Bestimmung eines aufspannenden Baumes mit annähernd minimalen Max-Stretch

Dem Fachbereich Mathematik
an der Technischen Universität Darmstadt
zur Erlangung des Grades eines

Bachelor of Science

eingereichte

B a c h e l o r a r b e i t

bearbeitet von

Frank Borchert

aus Kelkheim

Matrikel-Nr.: 1409817

Erstkorrektor: Dr. habil. Marco Lübbecke
Zweitkorrektor: Prof. Dr. Stefan Ulbrich

September 2010

Inhaltsverzeichnis

1 Einleitung

In den folgenden Kapiteln werden wir uns damit beschäftigen einen möglichst guten aufspannenden Baum ($t - Spanner$) eines Graphen G zu kreieren. t gibt dabei das Maximum aller möglichen Quotienten zwischen dem Abstand zweier beliebiger Knoten im Baum und dem Abstand dieser Knoten im Graph an. Die Aufgabe besteht also darin, einen Baum mit möglichst kleinem t zu finden. Für dieses *Minimum Max-Stretch Spanning Tree (MMST)*-Problem werden wir uns im nächsten Kapitel einige wichtige Definition ansehen, bevor wir danach den Algorithmus mit Hilfe eines Beispiels betrachten. Anschließend wollen wir die Güteklasse unserer entstandenen Lösung untersuchen und zuletzt einige Implementierungen in der Programmiersprache *Java* anschauen.

2 Definitionen

Bevor wir uns dem Algorithmus zum Finden eines aufspannenden Baum mit geringem Max-Stretch widmen, betrachten wir einige notwendige Definitionen. Dabei sind im Folgenden G ein Graph, n die Anzahl an Knoten, V_G die Menge aller in G enthaltenden Knoten und E_G die Menge aller in G enthaltenden Kanten.

2.1 Definitionen zur Problemstellung

Die Gewichtsfunktion eines ungewichteten Graphen ist defininiert durch $W_G(e) = 1$ für alle $e \in E_G$. Diese Gewichtsfunktion genügt der Dreiecksungleichung und damit induziert G mit der Funktion W_G eine Metrik, die wir im weiteren Verlauf mit M bezeichnen. Der Abstand zweier Knoten über M bezeichnen wir mit $dist_M(u, v)$, wobei $u, v \in V_G$.
Sei nun T ein aufspannender Baum von G, so ist der Stretch von zwei Knoten u und v definiert als:

$$Str_{T,G}(u, v) = \frac{dist_T(u, v)}{dist_G(u, v)}$$

Damit definieren wir den maximalen Stretch von T:

$$MaxStr(T, G) = \max_{x,y \in V_G, x \neq y} Str_{T,G}(x, y)$$

Um nun den minimalen Max-Stretch eines Graphen zu bestimmen, betrachten wir alle möglichen aufspannenden Bäume T und wählen einen, der den geringsten Max-Stretch hat.

$$MaxStr(G) = \min_{T} MaxStr(T, G)$$

2.2 Definitionen zum Algorithmus

Die weiteren Definitionen sind zwar nicht notwendig um das *MMST-Problem* zu verstehen, jedoch sind sie wesentlich für den Algorithmus im nächsten Kapitel.
Beschäftigen wir uns zunächst mit der *D-Nachbarschaft*, $\Gamma_D(u, M)$, eines Knoten. In dieser *D-Nachbarschaft* sind alle Knoten enthalten, die ausgehend vom Startknoten u in einer Entfernung von maximal D liegen. Damit ist $\Gamma_D(u, M) = \{v \in V_G | dist_M(u, v) \leq D\}$.

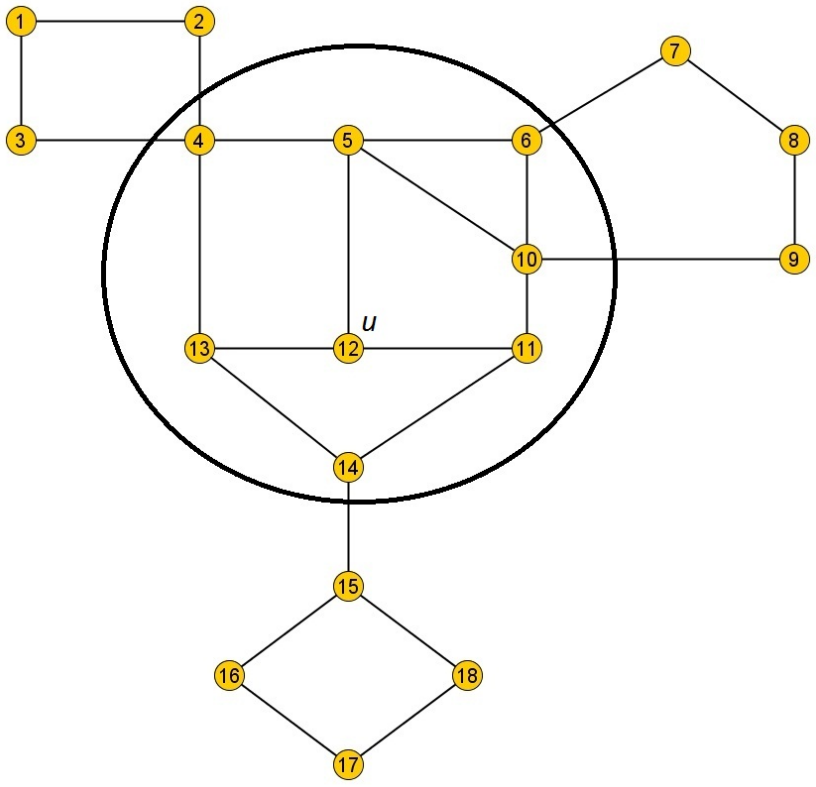

Abbildung 1: $\Gamma_2(12, M)$

Des Weiteren sei $P = \{U_1, \ldots, U_k\}$ eine Partition von G. Das heißt, es gilt sowohl $U_i \subseteq V_G$ und $U_i \cap U_j = \emptyset$ für alle $1 \leq i < j \leq k$, als auch $\cup_{i=1}^{k} U_i = V_G$. Die einzelnen Komponenten U_i von P heißen Cluster. Dabei ist es aber nicht notwendig, dass die einzelnen Cluster im Graph zusammenhängend sind.
Sei $\mathcal{E}(U_i)$ die Menge aller Kanten, die innerhalb des Cluster U_i liegen. Sei des Weiteren $\mathcal{E}(P) = \cup_{U_i \in P} \mathcal{E}(U_i)$ die Vereinigung aller inneren Kanten jedes Clusters. Alle externen bzw. alle Kanten zwischen den Clustern beschreiben wir durch $\overline{\mathcal{E}(P)} = E_G \backslash \mathcal{E}(P)$.
Entfernen wir nun alle Kanten der $D - Nachbarschaft$ vom Knoten u, also $\mathcal{E}(\Gamma_D(u, M))$, erhalten wir zusammenhängende Teilgraphen $G^1, \ldots, G^r$. Gilt nach Entfernen dieser Kanten $|V_{G^i}| \leq n/2$ für alle $1 \leq i \leq r$, dann ist u ein $D - Center$. $\mathcal{I}(P, X)$ gibt alle Cluster $U_i \in P$ an, die sich mit der Menge X überschneiden.

3 Algorithmus

Unser Algorithmus *Solve-MMST* bekommt einen Graph G gegeben und erzeugt mit Hilfe der rekursiven Prozedur *Rec-Cons-Tree* einen Baum T. Zunächst initialisiert der Algorithmus einen leeren Baum T, der während des Prozesses sukzessiv gefüllt wird. Zudem wird eine Partition P von G erstellt, wobei jedes Cluster dieser Partition nur einen einzigen Knoten enthält.
Danach ruft *Solve-MMST* die Prozedur $Rec-Cons-Tree(G, M(G), P, D, 0)$ mit $D = 0$ auf. Diese versucht nun mit gegebenem D ein *D-Center* zu finden. Gelingt dies nicht, so wird eine Fehlermeldung zurückgegeben. Der Algorithmus *Solve-MMST* erhöht D solange um 1 bis *Rec-Conc-Tree* schließlich ein *D-Center* findet. Abbildung 2 und 3 zeigen an Hand eines Beispielgraphen den Versuch für $D = 1$ bzw. $D = 2$ ein *D-Center* zu finden. Besteht unser Graph G nur aus einem einzigen Knoten so wird als Ergebnis generell die leere Menge ausgegeben.

Da $\left|\overline{G^1}\right| > 9$ gilt, ist der Knoten 12 kein *D-Center* und die Prozedur erzeugt eine Fehlermeldung. Das Selbe gilt auch für alle anderen Knoten mit $D = 1$. Für $D = 2$ gilt stets $|G^i| \leq 9, \forall 1 \leq i \leq 7$. Somit haben wir für ein möglichst kleines D ein *D-Center* gefunden.

