Institut für Theoretische Informatik
Lehrstuhl für Algorithmik I

Seminarband

Algorithmentechnik

Wintersemester 2015/16

Lehrstuhl für Algorithmik I
Institut für Theoretische Informatik
Fakultät für Informatik
Karlsruher Institut für Technologie

Vorwort

Im Rahmen des Seminars *Algorithmentechnik* werden ausgewählte aktuelle Forschungsergebnisse aus der Algorithmik behandelt. Die einzelnen Themen stammen insbesondere aus den Bereichen Graphenalgorithmen, geometrische Algorithmen, Algorithmen für Sensornetze und Algorithmen zum Graphenzeichnen. Damit vertieft das Seminar einzelne Themen aus dem Spektrum der übrigen am Lehrstuhl angebotenen Vertiefungsvorlesungen. Das Seminar ist Bestandteil der Vertiefungsfächer Algorithmentechnik und Theoretische Grundlagen im Master-Studium. Ziel des Seminars ist es, dass die Teilnehmer lernen, sich in wissenschaftliche Originalarbeiten einzuarbeiten und Literaturrecherche zu betreiben. Insbesondere müssen die Teilnehmer anhand eines Fachvortrages und einer Seminarausarbeitung die von Ihnen bearbeiteten Themen in ansprechender Form präsentieren. Damit werden Fähigkeiten erworben und erweitert, die auch zum Verfassen einer Masterarbeit in der Algorithmik erforderlich sind.

Das Seminar fand im Wintersemester 2015/2016 mit 10 Teilnehmern statt. Der vorliegende Seminarband umfasst die schriftlichen Ausarbeitungen der Teilnehmer.

1 Inhaltsverzeichnis

2 Zufällige, verbundene Graphen mit gegebener Gradsequenz

Maximilian Czerny

Zusammenfassung

Um zufällige, zusammenhängende und einfache Graphen mit gegebener Gradsequenz zu generieren, werden verschiedene Algorithmen dargestellt. Besonderer Fokus liegt auf der Effizienz und Einfachheit der vorgestellten Ansätze. Viger und Latapy [13] haben ein bereits bestehendes heuristisches Verfahren optimiert. Darüber hinaus konnten sie einen neuartigen probabilistischen Zusammenhangstest entwerfen, der in Verbindung mit dem gängigsten Algorithmus zum zufälligen Mischen von Graphen die bislang besten theoretischen Kosten hat. Zudem erreicht dieser in Experimenten je nach zugrunde liegender Verteilung eine um den Faktor 20 schnellere Laufzeit.

2.1 Einführung und Motivation

In Verbindung mit Graphalgorithmen besteht stets das Verlangen nach aussagekräftigen Experimenten zur Validierung von Laufzeitanalysen. Um bestimmte Graphstrukturen mit Millionen von Knoten und Kanten zu erstellen, sind effiziente Algorithmen von Nöten. Dabei sollte der Fokus auf die Einfachheit bei der Implementierung gelegt werden, sodass für Algorithmenentwickler Testgraphen leicht und schnell zu erstellen sind. Nicht nur für experimentelle Analysen von Graphen und Algorithmen sind Verfahren zum zufällig gleichverteilten Generieren von Graphen mit bestimmten Eigenschaften aus einer Menge aller Graphen mit diesen Eigenschaften essentiell. Ebenfalls zur Weiterverarbeitung kann solch ein Graph am Ausgangspunkt der Berechnungen stehen. In den nachfolgenden Abschnitten werden die drei geforderten Eigenschaften einfach, zusammenhängend und die vorgegebene Gradsequenz erfüllend besprochen. Das Ziel ist es, Graphen mit diesen Eigenschaften zufällig zu erzeugen. Selbst wenn man eine konkrete Graphinstanz dieser Graphenklasse hat, kann man sich für zufällige Graphen mit gleichen Eigenschaften interessieren. Beispielsweise können somit konkrete Messwerte eines Graphalgorithmus mit den erwartete Ergebnissen für Graphen mit gleichen Eigenschaften in Perspektive gesetzt werden. Schlauch et al. [11] listen weitere Möglichkeiten des Vergleichs zwischen konkreten Instanzen und zufällig generierten Graphen. Unter anderem wird der Stichproben p–Wert (*sample p value*), definiert als der Anteil ähnlich extremer Ausprägungen eines Wertes in den zufälligen Instanzen verglichen mit den konkreten Messwerten, genannt. Durch solche Stichprobenwerte lässt sich zum Beispiel auch die Qualität von verschiedenen heuristischen Verfahren überprüfen.

Einfache Graphen

Ein Graph heißt *einfach*, wenn er weder Mehrfachkanten zwischen zwei Knoten noch Schlingen (auch Schleifen genannt) aufweist. Im Kontext dieser Ausarbeitung werden nur ungerichtete Graphen betrachtet. Einfache Graphen sind häufig Voraussetzung für bestimmte Algorithmen oder ergeben sich aus der Definition der Topologie. Beispielsweise ist ein Graph, der die Freundschaftsbeziehungen zwischen Kontakten darstellt, einfach. Einerseits ist eine Person nicht mit sich selbst befreundet und entsprechend schlingenfrei und andererseits ergeben sich normale ungerichtete Kanten aus der Symmetrie der Beziehung.

Zusammenhängende Graphen

Ein Graph heißt *zusammenhängend*, wenn es einen Pfad zwischen allen Knotenpaaren des Graphen gibt. Oftmals sind zusammenhängende Graphen inhärent an bestimmte Aufgabenstellungen gebunden. Um beispielsweise die Kommunikation im Internet zu ermöglichen, muss jeder Server (Knoten im Graph) über Leitungen (Kanten im Graph) an das globale Netzwerk angeschlossen sein. Lokale Weiterleitungsentscheidungen können beispielsweise auf einem minimalen Spannbaum (MST) basieren. Ein MST lässt sich entsprechend nur für zusammenhängende Graphen berechnen. Weitere Berechnungen, die nur auf zusammenhängenden Graphen sinnvoll sind, sind unter anderem Routenplanung, Analyse sozialer Netzwerke und topologische Sortierung.

Graphen mit vorgegebener Gradsequenz

Ein Graph $G = (V, E)$ erfüllt eine vorgegebene Gradsequenz $grad[.]$, falls für jeden Knoten $v \in V$ und dessen Knotengrad $d(v) = grad[v]$ gilt. Verschiedene Verteilungen der Knotengrade resultieren in Graphen mit verschiedenen strukturellen Charakteristiken und simulieren somit verschiedene in der Praxis auftauchende Netzwerke oder Strukturen. Zu simulierende Netzwerke sind zum Beispiel das World Wide Web oder Freundschaftsbeziehungen in sozialen Netzwerken. Für eine Erklärung zum Aufbau verschiedener komplexer Netzwerke und deren Eigenschaften sei auf [9, 1] verwiesen. Einige gängige Verteilungen seien an dieser Stelle genannt:

- Gleichverteilung: Jeder Knotengrad ist gleich wahrscheinlich. Eine Gleichverteilung kann in komplexen Netzwerken nicht vorausgesetzt werden.
- Binomialverteilung: Die Vorkommen von Knotengraden häufen sich um einen Mittelwert. Derartige Verteilungen kann man bei Graphen, die geometrische Netzwerke in der Ebene darstellen, beobachten.
- Pareto-Verteilung (oder andere Potenzgesetze): Wenige Knoten haben einen sehr hohen Knotengrad währenddessen die meisten Knoten einen kleinen Grad aufweisen. Verteilungen nach dem Potenzgesetz finden sich in vielen komplexen Netzwerken wie zum Beispiel sozialen Netzwerken wieder.

Aufbau der Ausarbeitung

Diese Ausarbeitung basiert auf den Erkenntnissen von Viger und Latapy (siehe [13]) und fokussiert dabei zunächst die theoretischen Grundlagen von Markov-Ketten und probabilistischen Algorithmen in Kapitel 2.2. Nachfolgend wird in Kapitel 2.3 ein bereits existierender, naiver Algorithmus zur zufälligen Generierung von zusammenhängenden, einfachen Graphen mit gegebener Gradsequenz präsentiert. Kapitel 2.4 und Kapitel 2.5 stellen dafür bekannte effiziente heuristische Methoden und einen neuartigen Zusammenhangstest für den Mischalgorithmus vor. Die verschiedenen Ansätze werden in Kapitel 2.6 theoretisch und experimentell verglichen. Abschließend folgt Kapitel 2.7 mit einem Fazit.

2.2 Vorwissen

Nachfolgend wird zunächst auf die theoretischen Grundlagen von Markov-Ketten in Abschnitt 2.2.1 eingegangen, welche zum Generieren von Stichproben aus der gegebenen Graphenklasse benutzt werden Danach wird die Klasse von Monte-Carlo-Algorithmen in Ab-

schnitt 2.2.2 genauer beschrieben. Für die restliche Ausarbeitung ist $n = |V|$ die Anzahl an Knoten in G und $m = |E|$ analog die Anzahl an Kanten.

2.2.1 Markov-Ketten

Da alle der in den folgenden Kapiteln vorgestellten Algorithmen auf der Theorie von Markov-Ketten beruhen, sei ein kurzer Überblick zu wichtigen Erkenntnissen gegeben. Dabei wurden das Skript der Vorlesung *Randomisierte Algorithmen* [14] sowie die Darstellungen von Behrends [2] als Grundlage verwendet. In der Stochastik werden häufig zeilenstochastische Matrizen verwendet, für die gilt, dass jeder Eintrag einer Zeile zwischen 0 und 1 liegt und die Summe der Zeileneinträge 1 ist. Eine *Markov-Kette* ist ein Paar (S, R) bestehend aus einer Zustandsmenge $S = \{1, \ldots, n\}$ und einer zeilenstochhastischen Matrix $P \in \mathbb{R}^{n \times n}$ wobei der Eintrag p_{ij} gerade die Übergangswahrscheinlichkeit von Zustand $i \in S$ nach $j \in S$ ist. Diese Übergangsmatrix definiert gewichtete Kanten zwischen den Zuständen für $p_{ij} > 0$ und indiziert somit einen Graph G.

Weiterhin heißt eine Markov-Kette *irreduzibel*, falls es sich bei G um einen zusammenhängenden Graph handelt, bei dem es von jedem Knoten zu jedem anderen Knoten einen Pfad gibt. Eine Markov-Kette heißt *aperiodisch*, falls für die Menge der Anzahl der Zustandsübergänge in G von einem beliebigen Knoten i zurück zu sich selbst gilt, dass der größte gemeinsame Teiler 1 ist.

Für aperiodische und irreduzible Markov-Ketten gibt es eine zeilenstochastische Matrix W, sodass gilt $\lim_{t\to\infty} P^t = W$ und $wP = w$, wobei die Zeilenvektoren von W alle gleich w sind. In anderen Worten konvergieren solche Markov-Ketten gegen die eindeutige stationäre Verteilung w. Es existieren Eigenschaften, die zudem zeigen können, dass diese Konvergenz in gewissem Maße schnell passiert. Ist die Übergangsmatrix P beispielsweise symmetrisch, so wird sie *reversibel* genannt. Reversible Markov-Ketten sind *schnell mischend* und konvergieren somit besonders schnell gegen die stationäre Verteilung.

2.2.2 Monte-Carlo-Algorithmen

Bei randomisierten Algorithmen unterscheidet man zwischen Monte-Carlo-Algorithmen und Las-Vegas-Algorithmen. Unter einem *Monte-Carlo-Algorithmus* versteht man einen randomisierten Algorithmus, welcher im Gegensatz zu einem *Las-Vegas-Algorithmus* nicht immer das korrekte Ergebnis liefert. Für eine genauere Betrachtung sei wieder auf [14] verwiesen. Da ein randomisierter Algorithmus unter Verwendung von Zufallsbits Entscheidungen fällen kann, können sowohl die Laufzeit, der Ressourcenverbrauch als auch das Ergebnis Zufallsvariablen sein. Im Allgemeinen interessiert man sich hier für Erwartungswerte. Da Monte-Carlo-Algorithmen nicht immer das korrekte Ergebnis liefern, ist die erwartete Fehlerwahrscheinlichkeit ein wichtiges Maß zur Abschätzung der Güte. Hat man einen Monte-Carlo-Algorithmus mit beschränkter Fehlerwahrscheinlichkeit ϵ (bspw. $\epsilon < 1/2$) so kann man häufig durch mehrfaches Ausführen des gleichen Algorithmus eine höhere Erfolgswahrscheinlichkeit zu Lasten der Laufzeit erzielen.

2.3 Zufällige Generierung von zusammenhängenden, einfachen Graphen mit gegebener Gradsequenz

Bereits seit 1959 gibt es Modelle zur zufälligen Generierung von Graphen ohne vorgegebene Gradsequenz. Das Erdős–Rényi Modell [4] basiert beispielsweise auf einer Gleichverteilung

und ist damit ungenügend für die Modellierung von beispielsweise dem World Wide Web oder sozialen Netzen, die eher dem Potenzgesetz folgen. Weiterhin gibt es neuere Modelle zur zufälligen Generierung von Graphen mit vorgegebener Gradsequenz zur detaillierten Definition der gewünschten Verteilung [11]. Dabei unterteilt man die Modelle in solche mit einer exakten und solche mit einer approximierten Gradsequenz. Sämtliche folgenden Beschreibungen beziehen sich nun auf *FDSM* (engl. fixed degree sequence model). Dieses ist als ein exaktes Model bezüglich der Gradsequenz zu klassifizieren, da die generierten Graphen Knotengrade entsprechend der gegebenen Gradsequenz aufweisen. Das Modell FDSM ist somit mit aufwändigeren Berechnungen verbunden als die approximierten Gegenstücke wie beispielsweise SIM (engl. simple independence model), das allerdings gerade bei großen Netzwerken an Genauigkeit verliert. Latapy und Viger setzten es sich in ihrer Ausarbeitung zum Ziel, Graphen mit exakten Eigenschaften zu generieren und den Prozess zu beschleunigen, um trotz des exakten Modells FDSM effizienter umzusetzen.

Zur zufälligen Generierung von Graphen mit den gewünschten Eigenschaften im FDSM kann man generell in zwei Schritten vorgehen [5, 8]. Zuerst wird ein Ausgangsgraph G_0 generiert, der sowohl zusammenhängend als auch einfach ist und dabei die gegebene Gradsequenz genau erfüllt. Hierbei fehlt nur noch das geforderte Merkmal des Zufalls. Danach wird G_0 bei strenger Erhaltung dieser Eigenschaften unter Verwendung von Markov-Ketten gemischt. Im Folgenden beschäftigt sich Abschnitt 2.3.1 mit der Erzeugung von G_0 und Abschnitt 2.3.2 mit der Weiterverarbeitung durch einen grundlegenden Markov-Kette-Monte-Carlo-Algorithmus. Danach wird zunächst in Abschnitt 2.3.2.3 die dafür benötigte Datenstruktur und die darauf unterstützten Operationen eingeführt. Zuletzt kann dadurch in Abschnitt 2.3.2.4 eine theoretische Analyse erfolgen.

2.3.1 Zusammenhängenden, einfachen Ausgangsgraph mit gegebener Gradsequenz generieren

Ein Graph G heißt *realisierbar für eine gegebene Gradsequenz*, wenn die Knotengrade aus G genau denen in der Gradsequenz entsprechen. Das Graphrealisierungsproblem, welches einen Graphen sucht, der eine gegebene Gradsequenz realisiert, wird durch den Havel-Hakimi Algorithmus [6] gelöst. Dabei terminiert der Algorithmus auch, sobald festgestellt wird, dass die Gradsequenz nicht realisiert werden kann.

Um aus einer realisierbaren Gradsequenz $grad[\cdot]$ einen Ausgangsgraph $G_0 = (V_0, E_0)$ zu erstellen, wird in den nachfolgend besprochenen drei Schritten vorgegangen. Algorithmus 1 stellt den Ablauf konkret dar. Alle Methodenaufrufe in dem Algorithmus haben sprechende Namen und werden in Abschnitt 2.3.2.3 genauer erklärt.

1. Jeder Knoten $v \in V_0$ bekommt entsprechend dem zu erfüllenden Grad $grad[v]$ viele Teilkanten $(v, -)$ ohne Zielknoten zugewiesen. Sei $m = \sum_{i=0}^{n-1} grad[i]$, dann ist offensichtlich $O(m + n)$ die Laufzeit.
2. Im nächsten Schritt wird nach dem Graphrealisierungsproblem mit dem Havel-Hakimi Algorithmus die gegebene Gradsequenz realisiert. Die Knoten seien dafür nun in einer Prioritätswarteschlange für Integers verwaltet, um schnell auf den Knoten mit den meisten freien Teilkanten zuzugreifen. Um die Gradsequenz zu realisieren werden nun jeweils zwei Teilkanten $(u, -)$ und $(v, -)$ mit $u \neq v$ verknüpft zu der ungerichteten Kante $\{u, v\}$. Wird immer der Knoten mit den meisten freien Teilkanten gewählt und ist die Gradsequenz realisierbar, dann ergibt sich ein Graph in dem für jeden Knoten $v \in V_0$ gilt $d(v) = grad[v]$ [3]. Abgesehen von der Verwaltung der Prioritätswarteschlange resultiert wieder eine Laufzeit von $O(m + n)$. Die Verwaltung der Prioritätswarteschlange für

Integers beschränkt sich auf Operationen mit konstanter Laufzeit. Ist die Gradsequenz nicht realisierbar kann entsprechend nach diesem Schritt abgebrochen werden.

3. Gegebenenfalls ist der bisher entstandene Graph G_0 nicht zusammenhängend. Um diese Eigenschaft zu gewährleisten, werden nicht verbundene Komponenten solange verschmolzen, bis der Graph zusammenhängend ist. Dafür führt man eine Breitensuche ausgehend von je zwei Knoten $x, y \in V_0, x \neq y$ aus. $C_x \subset G_0$ sei die Komponente, die von der Breitensuche bei x entdeckt wurde. Gilt $C_x \neq C_y$ und gibt es eine Kante $\{a, b\}$ in C_x oder C_y, die nicht für den Zusammenhalt der Komponente relevant ist, dann können Komponenten verbunden werden. Dafür sei die Kante $\{c, d\}$ aus der jeweils anderen Komponente beliebig gewählt. Die Komponenten werden wie folgt verbunden $E_0' = E_0 \setminus \{\{a, b\}, \{c, d\}\} \cup \{\{a, c\}, \{b, d\}\}$ und das offensichtlich unter Erhalt der Gradsequenz. Diese Operation wird nachfolgend als *Kantentausch* bezeichnet und ist in Abbildung 1 dargestellt. Findet man keine Komponente mehr, die verbunden werden kann, dann ist die Gradsequenz nicht realisierbar für zusammenhängende Graphen und der Algorithmus wird abgebrochen. Dies ist außerdem der Fall, wenn $m \leq n - 2$. Dann kann kein Baum über die Knoten des Graphen gespannt werden kann und der Graph ist somit nicht zusammenhängend. Falls $m \geq n - 1$ ist und der Graph noch aus mehreren Komponenten besteht, gibt es offensichtlich immer solch eine überflüssige Kante $\{a, b\}$ in einer Komponente.
   Da es durch die vorher beschriebenen Schritte zu maximal $\lceil n/2 \rceil$ Komponenten kommt, ergibt sich mit der Breitensuche eine Laufzeit von $O(n \cdot (n + m))$.

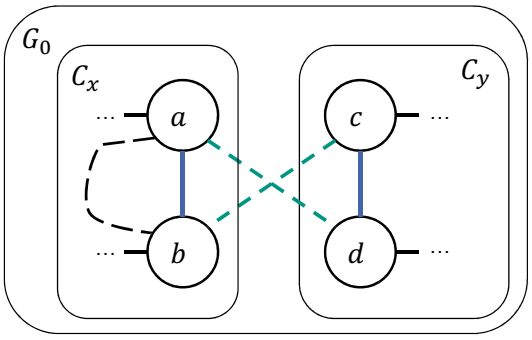


Abbildung 1 Verbinden von zwei Zusammenhangskomponenten in G_0 durch einen Kantentausch. Die aus dem Kantentausch resultierenden Kanten $\{a, d\}$ und $\{b, c\}$ verbinden die beiden Komponenten und reduzieren die Anzahl an Zusammenhangskomponenten in G_0 um eins, falls es in C_x einen Pfad von a nach b gibt.

2.3.2 Kantentauschoperationen und der Monte-Carlo-Algorithmus

Die in Abbildung 1 dargestellte Kantentauschoperation wird weiterhin genutzt, um den in Abschnitt 2.3.1 generierten Ausgangsgraph G_0 unter Verwendung von Markov-Ketten zu mischen, bis ein zufälliger Graph mit gleichen Eigenschaften entstanden ist.

2.3.2.1 Der Markov-Kette-Monte-Carlo-Algorithmus

Der Algorithmus 2 zeigt ein Vorgehen, um einen zufälligen, zusammenhängenden, einfachen Graph mit gegebener Gradsequenz zu generieren. Diesem Algorithmus gehen die Berechnungen für den Ausgangsgraph G_0 aus Algorithmus 1 voraus. Es bezeichne $T \in \Omega(m)$ die gewählte

Algorithmus 1 : Ausgangsgraph G_0 generieren

Eingabe : Gradsequenz $grad[n]$
Ausgabe : zusammenhängender, einfacher Ausgangsgraph G_0 mit Gradsequenz $grad[n]$

```
for i = 0 to n − 1 do
    grad[i] Teilkanten zu Knoten i zuweisen;
end
while Knoten a mit freien Teilkanten existiert do
    for j = 1 to grad[a] do
        b ← Knoten mit j meisten freien Teilkanten;
        G_0.kante_einfuegen((a − b));
        grad[a] ← grad[a] − 1;
        grad[b] ← grad[b] − 1;
    end
end
while G_0 nicht zusammenhängend do
    C_x, C_y, (a − b) ∈ C_x aus Zusammenhangstest;
    (c − d) ← G_0.zufaellige_kante();
    G_0.kantentausch((a − b), (c − d));
end
```

ausreichende Anzahl an Kantentauschoperationen bis zur genügenden Konvergenz der Markov-Kette. Dies wird in Abschnitt 2.3.2.2 genauer besprochen. Die Operation $kantentausch((a-b),(c-d))$ führt den bereits vorgestellten Kantentausch durch. Durch den Aufruf von $zufaellige\_kante()$ wird zufällig gleichverteilt eine der Kanten des Graphen ausgewählt. Die Überprüfung, ob der durchgeführte Kantentausch gültig ist, passiert durch die Validierungen $einfach()$ und $zusammenhaengend()$. Diese werden in Abschnitt 2.3.2.3 genauer besprochen. Die durch diesen naiven Mischalgorithmus definierte Markov-Kette ist in Abbildung 2 dargestellt.

2.3.2.2 Die Markov-Kette zum schnellen Mischen von G_0

Eine Kantentauschoperation heißt *gültig*, falls der entstehende Graph $G' = (V, E')$ einfach und zusammenhängend ist. Wie zuvor beschrieben, ändert ein Kantentausch an dem Knotengrad nichts. Analog wird diese als *ungültig* bezeichnet, sollte eine der beiden Eigenschaften verletzt werden. Es bezeichne G_t den Graph, der nach t gültigen, zufälligen Kantentauschoperationen entstanden ist. Der Graph G_t als Gesamtes hat konzeptionell genau zwei mögliche Übergänge in der Markov-Kette mit Wahrscheinlichkeit echt größer Null. Einerseits den Selbstübergang $G_t \to G_t$ bei einer ungültigen Operation und $G_t \to G_{t+1}$ ansonsten. Man beachte, dass viele verschiedene Kantentauschoperationen auf G_t geben kann, die entweder zu einem Selbstübergang oder zu dem nächsten gültigen Graph G_{t+1} führen. Anders gesehen kann jedes Kantenpaar aus G_t durch einen Kantentausch zu einem Übergang von G_t zu G_t oder G_{t+1} führen. Der initiale Zustand ist der Ausgangsgraph G_0.

▶ Theorem 1. Diese Beschreibung entspricht einer ergodischen, schnell mischenden Markov-Kette.

Beweis. Die verwendeten Begrifflichkeiten wurden in Kapitel 2.2.1 eingeführt und erklärt

Algorithmus 2 : Markov-Kette-Monte-Carlo-Algorithmus

Eingabe : zusammenhängender, einfacher Ausgangsgraph G_0 mit Gradsequenz $grad[n]$

Ausgabe : zufälliger, zusammenhängender, einfacher Graph G_T mit Gradsequenz $grad[n]$

```
for t = 0 to T do
    (a − b) ← G_t.zufaellige_kante();
    (c − d) ← G_t.zufaellige_kante();
    G' ← G_t.kantentausch((a − b), (c − d));
    if G'.einfach() und G'.zusammenhaengend() then
        G_{t+1} ← G';
    else
        G_{t+1} ← G_t;
    end
end
```

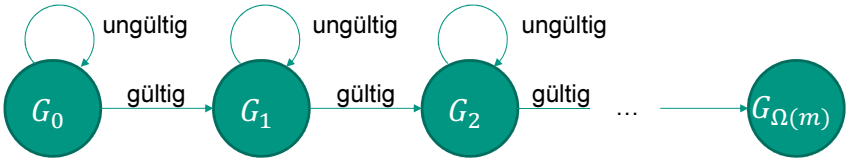


Abbildung 2 Markov-Kette des naiven Mischalgorithmus mit einem Zusammenhangstest und einem Test auf Einfachheit nach jeder Iteration (bzw. nach jedem Kantentausch). $G_{\Omega(m)}$ stehe symbolisch für den resultierenden Graph bei Konvergenz der Markov-Kette.

- Irreduzibel: $P_{G_t \to G_{t+1}}$ sei die Wahrscheinlichkeit für einen Übergang von G_t nach G_{t+1}. Da $P_{G_t \to G_t} > 0$ und jeder Zustand von G_0 aus erreichbar ist bzw. $P_{G_0 \xrightarrow{*} G_t} > 0$, ist die Markov-Kette irreduzibel.
- Aperiodisch: Durch den möglichen Selbstübergang ist es möglich in jedem Schritt in den gleichen Zustand zurückzukehren. Daher ist die Periode der Markov-Kette 1 und sie ist aperiodisch.
- Ergodisch: Da die Markov-Kette, wie gezeigt sowohl irreduzibel als auch aperiodisch ist und die Selbstübergänge dazu führen, dass man in jedem Schritt wieder in den Zustand zurückkehren kann, ist diese per Definition ergodisch.
- Schnell mischend: Die Markov-Kette ist *reversibel*, da ein Kantentausch $\{a,b\},\{c,d\} \to \{a,c\},\{b,d\}$ wieder rückgängig gemacht werden kann: $\{a,c\},\{b,d\} \to \{a,b\},\{c,d\}$. Eine reversible und ergodische Markov-Kette ist schnell mischend.

