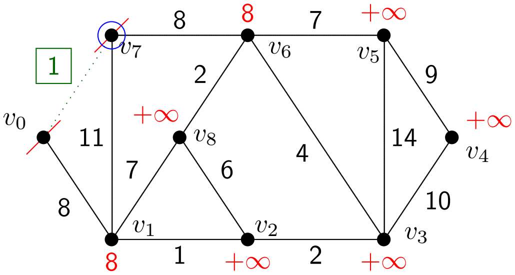


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$: $(v_1 = 8, v_7 = 4)$

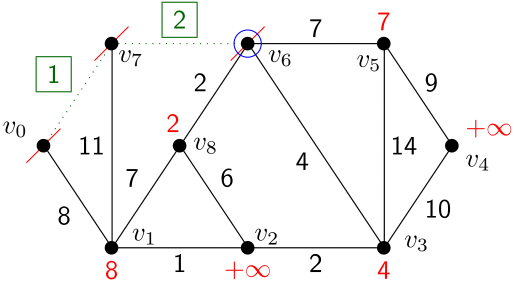


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$: $(v_6 = 8)$

Beispiel 105 (Prim's Algorithmus)

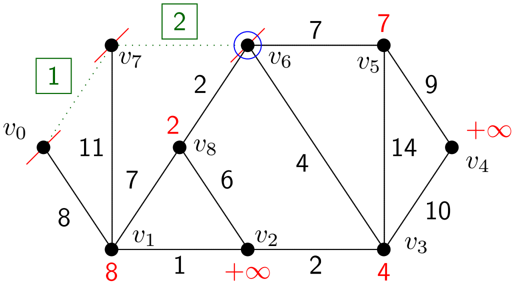


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$: $(v_6 = 8)$

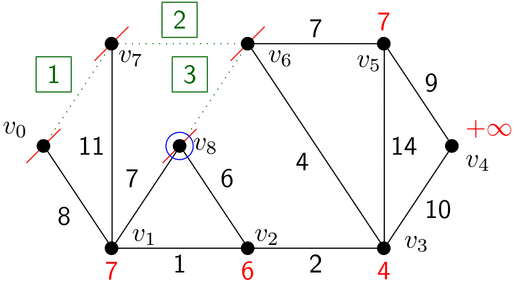


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$: $(v_3 = 4, v_5 = 7, v_8 = 2)$

Beispiel 105 (Prim's Algorithmus)

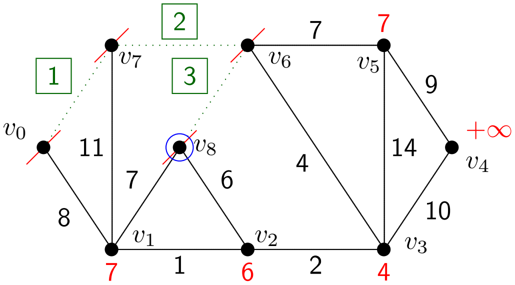


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u, v\}) <$ Schlüssel$[v]$: $(v_3 = 4, v_5 = 7, v_8 = 2)$

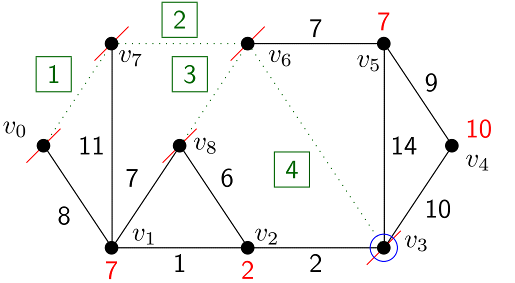


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u, v\}) <$ Schlüssel$[v]$: $(v_1 = 7, v_2 = 6)$

Beispiel 105 (Prim's Algorithmus)

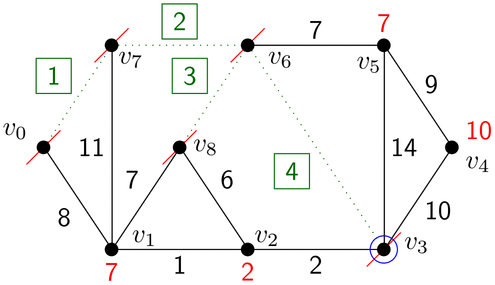


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\})<$ Schlüssel$[v]$: $(v_1=7, v_2=6)$