Als nächstes ruft *Rec-Cons-Tree* die Prozedur $Clst-Dijk(G, M, P, u, D)$ auf, eine leicht abgewandelte Version des Dijkstra-Algorithmus. In unserem Beispiel wäre jetzt $u = 12$, $D = 2$ und die einzelnen Cluster der Partition P enthielten nur jeweils einen Knoten. $Clst-Dijk$ erzeugt zunächst einen *cliqued-cluster-graph* $CC_M(G, P)$, wobei die Knotenmenge $V_{CC_{M(G,P)}}$ der Knoten-

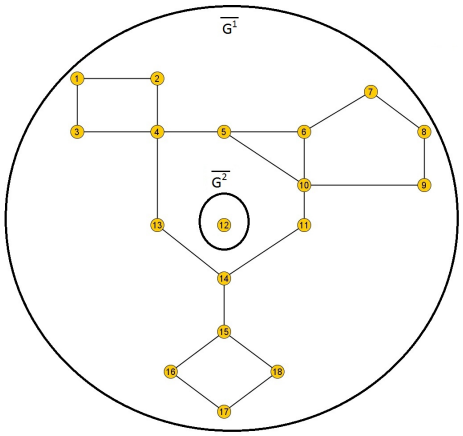

Abbildung 2: $G \backslash \mathcal{E}(\Gamma_1(12, M))$

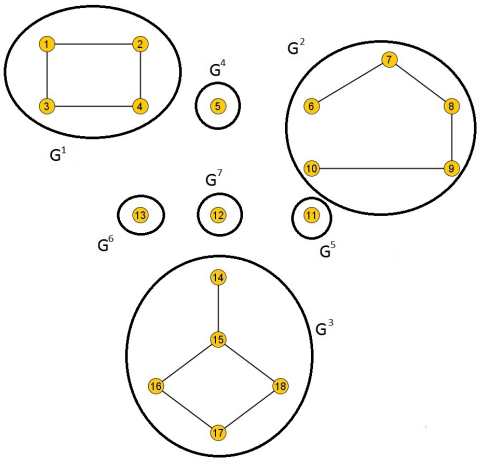

Abbildung 3: $G \backslash \mathcal{E}(\Gamma_2(12, M))$

menge von V_G enspricht und für die Menge der Kanten gilt $E_{CC_{M(G,P)}} = \overline{\mathcal{E}(P)} \cup \bigcup_{U_i \in P}(U_i \times U_i)$. Zur Erinnerung $\overline{\mathcal{E}(P)}$ sind alle Kanten, die die einzelnen Cluster verbinden und zusätzlich in E_G liegen. $U_i \times U_i$ macht unabhängig von den zur Verfügung stehenden Kanten aus jedem Cluster eine Clique, das heißt es existiert zwischen zwei beliebigen Knoten eine Kante. Da unsere Cluster zur Zeit nur aus genau einem Knoten bestehen, ist der entstandene *cliqued-cluster-graph* gerade der Graph G selbst. Nun startet der eigentliche Dijkstra-Prozess. Da wir einen ungewichteten Graphen betrachten, gleicht das Verfahren der normalen Breitensuche. Ausgehend vom *D-Center* werden die jeweils betrachteten Knoten und die Knoten, die im selben Cluster enthalten sind, in ein neues Cluster U' kopiert. Sofern der Knoten noch nicht Teil von U' ist, wird zudem die Kante, über der der betrachtete Knoten erreicht wurde, zur Kantenmenge T_{local} hinzugefügt. Das Verfahren stoppt beim Abstand von D zum Knoten u. In Abbildung 4 ist der *cliqued-cluster-graph* unseres Beispiels zu sehen, wobei alle Kanten, die zur Menge T_{local} hinzugefügt wurden, rot markiert sind.

Wir definieren eine neue Partition P' durch $P' = (P \cup \{U'\}) \backslash \mathcal{I}(P, \Gamma(u, M))$. Mit Hilfe dieser Partition und der zusammenhängenden Teilgraphen $G^1, \ldots, G^r$ wird der Graph zerlegt. Für diese neuen Teilgraphen gelten die Partitionen $P_i = \{U \cap V_{G^i} | U \in P'\}$. Veranschaulichen wir dies mit Hilfe des Beispiels:

$$P' = \{U_1, U_2, U_3, U_7, U_8, U_9, U_{15}, U_{16}, U_{17}, U_{18}, U'\}$$

wobei

$$U' = \{4, 5, 6, 10, 11, 12, 13, 14\}$$

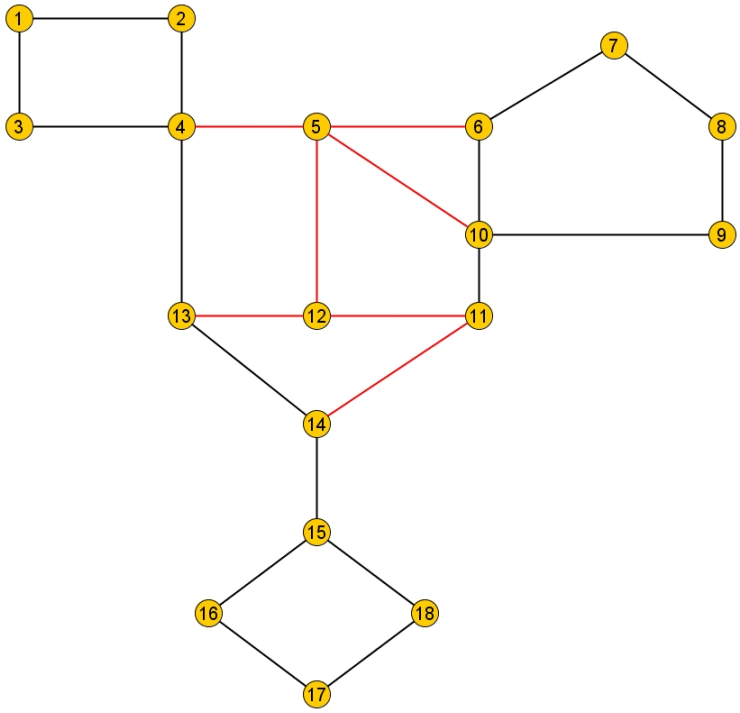

Abbildung 4: Zwischenergebnis - Die Kanten von T_{local} sind rot markiert.

Damit ergeben sich mit Hilfe der Knotenmengen $V_{G^1}, \ldots, V_{G^7}$ folgende neue Partitionen:

$$
\begin{aligned}
P_1 &= \{U_1 \cap V_{G^1}, U_2 \cap V_{G^1}, \ldots, U' \cap V_{G^1}\} = \{\{1\},\{2\},\{3\},\{4\}\} \\
P_2 &= \{U_1 \cap V_{G^2}, U_2 \cap V_{G^2}, \ldots, U' \cap V_{G^2}\} = \{\{7\},\{8\},\{9\},\{6,10\}\} \\
P_3 &= \{U_1 \cap V_{G^3}, U_2 \cap V_{G^3}, \ldots, U' \cap V_{G^3}\} = \{\{15\},\{16\},\{17\},\{18\},\{19\}\} \\
P_4 &= \{U_1 \cap V_{G^4}, U_2 \cap V_{G^4}, \ldots, U' \cap V_{G^4}\} = \{\{5\}\} \\
P_5 &= \{U_1 \cap V_{G^5}, U_2 \cap V_{G^5}, \ldots, U' \cap V_{G^5}\} = \{\{11\}\} \\
P_6 &= \{U_1 \cap V_{G^6}, U_2 \cap V_{G^6}, \ldots, U' \cap V_{G^6}\} = \{\{13\}\} \\
P_7 &= \{U_1 \cap V_{G^7}, U_2 \cap V_{G^7}, \ldots, U' \cap V_{G^7}\} = \{\{12\}\}
\end{aligned}
$$

Nun wird $Rec - Cons - Tree(G(V_{G^i}), M(V_{G^i}), P_i, D, l+1)$ aufgerufen und für jeden dieser Teilgraphen, sofern er nicht nur aus einem einzigem Knoten besteht, ein aufspannender Baum T_i erstellt. Die Eingabe $l+1$ gibt dabei den Grad der Rekursion an.

