24. Minimale Spannbäume

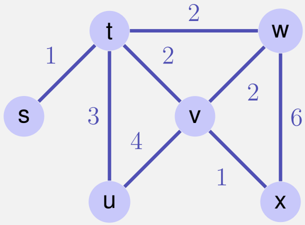
Motivation, Greedy, Algorithmus von Kruskal, Allgemeine Regeln, Union-Find Struktur, Algorithmus von Jarnik, Prim, Dijkstra, Fibonacci Heaps

[Ottman/Widmayer, Kap. 9.6, 6.2, 6.1, Cormen et al, Kap. 23, 19]

Problem

Gegeben: Ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$.

Gesucht: Minimaler Spannbaum $T = (V, E')$, $E' \subset E$, so dass $\sum_{e \in E'} c(e)$ minimal.



Anwendung: Billigstes / kürzestes Kabelnetzwerk

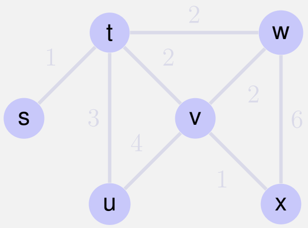
Greedy Verfahren

Zur Erinnerung:

- Gierige Verfahren berechnen die Lösung schrittweise, indem lokal die beste Lösung gewählt wird.
- Die meisten Probleme sind nicht mit einer greedy Strategie lösbar.
- Das Problem der minimalen Spannbäume bildet in diesem Sinne eine der Ausnahmen.

Greedy Idee

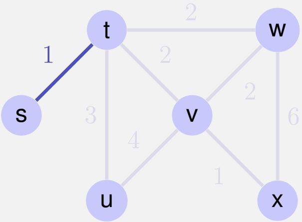
Konstruiere T indem immer die billigste Kante hinzugefügt wird, welche keinen Zyklus erzeugt.



(Lösung ist nicht eindeutig.)

Greedy Idee

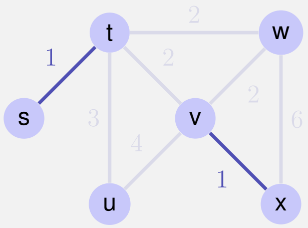
Konstruiere T indem immer die billigste Kante hinzugefügt wird, welche keinen Zyklus erzeugt.



(Lösung ist nicht eindeutig.)

Greedy Idee

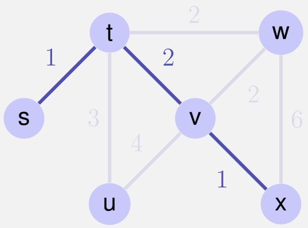
Konstruiere T indem immer die billigste Kante hinzugefügt wird, welche keinen Zyklus erzeugt.



(Lösung ist nicht eindeutig.)

Greedy Idee

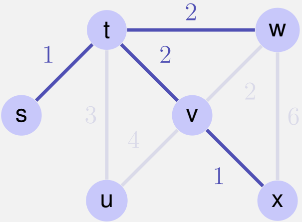
Konstruiere T indem immer die billigste Kante hinzugefügt wird, welche keinen Zyklus erzeugt.



(Lösung ist nicht eindeutig.)

Greedy Idee

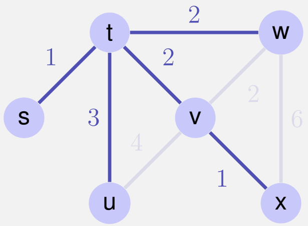
Konstruiere T indem immer die billigste Kante hinzugefügt wird, welche keinen Zyklus erzeugt.



(Lösung ist nicht eindeutig.)

Greedy Idee

Konstruiere T indem immer die billigste Kante hinzugefügt wird, welche keinen Zyklus erzeugt.



(Lösung ist nicht eindeutig.)