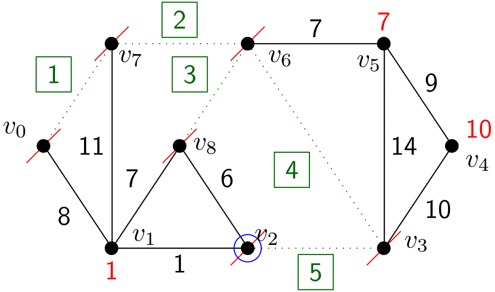


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\})<$ Schlüssel$[v]$: $(v_2=2, v_4=10)$

Beispiel 105 (Prim's Algorithmus)

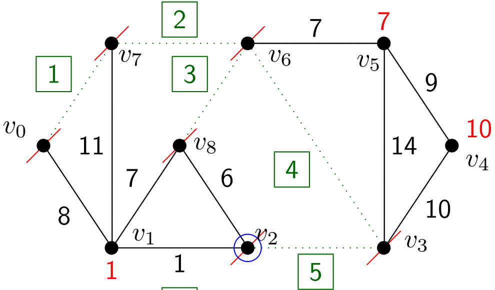


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$: $(v_2 = 2, v_4 = 10)$

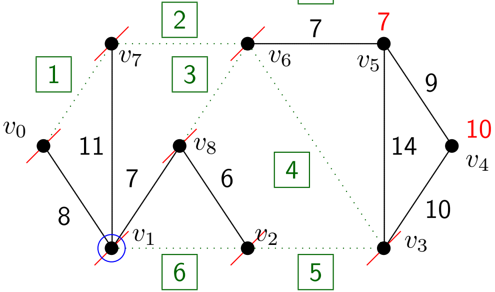


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$: $(v_1 = 1)$

Beispiel 105 (Prim's Algorithmus)

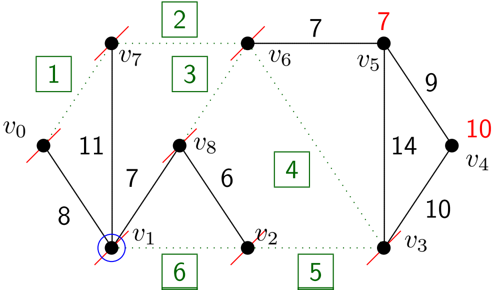


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$: $(v_1 = 1)$

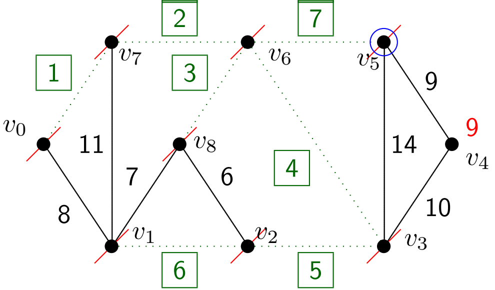


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$: solche Nachbarn existieren nicht

Beispiel 105 (Prim's Algorithmus)

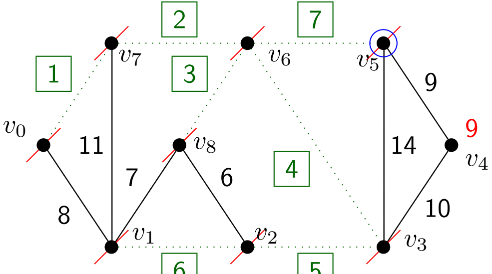


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\})$ < Schlüssel$[v]$: solche Nachbarn existieren nicht

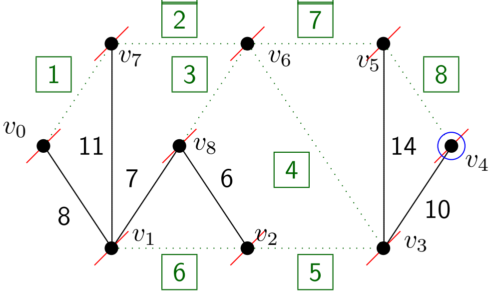


suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\})$ < Schlüssel$[v]$: $(v_4 = 9)$

Beispiel 105 (Prim's Algorithmus)



suche u := FindMin(PQ) und entferne u aus PQ setze Schlüssel der Nachbarn in PQ mit w$(\{u,v\}) <$ Schlüssel$[v]$: $(v_4 = 9)$



Endzustand:
suche u := FindMin(PQ) und entferne u aus PQ, damit ist PQ leer und der Algorithmus beendet

Korrektheit: ist klar.