Die Lösung T ensteht durch die Vereinigung aller aus *Rec-Cons-Tree* entstanden Bäume. Bei enstsprechender Wahl der *D-Center* könnte der durch diesen Algorithmus entstandene Baum so aussehen:

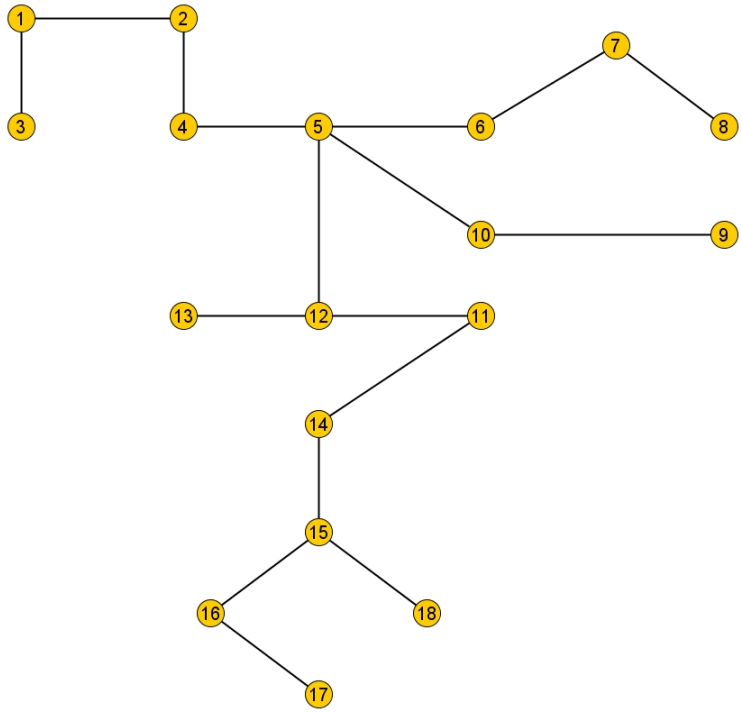

Abbildung 5: mögliches Endergebnis

Das komplette Beispiel einschließlich aller Teilschritte befindet sich im Anhang.

4 Güte der Lösung

In diesem Kapitel wollen wir unsere berechnete Lösung betrachten und uns Folgendes fragen: Ist unsere Lösung überhaupt ein Baum? Gibt es eine obere Schranke für den maximalen Stretch unserer Lösung?
Sei im Folgenden der Graph $\hat{G}$ die Eingabe und $\hat{T}$ die zugehörige errechnete Lösung.

Beobachtung 1
Die Kantenmenge T_{local} induziert einen Baum für die Cluster von $\mathcal{I}(P, U')$

Beobachtung 2
$\Gamma_D(u, M) \cap V_{G^i} \neq \emptyset, \forall 1 \leq i \leq r$

Lemma 3

Betrachte den Graph G während der Prozedur *Rec-Cons-Tree* im Rekursionslevel l. Seien x und y zwei Knoten in V_G. Dann gilt zu jedem Zeitpunkt des Algorithmus während des Rekursionslevels l:
Der Graph $\hat{T}$ enthält einen Pfad zwischen x und y, genau dann wenn ein Cluster U existiert, so dass $x, y \in U$.

Beweis:

„$\Leftarrow$“: Seien x und y Knoten in G. Sei U ein Cluster und es gilt $x, y \in U$ im Rekursionslevel l.
Beweis per Induktion nach l:
Sei $l = 0$, dann besteht U entweder aus genau einem Knoten und es gilt $x = y$ oder U ist das neu geformte Cluster U'. Im zweiten Fall existiert in $\hat{T}$ ein Pfad vom betrachteten *D-Center* u zum Knoten x bzw. y. Somit existiert auch ein Pfad zwischen x und y. Nehmen wir nun an, die Aussage gilt für jedes Rekursionslevel kleiner als l. Gehören x und y schon im Rekursionslevel $l-1$ zum selben Cluster, so enthält der Graph $\hat{T}$ mit Hilfe der Induktionsannahme einen Pfad zwischen x und y. Im anderen Fall ist das Cluster U, dem x und y angehören, das im Rekursionslevel l aus den Clustern $U_1 \dots U_m$ neu gebildetet Cluster. Nach Rekursionsannahme gilt für zwei Knoten v und v' in U_i, $\forall 1 \leq i \leq m$, der Graph $\hat{T}$ enthält einen Pfad zwischen v und v'. Beobachtung 1 sagt uns dann, es existiert in $\hat{T}$ ein Pfad zwischen zwei beliebigen Knoten in U, insbesondere zwischen x und y.

„$\Rightarrow$“: Seien x und y zwei Knoten in G, die während des Rekursionslevels l durch den Pfad π verbunden sind.
Beweis per Induktion nach der Länge von π:
Ist $|\pi| = 0$, so gilt $x = y$ und die Aussage ist wahr. Nehmen wir an, die Aussage sei bewiesen für jeden Pfad π der Länge kleiner als k und betrachten nun einen Pfad π zwischen x und y der Länge k. Sei $l' \leq l$ das Rekursionslevel in dem der Pfad π erstellt und $(x', y') \in E_\pi$ die letzte Kante, die zu $\hat{T}$ hinzugeführt wurde. Das heißt, zuvor existierte ein Pfad zwischen x und x' und ein weiterer zwischen y und y'. Nach der Rekursionsannahme müssen während des Rekursionslevel l' die Cluster U_x mit $x, x' \in U_x$ und U_y mit $y, y' \in U_y$ existiert haben. Nachdem die Prozedur *Clst-Dijk* die Kante (x', y') zu $\hat{T}$ hinzugefügt hat, ersteht ein neues Cluster, bestehend aus U_x und U_y. Somit sind x und y Mitglied des gleichen Clusters im Rekursionslevel l.

Lemma 4
Betrachte den *cliqued-cluster-graph* $CC_M(G, M)$, der wärend der Prozedur *Clst-Dijk* konstruiert wird. Es gilt:
$dist_{CC_M(G,P)}(x, y) = dist_{\hat{G}}(x, y), \forall x, y \in V_G$

Korollar 5
Für den in *Clst-Dijk* konstruierten Graphen $CC_M(G, P)$ gilt:
$\mu(CC_M(G, P)) = M$

Korollar 6
Nach der Beendingung der Prozedur *Clst-Dijk* gilt für das Cluster U':
$\Gamma_D(u, M) \subseteq U'$

Lemma 7
Seien $x, y \in V_{\hat{G}}$ und $(x, y) \in E_{\hat{G}}$. Dann gilt an einer Stelle während der Ausführung von *Rec-Cons-Tree*, dass $x, y \in \Gamma_D(u, M)$ für das betrachtete *D-Center* u.

Beweis:
Bei jedem Rekursionslevel gilt entweder $x, y \in \Gamma_D(u, M)$ oder es existert ein zusammenhängender Teilgraph G_i mit $x, y \in V_{G^i}$. In diesem Fall sind x und y auch im Teilgraphen, der im nächsten Rekursionslevel betrachtet wird, enthalten.

Lemma 8
Die Ausgabe $\hat{T}$ des Algorithmus *Solve-MMST* ist ein aufspannender Baum des eingebenen Graphen $\hat{G}$.

Beweis:
Zunächst wollen wir zeigen, dass $\hat{T}$ zusammenhängend ist. Sei die Kante $(x, y) \in E_{\hat{G}}$. Nach Lemma 7 gehören x und y an einer Stelle der Prozedur *Rec-Cons-Tree* der Menge $\Gamma_D(u, M)$ an. Nach Korollar 6 gilt, dass wenn die Prozedur *Clst-Dijk* terminiert, das neue Cluster U' sowohl x, als auch y enthält. Lemma 3 besagt, terminiert *Clst-Dijk*, dann enthält der Graph $\hat{T}$ einen Pfad zwischen x und y. Daraus folgt, dass für jede Kante $(x, y) \in E_{\hat{G}}$ der Graph $\hat{T}$ einen Pfad zwischen x und y enthält. Somit ist $\hat{T}$ zusammenhängend, sofern $\hat{G}$ zusammenhängend ist.
Nun müssen wir zeigen, dass $\hat{T}$ zudem keinen Kreis enthält. Wir nehmen an, $\hat{T}$ enthalte den Kreis $(v_1, v_2, \ldots, v_k, v_1)$. Sei o.B.d.A. die Kante (v_k, v_1) die letzte dieser Kanten, die zu $\hat{T}$ hinzugefügt wurden. Die Knoten v_k und v_1 ge-

hören verschiedenen Clustern an, da nur so die Prozedur *Clst-Dijk* die Kante (v_k, v_1) zu $\hat{T}$ hinzufügt. Nach Lemma 3 existert kein Pfad zwischen v_k und v_1, wenn sie nicht dem gleichen Cluster angehören. Das ist ein Widerspruch zu unserer Annahme, dass der Pfad $(v_1, \ldots, v_k)$ bereits existiert. Das bedeutet, die Annahme ist falsch und die Lösung $\hat{T}$ enthält keinen Kreis und ist somit ein aufspannender Baum von $\hat{G}$.