Algorithmus MST-Kruskal(G)

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

Sortiere Kanten nach Gewicht $c(e_1) \leq ... \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ **to** $|E|$ **do**
    if $(V, A \cup \{e_k\})$ kreisfrei **then**
        $A \leftarrow E' \cup \{e_k\}$

return (V, A, c)

Korrektheit

Zu jedem Zeitpunkt ist (V, A) ein Wald, eine Menge von Bäumen.

MST-Kruskal betrachtet jede Kante e_k einmal und wählt e_k oder verwirft e_k

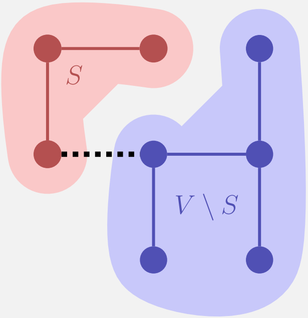
Notation (Momentaufnahme im Algorithmus)

- A: Menge gewählte Kanten
- R: Menge verworfener Kanten
- U: Menge der noch unentschiedenen Kanten

Schnitt

Ein Schnitt von G ist eine Partition $S, V \setminus S$ von V. ($S \subseteq V$).

Eine Kante kreuzt einen Schnitt, wenn einer Ihrer Endpunkte in S und der andere in $V \setminus S$ liegt.



Regeln

1. Auswahlregel: Wähle einen Schnitt, den keine gewählte Kante kreuzt. Unter allen unentschiedenen Kanten, welche den Schnitt kreuzen, wähle die mit minimalem Gewicht.
2. Verwerfregel: Wähle einen Kreis ohne verworfene Kanten. Unter allen unentschiedenen Kanten im Kreis verwerfe die mit maximalem Gewicht.

Regeln

Kruskal wendet beide Regeln an:

1. Ein gewähltes e_k verbindet zwei Zusammenhangskomponenten, sonst würde ein Kreis erzeugt werden. e_k ist beim Verbinden minimal, man kann also einen Schnitt wählen, den e_k mit minimalem Gewicht kreuzt.
2. Ein verworfenes e_k ist Teil eines Kreises. Innerhalb des Kreises hat e_k maximales Gewicht.

Korrektheit

Theorem

Jeder Algorithmus, welcher schrittweise obige Regeln anwendet bis $U = \emptyset$ ist korrekt.

Folgerung: MST-Kruskal ist korrekt.

Auswahlinvariante

Invariante: Es gibt stets einen minimalen Spannbaum, der alle gewählten und keine der verworfenen Kanten enthält.

Wenn die beiden Regeln die Invariante erhalten, dann ist der Algorithmus sicher korrekt. Induktion:

- Zu Beginn: $U = E$, $R = A = \emptyset$. Invariante gilt offensichtlich.
- Invariante bleibt erhalten.
- Am Ende: $U = \emptyset$, $R \cup A = E \Rightarrow (V, A)$ ist Spannbaum.

Beweis des Theorems: zeigen nun, dass die beiden Regeln die Invariante erhalten.

Auswahlregel erhält Invariante

Es gibt stets einen minimalen Spannbaum T, der alle gewählten und keine der verworfenen Kanten enthält.

Wähle einen Schnitt, den keine gewählte Kante kreuzt. Unter allen unentschiedenen Kanten, welche den Schnitt kreuzen, wähle eine Kante e mit minimalem Gewicht.

- Fall 1: $e \in T$ (fertig)
- Fall 2: $e \notin T$. Dann hat $T \cup \{e\}$ einen Kreis, der e enthält. Kreis muss einen zweite Kante e' enthalten, welche den Schnitt auch kreuzt.[34] Da $e' \notin R$ ist $e' \in U$. Somit $c(e) \leq c(e')$ und $T' = T \setminus \{e'\} \cup \{e\}$ ist auch minimaler Spannbaum (und $c(e) = c(e')$).

---

[34] Ein solcher Kreis enthält mindestens einen Knoten in S und einen in $V \setminus S$ und damit mindestens zwei Kanten zwischen S und $V \setminus S$.