Zeitkomplexität:

- n *ExtractMin*
- $\mathcal{O}(m)$ sonstige Operationen inclusive *DecreaseKey*

Implementierung der Priority Queue mittels Fibonacci-Heaps:

Initialisierung	$\mathcal{O}(n)$
ExtractMins	$\mathcal{O}(n \log n)$ $\quad(\leq n$ Stück)
DecreaseKeys	$\mathcal{O}(m)$ $\quad(\leq m$ Stück)
Sonstiger Overhead	$\mathcal{O}(m)$

Satz 106

Sei $G = (V, E)$ ein ungerichteter Graph (zusammenhängend, einfach) mit Kantengewichten w. Prim's Algorithmus berechnet, wenn mit Fibonacci-Heaps implementiert, einen minimalen Spannbaum von (G, w) in Zeit $\mathcal{O}(m + n \log n)$ (wobei $n = |V|,\ m = |E|$). Dies ist für $m = \Omega(n \log n)$ asymptotisch optimal.

Beweis:

s.o. $\square$

3.5 Prim's Algorithmus, zweite Variante

Die Idee der folgenden Variante von Prim's Algorithmus ist:

Lasse die Priority Queues nicht zu groß werden.

Seien dazu $G = (V, E)$, $|V| = n$, $|E| = m$, w Gewichtsfunktion, und k ein Parameter, dessen Wert wir erst später festlegen werden.

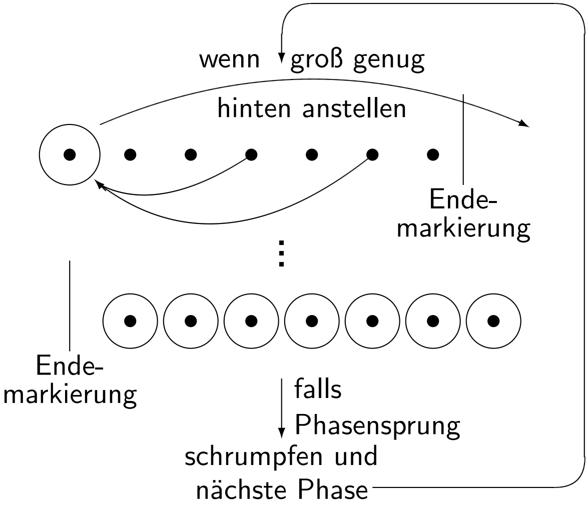
Der Algorithmus arbeitet nun in Phasen wie folgt:

1. Initialisiere eine Schlange von Bäumen, jeder Baum anfangs ein (Super-) Knoten. Zu jedem Baum initialisiere eine Priority Queue (Fibonacci-Heap) mit den Nachbarn der Knoten im Baum, die selbst nicht im Baum sind, als Elementen und jeweils dem Gewicht einer leichtesten Kante zu einem Knoten im Baum als Schlüssel.
2. Markiere jeden Baum in der Schlange mit der Nummer der laufenden Phase.

3. Bestimme k für die Phase
4.

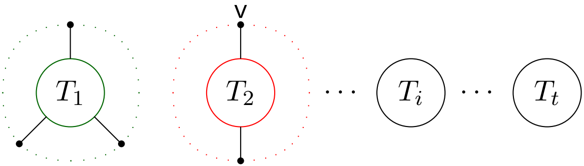
while vorderster Baum in der Schlange hat laufende Phasennummer **do**
    lasse ihn wachsen, solange seine Priority Queue höchstens
        k Elemente enthält (und noch etwas zu tun ist)
    if Priority Queue zu groß (mehr als k Elemente) **then**
        füge Baum mit inkrementierter Phasennummer am
        Ende der Schlange ein
    fi
od