◀

Die Autoren Viger und Latapy stellen ebenfalls dar, dass die Wahrscheinlichkeit für eine gültige Kantentauschoperation in einem beliebigen Zustand durch $\frac{\rho}{2z(z+1)}$ nach oben beschränkt ist, wobei ρ der Anteil an Knotenpaaren mit einem Kantenabstand von maximal 3 gegenüber der Anzahl aller möglichen Knotenpaare ist. Der Wert z steht für den durchschnittlichen Knotengrad und ist somit für eine Eingabe konstant. Entsprechend genügt es die Anzahl an Kantentauschoperationen zu analysieren, um Rückschlüsse auf die Anzahl an Transitionen bis zur Konvergenz der Markov-Kette zu ziehen. Experimentelle Untersuchungen von Milo et al. [8] haben gezeigt, dass $O(m)$ Kantentauschoperationen reichen, um einen

Graph mehr oder weniger zufällig zu mischen. Theoretisch konnte eine polynomielle Schranke jedoch noch nicht bewiesen werden. Abbildung 2 stellt die beschriebene ergodische, schnell mischende Markov-Kette dar.

2.3.2.3 Datenstruktur

Die zugrunde liegende Datenstruktur soll sowohl den Graph G repräsentieren, als auch schnell Anfragen zu Kanten beantworten können. Insbesondere wird kein Fokus auf eine dynamische Datenstruktur gelegt, die die Konnektivität des Graphen schnell beantworten kann, da die Autoren auf einen verwandten, komplizierten Ansatz [7] schlechterer Laufzeit verweisen.

- Grad: Ein Feld der Länge n speichert an Index i den Knotengrad von dem Knoten i.
- Nachbarn: Kanten werden über Nachbarschaftsbeziehungen in dynamischen Hashtabellen gespeichert. Jede dieser Hashtabellen speichert explizit die Knoten, zu denen Knoten i eine Kante hat. Ein Feld der Länge n bildet wiederum implizit von Knoten i auf die Adresse der dazugehörigen dynamischen Hashtabelle ab. Entsprechend wird jede Kante doppelt repräsentiert. Beispielsweise kann ein dynamisches Hashverfahren wie Kuckucks-Hashing gewählt werden, um mit effizientem Speicherverbrauch konstante Anfragezeiten durch geschickte Kollisionsauflösung zu erzielen [10].
- Größe: Ein Feld der Länge n verwaltet explizit die aktuelle Anzahl an Feldern (inklusive leerer Einträge) der einzelnen dynamischen Hashtabellen.
- Wahrscheinlichkeiten: Es wird eine Wahrscheinlichkeitsverteilung zu den Kanten explizit gespeichert, indem jeder Knoten in einem Feld der Länge $2m$ so oft wie sein Knotengrad gespeichert wird. Somit kann durch zufällig gleichverteilte Wahl eines Indizes in dem Intervall $[0..2m-1]$ ein Knoten gefunden werden. Danach wird ebenfalls zufällig gleichverteilt ein Index aus der Hashtabelle des Knoten gewählt. Ist dieser Eintrag leer, so wird so lange zufällig gleichverteilt ein neuer Index gezogen, bis eine existierende Kante gefunden wurde. Da Kuckucks-Hashing eine hohe Speichernutzung erzielt, benötigt das Finden eines gültigen Indexes nur erwartet konstanten Mehraufwand.

Die somit definierte Datenstruktur ermöglicht die folgenden Operationen mit den genannten Laufzeiten.

- $kantentausch((a-b),(c-d))$: Diese Operation wurde bereits in Abschnitt 2.3.2 formal eingeführt Es werden alle vier Hashtabellen der Knoten aktualisiert, indem der alte Nachbar entfernt und der neue hinzugefügt wird. Ebenfalls wird vermerkt, ob durch den Kantentausch eine Mehrfachkante oder eine Schlinge entsteht. Diese beiden Eigenschaften sind durch einfaches Nachschlagen in den entsprechenden Hashtabellen möglich. Insgesamt ergibt sich eine erwartete Laufzeit von $O(1)$.
- $zufaellige\_kante()$: Es wird wie oben beschrieben eine Kante $(a-b)$ zufällig gleichverteilt aus der Menge der Kanten gezogen. Demzufolge ist die Laufzeit erwartet $O(1)$.
- $zufaelliger\_kantentausch()$: Diese Operation sei zur Vereinfachung definiert. Sie wählt zunächst zwei Kanten $(a-b)$ und $(c-d)$ mittels $zufaellige\_kante()$ und führt danach den Kantentausch $kantentausch((a-b),(c-d))$ durch. Zusammen ist die Laufzeit immer noch erwartet $O(1)$.
- $zusammenhaengend()$: Es wird eine Breitensuche von dem ersten Knoten ausgehend ausgeführt und überprüft, ob alle Knoten besucht werden. Entsprechend ist die Laufzeit dieser Operation $O(m+n)$.
- $einfach()$: Hierbei findet eine Überprüfung von beispielsweise einem während $kantentausch$ gesetzten Flag statt. Dieses gibt an, ob der durch den Kantentausch entstandene Graph einfach ist. Demnach ist die Laufzeit $O(1)$.

2.3.2.4 Laufzeitbetrachtung

Der Algorithmus 2 soll nun unter Verwendung der in Abschnitt 2.3.2.3 vorgestellten Datenstruktur theoretisch analysiert werden. Dabei ist der exakte Wert von T bis zur Konvergenz der Markov-Kette nicht bekannt, sodass vielmehr die Arbeit pro Iteration als Laufzeitmaß verwendet werden soll. Es bezeichne fortan C_{xyz} die *Einheitskosten* eines Algorithmus xyz, wobei sich C_{xyz} aus der Laufzeit bis zur nächsten Transition $G_t \to G_{t+1}$ (nicht Selbsttransition $G_t \to G_t$) der Markov-Kette ergibt. Die Einheitskosten von Algorithmus 2 sind offensichtlich durch den Zusammenhangstest nach einem Kantentausch dominiert und betragen entsprechend $C_{naiv} \in O(m)$.

2.4 Gkantsidis-Heuristik

Da der dominierende Faktor der Einheitskosten von Algorithmus 2 der Zusammenhangstest ist, soll nun die Anzahl der nötigen Tests reduziert werden. Zu diesem Zweck haben Gkantsidis et al. [5] einen heuristischen Ansatz vorgeschlagen, der eine bestimmte Anzahl ungeprüfter Kantentauschoperationen erlaubt, bevor der Konnektivitätstest ausgeführt wird. Dafür wird in Abschnitt 2.4.1 zuerst das statische Konzept erläutert und danach in Abschnitt 2.4.2 dynamisch betrachtet.

2.4.1 Statische Betrachtung

Wenn man Algorithmus 2 so abwandelt, dass erst alle X Iterationen ein Zusammenhangstest durchgeführt und je nach Resultat fortgefahren oder zurückgesetzt wird, erhält man Algorithmus 3 mit der statischen Gkantsidis-Heuristik. Dabei werden zunächst X beliebige Kantentauschoperationen durchgeführt, bevor der resultierende Graph auf Gültigkeit überprüft wird. Die dadurch definierte Markov-Kette ist in Abbildung 3 dargestellt. Offensichtlich werden dadurch einerseits die Laufzeitkosten je Durchlauf um den Faktor X geringer. Andererseits steigt die Wahrscheinlichkeit, den Graph in nicht zusammenhängende Komponenten zu unterteilen (Entkopplungswahrscheinlichkeit), ebenfalls proportional zu X. Anders gesagt, wird die Markov-Kette um X Zustände zurückgesetzt. Da diese Entkopplungswahrscheinlichkeit von der eingegebenen Gradsequenz abhängt, ist ein konstanter Wert für X ungenügend. Abschnitt 2.4.2 zeigt daher, wie ein optimaler Wert für X während der Abarbeitung abgeleitet werden kann.

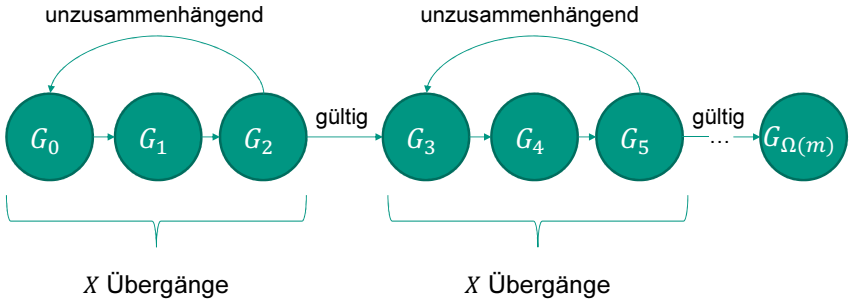


Abbildung 3 Markov-Kette der statischen Gkantsidis-Heuristik mit jeweils X Übergängen vor dem nächsten Zusammenhangstest. $G_{\Omega(m)}$ stehe symbolisch für den resultierenden Graph bei Konvergenz der Markov-Kette.

Algorithmus 3 : Markov-Kette-Monte-Carlo-Algorithmus mit statischer Gkantsidis-Heuristik

Eingabe : zusammenhängender, einfacher Ausgangsgraph G_0 mit Gradsequenz $grad[n]$

Ausgabe : zufälliger, zusammenhängender, einfacher Graph G_T mit Gradsequenz $grad[n]$

```
X ← Konstante;
G' ← G_0;
for t = 0 to T do
    G_{t+1} ← G_t.zufaelliger_kantentausch());
    if nicht G_{t+1}.einfach() then
        G_{t+1} ← G_t;
    end
    if X Durchläufe durchgeführt then
        if G'.zusammenhaengend() then
            G' ← G_{t+1};
        else
            G_{t+1} ← G';
        end
    end
end
```

2.4.2 Dynamische Betrachtung

Algorithmus 3 ist theoretisch vielversprechend. Allerdings ist nicht geklärt, welcher Wert für X optimal wäre. Anders formuliert, gibt es für jeden statisch festgelegten Wert eine Eingabesequenz, die zu einer schlechteren Laufzeit führen würde. Es ergibt sich die Frage nach einem optimalen Wert für X. Da dieser nicht statisch im Voraus festgelegt werden kann, ist die Idee der dynamischen Gkantsidis-Heuristik, ihn während der Berechnung zu ermitteln [5]. Die Vorgehensweise ist dabei ähnlich der Vermeidung von Netzwerküberlastungen in der Telematik. Der Algorithmus 4 beginnt mit $X = 1$ und tastet sich langsam an den optimalen Wert heran. Sobald der bearbeitete Graph nach einem Kantentausch nicht mehr zusammenhängend ist, wird diese Entkopplungswahrscheinlichkeit als zu hoch angenommen und X halbiert. Somit ist zu erwarten, dass sich X um einen optimalen Wert für die gegebene Gradsequenz einpendelt.

2.4.2.1 Erweiterung der dynamischen Gkantsidis-Heuristik

Um eine schnellere Annäherung an das optimale Fenster X^{opt} zu erzielen, haben die Autoren Viger und Latapy eine optimale Anpassungsmethode für den Algorithmus 4 entwickelt und analysiert. Dafür wird Algorithmus 4 insofern angepasst, als dass bei einer gültigen Transition X um einen festen Faktor q^+ erhöht wird. Im Falle einer ungültigen Transition wird entsprechend um einen Faktor q^- erniedrigt. Sie konnten herleiten, dass eine optimale Annäherung an X^{opt} geschieht, wenn $\frac{q^+}{q^-} = e - 1$ für das Verhältnis gilt. Die Größenordnung kann dabei frei gewählt und angepasst werden. Allerdings konnte in Experimenten festgestellt werden, dass sich für $\sqrt{q^+ q^-} = \frac{1}{10}$ die insgesamt besten Resultate ergeben. Analytisch resultieren mit dieser Anpassung die Einheitskosten $C_{neu} = O(1 + \bar{p} \cdot m)$, wobei $\bar{p}$ für den Durchschnittswert der Entkopplungswahrscheinlichkeit steht.

Algorithmus 4 : Markov-Kette-Monte-Carlo-Algorithmus mit dynamischer Gkantsidis-Heuristik

Eingabe : zusammenhängender, einfacher Ausgangsgraph G_0 mit Gradsequenz $grad[n]$
Ausgabe : zufälliger, zusammenhängender, einfacher Graph G_T mit Gradsequenz $grad[n]$

```
X ← 1;
G' ← G_0;
for t = 0 to T do
    G_{t+1} ← G_t.zufaelliger_kantentausch());
    if nicht G_{t+1}.einfach() then
        G_{t+1} ← G_t;
    end
    if X Durchlaufe durchgeführt then
        if G'.zusammenhaengend() then
            G' ← G_{t+1};
            X ← X + 1
        else
            G_{t+1} ← G';
            X ← ⌈X/2⌉;
        end
    end
end
```

2.5 Neuer Mischalgorithmus

Zusätzlich zu der optimierten Gkantsidis-Heuristik in Abschnitt 2.4.2.1 haben die Autoren in [13] einen neuartigen Mischalgorithmus vorgestellt. In Abschnitt 2.5.1 wird das grundlegende Konzept erörtert, wie man Entkopplungen des Graphen mit nur logarithmischen Kosten entdecken kann. Dieser sogenannte *K-Isolations-Test* wird anschließend in Abschnitt 2.5.2 zu einem effizienten Algorithmus ausgebaut.

2.5.1 Ansatz für einen Entkopplungstest

Diesem Abschnitt geht die Überlegung voraus, dass die meisten relevanten Graphen in der Praxis keine Bäume sind. Entsprechend gelte $\frac{m}{n} > 1 + \mu$, wobei μ reguliert wie stark zusammenhängend der Graph ist. Das bedeutet für eine zufällig gleichverteilt gewählte, zusammenhängende Komponente C_K mit K Knoten, dass es zusätzlich zu einem aufspannenden Baum mit $K - 1$ Kanten erwartet mindestens weitere μK *überflüssige Kanten* gibt. Wenn man nun annimmt, dass jeder Knoten in einem Graph mit jedem anderen Knoten verbunden sein kann, dann folgen für jede überflüssige Kante $n - K$ Zielknoten außerhalb von C_K.

Gibt es nun eine große, zusammenhängende Komponente mit mindestens $n/2$ Knoten (wie im Fall einer Entkopplung durch einen Kantentausch), so ist jede überflüssige Kante aus C_K mit Wahrscheinlichkeit mindestens $1/2$ mit dieser großen Komponente verbunden. Insgesamt ist die Wahrscheinlichkeit dafür, dass C_K über alle überflüssigen Kanten hinweg nicht mit der großen Komponente verbunden ist höchstens $2^{-\mu K}$.

Statt alle X Iterationen mit einer vollständigen, kostenintensiven Breitensuche den Graph auf Verbundenheit zu überprüfen, wird nun ein probabilistischer Ansatz gewählt. Führt

man die Breitensuche nur aus, bis man erfolgreich K Knoten entdeckt hat, kann man sich bereits mit Wahrscheinlichkeit $1 - 2^{-\mu K}$ sicher sein, dass der Graph zusammenhängend ist. Dabei geht man davon aus, dass der Graph erst zusammenhängend war und dann ein Kantentausch ausgeführt wurde, so wie das in dem naiven Algorithmus 2 der Fall war. Scheitert die Breitensuche bereits daran $K < n$ Knoten zu finden, ist der Graph entkoppelt. Diese Überlegungen führen zu der folgenden Operation.

Ein *K-Isolations-Test*, im Folgenden auch $k\_isoliert(K)$, überprüft nach einem Kantentausch $\{a, b\}, \{c, d\} \rightarrow \{a, c\}, \{b, d\}$ mit einer Breitensuche, ob es jeweils ausgehend von den Knoten a und b, die sich nun in unterschiedlichen Komponenten befinden könnten, $K < n$ zusammenhängende Knoten gibt. Schlägt der Test fehl, so ist der zugrunde liegende Graph definitiv nicht zusammenhängend. Offensichtlich dauert der Test gegenüber der vollständigen Breitensuche nicht $O(n+m)$, sondern hat eine Laufzeit von $O(K)$. Andererseits weist er eine Fehlerwahrscheinlichkeit von höchstens $2^{-2\mu K}$ auf. Für $K \geq \lceil n/2 \rceil$ ist der K-Isolations-Test jedoch immer korrekt, denn es werden im schlimmsten Fall alle Knoten besucht. Durch die einseitige Fehlerwahrscheinlichkeit handelt es sich bei dem K-Isolations-Test um ein probabilistisches Verfahren.

2.5.2 Mischalgorithmus mit K-Isolations-Test

Unter Verwendung des vorgestellten K-Isolations-Tests wird nun der Algorithmus 2 abgeändert. Dabei wird zunächst eine Kopie von dem Ausgangsgraph G_0 gemacht. Danach wird der bekannte Algorithmus ausgeführt, wobei die Breitensuche durch den K-Isolations-Test ersetzt wird. Da K zunächst sehr klein gewählt wurde, ergeben sich Einheitskosten von $O(K)$ und somit eine schnelle Konvergenz. Allerdings ist gerade bei kleinem K die Fehlerwahrscheinlichkeit recht hoch. Daher wird zum Abschluss noch ein vollständiger Konnektivitätstest durchgeführt. Sollte dieser zeigen, dass G_T nicht zusammenhängend ist, wird erneut von der Kopie aus, aber mit höherem K gestartet. Es resultiert der Algorithmus 5 mit der in Abbildung 4 dargestellten Markov-Kette.

Algorithmus 5 : Mischalgorithmus mit K-Isolations-Test

Eingabe : zusammenhängender, einfacher Ausgangsgraph G_0 mit Gradsequenz $grad[n]$

Ausgabe : zufälliger, zusammenhängender, einfacher Graph G_T mit Gradsequenz $grad[n]$

```
K ← 1;
G_init ← G_0;
do
    G_0 ← G_init;
    for t = 0 to T do
        G' ← G_t.zufaelliger_kantentausch();
        if G'.einfach() und G'.k_isoliert(K) then
            G_{t+1} ← G';
        else
            G_{t+1} ← G_t;
        end
    end
    K ← 2K;
until G_T.zusammenhaengend();
```

Naiv	Dynamisch	Gkantsidis (neu)	K-Isolation
$O(m)$	$O(\log n(\log\log n)^3)$	$O(1+\bar{p}\cdot m)$	$O(\log n)$

Tabelle 1 Vergleich der Einheitskosten verschiedener Ansätze zum zufälligen Generieren der Graphen

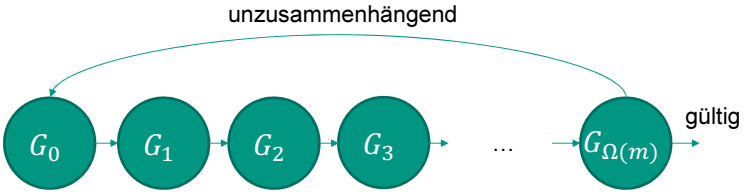


Abbildung 4 Markov-Kette des Mischalgorithmus unter Verwendung des K-Isolations-Tests nach jedem Übergang und einem vollständigen Zusammenhangstest am Schluss. $G_{\Omega(m)}$ stehe symbolisch für den resultierenden Graph bei Konvergenz der Markov-Kette.

Sei $N \leq \lceil n/2 \rceil$ die benötigte Anzahl an Iterationen der Do-Schleife, d.h. die Anzahl an Neustarts von dem initialen Graphen G_0 aus. Die Laufzeitkosten für den Algorithmus setzten sich zusammen aus dem initialen Kopieren von G_0, dem N-maligen Wiederherstellen der Kopie sowie N Mal dem Konvergieren gegen G_T unter Verwendung von K-Isolations-Tests und einem vollständigen Konnektivitätstest. Insgesamt ergibt sich für die gesamte Laufzeit von Algorithmus 5:

$$O(m) + N \cdot O(m) + \sum_{i=1}^{N} O((2^i + 1) \cdot T) = O(mN + T2^N) \tag{1}$$

Da $T \in \Omega(m)$ ist, folgen die Einheitskosten $C_{k-iso} \in O(2^N)$. Die Autoren sind durch experimentelle Untersuchungen zu der folgenden Erkenntnis gekommen: $E[C_{k-iso}] \in O(\log n)$.

2.6 Analytischer und experimenteller Vergleich der vorgestellten Ansätze

Im Folgenden werden die vorgestellten Ansätze hinsichtlich ihrer Einheitskosten und ihrer Performance in Versuchsreihen ausgewertet.

2.6.1 Einheitskosten

Viger und Latapy legen in ihrer Ausarbeitung besonderen Wert auf leicht verständliche Datenstrukturen und Algorithmen. Sie verweisen auf einen Ansatz von Henzinger und King [7] mit dynamischen Datenstrukturen, um die Konnektivitätstests schneller beantworten zu können. Zum Vergleich sind die dadurch erzielten Einheitskosten in Tabelle 1 mit dargestellt, wenn auch die Autoren mit der Kompliziertheit dieses und anderer Ansätze unzufrieden sind. Der dynamische Ansatz ist nicht nur komplex in der Implementierung, sondern bringt auch hohe konstante Faktoren mit sich, sodass dieser Ansatz in der Praxis keine Anwendung findet. Ebenfalls dargestellt sind die Einheitskosten des naiven Algorithmus 2, des optimierten Algorithmus 4 unter Verwendung der Gkantsidis-Heuristik und des Algorithmus 5 mit dem K-Isolations-Test.

z	C_{Gkan}	C_{neu}	C_{k-iso}
2.5	2606.12	1277.69	16.49
3	1082.64	287.20	10.40
4	259.02	16.61	5.81
5	56.75	2.20	4.24

Tabelle 2 Experimentelle Ergebnisse der gemessenen Einheitskosten für Binomialverteilungen mit einem durchschnittlichen Knotengrad von z. Von links nach rechts ist der Mischalgorithmus mit folgenden Erweiterungen dargestellt: die dynamische Gkantsidis-Heuristik, die optimale Gkantsidis-Heuristik und der K-Isolations-Test

2.6.2 Experimente

In experimentellen Untersuchungen wurden die vorgestellten Mischalgorithmen verglichen. Tabelle 2 stellt die gemittelten Einheitskosten dar. Es wird deutlich, dass gerade bei geringem mittleren Grad z der Mischalgorithmus mit K-Isolations-Test deutlich besser abschneidet. Da der mittlere Knotengrad $z = 5$ verglichen mit den anderen Werten hoch und gleichverteilt ist, führt die optimierte Gkantsidis-Heuristik zu einem rasanten Wachstum gegen ein optimal hohes Fenster X. Aus diesem Grund erklärt sich der bessere Wert gegenüber dem K-Isolations-Test, der wahrscheinlich erst nach wenigen Iterationen mehr zu einem zusammenhängenden Ergebnis kommt. Beide Ansätze sind jedoch deutlich besser als die dynamische Gkantsidis-Heuristik. Reale Netzwerke, wie zum Beispiel Freundschaftsbeziehungen in sozialen Netzwerken oder auch das World Wide Web, haben deutlich höhere durchschnittliche Knotengrade und es ist somit mit noch besseren Resultaten gegenüber der dynamischen Gkantsidis-Heuristik zu rechnen.

2.7 Fazit

Zusammenfassend wurden drei Ansätze besprochen, um aus einem generierten Ausgangsgraph G_0 mit Kantentauschoperationen zu einem konvergierten Zustand – also zufälligen Graph G_T – der Markov-Kette zu gelangen, wenn ausreichend viele Iterationen durchgeführt werden. Hierbei wurde auf einem naiven Ansatz, der in jedem Schritt einen vollständigen Konnektivitätstest durchführt, aufgebaut. Als Verbesserung wurden verschiedene heuristische Verfahren basierend auf der Gkantsidis-Heuristik vorgestellt, um unter Erhalt der Konvergenzeigenschaften weniger Konnektivitätstests zu benötigen. Am Ende konnte mit der Vorüberlegung des K-Isolations-Tests ein neuartiger Mischalgorithmus abgeleitet werden, der aktuell sowohl theoretisch, als auch praktisch die besten Resultate erzielt. Dabei wird der vollständige Konnektivitätstest nur zum Schluss der Konvergenz ausgeführt und während der Iterationen durch K-Isolations-Tests ersetzt. Dieser Algorithmus ist nicht nur effizienter, sondern auch simpler zu implementieren als direkt konkurrierende Ansätze über beispielsweise dynamische Datenstrukturen. Die Autoren Viger und Latapy stellen eine Implementierung zu diesem Algorithmus online zur Verfügung [12].

Referenzen

1 Dimitris Achlioptas, Aaron Clauset, David Kempe, and Cristopher Moore. On the bias of traceroute sampling: or, power-law degree distributions in regular graphs. In *Proceedings*

of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005, pages 694–703, 2005.

2 Erhard Behrends. *Introduction to Markov Chains*. Vieweg, 2000.

3 Paul Erdős and Tibor Gallai. Graphs with prescribed degree of vertices. *Mat. Lapok*, pages 264–274, 1960.

4 Paul Erdős and Alfréd Rényi. On random graphs i. *Publ. Math. Debrecen*, 6:290–297, 1959.

5 Christos Gkantsidis, Milena Mihail, and Ellen W. Zegura. The markov chain simulation method for generating connected power law random graphs. In *Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments, Baltimore, MD, USA, January 11, 2003*, pages 16–25, 2003.

6 S Louis Hakimi. On realizability of a set of integers as degrees of the vertices of a linear graph. i. *Journal of the Society for Industrial and Applied Mathematics*, 10(3):496–506, 1962.

7 Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

8 Ron Milo, Nadav Kashtan, Shalev Itzkovitz, Mark E. J. Newmann, and Uri Alon. Uniform generation of random graphs with arbitrary degree sequences. *arxiv:cond-mat/0312028*, 2003.

9 Mark E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45 (2):167–256, 2003.

10 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Algorithms—ESA 2001*, pages 121–133. Springer Berlin Heidelberg, 2001.

11 Wolfgang E. Schlauch, Emőke Ágnes Horvát, and Katharina A. Zweig. Different flavors of randomness: comparing random graph models with fixed degree sequences. *Social Network Analysis and Mining*, 5(1):1–14, 2015.

12 Fabien Viger and Matthieu Latapy. Implementierung des Mischalgorithmus mit K-Isolations-Test. https://www.liafa.jussieu.fr/~fabien/generation, 2005.

13 Fabien Viger and Matthieu Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. *Computing and Combinatorics*, pages 440–449, 2005.

14 Thomas Worsch. Randomisierte Algorithmen. http://liinwww.ira.uka.de/~thw/vl-rand-alg/skript-2014.pdf, 2014.

3 Trajektorienbasierte Gruppierung

Matthias Wolf

Zusammenfassung

In dieser Arbeit stellen wir eine von Buchin et al. [2] entwickelte Datenstruktur vor, die Gruppen von bewegten Objekten erfasst. Ausgehend von den Trajektorien von Objekten, die wir hier als stückweise linear annehmen, definieren wir zunächst, was wir unter Gruppen verstehen wollen. Wir beweisen eine obere Schranke für die Anzahl auftretender maximaler Gruppen. Den Kern bildet ein Algorithmus, der dazu mittels einer Sweepline die genannte Datenstruktur für n Objekte in $\mathcal{O}(\tau n^3)$ Zeit berechnet, wobei τ die Anzahl der Strecken pro Trajektorie bezeichne. Abschließend widmen wir uns auch der Frage, wie diese Datenstruktur robust gegen kurze Unterbrechungen gemacht und algorithmisch bestimmt werden kann.

