

# A Linear-Time Algorithm For Finding Tree-Decompositions Of Small Treewidth

Autor: Hans L. Bodlaender – [h.l.bodlaender@uu.nl](mailto:h.l.bodlaender@uu.nl)

Vortrag: Maximilian F. Göckel – [uzkns@student.kit.edu](mailto:uzkns@student.kit.edu)

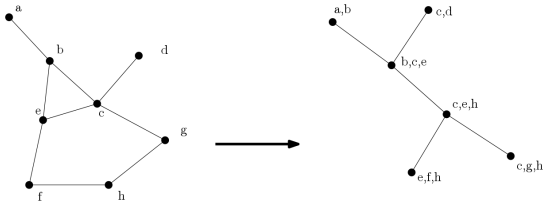
Institut für Theoretische Informatik - Proseminar Algorithmen für NP-schwere Probleme

Eine Baumzerteilung eines Graphen  $G = (V, E)$  ist ein Tupel  $(X, T)$  wo  $T = (I, F)$  ein Baum ist und  $X = \{X_i | i \in I\}$  eine Familie von Teilmengen von  $V$  wobei jedes  $X_i$  einen Knoten in  $T$  darstellt. Es gilt:

1.  $\bigcup_{i \in I} X_i = V$
2.  $\forall (v, w) \in E : \exists i \in I : v, w \in X_i$
3.  $\forall w \in X_i, X_j : \text{Jedes } X_k \text{ im Pfad zwischen } X_i, X_j \text{ enthält } w$

# Tree-decomposition

## Veranschaulichung



Jede Baumzerteilung hat eine "Baumweite" (treewidth).

- Baumweite einer Zerteilung:  $\max(|X_i|_{i \in I} - 1)$  ("Zerteilungsweite")
- Baumweite eines Graphen: Minimale Zerteilungsweite aller Zerteilungen

Eine Baumzerteilung der Weite max.  $k$  heißt auch  $k$ -Baumzerteilung oder  $k$ -Zerteilung.


Ein  $k$ -Tree  $G = (V, E)$  ist ...

- induktiv konstruiert aus einem  $k$ -Tree mit  $|V| - 1$  Knoten indem man einen Knoten zu einer Clique max.  $k$  hinzufügt oder
- ein Graph der eine Clique max.  $k$  ist

Ein  $k$ -Tree ist der maximale Graph mit Baumweite  $k$ .

Graph  $G = (V, E)$  ist partieller  $k$ -Tree  $\Leftrightarrow$

- $G$  ist Teilgraph eines  $k$ -Trees oder
- $G$  hat Baumweite max.  $k$

- Maximum-Weight Independent Set in Linearzeit lösbar
- Hohe Baumweite  $\Leftrightarrow$  Hohe Komplexität in der Systemanalyse
- Erkennungsalgorithmen von Graphen
  - Graphen mit geb. Pfadweite 
  - Partielle  $k$ -Trees

Eingabe: Graph  $G = (V, E)$  und Konstante  $k \in \mathbb{N}$ .

Der Graph wird als Adjazenzliste übergeben.

Ausgabe in  $O(n)$ :

- "Baumweite von  $G$  ist größer als  $k$ " oder
- "Baumweite von  $G$  ist maximal  $k$ "
  - Baumzerteilung von  $G$  mit Baumweite  $k$

Ist  $k$  Teil der Eingabe so ist das Problem NP-schwer und sogar NP-vollständig.



Ein Knoten  $v$  ist ...