Verwerfregel erhält Invariante

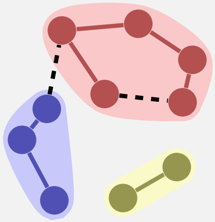
Es gibt stets einen minimalen Spannbaum T, der alle gewählten und keine der verworfenen Kanten enthält.

Wähle einen Kreis ohne verworfene Kanten. Unter allen unentschiedenen Kanten im Kreis verwerfe die Kante e mit maximalem Gewicht.

- Fall 1: $e \notin T$ (fertig)
- Fall 2: $e \in T$. Entferne e von T, Das ergibt einen Schnitt. Diesen Schnitt muss eine weitere Kante e' aus dem Kreis kreuzen. Da $c(e') \leq c(e)$ ist $T' = T \setminus \{e\} \cup \{e'\}$ auch minimal (und $c(e) = c(e')$).

Zur Implementation

Gegeben eine Menge von Mengen $i \equiv A_i \subset V$. Zur Identifikation von Schnitten und Kreisen: Zugehörigkeit der beiden Endpunkte einer Kante zu einer der Mengen.



Zur Implementation

Allgemeines Problem: Partition (Menge von Teilmengen) z.B.
$\{\{1, 2, 3, 9\}, \{7, 6, 4\}, \{5, 8\}, \{10\}\}$

Benötigt: ADT (Union-Find-Struktur) mit folgenden Operationen

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

Union-Find Algorithmus MST-Kruskal(G)

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

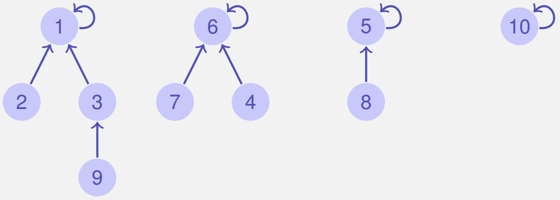
Sortiere Kanten nach Gewicht $c(e_1) \leq ... \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ **to** $|V|$ **do**
  MakeSet(k)

for $k = 1$ **to** $|E|$ **do**
  $(u, v) \leftarrow e_k$
  if Find(u) $\neq$ Find(v) **then**
    Union(Find(u), Find(v))
    $A \leftarrow A \cup e_k$

return (V, A, c)

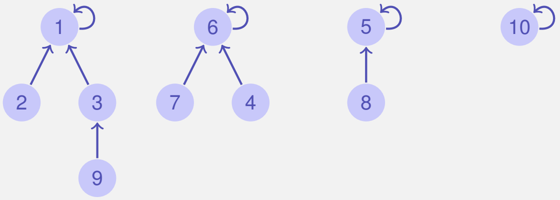
Implementation Union-Find

Idee: Baum für jede Teilmenge in der Partition, z.B.
$\{\{1,2,3,9\},\{7,6,4\},\{5,8\},\{10\}\}$



Baumwurzeln = Namen der Mengen,
Bäume = Elemente der Mengen

Implementation Union-Find



Repräsentation als Array:

Index	1	2	3	4	5	6	7	8	9	10
Parent	**1**	1	1	6	**5**	**6**	5	5	3	**10**

Implementation Union-Find

Index	1	2	3	4	5	6	7	8	9	10
Parent	**1**	1	1	6	**5**	**6**	5	5	3	**10**

Operationen:

- Make-Set(i): $p[i] \leftarrow i$; **return** i
- Find(i): **while** $(p[i] \neq i)$ **do** $i \leftarrow p[i]$; **return** i
- Union(i, j): $p[j] \leftarrow i$; **return** i

Optimierung der Laufzeit für Find

Baum kann entarten: Beispiel Union$(1, 2)$, Union$(2, 3)$, Union$(3, 4)$, ...
Idee: Immer kleineren Baum unter grösseren Baum hängen.
Zusätzlich: Grösseninformation g

Operationen:

- Make-Set(i): $p[i] \leftarrow i$; $g[i] \leftarrow 1$; **return** i
- Union(i, j):
  if $g[j] > g[i]$ **then** swap(i, j)
  $p[j] \leftarrow i$
  $g[i] \leftarrow g[i] + g[j]$
  return i

Beobachtung

Theorem

Obiges Verfahren Vereinigung nach Grösse konserviert die folgende Eigenschaft der Bäume: ein Baum mit Höhe h hat mindestens 2^h Knoten.