3.1 Einleitung

Durch die weite Verbreitung und günstige Herstellung von GPS-Empfängern ist es möglich geworden, große Mengen an Bewegungsdaten zu erfassen. Ein Gebiet, das besonders der Motivation dieser Arbeit dient, ist die Verhaltensforschung. Eine Möglichkeit Wildtiere zu erforschen besteht darin, diese mit GPS-Sendern auszustatten. Aus den so gewonnenen Trajektorien der Tiere wollen Biologen Rückschlüsse auf das Verhalten der Tiere ziehen.

Eine interessante Fragestellung betrifft dabei, wie sich die Tiere in Gruppen zusammenfinden und wie diese sich im Laufe der Zeit verändern. Um diese Fragen zu beantworten haben Buchin et al. [2] eine Datenstruktur zur trajektorienbasierten Gruppierung (engl. *trajectory grouping structure*) entwickelt. Wie diese aufgebaut werden kann, wollen wir in dieser Ausarbeitung darstellen. Dieselben Autoren haben die Datenstruktur auch implementiert und daraus Videos erstellt[1].

Alternativ zu der hier vorgestellten Gruppierung von Objekten, wurden auch diverse andere Modelle entworfen. Dazu zählen allen voran Herden (engl. *flocks*) [1], zu denen eine Reihe von Algorithmen entwickelt wurden [1, 3, 4]. Weitere Modelle sind Konvois [5], sich bewegende Cluster (engl. *moving clusters*) [6] und Schwärme (engl. *swarms*) [8]. Diese Modelle unterscheiden sich unter anderem darin, wie Objekte zu Gruppen räumlich zusammengefasst werden. Liegen beispielsweise alle Objekte einer Gruppe innerhalb einer Scheibe [1] oder werden sie – wie hier – mittels Ketten von benachbarten Objekten beschrieben. Zudem werden unterschiedliche Änderungen der Gruppen (z.B. Auftrennen und Verschmelzen) unterstützt.

Wie die in dieser Arbeit vorgestellte Datenstruktur auch auf Umgebungen mit Hindernisse übertragen werden kann, untersuchen Kostitsyna et al. [7]. Der wesentliche Unterschied zum dem hier vorgestellten Algorithmus besteht in der Berechnung der Zeitpunkte, an denen sich die Gruppen ändern können.

Der Rest der Arbeit ist wie folgt aufgebaut: Im nächsten Abschnitt definieren wir formal, was wir unter einer Gruppe verstehen wollen. Im darauf folgenden Abschnitt 3.3 führen wir den Reeb-Graphen als Datenstruktur zur Repräsentation von Gruppen ein. Wie dieser aufgebaut werden kann, beschreiben wir in Abschnitt 3.4. Abschließend widmen wir uns in Abschnitt 3.5 der Frage, wie die Definition von Gruppen robust gegenüber kurzzeitigen

[1] http://www.staff.science.uu.nl/~staal006/grouping

Unterbrechungen werden kann, d.h. es wird ignoriert, wenn Objekte die Gruppe für eine kurze Zeit verlassen. Zudem beschreiben wir, wie der Reeb-Graph adaptiert werden kann, um die Robustheit zu beachten.

3.2 Gruppen

In der gesamten Arbeit betrachten wir die folgende Situation: Gegeben sei eine Menge $\mathcal{X}$ von Objekten und deren Trajektorien im Zeitintervall $[t_0, t_\tau] \subseteq \mathbb{R}$, welche durch die Positionen jedes Objekts $x \in \mathcal{X}$ im $\mathbb{R}^n$ zu den $\tau + 1$ Zeitpunkten $t_0 = t_0^x, t_1^x, \ldots, t_\tau^x = t_\tau$ beschrieben werden. Zwischen diesen Zeitpunkten bewegen sich die Objekte mit konstanter Geschwindigkeit auf der geraden Verbindungsstrecke zur nächsten bekannten Position. Die Positionen eines Objekts x zu den Zeiten t_i^x bezeichnen wir auch als *bekannte Positionen*.

Buchin et al. [2] gehen in der Beschreibung ihres Algorithmus davon aus, dass die Zeitpunkte der bekannten Positionen für alle Objekte identisch sind. Wir betrachten hier jedoch durchweg die oben beschriebene, allgemeinere Version und zeigen, dass sich die asymptotische Laufzeit des Algorithmus nicht ändert.

Unser Ziel ist es, die auftretenden Gruppen zu bestimmen. Dazu benötigen wir zunächst eine möglichst intuitive Definition einer Gruppe. Wir stellen im Folgenden die von Buchin et al. [2] vorgeschlagene Definition vor. Grob gesprochen sollen Objekte dann zu Gruppen zusammengefasst werden, wenn sie längere Zeit nahe beieinander sind. Daher benötigen wir zunächst eine formale Definition von räumlicher Nähe.

▶ **Definition 1.** Für $\varepsilon > 0$ seien zwei Objekte $x, y \in \mathcal{X}$ zum Zeitpunkt t *direkt verbunden*, wenn sich die Bälle $B_\varepsilon(x)$ und $B_\varepsilon(y)$ mit Radius ε um x bzw. y sich schneiden. Dies ist genau dann der Fall, wenn der Abstand von x und y höchstens 2ε beträgt.

Zwei Objekte $x, y \in \mathcal{X}$ heißen *ε-verbunden*, wenn es eine Folge von Objekten $x = x_0, x_1, \ldots, x_k = y$ in $\mathcal{X}$ gibt, so dass je zwei aufeinander folgende Objekte direkt verbunden sind.

Bei beiden Definition ist zu beachten, dass dabei ein Zeitpunkt $t \in [t_0, t_\tau]$ festgehalten wird. Objekte können zu einem Zeitpunkt direkt oder ε-verbunden sein und zu einem anderen nicht. Wie leicht einzusehen ist, definieren beide Begriffe der Verbundenheit zu jedem Zeitpunkt eine Äquivalenzrelation auf der Menge der Objekte.

Eine andere Sichtweise bietet die Modellierung der Verbundenheit zum Zeitpunkt t als Graph G_t: Dabei sei die Menge der Objekte $\mathcal{X}$ genau die Menge der Knoten. Zwischen zwei Knoten verläuft genau dann eine Kante, wenn die entsprechenden Objekte zum Zeitpunkt t direkt verbunden sind. Zwei Objekte sind nun zum Zeitpunkt t genau dann ε-verbunden, wenn die entsprechenden Knoten in derselben Zusammenhangskomponente von G_t liegen, denn eine Folge $x_1, x_2, \ldots, x_k$ von direkt verbundenen Objekten wie in Definition 1 entspricht in G_t dem Pfad $x_1 x_2 \ldots x_k$. Ein Beispiel dazu zeigt Abbildung 1.

Aufbauend auf der Definition von ε-Verbundenheit definieren wir, was wir unter einer Gruppe verstehen wollen.

▶ **Definition 2** (Gruppen)**.** Gegeben $\varepsilon > 0, \delta > 0$ und $m \in \mathbb{N}$ bildet ein Paar (X, I) bestehend aus einer Menge von Objekten $X \subseteq \mathcal{X}$ und einem Intervall $I \subseteq [t_0, t_\tau]$ eine *Gruppe*, wenn folgende Bedingungen erfüllt sind:

1. Zu jedem Zeitpunkt $t \in I$ sind alle Objekte aus X paarweise ε-verbunden.
2. $|X| \geq m$
3. $|I| \geq \delta$

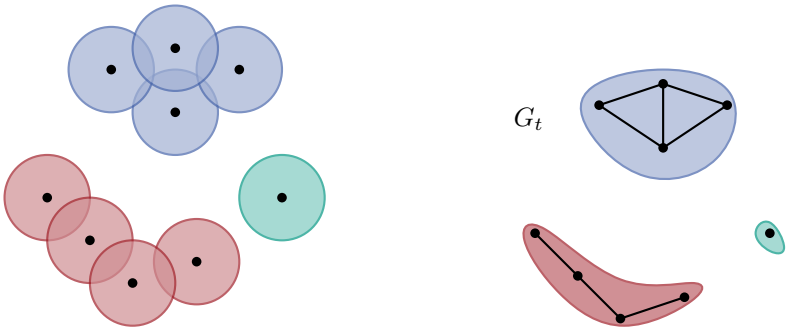


Abbildung 1 Positionen der Objekte zu einem Zeitpunkt t mit den ε-Scheiben und der entsprechende Graph G_t. Die zusammenhängenden Gebiete links entsprechen den Komponenten des Graphen G_t.

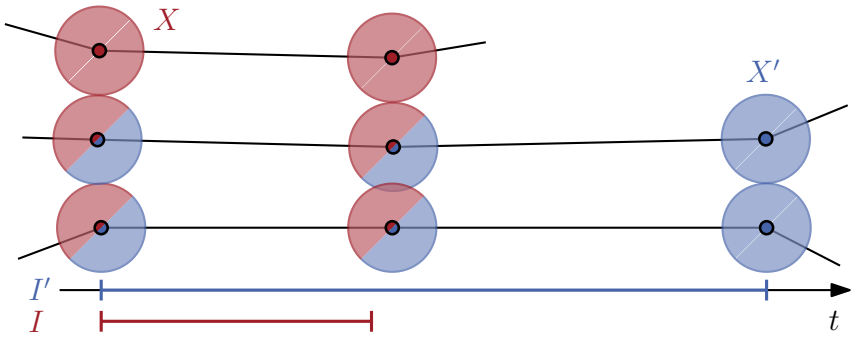


Abbildung 2 Objekte können gleichzeitig zu verschiedenen maximalen Gruppen gehören. Hier spaltet sich ein Objekt früher von den Objekten aus X' ab. Die Objekte in X bilden somit nur im kürzeren Intervall I eine Gruppe, diejenigen aus X' hingegen im gesamten Intervall I'. Sowohl (X, I) als auch (X', I') sind maximal. Somit gehören die unteren beiden Objekte im Intervall I zu zwei verschiedenen maximalen Gruppen.

Intuitiv bedeutet Bedingung 1, dass die Objekte der Gruppe nahe beieinander sein müssen. Bedingung 2 fordert eine Mindestgröße m und Bedingung 3, dass die Objekte mindestens über einen Zeitraum der Länge δ zusammen sind. Die erste Bedingung lässt sich auch mittels der Graphen G_t formulieren: Für alle Zeitpunkte $t \in I$ und Objekte $x, y \in G$ liegen x und y in derselben Komponente von G_t.

Eine Gruppe (X, I) *dominiert* eine Gruppe (X', I'), wenn sowohl $X' \subseteq X$ als auch $I' \subseteq I$ gilt. Intuitiv bedeutet dies, dass die dominierende Gruppe mindestens so groß ist und auch mindestens so lange existiert wie die dominierte Gruppe. Gilt für eine Gruppe $G = (X, I)$, dass das Zeitintervall nicht minimale Länge hat, d.h. $|I| > \delta$, so dominiert G auch alle Gruppen der Form (X, I') mit $I' \subseteq I$ und $|I'| \geq \delta$. Dies sind jedoch unendlich viele, weshalb wir uns im Folgenden darauf beschränken werden, die maximalen Gruppen zu bestimmen.

▶ **Definition 3** (Maximale Gruppen)**.** Eine Gruppe G ist *maximal*, wenn es keine andere Gruppe gibt, die G dominiert.

In Abschnitt 3.3.2 zeigen wir, dass es davon nur $\mathcal{O}(\tau n^3)$ Stück gibt. Zudem ist jede Gruppe in mindestens einer maximalen Gruppe enthalten, weshalb die maximalen Gruppen die Gruppierung der Objekte vollständig beschreiben.

Es ist jedoch zu beachten, dass Objekte gleichzeitig in verschieden maximalen Gruppen enthalten sein können. Dies ist beispielsweise dann der Fall, wenn ein Teil der Objekte sich

von einer Gruppe (X, I) ablöst und die restlichen Objekte X' länger beieinander bleiben. Sie formen dann eine Gruppe (X', I') mit $X' \subsetneq X$ und $I \subsetneq I'$. Somit dominiert keine der Gruppen die andere. In Abbildung 2 ist ein Beispiel für eine solche Situation dargestellt.

3.3 Der Reeb-Graph

In der Definition von Gruppen werden Objekte zusammengefasst, die eine gewisse Zeit lang ε-verbunden sind. Welche Objekte in welchen Intervallen ε-verbunden sind, kann durch den *Reeb-Graphen* repräsentiert werden – ein Konzept, das ursprünglich aus der Topologie stammt. Dabei entspricht eine Kante im Reeb-Graphen genau einer maximalen Menge von Objekten, die innerhalb eines Intervalls ε-verbunden sind. Die Knoten entsprechen Änderungen dieser maximalen Mengen.

3.3.1 Definition des Reeb-Graphen

Wir verzichten hier auf eine allgemeine Definition und beschränken uns darauf, den Reeb-Graphen für die Bewegung der Objekte auf ihren Trajektorien zu beschreiben und erläutern, wie er algorithmisch bestimmt werden kann (s. Abschnitt 3.4.1). Eine allgemeine, formale Definition findet sich in [9].

Zur Vereinfachung der Beschreibung nehmen wir ohne Beschränkung der Allgemeinheit an, dass zu keinem Zeitpunkt mehr als zwei Objekte direkt verbunden werden oder ihre direkte Verbindung verloren geht. Für die Graphen G_t bedeutet dies, dass keine zwei Kanten gleichzeitig eingefügt oder gelöscht werden. Diese Einschränkung erleichtert die Beschreibung des Reeb-Graphen und des Algorithmus, da dadurch weniger Fälle auftreten. Im Abschnitt 3.4.4 gehen wir kurz darauf ein, wie gleichzeitige Ereignisse behandelt werden können.

Der *Reeb-Graph* zu einer Menge von Objekten $\mathcal{X}$ und ihren Trajektorien im Zeitintervall $[t_0, t_\tau]$ ist ein gerichteter Graph, der sich wie folgt konstruieren lässt:

Füge zunächst für jede Komponente von G_{t_0} einen Knoten (*Start-Knoten*) mit einer ausgehenden (Halb-)Kante ein. Diese entspricht nun der Komponente.

Betrachte anschließend alle Zeitpunkte $t \in (t_0, t_\tau)$ aufsteigend. Falls sich zu einem Zeitpunkt t die Komponenten ändern, fügen wir einen Knoten in den Reeb-Graphen ein. Dabei können zwei Fälle auftreten:

- Zwei Komponenten verschmelzen: Füge einen *Merge-Knoten* ein und verbinde diesen mit den Halbkanten, die den verschmolzenen Komponenten entsprechen. Zusätzlich fügen wir eine ausgehende Halbkante ein, die der neuen Komponente entspricht.
- Eine Komponente trennt sich in zwei neue auf: Füge einen *Split-Knoten* ein, dessen eingehende Kante die Kante ist, die der aufgetrennten Komponente entspricht. Füge zwei ausgehende Halbkanten ein, die den neuen Komponenten entsprechen.

Abschließend fügen wir einen *End-Knoten* am Ende aller noch offenen Halbkanten ein.

Bei dieser Konstruktion unendlich viele Werte $t \in [t_0, t_\tau]$ betrachtet, wobei allerdings nur für endlich viele ein Knoten in den Reeb-Graphen eingefügt wird. In Abschnitt 3.4 beschreiben wir, wie die betrachteten Zeitpunkte auf eine möglichst kleine (und insbesondere endliche) Menge eingeschränkt werden kann.

Ein Beispiel eines Reeb-Graphen für die Trajektorien von Objekten findet sich in Abbildung 3. Die Objekte bewegen sich von unten nach oben und dementsprechend sind die Kanten des Reeb-Graphen auch von unten nach oben gerichtet. Dabei ist zu beachten, dass

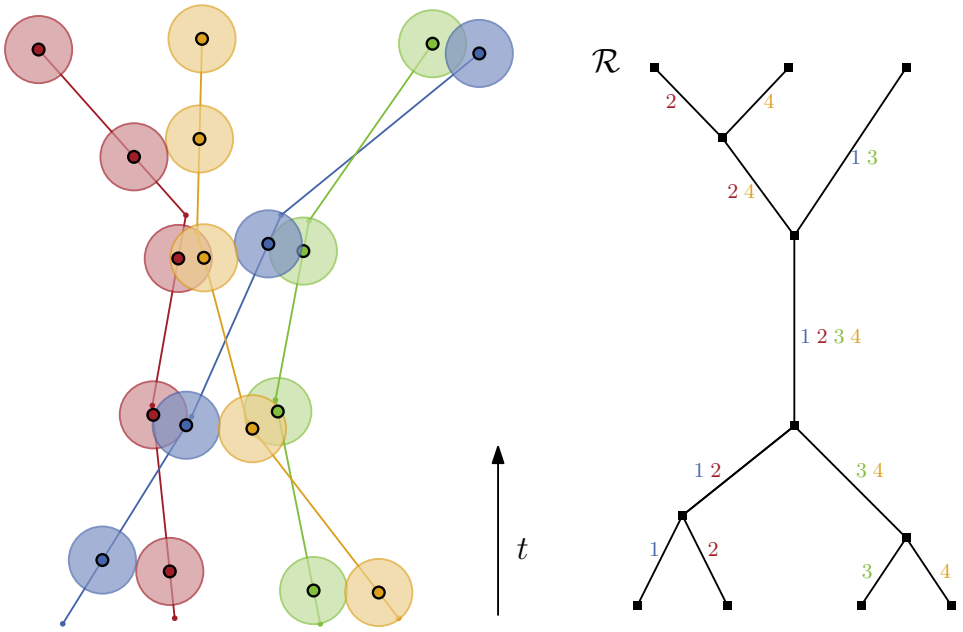


Abbildung 3 Links: Die Trajektorien der Objekte und deren ε-Scheiben zu Zeitpunkten, an denen sich die Komponenten ändern. Die Objekte bewegen sich von unten nach oben. Rechts: Der dazugehörende Reeb-Graph.

der Reeb-Graph im Allgemeinen kein Baum sein muss, sondern ein beliebiger azyklischer Graph sein kann. Die Bewegung eines Objekts wird im Reeb-Graph durch einen Pfad von einem *Start*- zu einem *End*-Knoten dargestellt. Das Objekt „bewegt sich“ entlang dieses Pfades.

Eine weitere Eigenschaft des Reeb-Graphen lässt sich direkt aus der Definition ableiten: Jeder Knoten hat Grad 1 (*Start*- und *End*-Knoten) oder Grad 3 (*Merge*- und *Split*-Knoten).

3.3.2 Obere Schranken

In diesem Abschnitt beweisen wir obere Schranken für die Größe des Reeb-Graphen zu einer Menge von n Objekten, deren Trajektorien durch τ Strecken dargestellt werden.

In der Definition des Reeb-Graphen werden alle Zeitpunkte $t \in [t_0, t_\tau]$ betrachtet. Jedoch ändert sich der Graph G_t genau zu den Zeitpunkten, an denen sich die direkte Verbindung zweier Objekte ändert, da genau dann eine Kante eingefügt oder gelöscht wird.

▶ **Lemma 4.** *Es gibt höchstens $\mathcal{O}(\tau n^2)$ Zeitpunkte, zu denen sich die direkte Verbindung der Objekte ändert.*

Beweis. Dieser Beweis basiert auf dem Beweis von Satz 2 in [2], verallgemeinert diesen jedoch, da wir hier nicht annehmen, dass die Zeitpunkte der bekannten Positionen für alle Objekte identisch sind.

Wir betrachten zwei Objekte $x, y \in \mathcal{X}$ und zeigen, dass ihre Eigenschaft, direkt verbunden zu sein, an höchstens $4\tau - 2$ Zeitpunkten ändert. Da es $\mathcal{O}(n^2)$ Paare von Objekten gibt, folgt daraus sofort die Behauptung.

Zunächst vereinigen wir die bekannten Zeitpunkte $t_0^x, \ldots, t_\tau^x$ und $t_0^y, \ldots, t_\tau^y$ zu einer Folge und sortieren diese aufsteigend. Die so erhaltene Folge sei $t'_0, \ldots, t'_{2\tau-1}$. In jedem der Intervalle $[t'_i, t'_{i+1}]$ bewegen sich beide Objekte linear. Dies impliziert, dass der Abstand von x und y in

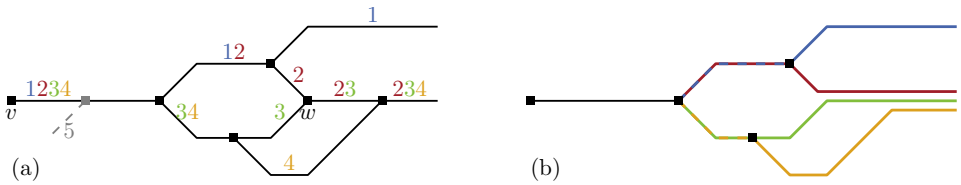


Abbildung 4 (a) Der von v ausgehende Reeb-Graph. (b) Der daraus entstandene Baum.

diesem Intervall eine konvexe Funktion ist. Somit gibt es höchstens einen Zeitpunkt, zu dem der Abstand unter 2ε fällt, und höchstens einen, zu dem der Abstand wieder über 2ε steigt. Dies sind genau die Zeitpunkte, an denen sich die direkte Verbindung ändert. Insgesamt gibt es in den $2\tau - 1$ Intervallen also höchstens $4\tau - 2$ kritische Zeitpunkte, was noch zu zeigen war. ◀

Daraus lässt sich auch direkt eine Aussage zur Komplexität des Reeb-Graphen ableiten.

▶ **Korollar 5.** *Der Reeb-Graph besitzt $\mathcal{O}(\tau n^2)$ Knoten und Kanten.*

Beweis. Pro Objekt gibt es einen *Start*- und einen *End*-Knoten, wobei auch mehrere Objekte denselben *Start*- oder *End*-Knoten haben können. Somit gibt es davon $\mathcal{O}(n)$ Stück. Die inneren Knoten werden ausschließlich von Änderungen der direkten Verbindungen induziert. Lemma 4 besagt, dass dies nur zu $\mathcal{O}(\tau n^2)$ Zeitpunkten geschieht. Folglich können auch nur $\mathcal{O}(\tau n^2)$ innere Knoten existieren.

Da jeder Knoten Grad 1 oder 3 hat, liegt auch die Anzahl der Kanten in $\mathcal{O}(\tau n^2)$. ◀

Davon ausgehend lässt sich auch eine obere Schranke für die Anzahl der maximalen Gruppen zeigen.

▶ **Theorem 6.** *Die Anzahl der maximalen Gruppen bei n Objekten, deren Trajektorien aus je τ Strecken bestehen, liegt in $\mathcal{O}(\tau n^3)$.*

Beweis. Wir beobachten zunächst, dass alle maximalen Gruppen bei einem *Start*- oder *Merge*-Knoten starten und bei einem *Split*- oder *End*-Knoten enden. Das Ziel ist es nun zu zeigen, dass an jedem Knoten des Reeb-Graphen $\mathcal{O}(n)$ maximale Gruppen beginnen. Da der Reeb-Graph laut Korollar 5 höchstens $\mathcal{O}(\tau n^2)$ Knoten besitzt, folgt daraus sofort die behauptete Schranke für die Anzahl der maximalen Gruppen.

Sei v ein *Start*- oder *Merge*-Knoten und X die Menge der Objekte, die seiner ausgehenden Kante entsprechen. Alle maximalen Gruppen, die bei v beginnen, bestehen ausschließlich aus Objekten in X. Daher betrachten wir den Teil $\mathcal{R}'$ des Reeb-Graphen, der aus den Knoten und Kanten besteht, welche auf von v ausgehenden, gerichteten Pfaden liegen. Dies sind genau die Kanten, welche nach v kommen und mindestens einem Objekt aus X entsprechen. Ein Beispiel zeigt Abbildung 4.

Wir beobachten außerdem, dass Objekte x und y sich trennen und später wieder treffen können (im Beispiel unter anderem x_2 und x_3 an Knoten w). Allerdings besitzt der Reeb-Graph dann einen weiteren *Merge*-Knoten w, bei dem die neue Gruppe mit x und y startet und dem wir diese Gruppe zurechnen.

Um diese Fälle zu umgehen, erstellen wir aus $\mathcal{R}'$ einen Baum T, indem wir die beiden Subgraphen nach einem *Split*-Knoten disjunkt machen. Dazu kopieren wir gegebenenfalls Teile von $\mathcal{R}'$, die wegen eines *Merge*-Knoten gemeinsam benutzt wurden. Abbildung 4(b) zeigt einen so entstandenen Baum.

Nun hat dieser Baum T höchstens $|X| \leq n$ Blätter (*End*-Knoten) und somit auch nur $\mathcal{O}(n)$ viele innere Knoten mit Grad größer als 2 (*Split*-Knoten). An jedem dieser Knoten u endet genau eine maximale Gruppen, nämlich jene, die aus genau den Objekten an der eingehenden Kante von u besteht. Folglich sind dies insgesamt $\mathcal{O}(n)$ maximale Gruppen, die am Knoten v beginnen. ◀

3.4 Algorithmus zur Berechnung des Reeb-Graphen

In diesem Abschnitt beschreiben wir einen Algorithmus zum Aufbau des Reeb-Graphen und zur Bestimmung der maximalen Gruppen. Dieser wurde von Buchin et al. [2] vorgestellt. Zunächst erläutern wir, wie die Struktur des Reeb-Graphen aus den Trajektorien erzeugt werden kann. Diese bildet auch das Kernstück des gesamten Algorithmus. Anschließend beschreiben wir, wie mittels geeigneter Beschriftung der Kanten des Reeb-Graphen die maximalen Gruppen bestimmt werden können (s. Abschnitt 3.4.2). In Abschnitt 3.4.3 gehen wir noch auf eine dynamische Datenstruktur zur Darstellung der Zusammenhangskomponenten eines Graphen ein, welche wir im Algorithmus verwenden.

3.4.1 Bestimmung des Reeb-Graphen

Der Algorithmus zum Aufbau des Reeb-Graphen orientiert sich an dessen Konstruktion in Abschnitt 3.3 und arbeitet mit einer Sweepline, welche sich entlang der Zeitachse im Intervall $[t_0, t_\tau]$ bewegt. Dabei ignorieren wir zunächst die Parameter δ und m aus der Definition einer Gruppe und betrachten nur den räumlichen Parameter ε.

Bei der Beschreibung des Algorithmus verwenden wir auch die Familie der Graphen $\{G_t\}_{t\in[t_0,t_\tau]}$. Zudem betrachten wir diese überwiegend als Zustände eines dynamischen Graphen G, d.h. G verändert sich im Intervall $[t_0, t_\tau]$ und ist zum Zeitpunkt t isomorph zu G_t.

Knoten im Reeb-Graphen $\mathcal{R}$ entsprechen Änderungen der Komponenten von G. Um diese schnell bestimmen zu können, nutzen wir dazu ST-Bäume F [10], welche einen aufspannenden Wald in G repräsentieren. Dieser Wald F ermöglicht es in $\mathcal{O}(\log n)$ Zeit zu überprüfen, ob sich die Komponenten ändern. Das Aktualisieren der Datenstruktur benötigt ebenfalls $\mathcal{O}(\log n)$ Zeit. Zudem liefern die ST-Bäume zu jeder Komponente einen Repräsentanten. In Abschnitt 3.4.3 wird gezeigt, dass es ausreicht, nur die ST-Bäume F zu speichern und den Graphen G nicht explizit zu repräsentieren. Zur Vereinfachung wird bei der Beschreibung des Algorithmus trotzdem auf den Graphen G verwiesen.