- ... *von niedrigem Grad* wenn  $\deg(v) \leq d$ 
  - $d := 2k^3 \cdot (k + 1) \cdot (4k^2 + 12k + 16)$
  - Analog: Hoher Grad  $\Leftrightarrow \deg(v) > d$
  - Auch "low-deg.-" und "high-deg.-Knoten" genannt
- ... *Friendly* wenn er low-deg. und adjazent zu einem weiteren low-deg.-Knoten ist
- ... *Simplizial* wenn alle Nachbarn eine Clique formen



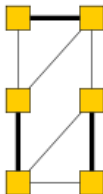
# Maximal Matching $M \subseteq E$



$M \subseteq E$  ist Matching in  $G = (V, E) \Leftrightarrow$  Keine 2 Kanten aus  $M$  haben gemeinsamen Endknoten

$M \subseteq E$  ist Maximal Matching wenn keine Kante mehr zu  $M$  hinzugefügt werden kann, sodass  $M$  Matching bleibt.

Ein Maximal Matching kann in  $O(|V|)$  gefunden werden, wenn die Baumweite durch ein  $k$  beschränkt ist.






### Lemma 4.2.

$G$  hat Baumweite max.  $k \Rightarrow$  Es gilt mindestens:

1.  $G$  hat min.  $\frac{|V|}{4k^2+12k+16} =: \lambda$  Friendly-Knoten, oder
2. Der Improved-Graph von  $G$  hat min.  $\frac{\lambda}{2}|V|$  l.simp.-Knoten

Die Anzahl Friendly-Knoten in  $G$  wird mit  $n_f$  notiert.

1. Finde Maximal Matching  $M \subseteq E$
2. Jede Kante in  $M$  kontrahieren  $\rightarrow \tilde{G} = (\tilde{V}, \tilde{E})$  entsteht 
3. Kompletten Algorithmus auf  $\tilde{G}$  ausführen um Baumzerteilung  $(Y, T)$  von  $\tilde{G}$  zu berechnen  
 $\rightarrow$  Wenn Baumweite von  $\tilde{G} > k \Rightarrow$  **STOP** (LEMMA 3.4.)
4. Mit LEMMA 3.3.  $(2k + 1)$ -Zerteilung  $(X, T)$  von  $G$  aus  $(Y, T)$  erstellen
5. Mit THEOREM 2.4. prüfen ob Baumweite von  $G > k$  ist, falls nein:  
 $\rightarrow k$ -Baumzerteilung von  $G$  ausgeben

1. Maximal Matching  $M \subseteq E$  finden

Ein Maximal Matching kann greedy in  $O(|V| + |E|)$  gefunden werden.

### Lemma 2.3.

$$\text{tw}(G) \leq k \Rightarrow |E| \leq k|V| - \frac{1}{2}k(k+1)$$

$$\Rightarrow |E| \in O(|V|)$$

$\Rightarrow$  Max. Matching kann in  $O(|V|)$  gefunden werden

2. Jede Kante in  $M$  kontrahieren um Graphen  $\tilde{G}$  zu erhalten

Eine Kante kann in  $O(1)$  kontrahiert werden, liegt der Graph als Adjazenzliste vor.

### Lemma 2.3.

$$\text{tw}(G) \leq k \Rightarrow |E| \leq k|V| - \frac{1}{2}k(k+1)$$

$$|M| \subseteq E \text{ und } E \in O(|V|)$$

$\Rightarrow$  Alle Kanten können in  $O(|V|)$  kontrahiert werden.

3. Kompletten Algorithmus auf  $\tilde{G}$  ausführen um Baumzerteilung  $(Y, T)$  von  $\tilde{G}$  auszugeben

Ein Maximal Matching hat min.  $\frac{n_f}{2d}$  Kanten.

Für jeden Friendly-Knoten gilt:

- Er ist Endpunkt von einem  $m \in M$ , oder
  - Er ist adjazent zu einem Friendly-Knoten, der Endpunkt ist
- $\Rightarrow \forall m \in M$  werden max.  $2d$  Friendly-Knoten assoziiert, die Endpunkt sind oder adjazent zu einem Friendly-Endpunkt sind.