Unmittelbare Folgerung: Laufzeit Find = $\mathcal{O}(\log n)$.

Beweis

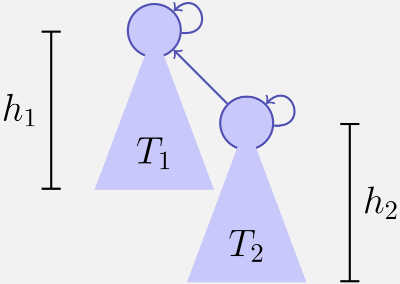
Induktion: nach Vorraussetzung haben Teilbäume jeweils mindestens 2^{h_i} Knoten. ObdA: $h_2 \leq h_1$.

- $h_2 < h_1$:

$$h(T_1 \oplus T_2) = h_1 \Rightarrow g(T_1 \oplus T_2) \geq 2^h$$

- $h_2 = h_1$:

$$g(T_1) \geq g(T_2) \geq 2^{h_2}$$
$$\Rightarrow g(T_1 \oplus T_2) = g(T_1) + g(T_2) \geq 2 \cdot 2^{h_2} = 2^{h(T_1 \oplus T_2)}$$



Weitere Verbesserung

Bei jedem Find alle Knoten direkt an den Wurzelknoten hängen.

Find(i):

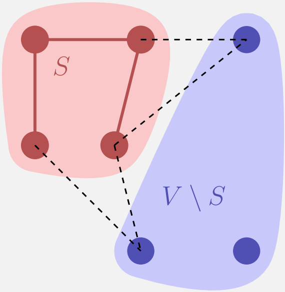
$j \leftarrow i$
while $(p[i] \neq i)$ **do** $i \leftarrow p[i]$
while $(j \neq i)$ **do**
  $t \leftarrow j$
  $j \leftarrow p[j]$
  $p[t] \leftarrow i$
return i

Amortisierte Laufzeit: amortisiert *fast* konstant (Inverse der Ackermannfunktion).

MST Algorithmus von Jarnik, Prim, Dijkstra

Idee: Starte mit einem $v \in V$ und lasse von dort unter Verwendung der Auswahlregel einen Spannbaum wachsen:

$S \leftarrow \{v_0\}$
for $i \leftarrow 1$ **to** $|V|$ **do**
  Wähle billigste (u, v) mit $u \in S$, $v \notin S$
  $A \leftarrow A \cup \{(u, v)\}$
  $S \leftarrow S \cup \{v\}$



Laufzeit

Trivial $\mathcal{O}(|V| \cdot |E|)$.

Verbesserungen (wie bei Dijskstras ShortestPath):

- Billigeste Kante nach S merken: für jedes $v \in V \setminus S$. Jeweils $\deg^+(v)$ viele Updates für jedes neue $v \in S$. Kosten: $|V|$ viele Minima + Updates: $\mathcal{O}(|V|^2 + \sum_{v \in V} \deg^+(v)) = \mathcal{O}(|V|^2 + |E|)$
- Mit Minheap, Kosten: $|V|$ viele Minima = $\mathcal{O}(|V| \log |V|)$, $|E|$ Updates: $\mathcal{O}(|E| \log |V|)$, Initialisierung $\mathcal{O}(|V|)$: $\mathcal{O}(|E| \cdot \log |V|.)$
- Mit Fibonacci-Heap: $\mathcal{O}(|E| + |V| \cdot \log |V|)$.