Wie bereits im Beweis von Korollar 5 gezeigt, werden nur zu den Zeitpunkten Knoten im Reeb-Graph eingefügt, an denen sich die direkte Verbindung zweier Objekte ändert. Deswegen wählen wir diese als *Ereignisse*, die wir während der Ausführung des Algorithmus betrachten wollen. Wir haben bereits in Lemma 4 gesehen, dass nur $\mathcal{O}(\tau n^2)$ Stück auftreten können.

Die Berechnung der Ereignisse erfolgt analog zu dem Vorgehen im Beweis von Lemma 4. Wir bestimmen alle Zeitpunkte, ab denen zwei Objekte neuerdings (*connect*-Ereignis) oder nicht mehr direkt verbunden sind (*disconnect*-Ereignis). Dies sind genau die Zeitpunkte, zu denen zwei Objekte genau den Abstand 2ε zueinander haben und dieser unmittelbar davor oder danach größer als 2ε ist.

Nach der Bestimmung der Ereignisse, bauen wir den Reeb-Graphen schrittweise auf. Dabei kommt es vor, dass wir bereits Kanten eingefügt haben, die zwar einen Startpunkt aber noch keinen Endpunkt besitzen. Jede dieser Halbkanten zum Zeitpunkt t ist mit einer Menge

von Objekten beschriftet, die zum Zeitpunkt t die Knoten einer Zusammenhangskomponente von G_t bilden. Umgekehrt gibt es auch für jede Komponente C von G_t eine Halbkante, so dass diese mit C beschriftet ist. Die Abbildung M von der Menge der Komponenten von G_t auf die Menge der Halbkanten zum Zeitpunkt t ist somit bijektiv. Wir schreiben $M(C)$ für die Halbkante, die mit den Objekten der Komponente C beschriftet ist. Diese Bijektion wird – beispielsweise als Hash-Tabelle oder binärer Suchbaum – explizit repräsentiert.

Zu Beginn initialisieren wir den Graphen G, indem wir für alle Paare x, y von Objekten deren Abstand $d(x, y)$ bestimmen und eine Kante $\{x, y\}$ in G einfügen, wenn der Abstand höchstens 2ε beträgt. Gleichzeitig initialisieren wir die ST-Bäume F, so dass diese immer die Komponenten von G repräsentiert.

Anschließend fügen wir für jede Komponente C von G (und somit auch von F) einen *Start*-Knoten mit einer ausgehenden Kante ein. Diese beschriften wir mit den Objekten der Komponente und fügen die Abbildung von C auf die ausgehende Kante in M ein.

Nach dieser Initialisierung betrachten wir die Ereignisse aufsteigend nach Zeit sortiert. Bei einem *connect*-Ereignis fügen wir zwischen den Objekten, die jetzt direkt verbunden sind eine Kante in den Graphen G ein und aktualisieren F. Falls dabei zwei Komponenten C_1 und C_2 verschmelzen, fügen wir einen neuen *Merge*-Knoten v mit eingehenden Kanten $M(C_1)$ und $M(C_2)$ in den Reeb-Graphen. Der Knoten v besitzt zusätzlich eine ausgehende Kante e, die wir mit $C = C_1 \cup C_2$ beschriften. Des Weiteren ersetzen wir C_1 und C_2 in M durch die Abbildung von C auf e.

Die Bearbeitung eines *disconnect*-Ereignisses läuft ähnlich ab. Wir entfernen die Kante zwischen den nicht mehr direkt verbundenen Objekten aus G und prüfen mittels F, ob dadurch eine Komponente C in zwei Komponenten C_1 und C_2 zerfällt. Wenn dies der Fall ist, fügen wir einen *Split*-Knoten v und in den Reeb-Graphen ein und setzen den Endpunkt der Kante $M(C)$ auf v. Zusätzlich besitzt v zwei ausgehende Halbkanten e_1 und e_2 mit Beschriftungen C_1 bzw. C_2. Die Abbildung M halten wir aktuell, indem wir C entfernen und stattdessen C_1 auf e_1 und C_2 auf e_2 abbilden.

Nachdem alle Ereignisse abgearbeitet wurden, gibt es im Reeb-Graphen noch Kanten, für die wir noch keine Endpunkte festgelegt haben. Diese entsprechen (mittels M^{-1}) den Komponenten von G_τ. Wir fügen für jede dieser Kanten einen *End*-Knoten in den Reeb-Graphen ein.

▶ **Theorem 7.** *Der oben vorgestellte Algorithmus konstruiert den Reeb-Graphen zu einer Menge von n Trajektorien bestehend aus τ Strecken in $\mathcal{O}(\tau n^2 \log n)$ Zeit.*

Beweis. Die Struktur des Algorithmus folgt der Konstruktionsvorschrift aus Abschnitt 3.3. Zudem haben wir bereits im Beweis von Korollar 5 gesehen, dass die im Algorithmus betrachteten Ereignisse ausreichen, um alle *Merge*- und *Split*-Knoten zu erhalten. Somit ist der Algorithmus korrekt. Zu zeigen bleibt also noch die Laufzeit.

Die Initialisierung betrachtet alle Paare von Objekten, wovon es $\mathcal{O}(n^2)$ viele gibt. Für jedes Paar wird F höchstens einmal aktualisiert, was jeweils $\mathcal{O}(\log n)$ Zeit kostet (s. Abschnitt 3.4.3). Insgesamt benötigt dies somit $\mathcal{O}(n^2 \log n)$ Zeit.

Aus Lemma 4 wissen wir, dass es $\mathcal{O}(\tau n^2)$ Ereignisse gibt. Diese können in $\mathcal{O}(\tau n^2 \log n)$ Zeit sortiert werden. Für jedes Ereignis stellen wir eine konstante Anzahl von Anfragen an F und M und aktualisieren diese auch nur konstant oft, was jeweils $\mathcal{O}(\log n)$ Zeit benötigt. Das Einfügen von neuen Knoten und Kanten kann in konstanter Zeit geschehen. Pro Ereignis wird somit $\mathcal{O}(\log n)$ Zeit aufgewandt, was für die Bearbeitung aller Ereignisse auf $\mathcal{O}(\tau n^2 \log n)$ Zeit hinausläuft.

Das abschließende Einfügen der *End*-Knoten erfordert $\mathcal{O}(n)$ Zeiteinheiten. Aufsummieren

der einzelnen Zeiten ergibt, dass der gesamte Reeb-Graph in $\mathcal{O}(\tau n^2 \log n)$ Zeit berechnet wird. ◀

3.4.2 Maximale Gruppen

Nachdem wir einen Algorithmus zur Konstruktion des Reeb-Graphen kennen gelernt haben, widmen wir uns im Folgenden der Frage, wie wir darin die (maximalen) Gruppen repräsentieren und diese auch ausgeben können.

Die Idee ist es dabei, den Reeb-Graphen mit Informationen anzureichern, welche uns die Bestimmung der maximalen Gruppen ermöglicht. Sei dazu (u, v) eine Kante des Reeb-Graphen, welche einer Komponente C im Graphen G entspricht, die dort vom Zeitpunkt t_u bis t_v existiert. Mit anderen Worten bedeutet dies, dass der Knoten u zum Zeitpunkt t_u und v zum Zeitpunkt t_v in den Reeb-Graphen eingefügt wurde. An dieser Kante speichern wir eine Menge von Paaren (X, t), wobei $X \subseteq C$ eine Teilmenge der Objekte und $t_0 \leq t \leq t_u$ gilt. Wir interpretieren diese Paare wie folgt: t ist der früheste Zeitpunkt, ab dem bis t_u alle Objekte aus X paarweise ε-verbunden sind, also in G in derselben Zusammenhangskomponente liegen.

Dieser Zeitpunkt ist für uns interessant, denn wenn die Objekte X während des Intervalls $[t_u, t_v]$ eine maximale Gruppe (X, I) (mit $[t_u, t_v] \subseteq I$) bilden, so muss dieses Intervall I bei t beginnen. Früher ist dies nicht möglich, da t gerade so gewählt ist, dass davor nicht mehr alle Objekte aus X paarweise ε-verbunden sind. Würde das Intervall später starten, so könnte es vergrößert werden, so dass es bei t beginnt und die Gruppe wäre nicht maximal.

Abbildung 5(a) zeigt die Beschriftung der Kanten des Reeb-Graphen zu den Trajektorien aus Abbildung 3. Daran ist schon zu erkennen, dass die Beschriftungen recht groß werden können. Es lassen sich Graphen konstruieren, in denen die durchschnittliche Größe der Kantenbeschriftung in $\Omega(n^2)$ liegt [2].

Allerdings sind die Beschriftungen von benachbarten Kanten nicht unabhängig voneinander. So tauchen beispielsweise alle Paare, die an den eingehenden Kanten eines *Merge*-Knotens stehen auch wieder an dessen ausgehender Kante aus. Diese Beobachtung kann benutzt

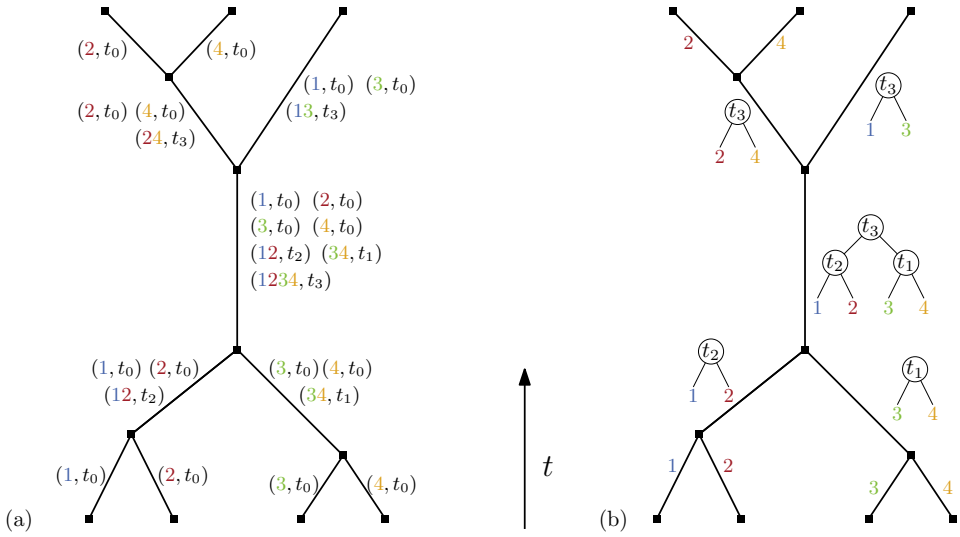


Abbildung 5 (a) Die Datenstruktur zur trajektorienbasierten Gruppierung mit expliziter Darstellung der Gruppen. Eine Menge von von Objekten $\{x_1, \ldots, x_k\}$ schreiben wir dabei der Kürze wegen als $x_1 \ldots x_k$. (b) Derselbe Graph, wobei die Gruppen mittels Bäumen repräsentiert werden. Der Zeitpunkt t_0 ist dabei implizit.

werden, um die Paare in einer kompakteren Form zu repräsentieren, welche im Wesentlichen den vergangenen Verschmelzungen von Komponenten entspricht. Genauer speichern wir an jeder Kante einen Baum, wobei die Blätter Mengen von Objekten und die inneren Knoten Zeitpunkte, zu denen die Objekte in den Unterbäumen verschmolzen sind, repräsentieren. Jede Kante hält dann einen Zeiger auf die Wurzel der Baumes. Diese Darstellung wird in Abbildung 5(b) beispielhaft gezeigt.

Um diese Bäume zu berechnen, erweitern wir unseren Algorithmus aus dem vorherigen Abschnitt wie folgt: Wir berechnen die Beschriftung einer Kante (u, v) immer zum Zeitpunkt t_u, zu dem ihr Anfangsknoten u in den Reeb-Graphen eingefügt wird. Dabei ist zu beachten, dass v zu diesem Zeitpunkt noch gar nicht bekannt ist und erst später festgelegt wird. Wir bezeichnen im Folgenden die Objekte der Komponente, die (u, v) entspricht, mit C. Bei der Berechnung der Beschriftung müssen wir abhängig vom Typ des Knotens u drei Fälle unterscheiden:

- u ist ein *Start*-Knoten: Dies tritt ausschließlich in der Initialisierung auf und in diesem Fall sind alle Objekte der Komponente seit dem Zeitpunkt t_0 paarweise ε-verbunden. Die Beschriftung besteht ausschließlich aus einem Blattknoten, der alle Objekte der Komponente enthält. Dies lässt sich für alle *Start*-Knoten in Zeit $\mathcal{O}(n)$ erledigen.
- u ist ein *Merge*-Knoten: In diesem Fall genügt es einen Wurzelknoten mit Beschriftung t_u einzufügen, der die Wurzeln der Teilbäume an den eingehenden Kanten von u als Kindknoten besitzt. Pro *Merge*-Knoten benötigen wir so $\mathcal{O}(1)$ Zeit.
- u ist ein *Split*-Knoten: Sei T der Baum an der eingehenden Kante von u. Wir bauen den Baum rekursiv auf. Für jeden Blattknoten von T mit Objekten X_{Blatt} fügen wir einen Blattknoten in den neuen Baum T' mit Beschriftung $X_{\text{Blatt}} \cap C$ ein, sofern diese Beschriftung nicht leer ist. Einen inneren Knoten aus T übernehmen wir in T', falls beide Kinder in T' existieren. So erreichen wir, dass alle inneren Knoten von T' zwei Kinder haben und alle Blätter nichtleer sind. Dies kann durch Traversieren des Baums T und parallelem Aufbau von T' in $\mathcal{O}(|T|) \subseteq \mathcal{O}(n)$ Zeit geschehen.

Der so beschriftete Reeb-Graph bildet die Datenstruktur zur trajektorienbasierten Gruppierung. Im Gegensatz zum Aufbau des Reeb-Graphen benötigt das Einfügen eines Knotens in den Reeb-Graphen nun $\mathcal{O}(n)$ Zeit, da noch die Beschriftungen generiert werden müssen. Daraus ergibt sich analog zu Theorem 7 die folgende Aussage.

▶ **Theorem 8.** *Die trajektorienbasierte Gruppierung lässt sich in $\mathcal{O}(\tau n^2 \log n + n \cdot |\mathcal{R}|)$ Zeit berechnen und ihre Größe beträgt $\mathcal{O}(n \cdot |\mathcal{R}|)$. Dabei bezeichne $|\mathcal{R}|$ die Anzahl Knoten und Kanten des Reeb-Graphen.*

Gemeinsam mit der oberen Schranke von $\mathcal{O}(\tau n^2)$ für die Komplexität des Reeb-Graphen aus Korollar 5 ergeben sich obere Schranken, die unabhängig von der Ausgabegröße sind.

▶ **Korollar 9.** *Die trajektorienbasierte Gruppierung lässt sich in $\mathcal{O}(\tau n^3)$ Zeit aufbauen und sie benötigt $\mathcal{O}(\tau n^4)$ Speicherplatz.*

Nachdem wir dargestellt haben, wie der Reeb-Graph mit Beschriftungen abgereichert werden kann, wollen wir im Folgenden zeigen, wie damit die maximalen Gruppen bestimmt und ausgegeben werden können. Dies kann während des Aufbaus des Reeb-Graphen oder anschließend in einem separaten Schritt geschehen. Grundsätzlich sollten wir uns dabei in Erinnerung rufen, dass die Definition einer Gruppen auch eine Mindestdauer δ und eine Mindestgröße m voraussetzt, welche wir beim Aufbau der Datenstruktur bisher ignoriert haben.

Bei der Ausgabe der maximalen Gruppen helfen uns die folgenden Beobachtungen: Alle maximalen Gruppen beginnen an *Start*- oder *Merge*-Knoten und enden an *Split*- oder *End*-Knoten. Zudem wird das Enden einer maximalen Gruppe an einem *Split*-Knoten immer dadurch induziert, dass die Objekte der Gruppe sich an diesem Zeitpunkt trennen und ab da (zumindest für kurze Zeit) nicht mehr paarweise ε-verbunden sind. Da die Startzeitpunkte und Zusammensetzung der maximalen Gruppen bereits in die Kantenbeschriftungen eingeflossen sind, müssen wir jetzt nur noch den Endzeitpunkt der Gruppe bestimmen.

Wir gehen daher alle *Merge*-Knoten durch und betrachten alle Beschriftungen an der eingehenden Kante. Stellen wir fest, dass bei einer Beschriftung (X, t_X) die Menge der Objekte X durch das *disconnect*-Ereignis zum Zeitpunkt t getrennt werden, so erhalten wir $G = (X, [t_X, t])$ als potentielle Gruppe. Wir müssen nur noch überprüfen, ob G auch die Mindestlänge δ, d.h. $t - t_X \geq \delta$, einhält und, ob $|X| \geq m$ gilt. Ist dies der Fall, formt G auf jeden Fall eine Gruppe. Diese ist auch maximal, denn nach Konstruktion der Beschriftungen kann weder X vergrößert noch t_X verringert werden. Da die Gruppe sich am Zeitpunkt t auftrennt, kann auch das Ende des Intervalls nicht weiter nach hinten geschoben werden.

An den *End*-Knoten führen wir die gleiche Überprüfung durch, bis darauf, dass wir nicht testen müssen, ob die Objekte anschließend noch zusammen bleiben.

Insgesamt erhalten wir daraus den folgenden Satz.

▶ **Theorem 10.** *Die maximalen Gruppen zu n Trajektorien von Objekten bestehend aus je τ Strecken können in $\mathcal{O}(\tau n^2 \log n + n \cdot |\mathcal{R}| + N)$ Zeit berechnet und ausgegeben werden, wobei $|\mathcal{R}|$ die Komplexität des Reeb-Graphen und N die Ausgabegröße bezeichne.*

Beweis. Laut Theorem 7 kann die Datenstruktur zur trajektorienbasierten Gruppierung in $\mathcal{O}(\tau n^2 \log n + |\mathcal{R}|)$ Zeit aufgebaut werden. Pro *Split*- und *End*-Knoten benötigt die Überprüfung, welche Gruppen dort enden $\mathcal{O}(n)$ Zeit, da dazu einmal der Baum an der eingehenden Kante traversiert werden muss. Über alle Knoten summiert ergibt sich somit für die Bestimmung und Ausgabe der maximalen Gruppen eine Laufzeit von $\mathcal{O}(n \cdot |\mathcal{R}| + N)$, was zusammen mit der Zeit zum Aufbau der Datenstruktur die behauptete Laufzeit ergibt. ◀

Bei der Laufzeit zur Berechnung der trajektorienbasierten Gruppierung (s. Theorem 7) spielen zum einen die Anzahl der Ereignisse und zum anderen die Größe des Reeb-Graphen eine entscheidende Rolle. Dabei ist klar, dass es mindestens so viele Ereignisse wie innere Knoten (d.h. *Merge*- und *Split*-Knoten) geben muss, da jedes Ereignis höchstens einen solchen Knoten induziert. Wir wollen im Folgenden untersuchen, wie große das Verhältnis von Anzahl Ereignissen zu Anzahl Knoten des Reeb-Graphen werden kann. Intuitiv bedeutet ein großes Verhältnis, dass der Algorithmus viele Ereignisse betrachtet, von denen jedoch nur wenige einen Einfluss auf den Reeb-Graphen haben.

Da es laut Lemma 4 höchstens $\mathcal{O}(\tau n^2)$ viele Ereignisse gibt und der Reeb-Graph mindestens zwei Knoten enthält, erhalten wir $\mathcal{O}(\tau n^2)$ als obere Schranke für dieses Verhältnis. Wie das folgende Lemma zeigt, ist die Schranke auch scharf.

▶ **Lemma 11.** *Für $n, \tau \in \mathbb{N}$ gibt es n Trajektorien mit je τ Strecken, so dass der Sweepline-Algorithmus zu Bestimmung des Reeb-Graphen $\theta(\tau n^2)$ viele Ereignisse betrachtet, der Reeb-Graph jedoch nur aus einer Kante besteht.*

Beweis. Wir geben eine explizite Beschreibung einer solchen Instanz an, indem wir die Bewegung der Objekte während eines Zeitintervalls $[t_i, t_{i+1}]$ betrachten. In diesem Intervall bewegen sich alle Objekte entlang einer Strecke (oder bleiben stehen). Im folgenden Zeitintervall bewegen sich die Objekte auf der gleichen Strecke zu ihrem Ausgangspunkt zurück. Dieses Vorgehen wiederholen wir für alle τ Intervalle.

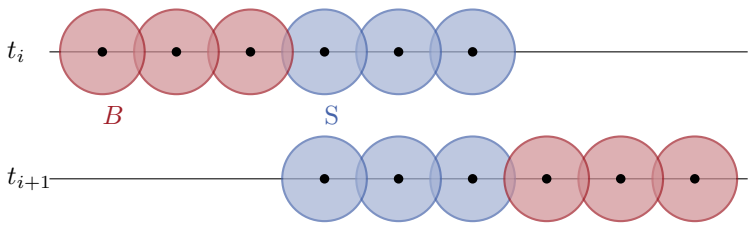


Abbildung 6 Bewegung der Objekte B an denen aus S vorbei ergibt $\theta(n^2)$ Ereignisse.

Zunächst teilen wir die Objekte in eine Menge $B = \{x_1, \ldots, x_{n/2}\}$ von bewegten und eine Menge $S = \mathcal{X} \setminus B$ von stationären Objekten ein. Zu Beginn sind alle Objekte auf einer Geraden angeordnet, wobei die Objekte in B links platziert werden. Benachbarte Objekte haben dabei den Abstand $d < \varepsilon$. Während des Intervalls $[t_i, t_{i+1}]$ bewegen sich die Objekte in B mit konstanter Geschwindigkeit nach rechts bis sie die stationären Objekte aus S passiert haben. Zum Zeitpunkt t_{i+1} sind so wieder alle Objekte mit Abstand d auf der Geraden angeordnet, nur dass sich diesmal die Objekte aus S links von denen in B befinden. Die Anordnung zu den Zeiten t_i und t_{i+1} zeigt Abbildung 6

Während des gesamten Intervalls sind alle Objekte paarweise ε-verbunden, der zugehörige Reeb-Graph besitzt also weder *Merge*- noch *Split*-Knoten. Über den gesamten Zeitraum betrachtet besteht er folglich nur aus einer Kante.

Bis auf x_1 und $x_{n/2}$ passiert jedes Objekt aus B alle Objekte in S und generiert so $\theta(|S|) = \theta(n)$ viele Ereignisse. Insgesamt betrachtet der Algorithmus im Intervall $[t_i, t_{i+1}]$ somit $\theta(n^2)$ viele Ereignisse. Über alle τ Intervalle ergibt sich die behauptete Anzahl von $\theta(\tau n^2)$ vielen Ereignissen. ◀

3.4.3 Der maximale Wald

Der vorgestellte Algorithmus benötigt eine Datenstruktur, mit der schnell entschieden werden kann, ob das Hinzufügen oder Löschen einer Kante aus dem Graphen G dessen Zusammenhangskomponenten ändert. Dazu verwenden wir eine Technik von Parsa [9], welche selbst auf einer dynamischen Datenstruktur für maximale aufspannende Wälder in einem sich dynamisch verändernden Graphen aufbaut. Als solche können beispielsweise ST-Bäume benutzt werden, welche von Sleator und Tarjan [10] vorgestellt wurden. In diesem Abschnitt wollen wir genauer vorstellen, wie diese bei der Berechnung von Reeb-Graphen eingesetzt werden können. Zudem wird gezeigt, dass es ausreicht, nur einen aufspannenden Wald von G und nicht den Graphen G selbst explizit zu speichern.

Dabei verzichten wir auf eine detaillierte Darstellung der ST-Bäume. Im Folgenden genügt es sich diese schlicht als einen aufspannenden Baum pro Komponente des Graphen G vorzustellen, wobei die folgenden Operationen jeweils in $\mathcal{O}(\log n)$ unterstützt werden: Einfügen und Löschen einer Kante, Bestimmung der Kante mit dem niedrigsten Gewicht auf dem eindeutigen Pfad zwischen zwei Knoten, Bestimmung eines Repräsentanten einer Komponente.

Zunächst gewichten wir jede Kante des Graphen G mit der Zeit, zu der sie wieder aus dem Graphen G entfernt wird. Sollte die Kanten nicht mehr aus dem Graphen entfernt werden, setzten wir als Gewicht den Endzeitpunkt t_τ. Durch diese Gewichtung erreichen wir, dass Kanten, die zwar noch in G vorhanden sind, jedoch nicht zum maximalen Wald F gehören, auch zu keinem späteren Zeitpunkt in F liegen.

Im Algorithmus werden Kanten in den Graphen G eingefügt und gelöscht. Dabei sind wir daran interessiert zu erfahren, ob sich die Zusammenhangskomponenten von G ändern.

Wir zeigen nun, wie diese Operationen implementiert werden können, ohne den Graphen G explizit zu repräsentieren.

Beim Einfügen einer Kante $e = \{u, v\}$ mit Gewicht $w(e)$ überprüfen wir zunächst, ob die Endpunkte der Kante in derselben Zusammenhangskomponente von G liegen, indem wir für die beiden Endpunkte von e jeweils den Repräsentanten ihrer Komponente bestimmen. Sind diese verschieden, so wird die Kante in F eingefügt. Liegen u und v in derselben Zusammenhangskomponente, bestimmen wir die Kante e' auf dem eindeutigen Pfad von u nach v mit minimalem Gewicht. Falls $w(e') < w(e)$ gilt, löschen wir e' aus F und fügen e ein. Andernfalls wird F nicht verändert. Wir rufen eine konstante Anzahl an Operationen auf dem aufspannenden Wald F auf, welche jeweils $\mathcal{O}(\log n)$ Zeit benötigen. Somit benötigt auch das gesamte Einfügen einer Kante in G (was wir aber nur implizit tun) und das Aktualisieren von F eine Laufzeit von $\mathcal{O}(\log n)$

Das Löschen einer Kante $e = \{u, v\}$ aus G behandeln wir wie folgt: Zunächst überprüfen wir, ob die Kante in F auftritt. Ist dies der Fall, löschen wir sie aus F. Dadurch zerfällt der Baum, der e enthielt, in zwei Teile, wobei einer den Knoten u und einer v enthält. Ist e in keinem der Bäume vorhanden, so muss auch F nicht aktualisiert werden, da der aufspannende Wald maximal bleibt. Das Ergebnis des zweiten Falls ist offensichtlich korrekt. Die Korrektheit des ersten Falls folgt aus dem folgenden Lemma.

▶ **Lemma 12.** *Wird eine Kante e aus dem Graphen G entfernt, die außerdem zu einem Baum aus F gehört, so ist $F' = F - e$ ein maximaler aufspannenden Wald von $G - e$.*

Beweis. Wir bezeichnen die Graphen vor und nach Löschen von e mit G bzw. G', d.h. $G' = G - e$. Analog sei F ein maximaler aufspannender Wald von G und $F' = F - e$. Durch Löschen der Kante e aus F zerfällt ein Komponente F in zwei Teile mit Knotenmengen V_1 und V_2. Es ist zu zeigen, dass die entsprechende Komponente in G genauso geteilt wird. Dies wäre genau dann nicht der Fall, wenn in G' eine Kante e' zwischen V_1 und V_2 existieren würde. Dann hätte e' aber ein größeres Gewicht als e, da e' ja nach e gelöscht wird. Nun wäre jedoch $F'' = F' + e' = F - e + e'$ ein aufspannender Wald von G mit größerem Gewicht als F. Dies widerspricht allerdings der Gewichtsmaximalität von F.