Sei im Folgenden für eine Menge von Cluster $C = \{U_1, \ldots, U_k\}$

$$\varphi(C) = \sum_{U_i \in C} diam_{\hat{T}}(U_i),$$

wobei der Diameter ($diam$) eines Clusters die Größe des längsten Pfades in einem Cluster über den betrachteten Graph angibt. Das heißt, es gilt:

$$diam_{\hat{T}}(U) = \max_{u,v \in U} \{dist_{\hat{T}}(u, v)\}$$

Lemma 9
Nach Beendigung der Prozedur $Clst - Dijk$ gilt:

$$diam_{\hat{T}}(U') \leq 2D + \varphi(\mathcal{I}(P, U'))$$

Beweis:
Die Prozedur $Clst - Dijk$ stoppt beim Abstand D von der Quelle u. Das bedeutet, der von T_{local} über die Cluster von $\mathcal{I}(P, U')$ induzierte Baum hat höchstens eine Tiefe von D, wobei der Knoten u als Wurzel angesehen wird. Betrachte nun zwei Knoten x und y in $\hat{T}$ und sei π der eindeutige Pfad zwischen diesen Knoten. Dann beinhaltet π höchstens $2D$ Kanten von T_{local}. Der Rest des Pfades besteht aus Kanten, die innerhalb der Cluster, die von U' verdrängt wurden, liegen. Die Anzahl dieser ist damit höchstens $\varphi(\mathcal{I}(P, U'))$. Damit gilt: $diam_{\hat{T}}(U') \leq 2D + \varphi(\mathcal{I}(P, U'))$

Lemma 10
Für jedes Cluster $U_i \in P' \backslash U'$ existiert ein Cluster $\overline{U_i}$ mit $U_i \subseteq \overline{U_i}$ und $\overline{U_i} \cap U' = \emptyset$.

Lemma 11

Für jedes zusammenhängende G^i, enstanden während der Prozedur *Rec-Cons-Tree* im Rekursionslevel l, gilt:
$\varphi(\mathcal{I}(P', V_{G^i})) \leq 2(l+1)D$

Beweis:

Beweis durch Induktion nach l:
Im Rekursionslevel 0 enthält die Partition P' nur ein Cluster, das aus mehr als einem Knoten besteht, nämlich U'. Der Diameter dieses Clusters beträgt $2D$. Der Diameter eines Clusters, das nur aus einem einzigen Knoten besteht ist 0. Das bedeutet, die Summe der Diameter aller Cluster die V_{G^i} schneiden beträgt $2D$. Die Aussage stimmt demnach für $l = 0$.
Nehmen wir nun an, die Aussage ist bewiesen für alle Rekursionslevel kleiner l und betrachten ein G^i im Rekursionslevel l. Nach Lemma 10 existiert für jedes Cluster $U \in P' \backslash \{U'\}$ ein Cluster $\bar{U} \in \mathcal{I}(P'_{l-1}, V_{G^j_{l-1}}) \backslash \mathcal{I}(P'_{l-1}, U')$ mit $U \subseteq \bar{U}$. Damit gibt es auch für jedes $U \in \mathcal{I}(P', V_{G^i}) \backslash \{U'\}$ ein $\bar{U} \in \mathcal{I}(P'_{l-1}, V_{G^j_{l-1}}) \backslash \mathcal{I}(P'_{l-1}, U')$ mit $U \subseteq \bar{U}$. Da $diam_{\hat{T}}(U) \leq diam_{\hat{T}}(\bar{U})$ für $U \subseteq \bar{U}$ gilt, folgt daraus:

$$\begin{gathered}\varphi(\mathcal{I}(P', V_{G^i})) - diam_{\hat{T}}(U') \leq \\ \varphi(\mathcal{I}(P'_{l-1}, V_{G^j_{l-1}}) \backslash \mathcal{I}(P'_{l-1}, U')) = \\ \varphi(\mathcal{I}(P'_{l-1}, V_{G^j_{l-1}})) - \varphi(\mathcal{I}(P'_{l-1}, U'))\end{gathered}$$

Damit gilt:

$$\varphi(\mathcal{I}(P', V_{G^i})) \leq diam_{\hat{T}}(U') + \varphi(\mathcal{I}(P'_{l-1}, V_{G^j_{l-1}})) - \varphi(\mathcal{I}(P'_{l-1}, U'))$$

Da jedes Cluster in P eine Teilmenge eines Clusters in P'_{l-1} ist, folgt:

$$\varphi(\mathcal{I}(P, U')) \leq \varphi(\mathcal{I}(P'_{l-1}, U'))$$

und damit:

$$\varphi(\mathcal{I}(P', V_{G^i})) \leq diam_{\hat{T}}(U') - \varphi(\mathcal{I}(P, U')) + \varphi(\mathcal{I}(P'_{l-1}, V_{G^j_{l-1}}))$$

Nach Anwendung von Lemma 9 haben wir:

$$\varphi(\mathcal{I}(P', V_{G^i})) \leq 2D + \varphi(\mathcal{I}(P'_{l-1}, V_{G^j_{l-1}}))$$

Mit Hilfe der Induktionsannahme $\varphi(\mathcal{I}(P'_{l-1}, V_{G^j_{l-1}})) \leq 2lD$ gilt letztendlich:

$$\varphi(\mathcal{I}(P', V_{G^i})) \leq 2D + 2lD = 2(l+1)D$$

Beobachtung 12

Für jedes zusammenhängende G^i, enstanden während der Prozedur *Rec-Cons-Tree* im Rekursionslevel l, gilt:
$|V_{G^i}| \leq n/2^{l+1}$

Beobachtung 13

Während der gesamten Ausübung der Prozedur *Rec-Cons-Tree* gilt für jedes Rekursionslevel l: $l \leq \lceil \log n \rceil$

Theorem 14

Seien $x, y \in V_{\hat{G}}$ und $(x, y) \in E_{\hat{G}}$. Dann gilt:
$dist_{\hat{T}}(x, y) \leq 2\lceil \log n \rceil D$

Beweis:
Nach Lemma 7 gibt es eine Stelle während der Prozedur *Rec-Cons-Tree* bei der $x, y \in \Gamma_D(u, M)$ gilt. Sei das zugehörige Rekursionslevel l. Mit Korollar 6 gilt zudem $x, y \in U'$. Betrachte nun einen Teilgraphen G^i dieses Rekursionslevel. Beobachtung 2 und Korollar 6 sagt uns, dass $V_{G^i} \cap U' \neq \emptyset$ und damit ist $U' \in \mathcal{I}(P', V_{G^i})$. Zudem gilt

$$dist_{\hat{T}}(x, y) \leq diam_{\hat{T}}(U') \leq \varphi(\mathcal{I}(P', V_{G^i}))$$

Nach Lemma 11 und Korollar 7 gilt

$$\varphi(\mathcal{I}(P', V_{G^i})) \leq 2(l+1)D \leq 2\lceil \log n \rceil D.$$

Damit gilt letztendlich $dist_{\hat{T}}(x, y) \leq 2\lceil \log n \rceil D$.

Korollar 15

Für die Ausgabe $\hat{T}$ gilt:
$MaxStr(\hat{T}) \leq 2\lceil \log n \rceil D$

5 Implementierung

In diesem Kaptitel betrachten wir einige Implementierungen. Die Methode *MaxStretch* errechnet mit Hilfe der Methode *Strecke*, die wiederum auf *BFS* zurückgreift, den maximalen Stretch eines Baumes. *DCenter* findet durch Löschen (Methode *delete*) der Kanten innerhalb der *D-Nachbarschaft* ein *D-Center*. Die *D-Nachbarschaft* eines Knoten wird wiederrum durch die Methode *DNachbarn* bestimmt.