Fibonacci Heaps

Datenstruktur zur Verwaltung von Elementen mit Schlüsseln.
Operationen

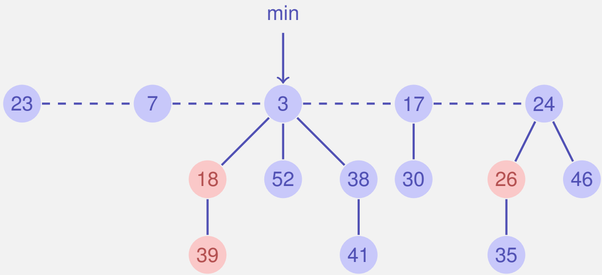
- MakeHeap(): Liefere neuen Heap ohne Elemente
- Insert(H, x): Füge x zu H hinzu
- Minimum(H): Liefere Zeiger auf das Element m mit minimalem Schlüssel
- ExtractMin(H): Liefere und entferne (von H) Zeiger auf das Element m
- Union(H_1, H_2): Liefere Verschmelzung zweier Heaps H_1 und H_2
- DecreaseKey(H, x, k): Verkleinere Schlüssel von x in H zu k
- Delete (H, x): Entferne Element x von H

Vorteil gegenüber Binary Heap?

	Binary Heap (worst-Case)	Fibonacci Heap (amortisiert)
MakeHeap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
ExtractMin	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(n)$	$\Theta(1)$
DecreaseKey	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$

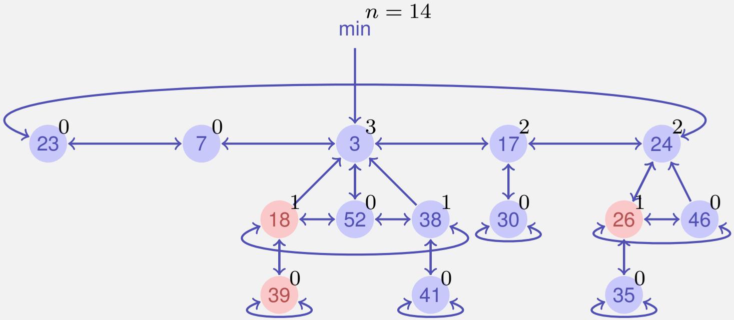
Struktur

Menge von Bäumen, welche der Min-Heap Eigenschaft genügen.
Markierbare Knoten.



Implementation

Doppelt verkettete Listen von Knoten mit marked-Flag und Anzahl Kinder. Zeiger auf das minimale Element und Anzahl Knoten.



Einfache Operationen

- MakeHeap (trivial)
- Minimum (trivial)
- Insert(H, e)
  1. Füge neues Element in die Wurzelliste ein
  2. Wenn Schlüssel kleiner als Minimum, min-pointer neu setzen.
- Union (H_1, H_2)
  1. Wurzellisten von H_1 und H_2 aneinander hängen
  2. Min-Pointer neu setzen.
- Delete(H, e)
  1. DecreaseKey($H, e, -\infty$)
  2. ExtractMin(H)

ExtractMin

1. Entferne Minimalknoten m aus der Wurzelliste
2. Hänge Liste der Kinder von m in die Wurzelliste
3. Verschmelze solange heapgeordnete Bäume gleichen Ranges, bis alle Bäume unterschiedlichen Rang haben:
   Rangarray $a[0, \ldots, n]$ von Elementen, zu Beginn leer. Für jedes Element e der Wurzelliste:
   a. Sei g der Grad von e.
   b. Wenn $a[g] = nil$: $a[g] \leftarrow e$.
   c. Wenn $e' := a[g] \neq nil$: Verschmelze e mit e' zu neuem e'' und setze $a[g] \leftarrow nil$. Setze e'' unmarkiert Iteriere erneut mit $e \leftarrow e''$ vom Grad $g + 1$.

DecreaseKey (H, e, k)

1. Entferne e von seinem Vaterknoten p (falls vorhanden) und erniedrige den Rang von p um eins.
2. Insert(H, e)
3. Vermeide zu dünne Bäume:
    a. Wenn $p = nil$, dann fertig
    b. Wenn p unmarkiert: markiere p, fertig.
    c. Wenn p markiert: unmarkiere p, trenne p von seinem Vater pp ab und Insert(H, p). Iteriere mit $p \leftarrow pp$.