Somit ist F' zumindest ein aufspannender Wald von G'. Da F maximal ist, gilt dies auch für F', denn jedem aufspannenden Wald in G' entspricht durch Hinzufügen der Kante e bijektiv ein aufspannender Wald von G. ◀

Wenn eine Kante aus F entfernt wird, müssen wir also nicht überprüfen, ob wir stattdessen eine andere Kante aus $G - e$ einfügen müssen. Dies erlaubt es uns den Graphen G gar nicht explizit zu speichern.

3.4.4 Degenerierte Fälle

In der Beschreibung wurde bislang angenommen, dass keine zwei Ereignisse gleichzeitig geschehen. Im Folgenden wird kurz darauf eingegangen, was getan werden kann, wenn dies doch der Fall ist. Dazu sortieren wir gleichzeitige Ereignisse so, dass *connect*-Ereignisse vor *disconnect*-Ereignissen betrachtet werden.

Wird der Algorithmus nun wie oben dargestellt durchgeführt, kann es geschehen, dass im Reeb-Graphen Knoten benachbart sind, die wegen gleichzeitig auftretenden Ereignissen eingefügt wurden. Die obige Sortierung garantiert, dass dabei *Merge*-Knoten immer vor *Split*-Knoten eingefügt wurden. Dies führt dazu dass alle Knoten mit dem Zeitpunkt t einen Wald $\mathcal{R}_t$ im Reeb-Graphen $\mathcal{R}$ induzieren. Wir kontrahieren nun jede Zusammenhangskomponente

von $\mathcal{R}_t$ zu einem Knoten. Analog kontrahieren wir in allen Kantenbeschriftungen benachbarte Knoten mit demselben Zeitpunkt.

Diese Korrektur kann bereits während des Aufbaus des Reeb-Graphen geschehen. Dazu wird beim Einfügen eines Knoten geprüft, ob Nachbarknoten mit dem selben Zeitpunkt existieren und bei Bedarf entsprechend verschmolzen. Pro Knoten ist dies nur ein konstanter Mehraufwand und ändert somit die Gesamtlaufzeit asymptotisch nicht. Auch die obere Schranke für die Komplexität des Reeb-Graphen aus Korollar 5 gilt weiterhin, da die Anzahl der Knoten und Kanten durch das Kontrahieren nur abnimmt.

3.5 Robuste Gruppen

Abschließend betrachten wir noch eine Verallgemeinerung des Gruppenbegriffs, bei dem Objekte einer Gruppe kurzzeitig unverbunden sein können. Nach der bisher betrachteten Definition endet eine Gruppe sofort, wenn die Objekte sich trennen. Dabei ist es unerheblich, ob diese kurz darauf wieder zusammentreffen oder ob sie länger getrennt sind. Bei der Untersuchung von Gruppen, die über mehrere Stunden oder gar Tage bestehen, möchten wir möglicherweise Unterbrechungen von wenigen Minuten ignorieren. Dazu erweitern wir die Definition einer Gruppe um einen Parameter α, welcher die maximale Länge von Unterbrechungen beschreibt, die wir ignorieren wollen.

▶ **Definition 13.** Zwei Objekte x und y sind zum Zeitpunkt t *α-direkt verbunden*, wenn ein Zeitpunkt $t' \in [t - \alpha/2, t + \alpha/2]$ existiert, so dass x und y zum Zeitpunkt t' direkt verbunden sind.

Wir verallgemeinern die Definitionen von ε-verbunden und (maximalen) Gruppen, indem wir in deren Definitionen „direkt verbunden“ durch „α-direkt verbunden“ ersetzen. Zusätzlich betrachten wir den dynamischen Graphen G^α mit Knotenmenge $\mathcal{X}$, wobei zum Zeitpunkt $t \in [t_0, t_\tau]$ genau dann eine Kante zwischen x und y existiert, wenn die Objekte x und y zum Zeitpunkt t α-direkt verbunden sind. Beachte dabei, dass diese Definitionen für $\alpha = 0$ genau denen aus Abschnitt 3.2 entsprechen.

Zunächst wollen wir verstehen, wie sich der Graph G^α von dem Graphen $G = G^0$ unterscheidet. Aus Definition 13 folgt sofort, dass alle *connect*-Ereignisse um $\alpha/2$ früher und alle *disconnect*-Ereignisse um $\alpha/2$ später auftreten. Mit anderen Worten die Komponenten verschmelzen früher und trennen sich auch erst später wieder. Der zu G^α gehörenden Reeb-Graph $\mathcal{R}_{\alpha/2}$ unterscheidet sich also von $\mathcal{R} = \mathcal{R}_0$ darin, dass die *Merge*-Knoten $\alpha/2$ weiter vorne liegen und die *Split*-Knoten $\alpha/2$ weiter hinten. Lässt man α von 0 gegen unendlich laufen, wandern die *Merge*-Knoten zeitlich nach vorne und ziehen dabei ihre beiden Eingangskanten ähnlich wie ein Reißverschluss zusammen. Analog bewegen sich die *Split*-Knoten zeitlich nach hinten.

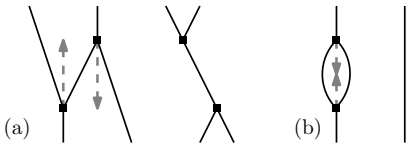


Abbildung 7 Mögliche Fälle, die auftreten können, wenn ein *Merge*- an einem *Split*-Knoten vorbei läuft.

Nun kann es geschehen, dass bei einem gewissen Wert γ in $\mathcal{R}_\gamma$ ein *Merge*-Knoten an einem *Split*-Knoten „vorbei läuft“. Dabei können die beiden Knoten durch eine oder durch zwei Kanten verbunden sein (s. Abbildung 7). Im ersten Fall tauschen die beiden Knoten die Reihenfolge und zwischen ihnen entsteht kurzzeitig eine größere Komponente, welche durch eine Kante im Reeb-Graphen repräsentiert wird. Der zweite Fall entspricht einer kurzzeitigen

Auftrennung der Objekte einer Gruppe, die nun ignoriert wird. Die beiden Knoten heben sich gegenseitig auf und übrig bleibt eine Kante. Die Werte von γ, an denen sich die Struktur des Reeb-Graphen $\mathcal{R}_\gamma$ ändert, nennen wir *kritisch*.

Die Idee ist es nun, aus dem gewöhnlichen Reeb-Graphen $\mathcal{R}$ schrittweise den Graphen $\mathcal{R}_{\alpha/2}$ zu konstruieren, indem wir die kritische Werte von γ in nicht-absteigender Folge betrachten und den Reeb-Graphen entsprechend anpassen. Wir beobachten dazu zunächst, dass ein *Merge*- und ein *Split*-Knoten nur dann als nächstes aneinander vorbei laufen können, wenn diese durch eine Kante verbunden sind. Daher genügt es auch nur diese Paare von Knoten zu speichern. Wir halten diese in einer Prioritätswarteschlange Q geordnet nach ihrem kritischen Wert von γ. Dieser entspricht genau der Differenz vom Zeitpunkt des *Merge*- und des *Split*-Knotens.

Als erstes initialisieren wir Q, indem wir über alle Kanten des Reeb-Graphen iterieren und diejenigen von einem *Split*- zu einem *Merge*-Knoten in Q einfügen. Dies kann in $\mathcal{O}(\tau n^2 \log \tau n^2)$ Zeit geschehen, da der Reeb-Graph laut Korollar 5 nur $\mathcal{O}(\tau n^2)$ viele Kanten besitzt.

Anschließend extrahieren wir das Minimum aus Q, solange für dessen kritischen Wert $\gamma \leq \alpha$ gilt. Wir führen die oben beschriebenen Änderungen am Reeb-Graphen durch und fügen potentiell weitere Ereignisse (aber höchstens konstant viele) in Q ein. Buchin et al. [2] haben folgende obere Schranke für die Anzahl der betrachteten Ereignisse bewiesen:

▶ **Lemma 14.** *Es werden $\mathcal{O}(\tau n^3)$ kritische Zeitpunkte betrachtet.*

Wir verzichten an dieser Stelle auf einen vollständigen Beweis. Grob gesagt verläuft der Beweis so, dass jeder *Split*-Knoten an höchstens n *Merge*-Knoten vorbei läuft. Denn wären es mehr Ereignisse, an denen er beteiligt ist, so wäre vorher bereits die Situation aus Abbildung 7(b) aufgetreten und der Knoten wäre ganz verschwunden. Pro kritischem Zeitpunkt, den wir betrachten, benötigen wir somit $\mathcal{O}(\log \tau n)$ Zeit, was folgende Gesamtlaufzeit ergibt.

▶ **Theorem 15.** *Den robusten Reeb-Graphen zu n Objekte, deren Trajektorien aus τ Strecken bestehen, lässt sich in $\mathcal{O}(\tau n^3 \log \tau n)$ berechnen.*

3.6 Zusammenfassung

Wir haben eine von Buchin et al. [2] entwickelte Datenstruktur zur Repräsentation von maximalen Gruppen von bewegten Objekten vorgestellt. Diese lässt sich in $\mathcal{O}(\tau n^3)$ Zeit berechnen, wobei n die Anzahl der Objekte und τ die Anzahl der Strecken auf deren Trajektorien bezeichnet. Zudem haben wir an einem Beispiel gesehen, dass der Algorithmus zum Aufbau dieser Datenstruktur möglicherweise die maximale Anzahl von $\Omega(\tau n^2)$ viele Ereignisse betrachtet, obwohl das Ergebnis nur aus einer Kante besteht.

Abschließend haben wir noch untersucht, wir die Gruppen robust gegen kurzzeitige Unterbrechungen gemacht werden kann. Dazu haben wir den Begriff einer Gruppe erweitert und dargelegt, wie die Datenstruktur zur trajektorienbasierten Gruppierung daran angepasst werden kann.

Referenzen

1 Marc Benkert, Joachim Gudmundsson, Florian Hübner, and Thomas Wolle. Reporting flock patterns. *Computational Geometry*, 41(3):111 – 125, 2008.

2 Kevin Buchin, Maike Buchin, Marc van Kreveld, Bettina Speckmann, and Frank Staals. Trajectory grouping structure. *Journal of Computational Geometry*, 6(1):75–98, 2015.

3 Joachim Gudmundsson, Marc Kreveld, and Bettina Speckmann. Efficient detection of patterns in 2D trajectories of moving points. *GeoInformatica*, 11(2):195–215, 2007.

4 Joachim Gudmundsson and Marc van Kreveld. Computing longest duration flocks in trajectory data. In *Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems*, GIS '06, pages 35–42. ACM, 2006.

5 Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, Christian S Jensen, and Heng Tao Shen. Discovery of convoys in trajectory databases. *Proceedings of the VLDB Endowment*, 1(1):1068–1080, 2008.

6 Panos Kalnis, Nikos Mamoulis, and Spiridon Bakiras. On discovering moving clusters in spatio-temporal data. In *Advances in spatial and temporal databases*, pages 364–381. Springer, 2005.

7 Irina Kostitsyna, Marc van Kreveld, Maarten Löffler, Bettina Speckmann, and Frank Staals. Trajectory Grouping Structure under Geodesic Distance. In *31st International Symposium on Computational Geometry (SoCG 2015)*, volume 34, pages 674–688, 2015.

8 Zhenhui Li, Bolin Ding, Jiawei Han, and Roland Kays. Swarm: Mining relaxed temporal moving object clusters. *Proceedings of the VLDB Endowment*, 3(1-2):723–734, 2010.

9 Salman Parsa. A deterministic $O(m \log m)$ time algorithm for the Reeb graph. *Discrete & Computational Geometry*, 49(4):864–878, 2013.

10 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

4 Dreiecke im External Memory

Sebastian Gieße

Zusammenfassung

In dieser Ausarbeitung werden Xiaocheng Hu, Yufei Tao und Chin-Wan Chung's Ergebnisse aus I/O-Efficient Algorithms on Triangle Listing and Counting [6] vorgestellt. Das Zählen und Aufzählen von Dreiecken in Graphen ist eine wichtige Suboperation vieler Graphenprobleme. Das Problem ist im internen Speicher ausgiebig analysiert. Nun stellen die Autoren einen neuen Algorithmus, für den Fall dass der Speicher nicht ausreicht um den kompletten Graphen zu speichern, vor. Dieser neue Algorithmus ist im Gegensatz zu bisherigen Algorithmen sowohl optimal bezüglich der Anzahl der I/O Operationen als auch bezüglich der CPU Laufzeit.

4.1 Einleitung

Ein *Dreieck* in einem Graphen G=(V,E) [1] ist eine aus 3 Knoten bestehende Clique im Graphen. Formen 3 Knoten $u, v, w \in V$ ein Dreieck, dann schreiben wir für dieses Dreieck Δ_{uvw}. Das *Triangle Listing Problem* fordert die Ausgabe aller Dreiecke in einem Graphen. Das *Triangle Counting Problem* fordert die Ausgabe der Anzahl der im Graphen vorhandenen Dreiecke.

Wir nehmen an, dass der Graph nicht in den internen Speicher passt. Entspricht M der Größe des Speichers dann gelte $M < c|E|$ für ein $c < 1$. Weiter bezeichne B die Größe eines Speicherblocks und K die Anzahl der Dreiecke.

Den Knotengrad eines Knoten u bezeichnen wir mit $deg(u)$. Außerdem hat jeder Knoten eine eindeutige Identifikationsnummer $id(u)$.

4.1.1 Motivation

Das Bestimmen aller Dreiecke bzw. der Anzahl der Dreiecke eines Graphen ist eine wichtige Suboperation vieler Graphenprobleme.

- Der *Clusterkoeffizient* C ist eine wichtige Kenngröße zur Charakterisierung eines Graphen. Er ist definiert als $C = \frac{3 \cdot Anzahl\ Dreiecke}{Anzahl\ verbundener\ Tripel}$, wobei ein Tripel uvw genau dann im Graphen vorliegt wenn Kanten $(u, v), (v, w) \in E$.
- Das *Dense Subgraph Mining* Problem beschäftigt sich mit der Bestimmung von Dense-Neighborhood Graphen. Ein Dense-Neighborhood Graph ist ein Teilgraph in welchem für jedes verbundene Knotenpaar (u, v) der Schnitt ihrer Nachbarschaften $N(v) \cap N(u)$ mindestens eine bestimmte Größe haben muss. Der effizienteste bekannte Algorithmus benutzt das Triangle Listing Problem als Subroutine (siehe [9]).

4.1.2 Stand der Forschung

Tabelle 1 zeigt eine Übersicht bisher entwickelter Algorithmen.

Der EM-CF Algorithmus [7] hat eine wesentliche Schwäche. Die Anzahl der I/Os ist komplett unabhängig von der Größe des verfügbaren Speichers. Gewünscht ist ein Algorithmus, der bei größer werdendem Speicher I/Os spart.

[1] Wenn nicht näher spezifiziert, ist ein Graph in dieser Arbeit immer ungerichtet und einfach.

Tabelle 1 Übersicht bisheriger Algorithmen

Algorithmen	I/O	
External Memory Compact Forward (EM-CF)	$O\left(\lvert E\rvert + \frac{\lvert E\rvert^{1.5}}{B}\right)$	
External Memory Node Iterator (EM-NI)	$O\left(\frac{\lvert E\rvert^{1.5}}{B \cdot log_{M/B}\frac{\lvert E\rvert}{B}}\right)$	
Graph Partition	$O\left(\frac{\lvert E\rvert^2}{MB} + \frac{K}{B}\right)$	Nur unter bestimmten Annahmen

Beim EM-NI [5] Algorithmus wirkt die Größe des Speichers in der Basis des Logarithmus im Nenner. Man spart also I/Os wenn man mehr Speicher hat. Allerdings ist die Auswirkung sehr gering.

Chu und Cheng [4] entwickelten einen auf Graph Partitionierung basierenden Algorithmus. Die Anzahl der I/Os ($O(|E|^2/MB + K/B)$) entspricht der optimalen Anzahl I/Os. Allerdings gilt diese Anzahl nur unter bestimmten Annahmen.

A_1: jeder erweiterte Subgraph passt in den Speicher (1)

Ein erweiterter Subgraph wird durch die Kanten eines Subgraphen induziert.

A_2: $|V| \leq M$ (2)

A_3: $M \in \Omega\left(\sqrt{E \cdot B}\right)$ (3)

Es werden die zwei Implementierungen Dominating-Set-based Graph Partitioning (DGP) und Randomized Graph Partition (RGP) vorgeschlagen. Für DGP muss A_1, A_2 und A_3, für RGP muss A_2 und A_3 erfüllt sein. In 4.3 wird ein Algorithmus mit der asymptotisch gleichen Anzahl I/Os vorgestellt, der für jede Eingabe gilt.

4.2 Grundlagen

4.2.1 Arboricity

Die arboricity eines Graphen ist eine wichtige Eigenschaft zur Beschreibung der Dichte eines Graphens.

▶ **Definition 1.** Die *arboricity* α eines Graphen G=(V,E) enspricht der kleinsten Anzahl kantendisjunkter Wälder, die E überdecken.

Dass die arboricity die Dichte eines Graphen beschreibt wird in Lemma 2 deutlich.

▶ **Lemma 2** (Nash-Williams 1964 [8]). *Für jeden Graphen $G = (V, E)$ mit mindestens 2 Knoten gilt:*

$\alpha = max_{\forall G'} density(G')$ (4)

wobei G'=(V',E') ein Subgraph von G mit $|V'| \geq 2$ und $density(G') = \lceil \frac{|E'|}{|V'|-1} \rceil$.

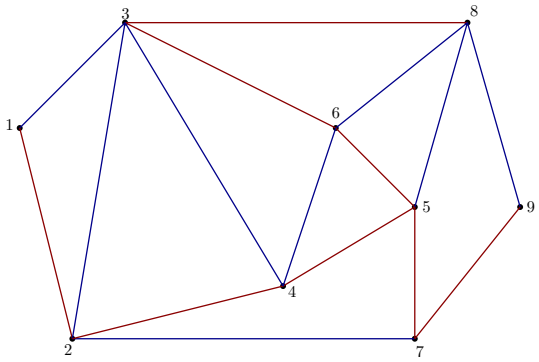


Abbildung 1 Ein Graph mit arboricity $\alpha = 2$. Die kantendisjunkten Wälder sind durch unterschieldiche Farben dargestellt.

Weiter gilt Korollar 3.

▶ **Korollar 3** (Chiba und Nishizeki 1985 [3]). *Für die arboricity α eines Graphen $G = (V, E)$ gilt:*

- $\alpha < \lceil \sqrt{|E|} \rceil$
- $\alpha = O(1)$ *(wenn G planar)*

Es besteht außerdem ein Zusammenhang zwischen der arboricity und der Anzahl der Dreiecke K in einem Graphen. Für ein Dreieck Δ_{uvw} müssen Kanten (u, v),(u, w) und (v, w) existieren. Die Anzahl der Dreiecke in denen (u, v) enthalten sein kann wird durch den kleineren Grad von u und v beschränkt. Weiter werden 3 Kanten für ein Dreieck benötigt. Es gilt also:

$$3K \leq \sum_{(u,v)\in E} min\{deg(u), deg(v)\}. \tag{5}$$

Die rechte Seite der Gleichung wird durch Lemma 4 beschränkt.

▶ **Lemma 4** (Chiba und Nishizeki 1985 [3]).

$$\sum_{(u,v)\in E} min\{deg(u), deg(v)\} \leq O(\alpha|E|) \tag{6}$$

Somit gilt

$$3K \leq O(\alpha|E|). \tag{7}$$

4.2.2 Orientierter Graph

Viele Algorithmen zur Auflistung von Dreiecken transformieren den gegebenen, einfachen, ungerichteten Graphen in einen sogenannten orientierten Graphen.

▶ **Definition 5** (Orientierter Graph). Für einen einfachen, ungerichteten Graphen G=(V,E) definiere eine Ordnung $\prec$ auf V, so dass $u \prec v$ wenn:

- $deg(u) < deg(v)$

- $deg(u) = deg(v)$ und $id(u) < id(v)$

$G^* = (V, E^*)$ ist orientierter Graph für G, wobei E^* für jede Kante $(u, v) \in E$ eine von u nach v gerichtete Kante enthält, wenn $u \prec v$.

Für einen orientierten Graphen $G^* = (V, E^*)$ gelten im Folgendem diese Schreibweisen:

- $N^+(v) = \{u|(v, u) \in E^*\}$ ist v's Menge von out-Nachbarn.
- $deg^+(v) = |N^+(v)|$ ist der Ausgangsgrad von v.

Die Ordnung $\prec$ sorgt dafür dass die Ausgangsgrade im orientiertem Graphen klein sind.

4.3 Massive-Graph-Triangulation Algorithmus

Im Folgenden wird der neue Triangle Listing Algorithmus *Massive Graph Triangulation (MGT)* vorgestellt. Wie in 4.3.5 gezeigt benötigt er $O(|E|^2/MB + K/B)$ I/Os und $O(|E|log|E| + |E|^2/M + \alpha|E|)$ CPU Zeit und ist somit optimal im Worst-Case.

4.3.1 Grundprinzip

Der MGT Algorithmus arbeit auf dem in 4.2.2 vorgestellten orientierten Graphen. Für ein Dreieck Δ_{uvw} gelte ohne Beschränkung der Allgemeinheit $u \prec v \prec w$. Dann bezeichnen wir u als *cone Knoten* und die Kante von v nach w als *Pivot Kante*.

Der Algorithmus arbeitet in Iterationen. In jeder Iteration werden zwei Schritte ausgeführt. In Schritt 1 wird eine Menge E_{mem} von Kanten in den Speicher geladen. Jede Kante wird dabei in genau einer Iteration geladen. Der Algorithmus terminiert wenn jede Kanten genau einmal in den Speicher geladen wurde. In Schritt 2 werden all die Dreiecke, deren Pivot Kante in E_{mem} liegt, ausgegeben.

Da jede Kante genau einmal in den Speicher geladen wurde und jedes Dreieck genau eine Pivot Kante hat, gibt der Algorithmus jedes Dreieck genau einmal aus und ist somit korrekt.

Input : $G = (V, E)$
Output : alle Dreiecke in G
1 $G^* = (V, E^*) \leftarrow oriented(G)$
2 **for** *es existieren Kanten, die noch nicht in den Speicher geladen wurden* **do**
3 | (1) Lade die nächsten cM Kanten in den Speicher, nenne sie E_{mem} [a]
4 | (2) Gebe Dreiecke, deren Pivot in E_{mem} liegt, aus
5 **end**

Algorithmus 1 : Masssive Graph Triangulation

[a] $c < 1$

4.3.2 Ein I/O effizienter Algorithmus

Um eine Aussage über die I/O Komplexität des MGT Algorithmus zu treffen wird zunächst eine grobe Version von Schritt 2 (Algorithmus 2) betrachtet. Hier werden bereits deutlich welche I/O Operationen ausgeführt werden. Die Details der Implementierung werden dann in Algorithmus 4 dargestellt.

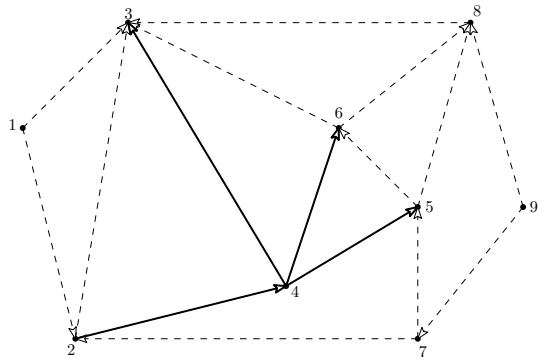


Abbildung 2 Beispiel für MGT Algorithmus. Die fetten Kanten repräsentieren E_{mem}.

Input : $G^* = (V, E^*)$, E_{mem}
Output : Dreiecke, deren Pivot in E_{mem} liegt
1 $V_{mem} \leftarrow$ induzierteKnoten(E_{mem})
2 **for** *each* $u \in V$ **do**
3 $\quad N_{mem}(u) \leftarrow leseAusSpeicher(N^+(u)) \cap V_{mem}$
4 $\quad S \leftarrow$ Kanten (u, v) mit $v \in N_{mem}(u)$
5 $\quad$ finde Dreiecke in $S \cup E_{mem}$, deren cone u ist
6 $\quad$ entferne N_{mem} und S aus Speicher
7 **end**

Algorithmus 2 : Schritt 2

Zunächst erstellt der Algorithmus die Menge V_{mem}, der durch E_{mem} induzierten Knoten. In Abbildung 2 ist also $V_{mem} = \{2, 3, 4, 5, 6\}$. Dann wird über jeden Knoten iteriert. Es wird die Menge $N_{mem}(u) = N^+(u) \cap V_{mem}$ gebildet. Dazu muss $N^+(u)$, also die Adjazenzliste von u, aus dem Speicher gelesen werden. Im Beispiel ist dann $N_{mem}(2) = \{3, 4\}$. Weiter wird die Menge S, als die Menge der Kanten von u nach $N_{mem}(u)$, gebildet. Diese wird mit E_{mem} vereint und in der entstanden Menge werden dann Dreiecke gesucht, deren cone Knoten u ist. Im Beispiel, für $u = 2$, wird also in $\{(2, 3), (2, 4), (4, 3), (4, 5), (4, 6)\}$ nach den Dreiecken gesucht, deren cone Knoten 2 ist. Es wird Δ_{234} gefunden.

In Schritt 2 wird für jeden Knoten dessen gesamt Adjazenzliste aus dem Speicher gelesen. Es werden also $O(|E|/B)$ I/Os benötigt. Der MGT Algorithmus liest in jeder Iteration $\Theta(M)$ Kanten und hat somit $\Theta(|E|/M)$ Iterationen. Zusätlich kommen noch $O(K/B)$ I/Os für die Ausgabe aller Dreiecke hinzu. Insgesamt braucht der MGT Algorithmus also $O(|E|^2/MB + K/B)$ I/Os.

4.3.3 Ein CPU effizienter Algorithmus

Im Folgendem betrachten wir Schritte 1 und 2 des MGT Algorithmus im Detail und erhalten dann die CPU Laufzeit des Algorithmus.

Zunächst begrenzen wir die Ausgangsgrade mit der *Kleiner-Grad-Annahme*:

$$\forall v \in V : deg^+(v) \leq cM/2 \tag{8}$$

Wir werden später sehen, dass der MGT Algorithmus trotz der Kleiner-Grad-Annahme für alle Graphen die gezeigte CPU Laufzeit und Anzahl I/Os hat.