5. Falls „Phasensprung“: Schrumpfe alle Bäume zu Superknoten, reduziere Kantenmenge (d.h., behalte zwischen zwei Knoten jeweils nur die leichteste Kante)
6. Beginne nächste Phase mit Schritt 1.!


wenn groß genug
hinten anstellen
Ende-
markierung
Ende-
markierung
falls
Phasensprung
schrumpfen und
nächste Phase

Analyse des Zeitbedarfs:

Sei t die Anzahl der Bäume in der Schlange zu Beginn einer Phase. Betrachte die Schlange von Bäumen:

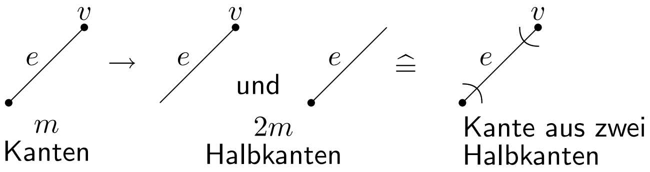


Abgesehen von den Operationen, die bei der Vereinigung von Superknoten anfallen, beträgt der Zeitaufwand pro Phase $\mathcal{O}(m)$.

Vereinigung zweier Superknoten, z.B. T_1 und T_2:
Für jeden (Super-)Knoten v in T_2's Priority Queue:

1. $v \in T_1$: $\surd$ (nichts zu tun)
2. v in Priority Queue von T_1: *DecreaseKey*. Hilfsdatenstruktur: für alle Knoten in den Priority Queues ein Zeiger auf den Superknoten, in dessen Priority Queue der Knoten letztmals am Anfang der Schlange stand.
3. sonst: Einfügen

Betrachte Knoten mit „Halbkanten": jede Halbkante kommt nur 1x in allen Bäumen der Queue vor. Mit m Kanten ergeben sich $2m$ Halbkanten.



Zeitaufwand pro Phase (mit Bildung der Superknoten zu Beginn):

- Initialisierung: $\mathcal{O}(m)$
- *ExtractMin*: $< t$ Operationen
- sonstige Priority Queue-Operationen, Overhead: Zeit $\mathcal{O}(m)$

Da die Priority Queues höchstens k Elemente enthalten, wenn darauf eine „teure" Priority Queue-Operation durchgeführt wird, sind die Kosten pro Phase

$$\mathcal{O}(t \log k + m)\,.$$

Setze $k = 2^{\frac{2m}{t}}$ (damit $t \log k = 2m$). Damit sind die Kosten pro Phase $\mathcal{O}(m)$.

Wieviele Phasen führt der Algorithmus aus?

t ist die Anzahl der Superknoten am Anfang einer Phase, t' sei diese Zahl zu Beginn der nächsten Phase. Sei a die durchschnittliche Anzahl ursprünglicher Knoten in jeder der t Priority Queues zu Anfang der Phase, a' entsprechend zu Beginn der nächsten Phase.

Wir haben:

1. $a = \frac{2m}{t}$
2. $t' \leq \frac{2m}{k}$ (mit Ausnahme ev. der letzten Phase)

Also:

$$a' = \frac{2m}{t'} \geq k = 2^{\frac{2m}{t}} = 2^a$$

Für die erste Phase ist $a = \frac{2m}{n}$, für jede Phase ist $a \leq n - 1$. Also ist die Anzahl der Phasen

$$\leq 1 + \min\left\{i; \log_2^{(i)}(n-1) \leq \frac{2m}{n}\right\}.$$

Setzen wir $\beta(m,n) := \min\left\{i; \log_2^{(i)} n \leq \frac{m}{n}\right\}$, dann gilt

$$\beta(m,n) \leq \log^* n \text{ für } n \leq m \leq \binom{n}{2}.$$

Satz 107

Für gewichtete, zusammenhängende (ungerichtete) Graphen mit n *Knoten,* m *Kanten kann ein minimaler Spannbaum in Zeit* $\mathcal{O}(\min\{m \cdot \beta(m,n), m + n\log n\})$ *bestimmt werden.*