Abschätzung für den Rang

Theorem

Sei p Knoten eines F-Heaps H. Ordnet man die Söhne von p in der zeitlichen Reihenfolge, in der sie an p durch Zusammenfügen angehängt wurden, so gilt: der i-te Sohn hat mindestens Rang $i - 2$

Beweis: p kann schon mehr Söhne gehabt haben und durch Abtrennung verloren haben. Als der $i$te Sohn p_i angehängt wurde, müssen p und p_i jeweils mindestens Rang $i - 1$ gehabt haben. p_i kann maximal einen Sohn verloren haben (wegen Markierung), damit bleibt mindestens Rang $i - 2$.

Abschätzung für den Rang

Theorem

Jeder Knoten p vom Rang k eines F-Heaps ist Wurzel eines Teilbaumes mit mindestens F_{k+1} Knoten. (F: Fibonacci-Folge)

Beweis: Sei S_k Minimalzahl Nachfolger eines Knotens vom Rang k in einem F-Heap plus 1 (der Knoten selbst). Klar: $S_0 = 1$, $S_1 = 2$. Nach vorigem Theorem $S_k \geq 2 + \sum_{i=0}^{k-2} S_i, k \geq 2$ (p und Knoten p_1 jeweils 1). Für Fibonacci-Zahlen gilt (Induktion) $F_k \geq 2 + \sum_{i=2}^{k} F_i, k \geq 2$ und somit (auch Induktion) $S_k \geq F_{k+2}$.

Fibonacci-Zahlen wachsen exponentiell ($\mathcal{O}(\varphi^k)$) Folgerung: Maximaler Grad eines beliebigen Knotens im Fibonacci-Heap mit n Knoten ist $\mathcal{O}(\log n)$.

Amortisierte Worst-case-Analyse Fibonacci Heap

$t(H)$: Anzahl Bäume in der Wurzelliste von H, $m(H)$: Anzahl markierte Knoten in H ausserhalb der Wurzelliste, Potentialfunktion $\Phi(H) = t(H) + 2 \cdot m(H)$. Zu Anfang $\Phi(H) = 0$. Potential immer nichtnegativ.

Amortisierte Kosten:

- Insert(H, x): $t'(H) = t(H) + 1$, $m'(H) = m(H)$, Potentialerhöhung 1, Amortisierte Kosten $\Theta(1) + 1 = \Theta(1)$
- Minimum(H): Amortisierte Kosten = tatsächliche Kosten = $\Theta(1)$
- Union(H_1, H_2): Amortisierte Kosten = tatsächliche Kosten = $\Theta(1)$

Amortisierte Kosten ExtractMin

- Anzahl der Bäume in der Wurzelliste $t(H)$.
- Tatsächliche Kosten der ExtractMin Operation: $\mathcal{O}(\log n + t(H))$.
- Nach dem Verschmelzen noch $\mathcal{O}(\log n)$ Knoten.
- Anzahl der Markierungen kann beim Verschmelzen der Bäume maximal kleiner werden.
- Amortisierte Kosten von ExtractMin also maximal

$$\mathcal{O}(\log n + t(H)) + \mathcal{O}(\log n) - \mathcal{O}(t(H)) = \mathcal{O}(\log n).$$

Amortisierte Kosten DecreaseKey

- Annahme: DecreaseKey führt zu c Abtrennungen eines Knotens von seinem Vaterknoten, tatsächliche Kosten $\mathcal{O}(c)$
- c Knoten kommen zur Wurzelliste hinzu
- Löschen von $(c-1)$ Markierungen, Hinzunahme maximal einer Markierung
- Amortisierte Kosten von DecreaseKey:

$$\mathcal{O}(c) + (t(H) + c) + 2 \cdot (m(H) - c + 2)) - (t(H) + 2m(H)) = \mathcal{O}(1)$$