Für die Implementierung von Schritt 1 fordern wir die *Alles-Oder-Nichts-Vorraussetzung*: Für alle $v \in V$ sind entweder alle ausgehenden Kanten in E_{mem} oder nicht. Algorithmus 3 erfüllt die Alles-Oder-Nichts-Vorraussetzung. Es wird über alle nichtmarkierten Knoten iteriert. Ein Knoten ist dann markiert wenn irgendwann im Laufe der Ausführung des MGT Algorithmus (beliebige Iteration) seine Ausgangskanten zu E_{mem} hinzugefügt wurden. Dann wird überprüft ob alle Ausgangskanten zu E_{mem} hinzugefügt werden können ohne die Beschränkung $|E_{mem}| \leq cM$ zu verletzen. Ist dies möglich werden die Kanten zu E_{mem} hinzugefügt und v markiert. Da die Alles-Oder-Nichts-Vorraussetzung erfüllt werden soll kann es dazu kommen dass E_{mem} kleiner als cM ist. Dies könnte die Zahl der Iterationen des MGT Algorithmus erhöhen. Durch die Kleiner-Grad-Annahme gilt aber $|E_{mem}| > cM/2$, da sonst die Kanten eines weiteren Knoten hinzugefügt werden könnten (außer in der letzen Iteration). Somit sorgt die Kleiner-Grad-Annahme dafür dass die Anzahl der Iterationen des MGT Algorithmus tatsächlich in $\Theta(|E|/M)$ liegt.

Input : $G^* = (V, E^*)$
Output : E_{mem} mit $cM/2 \leq |E_{mem}| \leq cM$
$E_{mem} \leftarrow \emptyset$
for *each* $v \in V$*:* v *nicht markiert* **do**
  if $|N^+(v)| + |E_{mem}| \leq cM$ **then**
    füge von v ausgehende Kanten zu E_{mem} hinzu
    markiere v
  end
  else
    break
  end
end
return E_{mem}

Algorithmus 3 : Schritt 1

Algorithmus 4 zeigt eine detailierte Implementierung von Schritt 2. Zunächst werden Hashtabellen für V_{mem}, V^+_{mem} und E_{mem} angelegt. Es gilt

$$V^+_{mem} = \{v \in V_{mem} | v \textit{ hat ausgehende Kante in } E_{mem}\} \tag{9}$$

Wegen der beschränkten Größe von V_{mem}, V^+_{mem} und E_{mem} können die Hashtabellen bei einem Speicherbedarf von $O(|E_{mem}|)$ Anfragen in $O(1)$ berechnen.

Dann wird über alle Knoten iteriert. Wir erhalten $N_{mem}(u)$ in $O(N^+(u))$ Zeit indem wir für alle Knoten der Adjazenzliste mit der Hashtabelle testen ob der Knoten in V_{mem} liegt. Weiter berechnen wir N^+_{mem} in $O(N^+(u))$ Zeit.

$$N^+_{mem}(u) = N_{mem}(u) \cap V^+_{mem} \tag{10}$$

In der folgenden inneren Schleife werden solche v und w gesucht, dass $v \in N^+_{mem}(u)$ und $w \in N_{mem}(u)$, also exisitieren Kanten von u nach v und von u nach w und v und w liegen in E_{mem}. Wenn nun $v \neq w$ und $(v, w) \in E_{mem}$ dann existiert das Dreieck Δ_{uvw} und die Pivot Kante liegt in E_{mem}. Die Laufzeit entspricht $O(N^+_{mem}(u) \cdot N_{mem}(u))$.

Insgesamt ergibt sich also für jedes $u \in V$ eine Laufzeit von $O(|N^+(u)|) + O(|N^+_{mem}(u)| \cdot |N_{mem}(u)|)$. Summiert man diese über alle Iterationen des MGT Algorithmus und alle Knoten ergibt sich, wie im Folgendem dargestellt, eine Laufzeit von $O(\alpha|E|)$.

Input : $G^* = (V, E^*)$, E_{mem}
Output : Dreiecke, deren Pivot in E_{mem} liegt

```
1  berechne V_mem und V+_mem aus E_mem
2  lege Hashtabellen V_mem, V+_mem und E_mem an
3  for each u ∈ V do
4  |  N_mem(u) ← leseAusSpeicher(N+(u)) ∩ V_mem
5  |  N+_mem(u) ← N_mem(u) ∩ V+_mem
6  |  for each v ∈ N+_mem(u) do
7  |  |  for each w ∈ N_mem(u) do
8  |  |  |  if v ≠ w und (v, w) ∈ E_mem then
9  |  |  |  |  gebe Δ_uvw aus
10 |  |  |  end
11 |  |  end
12 |  end
13 |  entferne N_mem und N+_mem(u) aus Speicher
14 end
```

Algorithmus 4 : Schritt 2

Um zu zeigen dass die Laufzeit in $O(\alpha|E|)$ liegt benötigen wir zunächst Lemma 6.

▶ **Lemma 6.** $\sum_{v\in V}(deg(v))^2 = O(\alpha|E|)$

Beweis. Für alle $v \in V$ gilt

$$(deg(v))^2 = deg^+(v) \cdot \sum_{u\in N^+(v)} 1 = \sum_{u\in N^+(v)} deg^+(v)$$

Also

$$\begin{aligned}
\sum_{v\in V}(deg(v))^2 &= \sum_{v\in V}\sum_{u\in N^+(v)} deg^+(v) \\
&= \sum_{(v,u)\in E^*} deg^+(v) \\
&\leq \sum_{(v,u)\in E^*} deg(v) \\
(nach\ Def.\ des\ orientierten\ Graphen) &= \sum_{(v,u)\in E} min\{deg(v), deg(u)\} \\
(nach\ Lemma\ 4) &= O(\alpha|E|)
\end{aligned}$$

◀

Die Laufzeit für jedes u in Schritt 2 ist $O(|N^+(u)|) + O(|N^+_{mem}(u)| \cdot |N_{mem}(u)|)$. Der erste Term addiert sich über alle $u \in V$ zu $O(|E^*|) = O(|E|)$. Über alle $\Theta(|E|/M)$ Iterationen ergibt sich daher eine CPU Laufzeit von $O(|E|^2/M)$.

Für die Analyse des Terms $O(|N^+_{mem}(u)| \cdot |N_{mem}(u)|)$ führen wir zunächst eine weitere Notation ein. Für $1 \leq i \leq h$ mit $h = \Theta(|E|/M)$ sei $N^+_{mem}(u, i)$ die Menge $N_{mem}(u)$ in der i-ten Iteration des MGT-Algorithmus.

▶ **Lemma 7.** $N^+_{mem}(u, 1), \ldots, N^+_{mem}(u, h)$ *sind paarweise disjunkt.*

Beweis. Wegen der Alles-Oder-Nichts-Vorraussetzung sind die V^+_{mem} der unterschiedlichen Iterationen des MGT Algorithmus paarweise disjunkt. Da $N^+_{mem}(u)$ eine Teilmenge von V^+_{mem} ist müssen daher alle $N^+_{mem}(u,1),\ldots,N^+_{mem}(u,h)$ paarweise disjunkt sein. ◀

Wegen Lemma 7 gilt

$$\sum_{i=1}^{h} |N^+_{mem}(u,i)| \leq |N^+(u)| = deg^+(u) \tag{11}$$

und somit ergibt sich insgesamt für den untersuchten Term

$$\begin{aligned}
&\sum_{i=1}^{h}\sum_{u\in V} O(|N^+_{mem}(u,i)|\cdot|N_{mem}(u)|)\\
=&\sum_{i=1}^{h}\sum_{u\in V} O(|N^+_{mem}(u,i)|\cdot|N^+(u)|)\\
=&\sum_{u\in V}\sum_{i=1}^{h} O(|N^+_{mem}(u,i)|\cdot|N^+(u)|)\\
=&\sum_{u\in V}(deg(u))^2\\
(\textit{nach Lemma 6})\ =&O(\alpha|E|)
\end{aligned}$$

Zusammengefasst ergibt sich eine CPU Laufzeit von $O(|E|^2/M+\alpha|E|)$. Hinzu kommt noch die Zeit um den Graphen in einen orientierten Graphen zu transformieren (nach [1]: $O(|E|log|E|)$. Der MGT Algorithmus hat so eine CPU Laufzeit von $O(|E|log|E|+|E|^2/M+\alpha|E|)$.

4.3.4 Entfernen der Kleiner-Grad-Annahme

Die Kleiner-Grad-Annhame fordert dass für alle v $deg^+(v)\leq cM/2$. In der Praxis ist dies meistens erfüllt. Da aber nicht garantiert ist dass die Kleiner-Grad-Annahme gilt muss noch der Fall bearbeitet werden dass es ein oder mehrere v gibt für die $deg^+(v)>cM/2$.

Dieser Fall wird durch eine Vorberechnung behandelt. Bevor G in den orientierten Graphen transformiert wird, wird G so in den (nicht-orientierten) Graphen G' transformiert dass G'^* die Kleiner-Grad-Annahme erfüllt. Alle Dreiecke in G, die der MGT Algorithmus in G' nicht finden kann, werden schon während dieser Vorberechnung ausgegeben.

Die Vorberechnung verläuft wie folgt.

1. Finde $u\in V$, sodass $deg(v)>cM/2$. Falls kein solches u existiert terminiere mit $G'=G$.
2. Lade $cM/2$ Kanten von u in den Speicher und nenne sie S.
3. Finde alle Dreiecke, die mindestens eine Kante in S haben. Gebe diese Dreiecke aus.
4. Entferne alle Kanten in S aus E und wiederhole ab (1).

Da in jeder Iteration $cM/2$ Kanten aus dem Speicher entfernt werden werden $O(|E|/M)$ Iterationen dieser Vorberechnung durchgeführt.

Die Menge S bildet einen Baum mit Höhe 2, u ist die Wurzel. Wir bezeichnen nun mit T die Blätter des Baumes. Alle in Schritt 3 auszugebende Dreiecke enthalten u und 1 oder 2 Knoten aus T. Im Folgenden seien Dreiecke mit 2 Knoten aus T Typ-1 Dreiecke und Dreiecke mit einem Knoten aus T Typ-2 Dreiecke. Typ-1 Dreiecke findet man durch Betrachten aller

Kanten in E. Seien für $(v, w) \in E$ $v, w \in T$ dann liegt ein Typ-1 Dreieck vor. Um Typ-2 Dreiecke zu finden betracheten wir die Adjazenzliste $N(v)$ für alle $v \neq u$. Wenn $u \notin N(v)$ machen wir mit dem nächsten Knoten weiter. Ansonsten bilden u und v mit allen Knoten in $N(v) \cap T$ ein Typ-2 Dreieck.

Somit kommt es pro Iteration zu $O(|E|/B)$ I/Os und eine Laufzeit von $O(|E|)$. Hinzu kommt noch die Ausgabe der Dreiecke. Insgesamt folgt also eine I/O Komplexität von $O(|E|^2/(MB) + K/B)$ und eine CPU Laufzeit von $O(|E|^2/M + K) = O(|E|^2/M + \alpha|E|)$. Die Vorberechnung übersteigt also asymptotisch weder die Anzahl der I/Os noch die Anzahl der CPU Operationen und somit gelten die zuvor gezeigten Schranken auch für den MGT Algorithmus inklusive Vorberechnung.

4.3.5 Optimal im Worst-Case

Wir wollen nun zeigen, dass kein Algorithmus asymptotisch besser sein kann als der MGT Algorithmus. Dazu zeigen wir dass sowohl die Anzahl der I/Os als auch der CPU Operationen asymptotisch nicht größer als die erforderliche Menge an Operationen für den Fall eines vollständigen Graphen ist. Dazu bestimmen wir die Anzahl der Dreiecke im vollständigen Graphen.

$$K = \binom{|V|}{3} = \Omega\left(|V|^3\right) = \Omega\left(|E|^{1.5}\right) = \Omega\left(\frac{|E|^2}{|V|}\right) = \Omega\left(\frac{|E|^2}{M}\right) \quad (f\ddot{u}r\ M \geq V) \tag{12}$$

Für jedes Dreieck ist eine I/O Operation nötig, also insgesamt $\Omega\left(|E|^2/(MB)\right)$ I/Os. Der MGT Algorithmus braucht

$$O\left(\frac{|E|^2}{MB} + \frac{K}{B}\right) = O\left(\frac{|E|^2}{MB} + \frac{|E|^2}{MB}\right) = O\left(\frac{|E|^2}{MB}\right) \tag{13}$$

I/Os und ist somit optimal.

Für die CPU Laufzeit gilt entsprechend auch dass eine Operation pro Dreieck, also $\Omega\left(|E|^2/M\right)$ Operationen benötigt werden. Der MGT Algorithmus benötigt

$$O\left(|E|log|E| + \frac{|E|^2}{M} + \alpha|E|\right) = O\left(\frac{|E|^2}{M} + \frac{|E|^2}{M} + \alpha|E|\right) \tag{14}$$

$$(nach\ Lemma\ 4) = O\left(\frac{|E|^2}{M} + \frac{|E|^2}{M} + \frac{|E|^2}{M}\right) \tag{15}$$

$$= O\left(\frac{|E|^2}{M}\right) \tag{16}$$

Operationen und ist somit auch optimal bezüglich der CPU Laufzeit.

4.4 Experimente

Auf einem Linux Rechner mit 2x3Ghz und 8GByte Arbeitsspeicher haben die Autoren in C++ die Algorithmen EM-CF, EM-NI, DGP, RGP und MGT implementiert. Den Implementierungen kann die Speichergröße M übergeben werden und es wird garantiert, dass nicht mehr als M bytes an Speicher benutzt werden.

Als Eingabe wurden die Graphen LJ, USRD, BTC, WebUK, und SubDomain benutzt. Tabelle 2 zeigt die Metadaten der benutzten Graphen.

Tabelle 2 Übersicht der in den Experimenten benutzten Graphen

	LJ	USRD	BTC	WebUK	SubDomain
$\|V\|$	4.846.609	23.947.347	164.660.997	62.338.347	89.247.739
$\|E\|$	42.851.237	28.854.312	386.411.047	938.715.528	1.940.007.864
$\|E\|/\|V\|$	8,84	1,20	2,35	15,06	21,74
$max_{v \in V} deg(v)$	20.333	9	1.637.619	48.822	3.032.590
$max_{v \in V} deg^{+}(v)$	686	4	645	5.692	10.695
disk size	364 M	403 M	4,1 G	7,5 G	15,1 G

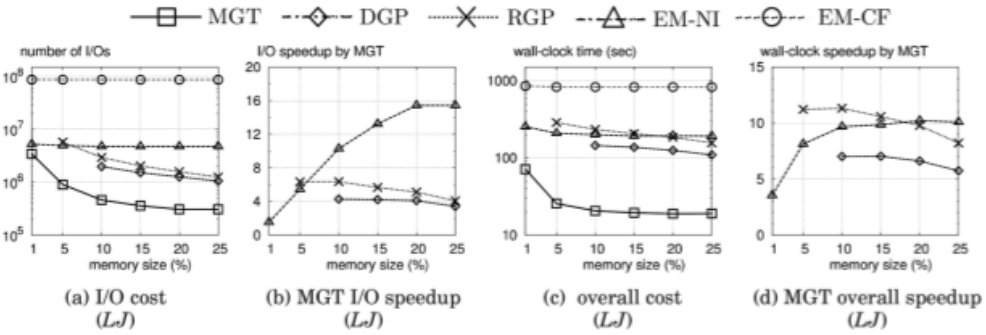


Abbildung 3 Ergebnisse der Experimente. Auswirkung der Größe des Speichers auf die Anzahl der I/Os und die Ausführungszeit. (Kopiert aus [6])

Zunächst wurde die Auswirkung der Speichergröße (gemessen an der Größe des Eingabegraphen) analysiert. Abbildung 3 zeigt die Ergebnisse für den Graphen LJ. Abbildung 3(a) zeigt dass weder EM-NI noch EM-CF merklich vom größeren Speicher profitieren. MGT, DGP und RGP profitieren vom größeren Speicher. Für jede Speichergröße benötigte der MGT Algorithmus weniger I/Os als jeder andere Algorithmus. Plot (b) zeigt den von MGT erzielten Speedup hinsichtlich der I/O Operationen gegenüber den anderen Algorithmen. Plots (c) und (d) zeigen das selbe hinsichtlich der Ausführungszeit. Insgesamt lässt sich feststellen, dass der MGT Algorithmus merklich effizienter als die anderen Algorithmen ist.

Außerdem wurden synthetisch Graphen erzeugt und die Auswirkung der Größe der Graphen analysiert. Dazu wurden die folgenden drei Modelle für synthetische Graphen benutzt:

- RANDom(RAND): Es wird ein Graph mit n Knoten und m zufälligen Kanten erzeugt. Doppelte Kanten werden anschließend eliminiert.
- Recursive MATrix (R-MAT) ([2]): Dieser Generator ist in der Lage eine große Bandbreite von Graphen zu generieren, deren Struktur stark der Struktur real vorkommender Graphen ähnelt.
- Small WORLD(S-WORLD) ([10]): n Knoten werden auf einem Kreis angeordnet und jeder Knoten mit seinen $m/2n$ nächsten linken und rechten Nachbarn verbunden. Mit Wahrscheinlichkeit p (hier $p = 0,01$) wird dann jede Kante durch eine zufällige Kante ersetzt. Schließlich werden doppelte Kanten eliminiert.

Abbildung 4(a)-(c) zeigt die Auswirkung der Größe des Graphen für die drei synthetischen Graphen. Plots (d)-(f) zeigen die entsprechenden Ausführungszeiten. Wieder sieht man dass der MGT Algorithmus immer effizienter ist als die anderen Algorithmen.

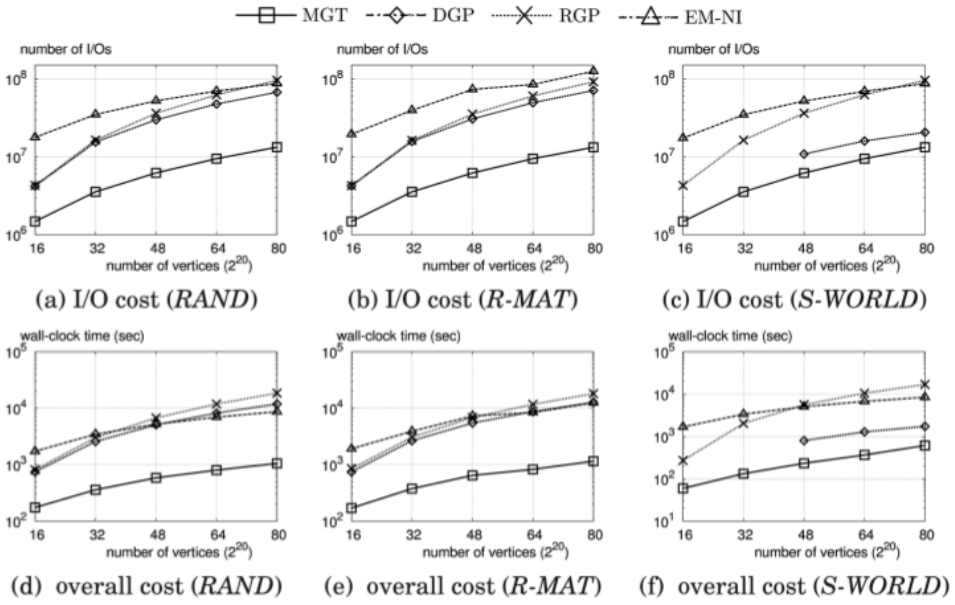


Abbildung 4 Ergebnisse der Experimente. Auswirkung der Größe des Graphen auf die Anzahl der I/Os und die Ausführungszeit. (Kopiert aus [6])

4.5 Zusammenfassung

Triangle Listing und Triangle Counting sind wichtige Operationen für Graphenalgorithmen. Obwohl sie im internen Speicher ausgiebig analysiert sind, entsprachen Algorithmen im external Memory nicht den Vorstellungen. Die Algorithmen waren entweder nicht in der Lage sich an die Menge des vorhandenen Speichers anzupassen oder waren nur unter bestimmten Vorraussetzungen gut. Mit dem hier vorgestellten MGT Algorithmus gibt es jetzt einen asymptotisch optimalen Algorithmus, der auch auf geläufigen Benchmark-Instanzen besser als alle bisher bekannten Algorithmen ist.

Referenzen

1 Alok Aggarwal, Jeffrey Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

2 Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

3 Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

4 Shumo Chu and James Cheng. Triangle listing in massive networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):17, 2012.

5 Roman Dementiev. *Algorithm engineering for large data sets.* PhD thesis, Saarland University, 2006.

6 Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. I/o-efficient algorithms on triangle listing and counting. *ACM Transactions on Database Systems (TODS)*, 39(4):27, 2014.

7 Bruno Menegola. An external memory algorithm for listing triangles. 2010.

8 C St JA Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, 1(1):12–12, 1964.

9 Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony KH Tung. On triangulation-based dense neighborhood graph discovery. *Proceedings of the VLDB Endowment*, 4(2):58–68, 2010.

10 Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

5 Genus, Baumweite und lokale Kreuzungszahl

Lars Gottesbüren

Zusammenfassung

Diese Seminararbeit basiert auf dem Artikel *Genus, Treewidth and Local Crossing Number* von Vida Dujmović, David Eppstein und David R. Wood [1]. Für auf topologischen Flächen gezeichnete Graphen G wird die scharfe asymptotische Abhängigkeit $\Theta\left(\sqrt{gkn}\right)$ für die Baumweite dieser Graphen von Größe n, Genus der Fläche g und Anzahl lokal notwendiger Kreuzungen k gezeigt. Dies verbessert die bisher beste bekannte obere Schranke $bw(G) \in O(k^{\frac{3}{4}}n^{\frac{1}{2}})$ für auf der Sphäre gezeichnete Graphen ($g = 0$) von Grigoriev und Bodlaender [4]. Im zweiten Teil wird gezeigt, wie man beliebige Graphen mit m Kanten auf der orientierbaren Fläche mit g Henkeln mit $O\left(\frac{m\log^2(g)}{g}\right)$ Kreuzungen pro Kante einbetten kann. Die lokale Kreuzungszahl kann also durch Betrachten einer komplizierteren Oberfläche um den Faktor $\frac{g}{\log^2(g)}$, also fast-linear reduziert werden. Das ist bis auf den Faktor $\log^2(g)$ optimal.

5.1 Einleitung und Grundlagen

In diesem Abschnitt werden die Begriffe Baumzerlegung und Baumweite, orientierbare Flächen und k- bzw. (g,k)-planare Graphen definiert. Anschließend werden einige Eigenschaften dieser anschaulich erläutert. Schlussendlich folgen noch Definitionen, die in den Beweisen benötigt werden. Die Begriffe und zugehörige Fragestellungen, auf die es hier eine Antwort gibt, werden parallel zu ihrer Erklärung in den entsprechenden Abschnitten dieses Grundlagenabschnitts algorithmisch motiviert. In Abschnitt 5.2 wird gezeigt, dass die Baumweite von k-planaren Graphen mit n Knoten in $\Theta\left(\sqrt{kn}\right)$, die von (g,k)-planaren Graphen in $\Theta\left(\sqrt{gkn}\right)$ liegt. Das verbessert die beste bisher bekannte Schranke $O(k^{\frac{3}{4}}n^{\frac{1}{2}})$ für k-planare Graphen und etabliert überhaupt erst ein Resultat für (g,k)-planare Graphen. Schlussendlich wird in Abschnitt 5.3 gezeigt, wie man beliebige Graphen mit n Knoten, m Kanten auf der orientierbaren Fläche mit g Henkeln mit $O\left(\frac{m\log^2(g)}{g}\right)$ als obere Schranke für die lokale Kreuzungszahl zeichnen kann. Die lokale Kreuzungszahl ist die Verbindung zwischen beiden Teilresultaten, welche vollkommen unabhängig voneinander bewiesen werden. Allerdings gibt die lokale Kreuzungszahlschranke für beliebige Graphen leider keine Neuerung bezüglich der Baumweite von beliebigen Graphen, wie später noch erklärt. Nach Erläuterung der Struktur dieser Ausarbeitung können wir nun die ersten Begriffe einführen und passende Fragestellungen motivieren.

5.1.1 Baumweite

Die Baumweite eines Graphen ist ein Maß für die Komplexität und wie ähnlich der Graph einem Baum ist. Eine hohe Baumweite bedeutet starken Zusammenhang, insbesondere haben Bäume Baumweite 1 (in Ausnamefällen 0). Man kann sich einen Graphen mit Baumweite k ungefähr aber eben nicht genau, als eine angedickte Version eines Baumes mit $(k+1)$-Cliquen statt Kanten vorstellen. Mehr zu dieser Intuition folgt gleich nach der richtigen Definition von Baumweite bei der Erläuterung der Verwandtschaft von Baumweite mit k-Bäumen. Die Baumweite ist ein interessantes Maß, da es für einige NP-schwere Probleme Algorithmen auf Baumzerlegungen gibt, deren Laufzeit exponentiell von der Weite der

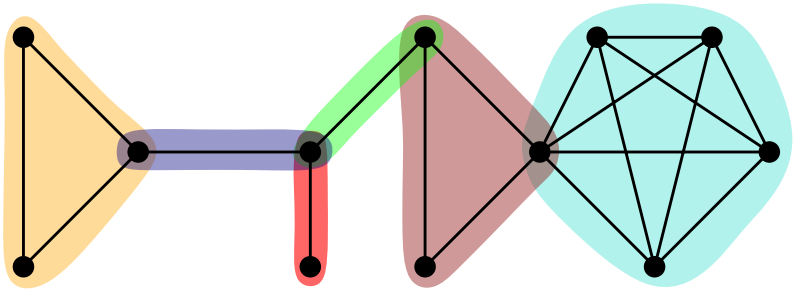


Abbildung 1 Eine optimale Baumzerlegung mit Baumweite 4. Die Taschen sind durch die farbigen Gebiete dargestellt. Sie sind verbunden, wo sich die Gebiete überlappen.

Zerlegung aber polynomiell von der Größe des Graphen abhängt. Zu diesen Problemen zählen unter anderem das Bestimmen der chromatischen Zahl beziehungsweise einer zugehörigen optimalen Färbung, Clique bzw. Independent Set sowie das Hamiltonkreisproblem. Hat eine Klasse von Graphen konstante oder logarithmisch in der Knotenzahl beschränkte Baumweite, sind dies Polynomialzeitalgorithmen für diese Klasse.

▶ **Definition 1** (Baumzerlegung (engl. tree decomposition))**.** Zu einem Graphen G ist eine *Baumzerlegung* von G ein Baum T, dessen Knoten Teilmengen der Knotenmenge von G sind. Die Knoten von T werden häufig *Taschen* genannt. Die Struktur von T muss außerdem die folgenden Bedingungen erfüllen:

- Jeder Knoten von G liegt in mindestens einer Tasche
- Die Endpunkte jeder Kante von G liegen in mindestens einer gemeinsamen Tasche
- Der Subbaum von T, induziert durch die Taschen, die einen festen Knoten $v \in V(G)$ gemeinsam haben, ist zusammenhängend

▶ **Definition 2** (Baumweite (engl. treewidth))**.** Die *Weite* einer Baumzerlegung T ist die Kardinalität der größten Tasche minus 1. Die Subtraktion von eins ist eine Schönheitskorrektur, damit schlussendlich Bäume Baumweite eins (oder null in Ausnahmen) statt zwei haben. Die *Baumweite* $bw(G)$ eines Graphen G ist die geringste Weite von allen möglichen Baumzerlegungen.