Ist ein Friendly-Knoten nicht assoziiert so ist  $M$  nicht maximal

$$\Rightarrow |M| \geq \frac{n_f}{2d}$$

3. Kompletten Algorithmus auf  $\tilde{G}$  ausführen um Baumzerteilung  $(Y, T)$  von  $\tilde{G}$  auszugeben

Ein Maximum Matching hat min.  $\frac{n_f}{2d}$  Kanten.

$$\Rightarrow |\tilde{V}| = \left(1 - \frac{1}{2d(4k^2+12k+16)}\right) \cdot |V|$$



4. Mit **LEMMA 3.3**. Zerteilung  $(X, T)$  von  $G$  aus  $(Y, T)$  erstellen

$$f_M : V \mapsto \tilde{V} : \begin{cases} f_M(v) = v & \text{Wenn } v \text{ nicht Endpunkt in } M \text{ ist} \\ f_M(v) = f_M(w) & \text{sonst} \end{cases}$$

Dabei ist  $f_M(w)$  der Knoten der bei der Kontraktion von  $(v, w) \in M$  bleibt.

Ist  $(Y, T)$  Zerteilung von  $\tilde{G}$ , so ist  $(X, T)$  mit  $X_i = \{v \in V \mid f_M(v) \in Y_i\}$  Zerteilung von  $G$  mit Weite max.  $2k + 1$

Hier kommt noch ein Bild von Schritt 4 hin



5.  $k$ -Zerteilung von  $G$  errechnen

5.1. prüfen ob Weite von  $G > k$  ist  $\Rightarrow$  **STOP**

**THEOREM 2.4.:** " $\forall k, l \in \mathbb{N} \exists$  Linearzeitalgorithmus, welcher aus einem Graph  $G = (V, E)$  und einer  $l$ -Zerteilung prüft ob die Baumweite von  $G$  max.  $k$  ist und eine  $k$ -Zerteilung errechnet"

Laufzeit:  $O(l^{l-2} \cdot ((2l+3)^{2l+3} \cdot (\frac{8}{3}2^{2k+2})^{2l+3})^{2l-1})) \in O(k^3)$  bei  $l \in O(k)$

### Lemma 4.2.

$G$  hat Baumweite max.  $k \Rightarrow$  Es gilt mindestens:

1.  $G$  hat min.  $\frac{|V|}{4k^2+12k+16} =: \lambda$  Friendly-Knoten, oder
2. Der Improved-Graph von  $G$  hat min.  $\frac{\lambda}{2}|V|$  l.simp.-Knoten

Die Anzahl Friendly-Knoten in  $G$  wird mit  $n_f$  notiert.

# Verbesserter Graph $G'$

## Erstellung und Eigenschaften

$G' = (V, E')$  ist  $G = (V, E)$  mit Kanten  $(v, w) \in E' \forall v, w \in V$  sodass  $v, w$  min.  $k + 1$  gem. Nachbarn mit Grad max.  $k$  haben.

### Lemma 4.1.

$\text{tw}(G) \leq k \Leftrightarrow \text{tw}(G') \leq k.$

Jede  $k$ -Zerteilung von  $G$  ist auch eine  $k$ -Zerteilung von  $G'$  und umgekehrt.



1. Improved-Graph  $G'$  berechnen  
 $\rightarrow \exists$  l.simp.-Knoten  $v$  mit  $\deg(v) = k + 1 \Rightarrow$  **STOP**
2. Alle l.simp.-Knoten in Menge  $SL$  und von  $G$  entfernen  $\Rightarrow \hat{G}$  entsteht  
 $\rightarrow |SL| < c_2 \cdot |V| \Rightarrow$  **STOP** (THEOREM 4.2.)
3. Algorithmus rekursiv auf  $\hat{G}$  ausführen  $\Rightarrow$  Ausgabe von Zerteilung  $(Y, T)$  von  $\hat{G}$   
 $\rightarrow \text{tw}(\hat{G}) > k \Rightarrow$  **STOP** ( $\hat{G}$  Teilgraph von  $G \Rightarrow \text{tw}(G) > k$ )
4. Füge  $SL$  wieder in die Zerteilung  $(Y, T)$  ein  
 $\Rightarrow$  Baumzerteilung  $(X, T)$  von  $G$  mit Baumweite max.  $k$