5.1 Berechnung des Max-Stretch

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Map;

public class Bachelorarbeit {

        //Das Graph-Interface stellt Knoten (Vertex)
        //als Strings dar.
        //Für jeden Knoten wird eine Collection
        //aller Knoten gespeichert, zu denen
        //von diesem Knoten aus eine Kante führt.
        interface Graph extends Map<String, Collection<String>> {
                public void addEdge(String i, String j);
                public void addVertex(String i);
        }


//Eine Methode zur Erzeugung eines leeren
//ungerichteten Graphen. Dieser Graph
//enthält weder Knoten noch Kanten.
public static Graph emptyGraph() {
        class MyGraph
                extends HashMap<String, Collection<String>>
                implements Graph {
                static final long serialVersionUID = 1234567;
```

```
                //Erzeugt einen Knoten,
                //sofern er noch nicht existiert
                public void addVertex(String i) {
                        if (!containsKey(i))
                                put(i, new ArrayList<String>());
                }
                //Erzeugt eine Hin- und Rückkante
                //zwischen zwei Knoten,
                //sofern sie noch nicht existieren
                //und ggf. die Knoten dazu.
                public void addEdge(String i, String j) {
                                addVertex(i);
                                addVertex(j);
                        if(!get(i).contains(j)){
                                get(i).add(j);
                                get(j).add(i);
                }
                }
        }

        return new MyGraph();
}

//Gibt, ausgehend von der Quelle, einen Baum
//mit kürzesten Wegen aus
//Die Ausgabe ist (Knoten, Vorgänger).
public static Map<String, String> BFS(Graph g, String quelle) {

        Map<String, String> result = new HashMap<String, String>();

        //Liste der Knoten, dessen Nachfolger
        //getestet werden müssen
        LinkedList<String> temp = new LinkedList<String>();
        //Liste der schon besuchten Knoten
        HashSet<String> visited = new HashSet<String>();
        //Füge Quelle zu result und temp hinzu.
        result.put(quelle, null);
        temp.add(quelle);

        //solange temp nicht leer ist
        while (!temp.isEmpty()){
```

```
                //Setze Nachfolger auf parent.
                String parent = temp.remove(0);
                //Füge parent zu visited hinzu.
                visited.add(parent);
                //Bestimme alle Nachfolger von parent.
                Collection<String> child = g.get(parent);

                //Betrachte alle Nachfolger.
                for (String it: child){
                        //Falls Nachfolger noch nicht besucht,
                        //füge ihn zu visited hinzu
                        if (!visited.contains(it)){
                        temp.add(it);}
                        //Falls Nachfolger noch nicht in result,
                        //füge ihn mit parent als Vorgänger hinzu
                        if(!result.containsKey(it)){
                        result.put(it, parent);}
        }
        }

        return result;
}


//Gibt die kürzeste Strecke von der Senke zur Quelle aus.
public static LinkedList<String>
Strecke(Graph g, String quelle, String senke) {

        LinkedList<String> result = new LinkedList<String>();
        LinkedList<String> temp = new LinkedList<String>();

        //Erstellt mit BFS (Dijkstra) einen Baum
        //mit kürzesten Wege.
        Map<String, String> Wege = BFS(g, quelle);

        //Füge Senke zu temp und result hinzu.
        if (Wege.containsKey(senke)){
                temp.add(senke);
                result.add(senke);
```

```
                //Solange temp nicht leer ist,
                //ziehe erstes Element heraus
                while (!temp.isEmpty()) {
                        String Nachfolger = temp.remove(0);

                        //Falls der betrachtete Knoten
                        //einen Vorgänger hat,
                        //füge ihn zu temp und result hinzu.
                        if (Wege.get(Nachfolger) != null){
                                String Vorgaenger =
                                 Wege.get(Nachfolger);
                                temp.add(Vorgaenger);
                                result.add(Vorgaenger);
                        }

                }
        }

        return result;
}


//Methode zum Errechnen des maximalen Stretch eines Baumes
public static int MaxStretch(Graph g, Graph t){

        int Stretch = 0;

        //Collection aller im Graph vorkommenden Knoten
        Collection<String> Knoten = g.keySet();

        //Betrachte jedes mögliche Paar zweier Knoten.
        for (String Knoten1 : Knoten){
                for (String Knoten2 : Knoten){
                        //sofern sie unterschiedlich sind
                        if (Knoten1 != Knoten2){

                                //Berechne kürzesten Weg
                                //im Graph und im Baum.
                                LinkedList<String> gWay =
                                        Strecke(g, Knoten1, Knoten2);
                                LinkedList<String> tWay =
```

```
                                        Strecke(t, Knoten1, Knoten2);
                                //Berechne Länge der Wege.
                                int gLaenge = gWay.size() -1;
                                int tLaenge = tWay.size() -1;
                                //Berechne maximalen Stretch.
                                if ((tLaenge/gLaenge) > Stretch){
                                        Stretch = (tLaenge/gLaenge);
                                }
                        }
                }
        }

        return Stretch;
}
```

5.2 Finden eines D-Centers

```
//Berechnet die Menge der Knoten in der D-Umgebung.
public static Collection<String>
DNachbarn(Graph g, String u, int D){

        LinkedList<String> temp =
                new LinkedList<String>();
        Map<String, Integer> visited =
                new HashMap<String, Integer>();

        //Füge Ausgangsknoten zu temp hinzu.
        temp.add(u);
        //Füge Ausgangsknoten mit Entfernung
        //zu visited hinzu.
        visited.put(u, 0);

        //Solange temp nicht leer ist
        while(!temp.isEmpty()){
                String parent = temp.remove(0);

                //falls Nachfolgerknoten
                //in D-Umgebung liegen
```

```
                if (visited.get(parent) < D){
                        Collection<String> child = g.get(parent);

                        for (String it: child){
                                //und noch nicht betrachtet wurden
                                if (!visited.containsKey(it)){
                                //Füge sie zu temp und
                                //inclusive Abstand zu u
                                //zu visited hinzu.
                                temp.add(it);
                                visited.put
                                (it, visited.get(parent)+1);}
                        }
        }
        }

        Collection<String> result = visited.keySet();

        return result;
}


//Methode zum Löschen von Kanten
public static Graph
delete(Graph g, Collection<String> Knoten){
        //Erstelle leeren Graph und Collection
        //aller vorkommenden Knoten.
        Graph neu = emptyGraph();
        Collection<String> temp = g.keySet();
        //Betrachte jeden Knoten.
        for (String it1: temp){
                //Erstelle Knoten im neuen Graph.
                neu.addVertex(it1);
                //Collection der
                //ursprünglichen Nachbarn
                Collection<String> nachbarn = g.get(it1);

                for (String it2: nachbarn){
                        //Falls die Kante keine der
                        //zu löschenden Kante ist,
                        if (!Knoten.contains(it1)
```

```
                                        || !Knoten.contains(it2)){
                                    //erstelle Kante
                                    //im neuen Graph.
                                    neu.addEdge(it1, it2);
                                }
                            }
                        }

                        return neu;
}


//Methode zum Finden eines D-Centers
//mit möglichst kleinem D
public static Map<String, Integer> DCenter(Graph g){

                        Map<String, Integer> result =
                                new HashMap<String, Integer>();
                        //Setze D auf 0 und
                        //berechne Anzahl der Knoten.
                        int D = 0;
                        int n = g.size();
                        //Collection aller Knoten
                        Collection<String> Knoten = g.keySet();

                        while (D < n){
                                //Erhöhe D nach und nach.
                                D = D + 1;

                                //Betrachte jeden Knoten.
                                for (String it: Knoten){
                                        //Erstelle neuen Graph
                                        //durch Löschen der Kanten
                                        //innerhalb der D-Umgebung.
                                        Collection<String> DNach =
                                                DNachbarn(g, it, D);
                                        Graph geloescht =
                                                delete(g, DNach);

                                        int temp = 0;
                                        //Betrachte für jeden Knoten
```