Abbildung 1 zeigt eine beispielhafte, optimale Baumzerlegung mit Weite 4. Zusätzlich zu ihrer Verbindung zur Algorithmik von NP-schweren Problemen ist die Baumweite stark korreliert mit der Größe $sn(G)$ eines minimalen Separators. Robertson und Seymour [10] zeigten, dass jeder Graph G einen Separator der Größe $bw(G) + 1$ hat. Norin und Dvořák [3] zeigten, dass $bw(G) \leq 105sn(G)$, die beiden Kennzahlen also insgesamt proportional zueinander sind.

Zur Stärkung der Intuition gibt es eine Charakterisierung der Baumweite über partielle k-Bäume. Ein Graph hat eine Baumzerlegung mit Weite k, genau dann wenn er ein partieller k-Baum ist. Dabei ist ein *k-Baum* ein Graph, der aus dem vollständigen Graphen K_{k+1} in beliebig vielen Schritten entsteht, indem in jedem Schritt ein neuer Knoten hinzugefügt und zu einer k-Clique des k-Baums aus dem vorherigen Schritt verbunden wird. Jede inklusionsmaximale Clique in einem k-Baum hat also exakt $k + 1$ Knoten. Beliebige Kantenteilgraphen (Kanten löschen ist erlaubt, Knoten löschen nicht) von k-Bäumen heißen *partielle k-Bäume*. Eine Richtung der Charakterisierung lässt sich einsehen, indem man eine Baumzerlegung für einen partiellen k-Baum G berechnet. Dazu betrachtet man die Knoten in der umgekehrten Reihenfolge, in der sie per Konstruktion eingefügt wurden. Man fängt also mit dem zuletzt

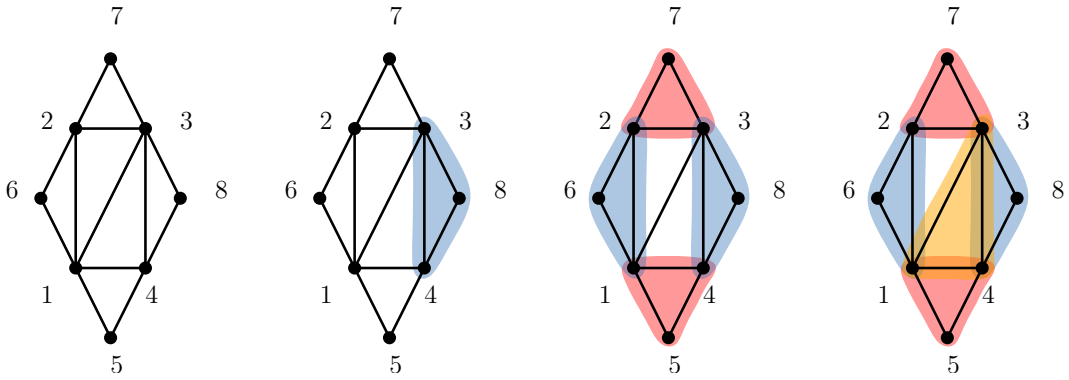


Abbildung 2 Verlauf des Algorithmus zum Bestimmen einer Baumzerlegung von k-Bäumen. Die Zahlindizes geben die Reihenfolge an, in der die Knoten bei der k-Baum-Konstruktion eingefügt wurden (hier $k = 2$). Der letzte Schritt und einige Zwischenschritte wurden ausgelassen.

hinzugefügten Knoten an. Für den momentan betrachteten Knoten v erschafft man eine Tasche, die diesen Knoten sowie seine höchstens k Nachbarn enthält. Anschließend wird v gelöscht und die erschaffene Tasche wird als Blatt an die Baumzerlegung angehängt, die durch Fortführung der Prozedur auf dem verbleibenden Graphen entsteht. Dieser Vorgang wird in Abbildung 2 noch einmal beispielhaft verdeutlicht. Genau umgekehrt erhält man aus einer Baumzerlegung eines Graphen G mit Weite k einen partiellen k-Baum, indem man alle Knoten einer Tasche zu einer Clique verbindet. Allerdings gibt es für beliebige Baumzerlegungen keine eins-zu-eins Korrelation zwischen den Taschen der Zerlegung und den Knoten des partiellen k-Baums. Es ist beispielsweise völlig legitim die gleiche Tasche beliebig oft hintereinander zu hängen.

Für das Verständnis der Beweise in Abschnitt 5.2 ist kein tieferes Verständnis des Baumzerlegungskonzepts nötig. Die Bedingungen an die Zerlegung werden im Beweis direkt adressiert. Wer dennoch eine tieferschürfende Intuition für den Begriff entwickeln möchte, sei verwiesen auf das Skript zur Vorlesung “Baumzerlegungen, Algorithmen und Logik“ von Isolde Adler an der Universität Frankfurt [1].

Ein wichtiges Hilfsmittel wird die Schichtbaumweite (engl. layered treewidth) von Graphen sein; eine mit der Baumweite ganz offensichtlich verwandte Kennzahl.

▶ **Definition 3** (Schichtbaumweite). Gegeben einen Graph G, sei $V_1 \dot\cup V_2 \dot\cup \cdots \dot\cup V_t = V(G)$ eine Partitionierung von G in Schichten sodass für jede Kante vw mit $v \in V_i, w \in V_j$ die V_i und V_j benachbart sind, also $|i - j| \leq 1$. Eine solche Partitionierung bieten zum Beispiel die Ebenen einer Breitensuche ausgehend von einem beliebigen Startknoten. Die *Schichtweite* (engl. layered width) einer Baumzerlegung ist die kleinste natürliche Zahl l sodass jede Tasche der Baumzerlegung pro Schicht höchstens l Knoten enthält. Die *Schichtbaumweite* ist dann unter allen möglichen Zerlegungen und Partitionierungen wie oben die kleinste aller Schichtweiten. Schichtbaumweite ist ein anderes Maß als Baumweite. Nimmt man die triviale Partitionierung in eine Schicht und eine bezüglich Baumweite optimale Baumzerlegung ist

[1] Skript zur Vorlesung *Baumzerlegungen, Algorithmen und Logik*, gelesen von Isolde Adler im Wintersemester 2009 an der Universität Frankfurt: http://www.tdi.informatik.uni-frankfurt.de/~adler/09-BAL/index.html#skript

die Schichtweite gerade die Baumweite plus eins. Für komplexere Partitionierungen hingegen ist die Schichtweite eine kleinere Zahl. Es kann sogar sein, dass eine für Schichtbaumweite optimale Baumzerlegung nicht die optimale Baumweite liefert. Im weiteren Verlauf der Seminararbeit wird der Begriff Schichten für eine Partitionierung mit obigen Bedingungen verwendet.

5.1.2 Orientierbare Flächen

Ein vielstudiertes Problem ist das Zeichnen von Graphen in der Ebene, der topologischen Fläche, die sich der Mensch am leichtesten bildlich vorstellen kann. Zeichnungen von Graphen dienen häufig der Informationsvisualisierung, da der Mensch in Bildern und nicht zwingend in Sprache oder Verbindungen denkt. Es gibt allerdings noch andere topologische Flächen als die Ebene und kreuzungsarme Einbettungen von Graphen in diese finden Anwendung in Gebieten wie der Computergrafik, dem Entwurf von Netzwerktopologien oder dem Chipbau. Hier wird speziell die Klasse der orientierbaren Flächen betrachtet, eine Unterklasse der kompakten, zusammenhängenden, zweidimensionalen Mannigfaltigkeiten. Aus Gründen der Anschaulichkeit wird sie hier induktiv wie folgt definiert und nicht-orientierbare zweidimensionale Mannigfaltigkeiten werden nicht weiter betrachtet.

▶ **Definition 4** (Orientierbare Fläche mit g Henkeln)**.** Die *orientierbare Fläche* mit 0 Henkeln ist die Sphäre, welche topologisch äquivalent zur Ebene ist. Die *orientierbare Fläche* mit g Henkeln entsteht aus der orientierbaren Fläche mit $g-1$ Henkeln, indem an zwei beliebigen Stellen auf der Oberfläche Kreisscheiben ausgeschnitten und die kreisförmigen Enden eines Zylinders angeklebt werden.

Diese Definition liefert für jede Henkelzahl $g \in \mathbb{N}$ eine (unter Homöomorphie) eindeutige Oberfläche, ein grundlegendes Resultat aus der Topologie, welches zum Beispiel in der Vorlesung *Einführung in Geometrie und Topologie* [2] bewiesen wird. In der Literatur spricht man manchmal von Löchern statt Henkeln. Als Notation wird Σ_g für die orientierbare Fläche mit g Henkeln verwendet. Abbildung 3 zeigt Beispielzeichnungen von Σ_1, Σ_2 und Σ_3. Das *Eulergenus* von Σ_g ist $2g$, eine Invariante von Flächen. Neben den orientierbaren Flächen haben auch die nicht-orientierbaren Flächen ein Eulergenus; der vereinte Begriff hilft dabei Zusammenhänge für beide Flächentypen zu formulieren. In dieser Seminararbeit werden orientierbare Flächen als Oberflächen zum Graphenzeichnen genutzt, wie schon anfangs angedeutet. Dabei werden Schranken abhängig von der Anzahl Henkel angegeben. Genauso könnte man das vom Eulergenus abhängig interpretieren. Da in der Literatur der Begriff *Genus* uneinheitlich genutzt wird, sprechen wir hier einfach von der Anzahl Henkel.

5.1.3 k- bzw. (g,k)-planare Graphen

Die Klasse der planaren Graphen und ihre Eigenschaften sind seit langer Zeit Bestandteil der Forschung. Einige nicht-planare Graphen sind nicht weit entfernt von Planarität, in dem Sinne, dass eventuell nur wenige Kreuzungen pro Kante die Planarität verhindern. Ein Maß dafür ist die *lokale Kreuzungszahl*, welche die notwendigen Kreuzungen pro Kante für Zeichnungen in der Ebene zählt.

---

[2] Vorlesung *Einführung in Geometrie und Topologie* an der Fakultät für Mathematik des KIT, gelesen von Prof. Roman Sauer im Wintersemester 2014/2015

[3] Bildquellen: Oleg Alexandrov bei https://commons.wikimedia.org/wiki/File:Torus_illustration.png, https://commons.wikimedia.org/wiki/File:Double_torus_illustration.png, https://commons.wikimedia.org/wiki/File:Triple_torus_illustration.png

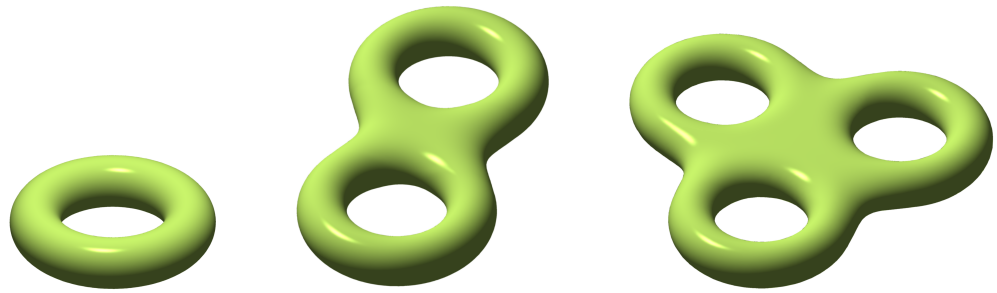


Abbildung 3 Die orientierbaren Flächen Σ_1, Σ_2 und Σ_3. Bildquellen [3]

▶ **Definition 5.** Ein Graph heißt k-planar, wenn er mit höchstens k Kreuzungen pro Kante in der Ebene gezeichnet werden kann.

Außerdem interessant zu betrachten können kreuzungsarme Zeichnungen auf allgemeineren zweidimensionalen Flächen sein. Zu den Anwendungsfällen gehören unter anderem das Design von Netzwerktopologien, der Chipbau sowie allgemein die Veranschaulichung von Informationen.
Nach Heawood's Formel gilt für jeden Graphen H der auf einer Fläche mit Eulergenus e kreuzungsfrei zeichenbar ist $\chi(H) \leq \lfloor \frac{7+\sqrt{1+24e}}{2} \rfloor$ und laut Ringel [9] ist diese Schranke scharf für alle Oberflächen bis auf die Ebene (orientierbar, 4 Farben) und die Klein'sche Flasche (nicht-orientierbar, 6 Farben). Es ist dementsprechend interessant Graphen auf zweidimensionalen Flächen kreuzungsfrei zu zeichnen. Als Erweiterung davon kann man nun die Frage stellen, wie viele Kreuzungen pro Kante in Kauf genommen werden müssen, für den Fall dass ein Graph nicht kreuzungsfrei zeichenbar ist.

▶ **Definition 6.** Ein Graph heißt (g, k)-planar, wenn er mit höchstens k Kreuzungen pro Kante auf der orientierbaren Fläche mit g Henkeln gezeichnet werden kann.

Intuitiv ist klar, dass Zeichnungen auf Flächen höherer Komplexität mit weniger Kreuzungen pro Kante auskommen. Im Fall der orientierbaren Flächen also, lässt sich die lokale Kreuzungszahl (bzgl. dieser Fläche, nicht der Ebene) verringern, indem weitere Henkel hinzugefügt werden. Eine Möglichkeit dies umzusetzen wird in Abschnitt 5.3 gezeigt.

5.1.4 Werkzeug

Die folgenden Definitionen stellen Werkzeug für das Führen der Beweise in dieser Seminararbeit bereit. Es ist ratsam sich zunächst einen groben Überblick zu verschaffen, bevor die Details der Beweise nachvollzogen werden. Die Definition der Separatoren wird für die Beweise zur Baumweite in Abschnitt 5.2 benötigt, während Expandergraphen und Kreisrang beim Zeichnen auf orientierbaren Flächen in Abschnitt 5.3 eine Rolle spielen werden.

▶ **Definition 7** (Separator)**.** Eine Knotenteilmenge $S \subset V(H)$ eines Graphen H ein *Separator*, wenn $H - S$ mehr Zusammenhangskomponenten als H hat.

Etwas spezieller ist bereits die Definition von ϵ-Separatoren.

▶ **Definition 8** (ϵ-Separator)**.** Gegeben einen Graphen H, $\epsilon \in (0, 1)$, ist eine Menge $S \subset V(H)$ ein *ϵ-Separator*, wenn jede Zusammenhangskomponente von $G - S$ höchstens $\epsilon|V(H)|$ Knoten enthält.

▶ **Definition 9** (Expandergraph)**.** Ein *Expandergraph* ist ein Graph H, sodass es für jedes $\epsilon \in (0,1)$ ein $\beta > 0$ gibt, sodass jeder ϵ-Separator von H mindestens $\beta|V(H)|$ Knoten enthält. Oft spricht man bei Expandergraphen von d-regulären Graphen für ein konstantes $d \in \mathbb{N}$, allerdings nicht immer.

Einige Möglichkeiten zur Berechnung von Expandergraphen finden sich in Hoory et al.[6].

▶ **Definition 10** (Kreisrang)**.** Der *Kreisrang* $r_c(G)$ eines Graphen $G = (V, E)$ ist die kleinste Anzahl Kanten, die man löschen muss, um einen kreisfreien Graphen zu erhalten. Sei c die Zahl der Zusammenhangskomponenten von G. Jeder Wald, der die Komponenten von G spannt, hat wie bekannt $n-c$ Kanten. Dementsprechend ist $r_c(G) = |E|-|V|+c$, unabhängig von der Struktur von G. Im Folgenden werden meist zusammenhängende Graphen betrachtet, die dementsprechend Kreisrang $|E| - |V| + 1$ haben.

5.2 Baumweite von k-planaren und (g,k)-planaren Graphen

Der Abschnitt Baumweite behandelt die asymptotisch scharfe Charakterisierung der Baumweite für die Klasse der k-planaren Graphen und der (g,k)-planaren Graphen. Begonnen wird mit dem einfacheren Fall der k-planaren Graphen, um die gewonnenen Erkenntnisse dann auf orientierbare Flächen auszuweiten.

5.2.1 Baumweite k-planarer Graphen

Dieser Abschnitt beschäftigt sich mit dem Beweis des folgenden Theorems, das die Baumweite von k-planaren Graphen asymptotisch scharf charakterisiert.

▶ **Theorem 11.** *Sei G ein k-planarer Graph mit n Knoten. Dann gilt*

$$bw(G) \in \Theta\left(\min\left\{\sqrt{kn}, n\right\}\right)$$

Beweis des Theorems. Die obere Schranke des Theorems wird durch Kombination der zwei folgenden Lemmata bewiesen. Die Schranke n ist trivial.

▶ **Lemma 12** (Norin [2])**.** *Jeder Graph mit n Knoten und Schichtbaumweite l hat höchstens Baumweite $2\sqrt{nl}$.*

Beweis Lemma 12. Gegeben eine Partitionierung $V_1 \dot\cup V_2 \dot\cup \cdots \dot\cup V_r \subset V(G)$ in Schichten und zugehörige Baumzerlegung T mit Schichtbaumweite l, bildet jede der inneren Schichten V_i mit $i \in \{2, 3, \ldots, m-1\}$ einen Separator S_i zu $(\bigcup_{j<i} V_j, \bigcup_{k>i} V_k)$, den Schichten oberhalb bzw. unterhalb von V_i, unabhängig vom Zusammenhang von G. Ein Beispiel dafür sieht man in Abbildung 4. Entfernt man statt nur einer Schicht direkt r innere Schichten, die nicht aufeinander folgen, so zerlegt man den Graph in mindestens $r+1$ viele Zusammenhangskomponenten. Nenne den Separator bestehend aus den r inneren, nicht aufeinander folgenden Schichten S. Sei $s_{\max}$ der größte Abstand zwischen zwei Separatorschichten, also unter allen Separatorschichten V_i und V_j ist $s_{\max} = \max_{V_i,V_j}(|i-j|)$. Jede Zusammenhangskomponente für sich hat höchstens Baumweite $l \cdot s_{\max}$. Durch Hinzufügen des Separators S zu jeder Tasche erhält man eine Baumzerlegung von G mit Weite $l \cdot s_{\max} + |S|$. Dieser Term wird minimiert, wenn zum einen $s_{\max}$ klein bleibt, also der Abstand zwischen verschiedenen Separatorschichten stets gleich ist. Zum anderen wird der Term minimiert, wenn der Separator S kleingehalten wird.

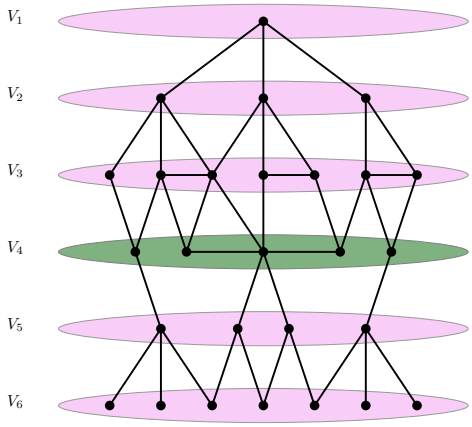
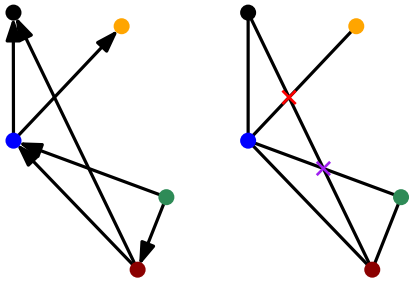


Abbildung 4 Eine Partitionierung in 6 Schichten $V_1, \ldots, V_6$. Man sieht, dass jede innere Schicht $V_2, \ldots, V_5$ einen Separator bildet.

Im Folgenden wird ein passender Separator W_i der Größe $\sqrt{nl}$ konstruiert, der dafür sorgt, dass jede Zusammenhangskomponente von $G - W_i$ gerade $\sqrt{\frac{n}{l}}$ Schichten enthält. Jede Zusammenhangskomponente benötigt also $l \cdot \sqrt{\frac{n}{l}} = \sqrt{nl}$ Knoten pro Tasche, was dann eine Baumzerlegung mit Weite höchstens $2\sqrt{nl}$ für G bedeutet, nach obiger Konstruktion. Sei $p = \lceil \sqrt{\frac{n}{l}} \rceil$. Für jede Zahl $j \in \{1, \ldots, p\}$ fasse angefangen bei V_j jede p-te Schicht in W_j zusammen, also $W_j = V_j \cup V_{j+p} \cup \cdots$. Da $\sum_{j=1}^{p} |W_j| = n$ gibt es ein W_i mit $|W_i| \leq \frac{n}{p} \leq \sqrt{nl}$. Jedes dieser W_j ist ein Separator und jede Zusammenhangskomponente von $G - W_i$ liegt in $p - 1$ aufeinanderfolgenden Schichten. ◀

▶ **Lemma 13** (Lemma). *Die Schichtbaumweite von k-planaren Graphen ist höchstens $6(k+1)$.*

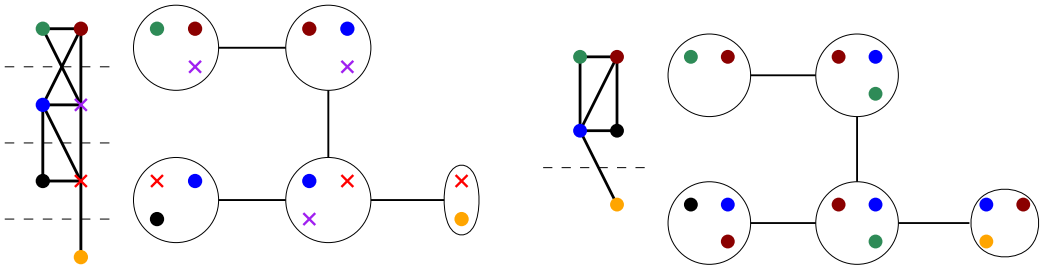
Beweis Lemma 13. Zeichne G mit höchstens k Kreuzungen pro Kante in der Ebene und weise jeder Kante $\{u, v\}$ eine beliebige aber eindeutige Orientierung (u, v) oder (v, u) zu. Der Nutzen der Orientierung wird im Laufe des Beweises ersichtlich. Die Zeichnung wird nun planarisiert, indem auf jede Kreuzung zweier Kanten ein neuer Knoten von Grad 4 gesetzt wird. Der resultierende Graph G' ist planar und hat nach [2] Schichtbaumweite höchstens drei. Abbildung 5 zeigt einen aus Anschaulichkeitsgründen planaren Beispielgraphen mit zwei Kreuzungen.
Aus einer Baumzerlegung T' von G' und einer Partitionierung in Schichten $V'_1, \dot\cup \cdots \dot\cup V'_r \subset V(G')$ lässt sich eine Baumzerlegung T für G konstruieren. Dazu wird in jeder Tasche von T' jeder Kreuzungsknoten durch die Startknoten der zwei sich an ihm kreuzenden Kanten ersetzt. In Abbildung 6 ist der Schritt von einer Baumzerlegung und Schichteneinteilung für G zu Baumzerlegung und Schichteneinteilung für G visualisiert, am Beispiel des Graphen aus Abbildung 5. Der lilafarbene und der rote Knoten sind Kreuzungsknoten. Unter anderem wird in jeder Tasche der lilafarbene Knoten, durch den dunkelroten und den grünen Knoten ersetzt. Der rote Knoten wird durch den orangenen und ebenso den dunkelroten Knoten ersetzt. In der Schichteneinteilung fallen der lilafarbene und der rote Knoten weg. Außerdem werden $k + 1 = 3$ aufeinanderfolgende Schichten zu einer Schicht zusammengefasst.
Die Knoten von G sind durch T abgedeckt, da $V(G) \subset V(G')$. Jede orientierte Kante (u, v) wird entweder nicht gekreuzt, dann muss sie bereits in einer Tasche von T' vorkommen oder es gibt in G' einen zu v adjazenten Kreuzungsknoten. Der Kreuzungsknoten wurde in allen Taschen durch u und einen zweiten Startknoten ersetzt; daher ist die Kante uv



(a) Ein (orientierter) Graph mit zwei Kreuzungen

(b) nach Platzieren von Kreuzungsknoten

Abbildung 5 Beispielhafte Planarisierung an einem (bereits planaren) Graphen mit zwei Kreuzungen, also $k = 2$.



(a) Links Schichteneinteilung, rechts Baumzerlegung für G'

(b) Zusammengefasste Schichten und resultierende Baumzerlegung für G

Abbildung 6 Die Baumzerlegung in (b) entsteht aus der in (a) indem in jeder Tasche die Kreuzungsknoten durch die zwei Startknoten der sich an ihr kreuzenden Kanten ersetzt werden. Die Schichteneinteilung entsteht durch Weglassen der Kreuzungsknoten und Zusammenfassen von $k = 2$ aufeinanderfolgenden Schichten. In (b) sind rein technisch gesehen die zwei oberen Tasche überflüssig.

durch T abgedeckt. Für jeden Knoten v ist der induzierte Teilgraph von G' auf v plus allen Kreuzungsknoten auf von v wegorientierten Kanten $(v, x) \in E(G)$ verbunden. Daher sind die zugehörigen Taschen in T' verbunden, nach der zweiten und dritten Eigenschaft einer Baumzerlegung. Diese sind nach Konstruktion genau die Taschen von T die v enthalten; letztere sind also in T verbunden. Daraus folgt, dass T eine gültige Baumzerlegung von G ist. Außerdem befinden sich in jeder Tasche höchstens $6 = 3 \cdot 2$ Knoten pro Schicht. Die bisherigen Schichten sind noch nicht passend, da sich Kanten von G über mehrere der V'_i spannen können.
Die Endpunkte jeder Kante $\{u, v\} \in E(G)$ sind in G' über die Kreuzungsknoten durch einen höchstens $k+2$ Knoten langen Pfad verbunden. Daher beträgt die Differenz der Schichtindizes von u und v höchstens $k + 1$. Durch das Zusammenfassen von $k + 1$ aufeinanderfolgenden Schichten entstehen gültige Schichten für G. Jede Schicht für G enthält 6 Knoten pro Schicht für G' und $k + 1$ Schichten für G', was eine Schichtbaumweite von höchstens $6(k + 1)$ ergibt. ◀

Die untere Schranke wird durch eine Menge von Graphen bewiesen, die asymptotisch mindestens Baumweite $c\sqrt{kn}$ haben; für eine positive Konstante c. Der nachfolgende Beweis ist in seiner Prägnanz und Klarheit nicht zu übertreffen, daher wird er in gleicher Form wie

in [1] wiedergegeben.

▶ **Lemma 14.** *Es gibt k-planare Graphen G' mit n' Knoten und $bw(G') \geq \sqrt{kn'}$.*

Beweis. Man betrachte einen 3-regulären Graphen G auf n Knoten mit $bw(G) \geq \epsilon n$ für eine positive Konstante ϵ, siehe Grohe und Marx [5] für ein Beispiel. Konstruiere den k-planaren Graphen G' mit n' Knoten aus G indem jede Kante durch einen Pfad ersetzt wird, der höchstens $\frac{3n}{2k}$ Knoten lang ist. Dadurch wird die Anzahl Kreuzungen pro Kante auf die Kanten des zugehörigen Pfades verteilt. Aus der Länge der Pfade folgt dass G' höchstens $n' \leq n + \frac{3n}{2k}\frac{3n}{2} = n + \frac{9n^2}{4k}$ Knoten hat. Nach Lozin und Rautenbach [8] verändert die Ersetzung von Kanten durch Pfade die Baumweite nicht, also gilt $bw(G') \geq bw(G) \geq \epsilon n \geq \frac{2\epsilon}{3}\sqrt{kn'}$. ◀

◀

5.2.2 Baumweite (g, k)-planarer Graphen

Für (g, k)-planare Graphen ergibt sich ein ähnliches Resultat wie für k-planare Graphen. Einzig der Faktor $\sqrt{g}$ kommt hinzu.