1. Den Improved-Graph  $G'$  berechnen

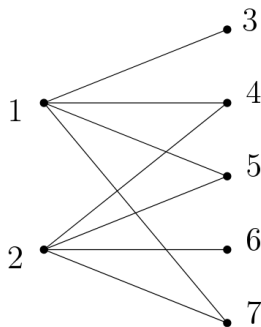
1.1. Alle I-simp.-Knoten von  $G$  in Menge  $SL$  zusammenfassen

Wir definieren  $Q = \{((v_i, v_j), -) \mid (v_i, v_j) \in E, i < j\}$   
 $\cup \{((v_i, v_j), v) \mid v_i, v_j \in N_G(v), i < j \wedge \deg(v) \leq k\}$

und  $Q_{v_i, v_j} = \{((v_i, v_j), v) \mid v_i, v_j \text{ fest, } v \in V\} \subseteq Q$

$Q$  für  $k = 2$ :

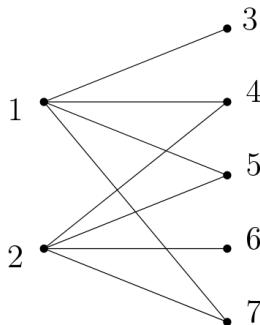
- $((1, 3), -)$
- $((1, 4), -)$
- $((1, 5), -)$
- $((1, 7), -)$
- $((2, 4), -)$
- $((2, 5), -)$
- $((2, 6), -)$
- $((2, 7), -)$





Q:

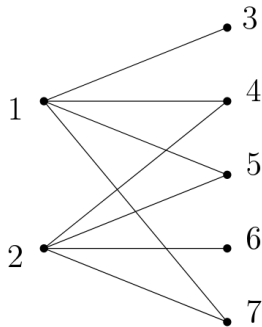
- $((1, 3), -)$
- $((1, 4), -)$
- $((1, 5), -)$
- $((1, 7), -)$
- $((2, 4), -)$
- $((2, 5), -)$
- $((2, 6), -)$
- $((2, 7), -)$
- $((1, 2), 4)$
- $((1, 2), 5)$
- $((1, 2), 7)$





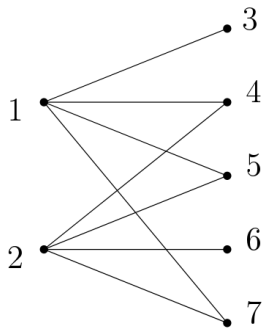
Q, erster BucketSort:

- $((1, 3), -)$
- $((1, 4), -)$
- $((1, 5), -)$
- $((1, 7), -)$
- $((1, 2), 4)$
- $((1, 2), 5)$
- $((1, 2), 7)$
- $((2, 4), -)$
- $((2, 5), -)$
- $((2, 6), -)$
- $((2, 7), -)$



$Q$ , zweiter BucketSort:

- $((1, 2), 4)$
- $((1, 2), 5)$
- $((1, 2), 7)$
- $((1, 3), -)$
- $((2, 4), -)$
- $((1, 4), -)$
- $((1, 5), -)$
- $((2, 5), -)$
- $((2, 6), -)$
- $((1, 7), -)$
- $((2, 7), -)$



$$Q_{v_i, v_j} = \{((v_i, v_j), v) \mid v_i, v_j \text{ fest}, v \in V\} \subseteq Q$$

Falls  $|Q_{v_i, v_j}| \geq k \rightarrow (v_i, v_j) \in E'$ , da  $v_i$  und  $v_j$  nun min.  $(k+1)$  gemeinsame Nachbarn haben.