```
                                //mit Hilfe von BFS
                                //die zusammenhängenden
                                //Teilgraphen.
                                for (String it2: Knoten){
                                        Map<String, String> G =
                                                BFS(geloescht, it2);
                                        //Falls die Bedingung gilt,
                                        //erhöhe temp um 1.
                                        if (G.size() <= n/2){
                                                temp = temp + 1;
                                        }
                                }
                                //temp=n falls obige Bedingung
                                //für jeden Knoten gilt.
                                if (temp == n){
                                        //Gib D-Center und D aus
                                        result.put(it, D);
                                        return result;
                                }
                        }
                        }
                return null;
}
```

Nun lassen wir uns für den Graph in Kapitel 2 und der von uns errechneten Lösung den maximalen Stretch ausgeben. Zudem berechnen wir noch ein mögliches *D-Center* inklusive D.

System.out.println(Beispiel);
$\Rightarrow \{17 = [16, 18], 18 = [15, 17], 15 = [14, 16, 18], 16 = [15, 17], 13 = [4, 12, 14],$
$14 = [11, 13, 15], 11 = [10, 12, 14], 12 = [5, 11, 13], 3 = [1, 4], 2 = [1, 4],$
$1 = [2, 3], 10 = [5, 6, 9, 11], 7 = [6, 8], 6 = [5, 7, 10], 5 = [4, 6, 10, 12],$
$4 = [2, 3, 5, 13], 9 = [8, 10], 8 = [7, 9]\}$

System.out.println(Baum);
$\Rightarrow \{17 = [16], 18 = [15], 15 = [14, 16, 18], 16 = [15, 17], 13 = [12],$
$14 = [11, 15], 11 = [12, 14], 12 = [5, 11, 13], 3 = [1, 4], 2 = [1], 1 = [2, 3],$
$10 = [5, 9], 7 = [6, 8], 6 = [5, 7], 5 = [4, 6, 10, 12], 4 = [3, 5], 9 = [10], 8 = [7]\}$

System.out.println(MaxStretch(Beispiel, Baum));
$\Rightarrow 5$

System.out.println(DCenter(Beispiel));
$\Rightarrow \{13 = 2\}$

Der von unserem Code ausgegebenen *D-Center* ist der Knoten 13. Der Unterschied zu der manuell errechneten Lösung liegt allein an der Reihenfolge in der die Methode die Knoten auf ihre *D-Center-Eigenschaft* prüft. Der zugehörige Wert D ist genau wie bei dem per Hand durchgerechnetem Beispiel 2.
Der *maximale Stretch* dieses Baumes beträgt 5.

6 Fazit

Der Algorithmus liefert mit Hilfe eines abgewandelten Dijkstra-Prozesses einen aufspannenden Baum. Der *Max-Stretch* dieses Baumes ist stets kleiner als $2\lceil \log n \rceil D$.

7 Literaturverzeichnis

- Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs
- Emek, Y./Peleg, D.: Approximating Minimum Max-Stretch Spanning Trees on Unweighted Graphs
- Peleg, D.: Low Stretch Spanning Trees
- Peleg, D./Tendler, D.: Low stretch spanning trees for planar graphs

8 Anhang

Anwendung des Algorithmus auf den Beispiel-Graph G:

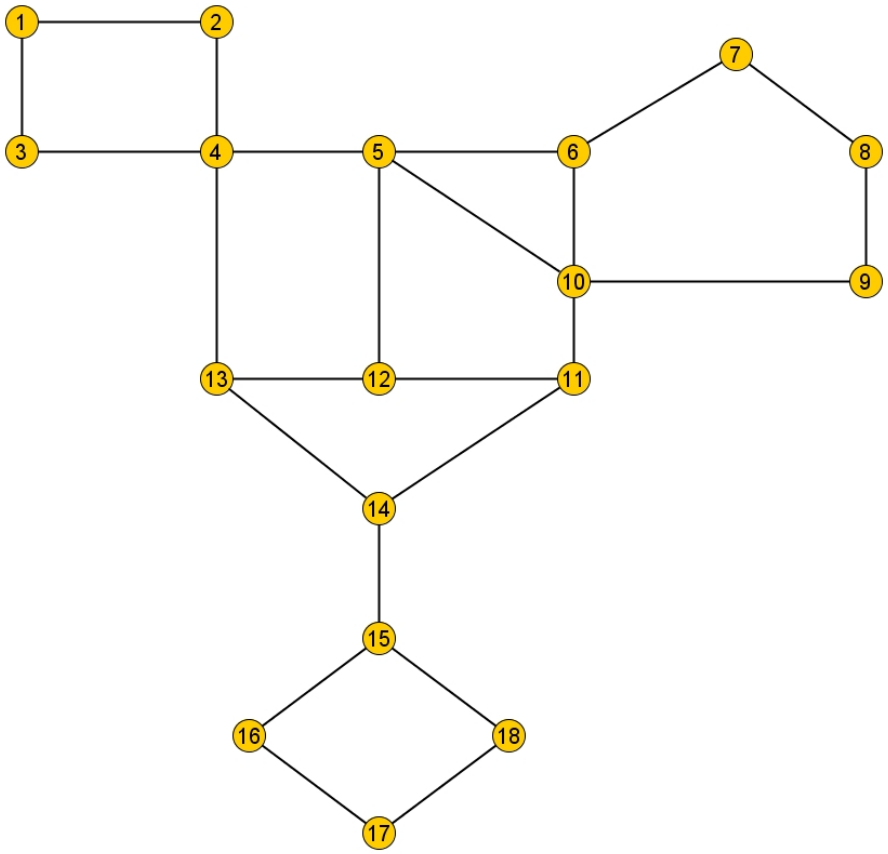

Abbildung 6: Beispielgraph G

Suche *D-Center* und betrachte als Kandidat den Knoten 12:

$V_{G^1} = \{1, 2, 3, 4\}$
$V_{G^2} = \{6, 7, 8, 9, 10\}$
$V_{G^3} = \{14, 15, 16, 17, 18\}$
$V_{G^4} = \{5\}$
$V_{G^5} = \{11\}$
$V_{G^6} = \{13\}$
$V_{G^7} = \{12\}$

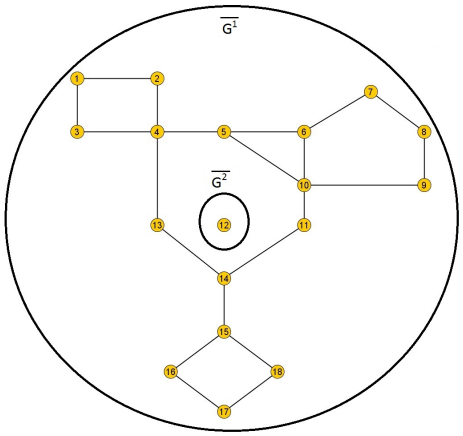

Abbildung 7: $E(G)\backslash\mathcal{E}(\Gamma_1(12, M))$

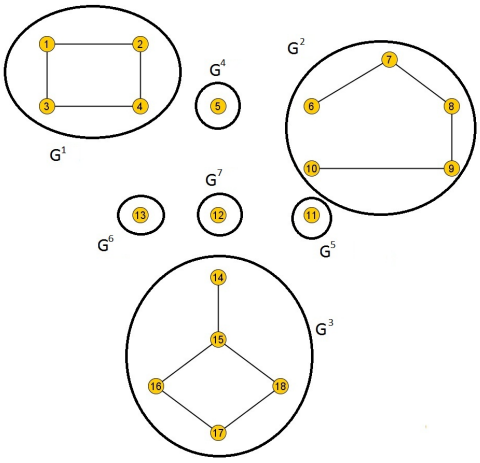

Abbildung 8: $E(G)\backslash\mathcal{E}(\Gamma_2(12, M))$

Für $D = 2$ gilt $V_{G^i} \leq n/2$.
$\Rightarrow$ Knoten 12 ist ein D-*Center* für $D = 2$.

$P = \{U_1, \ldots, U_{18}\}$
$U_1 = \{1\}$
$U_2 = \{2\}$
$U_3 = \{3\}$
$U_4 = \{4\}$
$U_5 = \{5\}$
$U_6 = \{6\}$
$U_7 = \{7\}$
$U_8 = \{8\}$
$U_9 = \{9\}$
$U_{10} = \{10\}$
$U_{11} = \{11\}$
$U_{12} = \{12\}$
$U_{13} = \{13\}$
$U_{14} = \{14\}$
$U_{15} = \{15\}$
$U_{16} = \{16\}$
$U_{17} = \{17\}$
$U_{18} = \{18\}$

$Clst - Dijk(G, M, P, 12, 2)$

Der zugehörige *cliqued-cluster-graph* $CC_M(G, P)$:

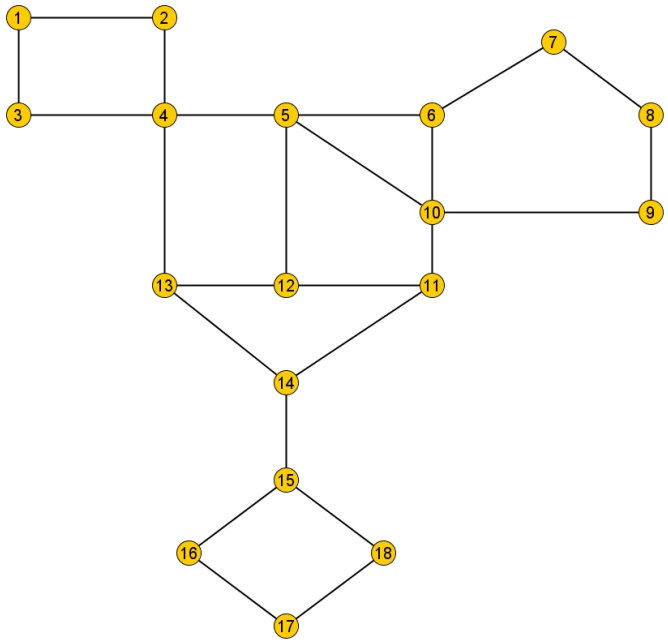

Abbildung 9: $CC_M(G, P)$

$\Gamma_2(12, M) = \{4, 5, 6, 10, 11, 12, 13, 14\}$
$U' = \{\}$ $\quad T_{local} = \{\}$

Führe den abgewandelten Dijkstra-Algorithmus durch:

Betrachte Knoten 12:
$12 \notin U' \wedge 12 \in \Gamma_2(12, M)$
$\Rightarrow U' = \{12\}$

Betrachte Knoten 5:
$5 \notin U' \wedge 5 \in \Gamma_2(12, M)$
$\Rightarrow U' = \{5, 12\}$
$\Rightarrow T_{local} = \{(5, 12)\}$

Betrachte Knoten 11:
$11 \notin U' \wedge 11 \in \Gamma_2(12, M)$
$\Rightarrow U' = \{5, 11, 12\}$
$\Rightarrow T_{local} = \{(5, 12), (11, 12)\}$

Betrachte Knoten 13:
$13 \notin U' \wedge 13 \in \Gamma_2(12, M)$
$\Rightarrow U' = \{5, 11, 12, 13\}$
$\Rightarrow T_{local} = \{(5, 12), (11, 12), (12, 13)\}$

Betrachte Knoten 4:
$4 \notin U' \wedge 4 \in \Gamma_2(12, M)$
$\Rightarrow U' = \{4, 5, 11, 12, 13\}$
$\Rightarrow T_{local} = \{(4, 5), (5, 12), (11, 12), (12, 13)\}$

Betrachte Knoten 6:
$6 \notin U' \wedge 6 \in \Gamma_2(12, M)$
$\Rightarrow U' = \{4, 5, 6, 11, 12, 13\}$
$\Rightarrow T_{local} = \{(4, 5), (5, 6), (5, 12), (11, 12), (12, 13)\}$

Betrachte Knoten 10:
$10 \notin U' \wedge 10 \in \Gamma_2(12, M)$
$\Rightarrow U' = \{4, 5, 6, 10, 11, 12, 13\}$
$\Rightarrow T_{local} = \{(4, 5), (5, 6), (5, 10), (5, 12), (11, 12), (12, 13)\}$

Betrachte Knoten 14:
$14 \notin U' \wedge 14 \in \Gamma_2(12, M)$
$\Rightarrow U' = \{4, 5, 6, 10, 11, 12, 13, 14\}$
$\Rightarrow T_{local} = \{(4, 5), (5, 6), (5, 10), (5, 12), (11, 12), (11, 14), (12, 13)\}$

Alle weiteren Knoten sind nicht Teil von $\Gamma_2(12, M)$.

$P' = (P \cup \{U'\}) \backslash \mathcal{I}(P, \Gamma_2(12, M))$
$= \{U_1, \ldots, U_{18}, U'\} \backslash \{U_4, U_5, U_6, U_{10}, U_{11}, U_{12}, U_{13}, U_{14}\}$
$= \{U_1, U_2, U_3, U_7, U_8, U_9, U_{15}, U_{16}, U_{17}, U_{18}, U'\}$

$P_i = \{U \cap V_{G^i} | U \in P'\}$

$P_1 = \{\{1\}, \{2\}, \{3\}, \{4\}\}$
$P_2 = \{\{7\}, \{8\}, \{9\}, \{6, 10\}\}$
$P_3 = \{\{14\}, \{15\}, \{16\}, \{17\}, \{18\}\}$
$P_4 = \{\{5\}\}$
$P_5 = \{\{11\}\}$
$P_6 = \{\{13\}\}$
$P_7 = \{\{12\}\}$

Da die Partitionen $P_4, \ldots, P_7$ jeweils nur aus einem Knoten bestehen, gibt der Algorithmus im nächsten Rekursionslevel die leere Menge zurück.

$Rec - Cons - Tree(G(V_{G^1}), M(V_{G^1}), P_1, 2, 1)$

Suche *D-Center* und betrachte als Kandidat den Knoten 1:

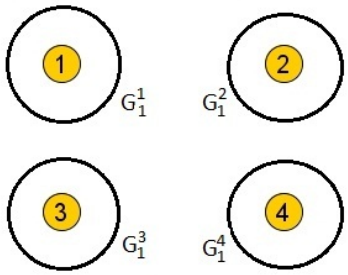

Abbildung 10: $G(V_{G^1}) \backslash \mathcal{E}(\Gamma_2(1, M(V_{G^1}))$

$V_{G_1^1} = \{1\}$
$V_{G_1^2} = \{2\}$
$V_{G_1^3} = \{3\}$
$V_{G_1^4} = \{4\}$

Für $D = 2$ gilt $V_{G_1^i} \leq n/2$.
$\Rightarrow$ Knoten 1 ist ein *D-Center* für $D = 2$.

$Clst - Dijk(G(V_{G^1}), M(V_{G^1}), P_1, 1, 2)$
Der zugehörige *cliqued-cluster-graph* $CC_{M(G^1)}(G^1, P_1)$:

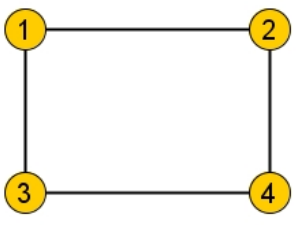

Abbildung 11: $CC_{M(G^1)}(G^1, P_1)$

$\Gamma_2(1, M) = \{1, 2, 3, 4\}$
$U_1' = \{\}$ $T_1 = \{\}$

Führe den abgewandelten Dijkstra-Algorithmus durch:

Betrachte Knoten 1:
$1 \notin U' \wedge 1 \in \Gamma_2(1, M)$
$\Rightarrow U' = \{1\}$

Betrachte Knoten 2:
$2 \notin U' \wedge 2 \in \Gamma_2(1, M)$
$\Rightarrow U' = \{1, 2\}$
$\Rightarrow T_1 = \{(1, 2)\}$

Betrachte Knoten 3:
$3 \notin U' \wedge 3 \in \Gamma_2(1, M)$
$\Rightarrow U' = \{1, 2, 3\}$
$\Rightarrow T_1 = \{(1, 2), (1, 3)\}$

Betrachte Knoten 4:
$3 \notin U' \wedge 4 \in \Gamma_2(1, M)$
$\Rightarrow U' = \{1, 2, 3, 4\}$
$\Rightarrow T_1 = \{(1, 2), (1, 3), (2, 4)\}$

$P_1' = (P_1 \cup \{U_1'\}) \backslash \mathcal{I}(P_1, \Gamma_2(1, M))$
$= \{\{1\}, \{2\}, \{3\}, \{4\}, \{1, 2, 3, 4\}\} \backslash \{\{1\}, \{2\}, \{3\}, \{4\}\}$
$= \{\{1, 2, 3, 4\}\}$

$P_{1,i} = \{U \cap V_{G_1^i} | U \in P_1'\}$

$P_{1,1} = \{\{1\}\}$
$P_{1,2} = \{\{2\}\}$
$P_{1,3} = \{\{3\}\}$
$P_{1,4} = \{\{4\}\}$

Da die Partitionen jeweils nur aus einem Knoten bestehen, gibt der Algorithmus im nächsten Rekursionslevel die leere Menge zurück.

$Rec - Cons - Tree(G(V_{G^2}), M(V_{G^2}), P_2, 2, 1)$