Für jedes Element aus der oberen Menge und wenn  $((v_i, v_j), -) \in Q$ :  
Füge  $(v_i, v_j)$  für jedes  $v \in V$  zu  $Q_v$  hinzu, sodass  $Q_v$  alle Kanten von Nachbarn von  $v$  enthält.



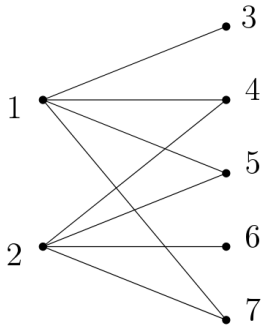


Q, zweiter BucketSort:

- 1  $((1, 2), 4)$
- 2  $((1, 2), 5)$
- 3  $((1, 2), 7)$
- $((1, 3), -)$
- $((2, 4), -)$
- ⋮

S:

1	2	3	4	5	6	7
leer	leer	leer	leer	leer	leer	leer



Der Graph  $G' = (V, E')$  kann also aus den verschiedenen  $Q_{v_i, v_j}$  ausgelesen werden.

Das finden von l-simp.-Knoten ist durch  $Q_v$  nun auch möglich: Da alle Nachbarn von  $v$  in  $Q_v$  sind, kann schnell geprüft werden ob  $N_{G'}(v)$  eine Clique formt.

Queue  $Q$  für Menge  $Q$ , Array  $S$  aus Listen für die  $Q_v$ 's.

1. Knoten ordnen ( $v_1, v_2 \dots v_n$ )
2.  $\forall (v_i, v_j) \in E, i < j$  : Lege  $((v_i, v_j), -)$  auf  $Q$
3.  $\forall v \in V$  : Lege alle  $((v_i, v_j), v)$  mit  $v_i, v_j \in N_G(v)$  und  $i < j$  auf  $Q$
4. Bucket-sortiere  $Q$  zwei mal: Ein mal nach dem ersten, dann nach zweiten Eintrag

Sind nach dem Sortieren von  $Q$   $(k + 1)$  Einträge für gleiches  $(v_i, v_j)$  in  $Q$  untereinander  $\rightarrow (v_i, v_j) \in E'$

Ist für solche  $v_i, v_j$  auch  $((v_i, v_j), -)$  in  $Q$ : Füge für jedes  $((v_i, v_j), v)$  das Tupel  $(v_i, v_j)$  in alle  $S[v]$  ein.

Ist in  $S[v]$  jedes  $v_i$  mit jedem  $v_j$  verbunden:  $N'_G(v)$  bildet Clique  $\Rightarrow v$  ist l-simp.  $\Rightarrow v \in SL$



2.  $SL$  von  $G$  entfernen  $\rightarrow \hat{G} = (\hat{V}, \hat{E})$  entsteht
3. Algorithmus rekursiv auf  $\hat{G}$  ausführen


Graph  $\hat{G}$  hat nach Entfernung von  $SL$   $(1 - c_2) \cdot |V|$  Knoten.

Wie in Fall 1 sind alle rekursiven Aufrufe in  $O(|V|)$  möglich.

4. Füge  $SL$  wieder in die Zerteilung  $(Y, T)$  ein

1.  $\forall v \in SL$ : Finde ein  $Y_{i_v} \in Y$  in dem alle Nachbarn von  $v$  sind  
 $(N_G(v) \subseteq Y_{i_v})$
2. Füge  $Y_{i_v} = \{\{v\} \cup N_G(v)\}$  zu  $Y$  hinzu und mache es adjacent zu  $Y_{i_v}$   
 $\Rightarrow$  Baumzerteilung von  $G$  mit Baumweite max.  $k$

$Y_{i_v}$  existiert für jedes  $v$ , da l-simp.-Knoten in  $G$  nicht adjacent sind und  $N_G(v)$  eine Clique formt.

**LEMMA 2.1.i)**: " $(X, T)$  Zerteilung von  $G$  und  $W \subseteq V$  formt Clique in  $G \Rightarrow \exists i \in I : W \subseteq X_i$ " 



- Wir haben:  $(Y, T)$   $k$ -Zerteilung von  $\tilde{G}$ 
  - $Y = \{Y_i | i \in I\}$
- Wir erstellen:  $\forall I \in \{1 \dots k\} : \text{Queue } Q_I$

1. Queue  $Q_I$ : Füge alle Paare  $((v_{i_1}, v_{i_2} \dots v_{i_l}), i)$  für alle  $i \in I$  zu dieser Queue hinzu
  - $v_{i_x} \in Y_i$
2. Füge zu  $Q_I$  noch alle Paare  $((v_{i_1}, v_{i_2} \dots v_{i_l}), v)$  mit  $v \in SL$  und  $v_{i_x} \in N_G(v)$  hinzu



$Q_1$ :

- $((u_1), Y_1)$
- $((v_1), Y_2)$
- $((w_1), Y_3)$
- $((x_1), Y_4)$

$Q_2$ :

- $((u_1, u_2), Y_1)$
- $((v_1, v_2), Y_2)$
- $((w_1, w_2), Y_3)$
- $((x_1, x_2), Y_4)$

$Q_3$ :

- $((u_1, u_2, u_3), Y_1)$
- $((v_1, v_2, v_3), Y_2)$
- $((w_1, w_2, w_3), Y_3)$
- $((x_1, x_2, x_3), Y_4)$

$Q_4$ :

- $((u_1, u_2, u_3, u_4), Y_1)$
- $((v_1, u_2, v_3, v_4), Y_2)$
- $((w_1, u_2, w_3, w_4), Y_3)$
- $((x_1, u_2, x_3, x_4), Y_4)$

$u_1$	$u_2$	$u_3$	$\cdots$	$u_k$
$v_1$	$v_2$	$v_3$	$\cdots$	$v_k$
$w_1$	$w_2$	$w_3$	$\cdots$	$w_k$
$x_1$	$x_2$	$x_3$	$\cdots$	$x_k$
$\vdots$				

Jedes  $Q_l$  enthält nun die ersten  $l$  Knoten jedes  $Y_i \in Y$  und  $l$  Nachbarn jedes  $v \in SL$ .

Bucket-sortiere jedes  $Q_l$   $l$ -mal, einmal für jeden Knoten  $v_{i_x} \rightarrow$  Für jedes  $((v_{i_1}, v_{i_2} \dots v_{i_l}), v)$  das passende Tupel  $((v_{i_1}, v_{i_2} \dots v_{i_l}), i)$  adjazent in  $Q$ .

So kann man den neuen Node direkt richtig erstellen und zu dem  $Y_i$  adjazent machen.



Sämtliche Operationen sind in  $O(n)$  wenn  $k$  Konstant ist.

Ist  $k$  nicht konstant, sondern als Variable Teil der Eingabe, so ist der Algorithmus nicht mehr linear und das Problem wird NP-schwer.

Der konstante Faktor ist deutlich zu hoch für praktische Anwendung, selbst schon für  $k = 4$ .

Allerdings wurde bei vielen Operationen grob geschätzt. Es ist zu erwarten, dass die Konstante noch sinken wird.



Der Algorithmus ist Basis für zwei weitere Theoreme:

- $\exists$  Linearzeit-Erkennungsalgorithmus für jede Klasse an Graphen die nicht alle planaren Graphen enthält und in ihren Minoren abgeschlossen ist
- $\forall k \in \mathbb{N} : \exists$  Linearzeitalgo der prüft ob  $G = (V, E)$  Pfadweite max.  $k$  hat und eine Pfad-Zerteilung ausgibt

Außerdem ist die Erkennung von l.-simp.-Knoten sehr effizient und kann gut als Grundlage für andere Algorithmen genutzt werden.