Suche *D-Center* und betrachte als Kandidat den Knoten 6:

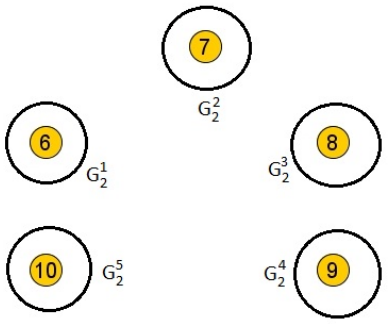

Abbildung 12: $G(V_{G^2})\backslash\mathcal{E}(\Gamma_2(6, M(V_{G^2}))$

$V_{G_2^1} = \{6\}$
$V_{G_2^2} = \{7\}$
$V_{G_2^3} = \{8\}$
$V_{G_2^4} = \{9\}$
$V_{G_2^5} = \{10\}$

Für $D = 2$ gilt $V_{G_2^i} \leq n/2$.
$\Rightarrow$ Knoten 6 ist ein *D-Center* für $D = 2$.

$Clst - Dijk(G(V_{G^2}), M(V_{G^2}), P_2, 6, 2)$
Der zugehörige *cliqued-cluster-graph* $CC_{M(G^2)}(G^2, P_2)$:

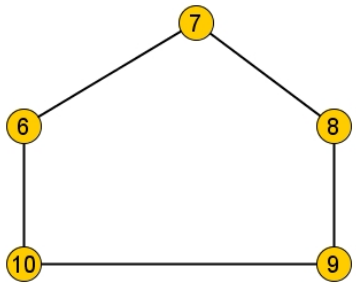

Abbildung 13: $CC_{M(G^2)}(G^2, P_2)$

$\Gamma_2(6, M) = \{6, 7, 8, 9, 10\}$
$U_2' = \{\}$ $T_2 = \{\}$

Führe den abgewandelten Dijkstra-Algorithmus durch:

Betrachte Knoten 6:
$6 \notin U_2' \wedge 6 \in \Gamma_2(6, M)$
$\Rightarrow U_2' = \{6, 10\}$

Betrachte Knoten 7:
$7 \notin U_2' \wedge 7 \in \Gamma_2(6, M)$
$\Rightarrow U_2' = \{6, 7, 10\}$
$\Rightarrow T_2 = \{(6, 7)\}$

Betrachte Knoten 10:
$10 \in U_2'$
$\Rightarrow$ Kante nicht zu T_2 aufnehmen.

Betrachte Knoten 8:
$8 \notin U_2' \wedge 8 \in \Gamma_2(6, M)$
$\Rightarrow U_2' = \{6, 7, 8, 10\}$
$\Rightarrow T_2 = \{(6, 7), (7, 8)\}$

Betrachte Knoten 7:
$9 \notin U_2' \wedge 9 \in \Gamma_2(6, M)$
$\Rightarrow U_2' = \{6, 7, 8, 9, 10\}$
$\Rightarrow T_2 = \{(6, 7), (7, 8), (9, 10)\}$

$P_2' = (P_2 \cup \{U_2'\}) \backslash \mathcal{I}(P_2, \Gamma_2(6, M))$
$= \{\{6, 10\}, \{7\}, \{8\}, \{9\}, \{6, 7, 8, 9, 10\}\} \backslash \{\{6, 10\}, \{7\}, \{8\}, \{9\}\}$
$= \{\{6, 7, 8, 9, 10\}\}$

$P_{2,i} = \{U \cap V_{G_2^i} | U \in P_2'\}$

$P_{2,1} = \{\{6\}\}$
$P_{2,2} = \{\{7\}\}$
$P_{2,3} = \{\{8\}\}$
$P_{2,4} = \{\{9\}\}$
$P_{2,5} = \{\{10\}\}$

Da die Partitionen jeweils nur aus einem Knoten bestehen, gibt der Algorithmus im nächsten Rekursionslevel die leere Menge zurück.

$Rec - Cons - Tree(G(V_{G^3}), M(V_{G^3}), P_3, 2, 1)$

Suche *D-Center* und betrachte als Kandidat den Knoten 15:

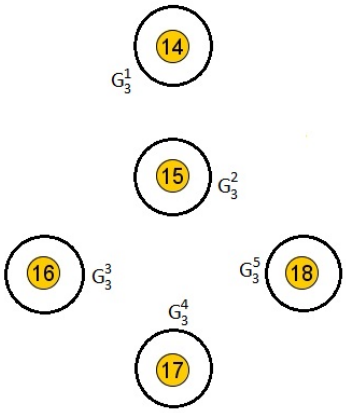

Abbildung 14: $G(V_{G^3})\backslash\mathcal{E}(\Gamma_2(15, M(V_{G^3}))$

$V_{G_3^1} = \{14\}$
$V_{G_3^2} = \{15\}$
$V_{G_3^3} = \{16\}$
$V_{G_3^4} = \{17\}$
$V_{G_3^5} = \{18\}$

Für $D = 2$ gilt $V_{G_3^i} \leq n/2$.
$\Rightarrow$ Knoten 15 ist ein*D-Center* für $D = 2$.

$Clst - Dijk(G(V_{G^3}), M(V_{G^3}), P_3, 15, 2)$

Der zugehörige *cliqued-cluster-graph* $CC_{M(G^3)}(G^3, P_3)$:

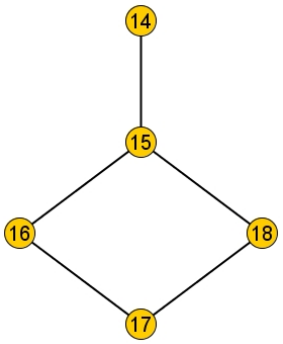

Abbildung 15: $CC_{M(G^3)}(G^3, P_3)$

$\Gamma_2(15, M) = \{14, 15, 16, 17, 18\}$
$U_2' = \{\}$ $T_2 = \{\}$

Führe den abgewandelten Dijkstra-Algorithmus durch:

Betrachte Knoten 15:
$15 \notin U_3' \wedge 15 \in \Gamma_2(15, M)$
$\Rightarrow U_3' = \{15\}$

Betrachte Knoten 14:
$14 \notin U_3' \wedge 14 \in \Gamma_2(15, M)$
$\Rightarrow U_3' = \{14, 15\}$
$\Rightarrow T_2 = \{(14, 15)\}$

Betrachte Knoten 16:
$16 \notin U_3' \wedge 16 \in \Gamma_2(15, M)$
$\Rightarrow U_3' = \{14, 15, 16\}$
$\Rightarrow T_2 = \{(14, 15), (15, 16)\}$

Betrachte Knoten 18:
$18 \notin U_3' \wedge 18 \in \Gamma_2(15, M)$
$\Rightarrow U_3' = \{14, 15, 16, 18\}$
$\Rightarrow T_2 = \{(14, 15), (15, 16), (15, 18)\}$

Betrachte Knoten 17:
$17 \notin U_3' \wedge 17 \in \Gamma_2(15, M)$
$\Rightarrow U_3' = \{14, 15, 16, 17, 18\}$
$\Rightarrow T_2 = \{(14, 15), (15, 16), (15, 18), (16, 17)\}$

$P_3' = (P_3 \cup \{U_3'\}) \backslash \mathcal{I}(P_3, \Gamma_2(15, M))$
$= \{\{14\}, \{15\}, \{16\}, \{17\}, \{18\}, \{14, 15, 16, 17, 18\}\} \backslash \{\{14\}, \{15\}, \{16\}, \{17\}, \{18\}\}$
$= \{\{14, 15, 16, 17, 18\}\}$

$P_{3,i} = \{U \cap V_{G_3^i} | U \in P_3'\}$

$P_{3,1} = \{\{14\}\}$
$P_{3,2} = \{\{15\}\}$
$P_{3,3} = \{\{16\}\}$
$P_{3,4} = \{\{17\}\}$
$P_{3,5} = \{\{18\}\}$

Da die Partitionen jeweils nur aus einem Knoten bestehen, gibt der Algorithmus im nächsten Rekursionslevel die leere Menge zurück.

$T = T_{local} \cup \bigcup_{1 \leq i \leq 7} T_i$

$\Rightarrow T$:

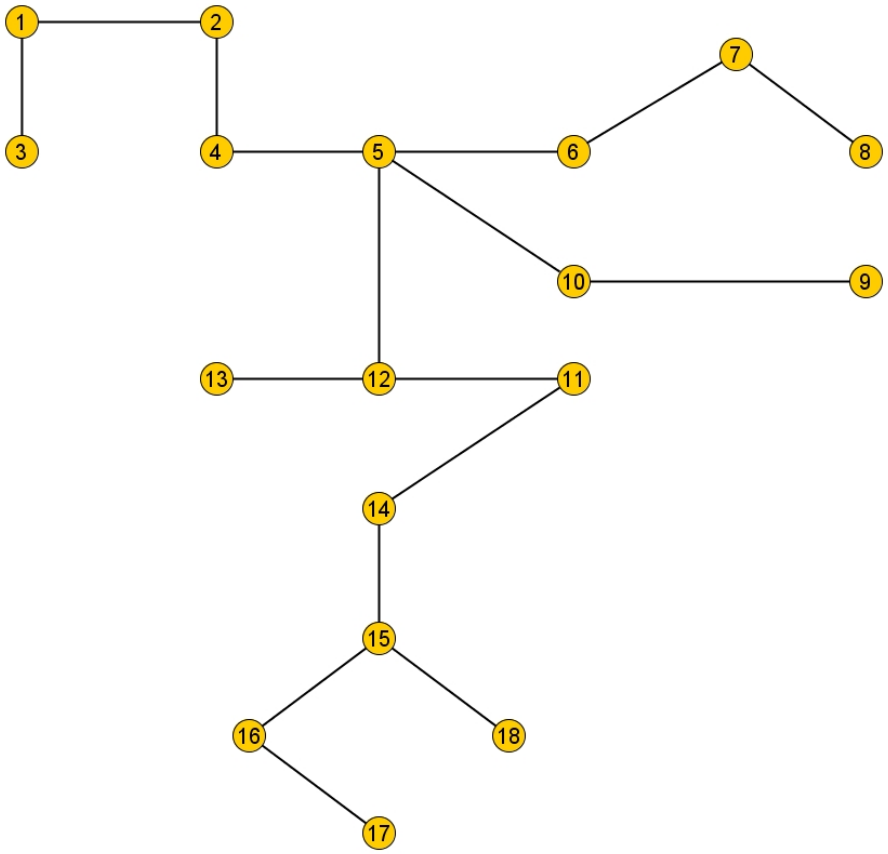

Abbildung 16: Lösung des Algorithmus