▶ **Theorem 15.** *Sei G ein (g, k)-planarer Graph mit n Knoten. Dann gilt*

$$bw(G) \in \Theta(\min\left\{\sqrt{gkn}, n\right\})$$

Eine obere Schranke für die Schichtbaumweite (g, k)-planarer Graphen ist $2g + 3$ [2]. Der Beweis der oberen Schranke der Baumweite erfolgt somit analog zum Beweis von Theorem 5.2.1. Einen Zeugen für die untere Schranke zu finden ist weitaus komplizierter. Der Beweis wird hier ausgelassen und auf das Originalpaper verwiesen [1]. Er basiert auf den Separationseigenschaften eines dreidimensionalen Gittergraphen und besteht aus einer langen Kette von Abschätzungen.

5.3 Zeichnung von Graphen auf orientierbaren Flächen

Dieser Abschnitt zeigt, wie man beliebige Graphen (mit n Knoten, m Kanten) auf der orientierbaren Fläche mit g Henkeln zeichnen kann, wobei jede Kante $O(m \log^2(g)/g)$ Kreuzungen in Kauf nehmen muss. Es ist also möglich die lokale Kreuzungszahl fast-linear zu reduzieren, indem man eine kompliziertere Oberfläche betrachtet. Nun kann man sich fragen, ob dies kombiniert mit der Baumweitenschranke aus Abschnitt 5.2 eine neue nichttriviale Schranke für die Baumweite beliebiger Graphen impliziert. Dies ist leider nicht der Fall, da die Schranke $\sqrt{gn}\sqrt{m \log^2(g)/g} = \sqrt{mn \log^2(g)}$ trivial ist. Nun aber zur Formulierung des eigentlichen Resultats, das alleingestellt sehr wohl eine Neuerung bedeutet.

▶ **Theorem 16** (Zeichnung). *Sei $g \in \mathbb{N}$ und G ein Graph mit n Knoten und m Kanten. Man kann G auf Σ_g mit höchstens $O(\frac{m \log^2(g)}{g})$ Kreuzungen pro Kante zeichnen.*

Die Lemmata werden zunächst dargelegt. Anschließend wird erläutert wie sie zusammengesteckt werden, um eine Zeichnung des Graphen zu konstruieren, die Theorem 16 beweist. Schlussendlich werden dann die Lemmata bewiesen, um den Beweis zu komplettieren. Die Ausgliederung der Lemmata erlaubt einen kompakteren Beweis zu Theorem 16 und kompaktere Beweise der einzelnen Beweiskomponenten. Es ist dabei von besonderer Bedeutung, sich zunächst nur die Aussagen der Lemmata klarzumachen und zu verstehen wie sie ineinandergreifen. Genau das Zusammengreifen wird im Beweis von Theorem 16 im Detail erläutert. Beginnen wir nun mit dem Hostgraphlemma.

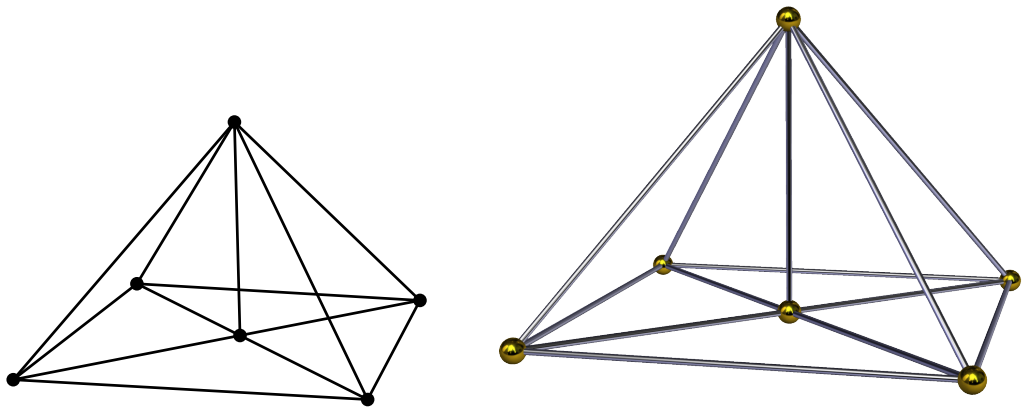


Abbildung 7 Links ein pyramidenartiger Graph. Rechts Σ_8 enstanden aus dem linken Graphen durch Ersetzen jedes Knotens durch eine Sphäre, und jeder Kante durch einen Zylinder. Bildquelle [4]

▶ **Lemma 17** (Hostgraph). *Aus gegebenem Graphen G, Henkelzahl $g \in \mathbb{N}$ und wählbarem (nicht notwendigerweise regulären) gradbeschränkten Expandergraphen Q (q Knoten, höchstens g Kanten) lässt sich ein Hostgraph H konstruieren, sodass*

- $V(H) = V(G) \dot\cup V(Q)$
- *Jede Kante von G wird in H auf einen Pfad der Länge $O(\log q)$ abgebildet, der sonst nur Knoten von Q besucht*
- $r_c(H) \leq g$
- *Jeder Knoten von Q wird durch $O(\frac{m \log q}{q})$ Pfade besucht*

Der Beweis folgt später, ist ein wenig technisch und benötigt noch weitere Werkzeuge. Jetzt soll es sich erstmal um den Beweis von Theorem 16 drehen. Die etwas aus der Luft gegriffene Bedingung von g Kanten für den Expandergraphen Q bewirkt übrigens, dass die Kreisrangbedingung für H eingehalten wird. Nun betrachten wir die Konstruktion, die einen Hostgraphen H aus dem Hostgraphlemma zu einer orientierbaren Fläche transformiert. Durch die Korrespondenz Hostgraph zu Fläche entsteht eine fast kanonische Zeichnung von G auf der Fläche, wie dann direkt anschließend im Beweis zu Theorem 16 gezeigt.

▶ **Lemma 18** (Transformation). *Gegeben einen Graph H aus dem Hostgraphlemma, also mit $r_c(H) \leq g$ lässt sich die orientierbare Fläche Σ_g aus H konstruieren, sodass die Kreise in H, Henkeln bzw. Henkeln in Σ_g entsprechen.*

Beweis Lemma 18 (Transformation). Ersetze jeden Knoten v von H durch eine Sphäre und schneide $\deg(v)$ disjunkte Kreisscheiben darauf aus. Auf diese Kreisscheiben werden später Zylinder angeklebt. Jede Kante von H wird durch einen Zylinder ersetzt und die Zylinderenden auf Kreisscheiben an den Endknoten der Kante geklebt. Der Kreisrang von H entspricht dabei der Anzahl Henkel der darauf entstehenden (abstrakten) Fläche. Falls $r_c(H) < g$ fügt man noch weitere Henkel beliebig hinzu, um Σ_g zu erhalten. ◀

Abbildung 7 zeigt die Anwendung des Transformationslemmas beispielhaft an einem pyramidenartigen Graphen mit 8 unabhängigen Kreisen. Man beachte, dass die Sphären- und Zylinderoberflächen zu betrachten sind, nicht das Innere des Körpers.

[4] Bildquelle Tom Ruen bei https://commons.wikimedia.org/wiki/File:Square_pyramid_pyramid.png

Mithilfe des Transformationslemmas und des Hostgraphlemmas können wir nun den Beweis von Theorem 16 skizzieren. Komplettiert wird er aber erst mit dem später folgenden Beweis des Hostgraphlemmas. Die Grundidee ist den Hostgraph H zu einem zu zeichnenden Graphen G in Σ_g zu transformieren und die Kanten von G entlang der Pfade zu zeichnen, auf die sie in H abgebildet wurden.

Beweis Theorem 16. Zunächst wird, gegeben einen Graphen G und eine gewünschte Henkelzahl $g \in \mathbb{N}$, mittels des Hostgraphlemmas ein Hostgraph H für G bestimmt. Dieser wird mit dem Transformationslemma zu Σ_g transformiert.

Durch die in Σ_g vorhandene Struktur von H und die in H vorhandene Struktur von G lässt sich nun G in Σ_g zeichnen. Eine Kante (x, y) von G wird auf einen Pfad in H abgebildet, der zwischen x und y über Kanten von H verläuft. Der Pfad in H wiederum induziert einen Weg entlang der Zylinder von Σ_g. Man zeichnet die Kanten von G also als Jordankurven entlang der zu einer Kante gehörenden Zylinder. Dabei sollen Kreuzungen nur an Sphären stattfinden. Das ist immer möglich, denn angenommen man zeichnet eine Menge an Kreuzungen auf einem Zylinder, kann man diese einfach auf eine der zwei Endsphären verschieben. Zusätzlich soll sich jedes Kantenpaar an jeder Sphäre höchstens einmal kreuzen. Das lässt sich erreichen, indem die topologische Äquivalenz (Homöomorphie) von Sphäre und Ebene ausgenutzt wird. In der Ebene kreuzen sich gerade Streckensegmente entweder unendlich oft (dieser Fall ist hier nicht relevant), nie oder eben einmal. Da Kreuzung unter Homöomorphie erhalten bleibt, gibt es also eine Zeichnung der Kanten mit paarweise höchstens einer Kreuzung auf der Sphäre.
Nachdem nun die Zeichnung beschrieben wurde, gilt es noch die Kreuzungen der Kanten auf der gesamten Fläche zu zählen. Sei Q der im Hostgraphlemma passend zu g gewählte Expandergraph mit q Knoten und $|E(Q)| \leq g$. Aus dem Hostgraphlemma wissen wir, dass jede durch $V(Q)$ induzierte Sphäre durch $O(\frac{m \log q}{q}) = O(\frac{m \log g}{g})$ Kurven besetzt wird, die sich paarweise auf der Sphäre höchstens einmal kreuzen. Die von $V(G)$ induzierten Sphären werden nur durch Kurven besetzt, deren korrespondierende Kanten inzident zum der Sphäre entsprechenden Knoten sind. Da sich diese Kanten aber am Knoten treffen, ist dort keine Kreuzung nötig. Außerdem geht jede Kurve durch $O(\log q) \subset O(\log g)$ Sphären. Dabei gilt $O(\log q) \subset O(\log g)$, da Q zusammenhängend ist, also $g \geq |E(Q)| \geq q - 1$. Das bedeutet für jede Kante insgesamt $O(\frac{m \log^2(g)}{g})$ Kreuzungen, was zu zeigen war.

◀

Nun gilt es noch das verbleibende Hostgraphlemma zu beweisen, um die zusammengesteckten, hohlen Bauteile mit Leben zu füllen. Dazu benötigen wir allerdings ein weiteres Lemma und das folgende Theorem, welches auf Mehrgüterflüssen basiert.

▶ **Theorem 19** (Leighton und Rao [7]). *Ein Graph G mit n Knoten und beschränktem Grad ist injektiv abbildbar auf einen beliebigen Expandergraphen H mit mindestens n Knoten. Dabei werden die Kanten von G auf Pfade in H der Länge $O(\log(n))$ und abgebildet. Außerdem wird jede Kante von H durch $O(\log(n))$ solcher Pfade besetzt.*

Für gradbeschränkte Graphen lässt sich das Theorem von Leighton und Rao direkt mit der Transformation aus Lemma 18 kombinieren. Man erreicht sogar ein deutlich besseres Ergebnis bezüglich notwendiger Kreuzungen pro Kante als in Theorem 16. Bei allgemeinen Graphen ist das Theorem von Leighton und Rao nicht direkt anwendbar. Mithilfe des nun folgenden Lastverteilungslemmas lassen sich allerdings gleichmäßige und gradbeschränkte

Blöcke konstruieren auf die man dann Theorem 19 anwenden kann. Genauere Details folgen in Kürze.

▶ **Lemma 20** (Lastverteilung). *Gegeben einen Graph G mit n Knoten, m Kanten und einen Graph Q mit q Knoten, gibt es einen bipartiten Graphen $\mathcal{B}$ mit den partiten Klassen $V(G)$ und $V(Q)$ sowie Kantengewichten $\omega : E(\mathcal{B}) \to \mathbb{N}$ sodass*

- *Der gewichtete Grad der Knoten von G in $\mathcal{B}$ entspricht ihrem Grad in G*
- *Der gewichtete Grad der Knoten von Q in $\mathcal{B}$ ist $\lfloor \frac{2m}{q} \rfloor$ oder $\lceil \frac{2m}{q} \rceil$*
- $|E(\mathcal{B})| \leq n + q - 1$

Der Beweis des Lastverteilungslemmas erfolgt konstruktiv über Algorithmus 1 der einen bipartiten Graphen $\mathcal{B}$ berechnet, der die Bedingungen des Lastverteilungslemmas erfüllt. Anstelle von Kantengewichten wird hier auch von einer Kapazitätsfunktion gesprochen, da die Kapazität einer Kante in $\mathcal{B}$ angibt wie viele Pfade (im Hostgraphlemma) über sie geroutet werden.

Beweis Lemma 20 (Lastverteilung). In Algorithmus 1 wird in jedem Schritt der while-Schleife eine Kante zwischen einem Knoten von Q und einem Knoten von G mit der größtmöglichen Restkapazität gezogen. Die Knoten von G haben die Kapazität ihres Grades in G, die Knoten von Q haben entweder Kapazität $\lceil \frac{2m}{q} \rceil$ oder $\lfloor \frac{2m}{q} \rfloor$, sodass sich ihre Kapazitäten genau zu $2m = \sum_{v \in V(G)} \deg_G(v)$ aufaddieren. Am Ende des Algorithmus hat jeder Knoten Restkapazität null, da die Gesamtkapazität auf beiden Seiten gleich ist und somit sind die Bedingungen an die gewichteten Grade erfüllt. Verbleibt die Anzahl Kanten von $\mathcal{B}$. Nach jedem Schritt hat einer der beiden Knoten keine Kapazität mehr, wird also in einem späteren Schritt nicht mehr betrachtet. Gleichzeitig wird in jedem Schritt eine Kante gezogen. Da außerdem im letzten Schritt zwei Knoten gleichzeitig saturiert werden, folgt dass $|E(\mathcal{B})| \leq n + q - 1$.

Input : Graph G mit n Knoten, m Kanten;
Graph Q mit q Knoten
Output : Bipartiter Graph $\mathcal{B}$, Kapazitätsfunktion $\omega : E(\mathcal{B}) \to \mathbb{N}$
$V(Q) \leftarrow \{v_1, \ldots, v_q\}$;
$\mathcal{B} \leftarrow (V(G) \cup V(Q), \emptyset)$;
$\forall v \in V(G) : \lambda[v] \leftarrow \deg(v)$;
for $i = 1$ **to** n **do**
  $\lambda[q_i] \leftarrow \begin{cases} \lceil \frac{2m}{q} \rceil & i \leq 2m \bmod q \\ \lfloor \frac{2m}{q} \rfloor & \text{i} > \text{2m mod q} \end{cases}$
end
while $\exists v \in V(G)$ *und* $w \in V(Q)$ *mit* $\lambda[v] > 0$ *und* $\lambda[w] > 0$ **do**
  $\zeta \leftarrow \min\{\lambda[v], \lambda[w]\}$;
  $\lambda[v] \leftarrow \lambda[v] - \zeta, \lambda[w] \leftarrow \lambda[w] - \zeta$;
  Füge Kante vw mit Kapazität ζ zu $\mathcal{B}$ hinzu ;
end

Algorithmus 1 : Lastverteilung

◀

Abbildung 8 zeigt einen möglichen bipartiten Lastverteilungsgraphen zwischen G und Q. Man beachte, dass die Struktur von G unwichtig ist und nur dessen Gradsequenz betrachtet werden muss. Ebenso ist nur die Anzahl von Knoten in Q relevant, keine weitere Struktur. Man beachte dass sich die Summe der gewichteten Grade in beiden Partitionen gleichen. Außerdem ergibt die Summe der Kantengewichte der zu einem Knoten inzidenten Kanten gerade wieder den Grad in G bzw. das Label in Q. Der resultierende Lastverteilungsgraph

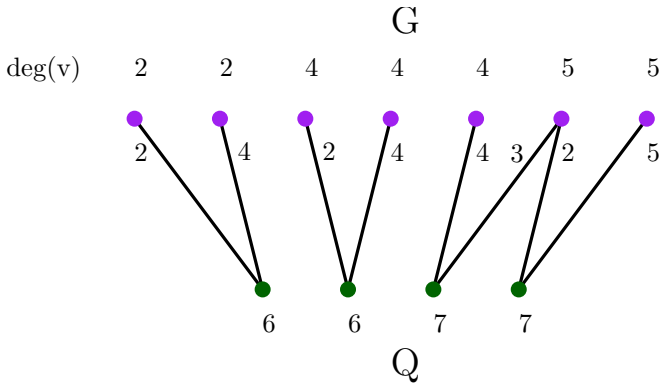


Abbildung 8 Ein bipartiter Lastverteilungsgraph für gegebene Gradsequenz von G

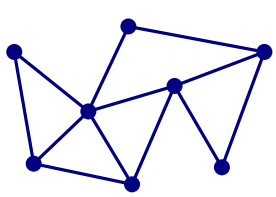
ist abhängig von der Reihenfolge in der Knoten mit verbleibender Kapazität im Kopf der while-Schleife gewählt werden. Zum Beispiel lässt sich der Lastverteilungsgraph in der Abbildung berechnen, indem die Knoten von rechts nach links traversiert werden.

Nach einem zweiten Ausschweifen kommen wir nun endlich zum Beweis des Hostgraphlemmas Lemma 17. Der Abschluss dieses Beweises komplettiert den Beweis von Theorem 16.

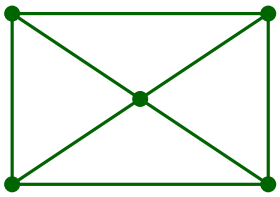
Beweis Lemma 17 (Hostgraph). Wähle ein $q \in \mathbb{N}$ und einen gradbeschränkten Expandergraphen Q mit höchstens g Kanten, zum Beispiel mittels einer Methode aus [6]. Die Anzahl Kanten ist bestimmt durch den maximalen Kreisrang, den der zu bestimmende Hostgraph H haben darf, nämlich g. Zunächst sollte die Konstruktionsvorschrift von H erläutert werden. Mithilfe des Lastverteilungslemmas wird der bipartite Graph $\mathcal{B}$ und die zugehörige Kapazitätsfunktion $\omega : E(B) \to \mathbb{N}$ zur Balancierung zwischen G und Q konstruiert. Dann sei der Hostgraph $H = (V(G) \dot\cup V(Q), E(Q) \dot\cup E(\mathcal{B}))$. Eine beispielhafte Konstruktion von H ist in Abbildung 9 schrittweise dargestellt. Da Q nach Voraussetzung zusammenhängend ist und jeder Knoten von G in H zu einem Knoten von Q verbunden ist, ist H zusammenhängend. Somit gilt für den Kreisrang von H:

$$r_c(H) = |E(H)| - |V(H)| + 1 \leq (|E(Q)| + (n + q - 1)) - (n + q) + 1 = |E(Q)| \leq g$$

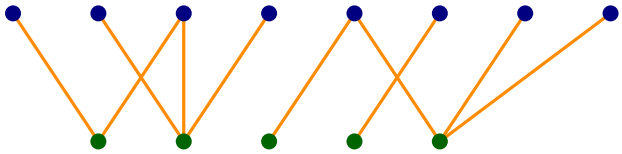
Dabei gilt $|E(H)| \leq |E(Q)| + (n + q - 1)$, da $\mathcal{B}$ nach dem Lastverteilungslemma höchstens $n + q - 1$ Kanten besitzt.
Es verbleibt für jede Kante $vw \in E(G)$ einen Pfad der Länge $O(\log q)$ zu finden, der intern nur Knoten von Q enthält. Dazu nimmt man pro Kante vw zwei Kanten $vp, wq \in E(\mathcal{B})$ mit verbleibender Restkapazität $\omega(vp), \omega(wq) > 0$, speichert das Paar pq zwischen und verbraucht eine Kapazitätseinheit von den Kanten vp und wq.
Nach abgeschlossener Zuweisung wird nun ein Hilfsgraph benötigt, nämlich der Graph mit den Knoten von Q und allen Paaren aus dem vorherigen Schritt als Kanten. Dieser Graph hat bis zu m Kanten. Eventuell werden verschiedene Kanten von G auf das gleiche Paar abgebildet. Die Kanten des Hilfsgraphen werden gleichmäßig in $\Theta(\frac{m}{q})$ Blöcke partitioniert, die dementsprechend $\Theta(q)$ viele Kanten enthalten. Das lässt sich zum Beispiel umsetzen, indem in jedem Block jedem Knoten konstant viele Kanten zugewiesen werden. Durch die Lastverteilung ist jeder Knoten von Q an gleich vielen Pfaden beteiligt, nämlich $\frac{2m}{q}$ vielen. Die Einteilung in $\Theta(\frac{m}{q})$ Blöcke bewirkt dann, dass jeder Block für sich gesehen konstante Knotengrade aufweist. Auf jeden dieser Blöcke lässt sich das Theorem von Leighton und Rao anwenden (mit der Identität als Einbettung von jedem Block nach Q), um einen Pfad



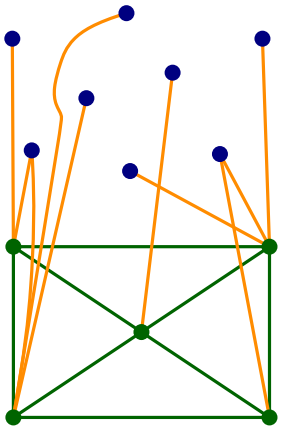
(a) Ursprungsgraph G



(b) Expandergraph Q



(c) Lastverteilungsgraph $\mathcal{B}$



(d) Hostgraph H

Abbildung 9 Schrittweise Konstruktion des Hostgraphen H zu einem gegebenen Graphen G. (a) zeigt den Eingabegraphen G, (b) zeigt einen gewählten Expandergraphen Q, (c) zeigt einen möglichen Lastverteilungsgraphen $\mathcal{B}$ zwischen Q und G; bestimmt mittels des Lastverteilungslemmas. Schlussendlich sieht man in (d) den Hostgraph H, zusammengesetzt aus den Kanten von $\mathcal{B}$ und Q

P_{pq} der Länge $O(\log q)$ zwischen jedem der Paare pq in einem Block über Kanten von Q zu erhalten. Der zu einer Kante vw in G gehörige Pfad in H, falls p der Partner von v und q der Partner von w ist, läuft von v nach p, über P_{pq} nach q und von dort nach w und hat Länge $O(\log(q) + 2)$. Das Theorem von Leighton und Rao garantiert, dass jede Kante von Q durch $O(\log q)$ Pfade pro Block besetzt wird. Es wird auf $O(\frac{m}{q})$ Blöcke angewandt. An jedem Knoten treffen sich also $O(m \log q/q)$ Pfade. ◀

5.4 Fazit

Im ersten Teil dieser Ausarbeitung wurde über explizite Konstruktion die asymptotisch scharfe Schranke $\Theta\left(\sqrt{kn}\right)$ (respektive $\Theta\left(\sqrt{gkn}\right)$) für die Baumweite von k-planaren Graphen (respektive (g, k)-planaren Graphen) gezeigt. Die Konstruktion erfolgte über die Schichtbaumweite von k-/(g, k)-planaren Graphen und das Lemma von Norin, welches ebenfalls über explizite Konstruktion besagt, dass jeder Graph mit Schichtbaumweite l höchstens Baumweite $2\sqrt{nl}$ hat. Im zweiten Teil wurde die Schranke $O\left(\frac{m \log^2(g)}{g}\right)$ für die lokale Kreuzungszahl auf orientierbaren Flächen für beliebige Graphen gezeigt. Dieses Ergebnis bietet leider keine Neuerung bezüglich der Baumweite von beliebigen Graphen, da die Schranke $\sqrt{gn}\sqrt{\frac{m \log^2(g)}{g}} = \sqrt{mn \log^2(g)}$ trivial ist. Besonders spannende Techniken im zweiten Abschnitt sind die Transformation von Graph zu orientierbarer Fläche sowie die Kantenblockpartitionierung, um das Theorem von Leighton und Rao anwenden zu können.

Referenzen

1 Vida Dujmović, David Eppstein, and David R Wood. Genus, treewidth, and local crossing number. In *Graph Drawing and Network Visualization*, pages 87–98. Springer, 2015.

2 Vida Dujmović, Pat Morin, and David R Wood. Layered separators in minor-closed families with applications. *arXiv preprint arXiv:1306.1595*, 2013.

3 Zdenek Dvorák and Sergey Norin. Treewidth of graphs with balanced separations. *arXiv preprint arXiv:1408.3869*, 2014.

4 Alexander Grigoriev and Hans L Bodlaender. Algorithms for graphs embeddable with few crossings per edge. *Algorithmica*, 49(1):1–11, 2007.

5 Martin Grohe and Dániel Marx. On tree width, bramble size, and expansion. *Journal of Combinatorial Theory, Series B*, 99(1):218–228, 2009.

6 Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

7 Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)*, 46(6):787–832, 1999.

8 VV Lozin and D Rautenbach. The tree-and clique-width of bipartite graphs in special classes. *Australasian Journal of Combinatorics*, 34:57, 2006.

9 Gerhard Ringel and John WT Youngs. Solution of the heawood map-coloring problem. *Proceedings of the National Academy of Sciences*, 60(2):438–445, 1968.

10 Neil Robertson and Paul D Seymour. Graph minors. v. excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 41(1):92–114, 1986.

6 Obere und untere Schranken für Online-Routing auf Delaunay-Triangulationen

Katharina Dormann

Zusammenfassung

In dieser Seminararbeit wird das Paper „Upper and Lower Bounds for Online Routing on Delaunay Triangulations“ von Bonichon et al. [1] vorgestellt. Sie beschäftigt sich mit dem Online-Routing auf Delaunay-Triangulationen. Es wird ein Online-Routing-Algorithmus beschrieben und ein Beweis für die obere Schranke von dessen Routing Ratio geführt. Weiterhin werden untere Schranken für die Routing Ratio und die Competitive Ratio auf Delaunay-Triangulationen angegeben.

6.1 Einleitung

6.1.1 Routing in Netzwerken

In ganz verschiedenen Bereichen wie der Robotik, beim Design von Kommunikationsnetzwerken und bei der Stadtplanung tritt des Problem des Routings in Netzwerken auf. Es geht dabei um das Finden von (kurzen) Pfaden innerhalb des Netzwerks. Die Modellierung erfolgt meist mit Hilfe von *geometrischen Graphen*. Ein geometrischer Graph ist ein Graph, dessen Knoten als Punkte in einer Ebene, und dessen Kanten als Strecken zwischen den Punkten interpretiert werden. Die Gewichte der Kanten entsprechen den Abständen der Knoten in der Ebene. Das Problem des Routings ist nun das Problem des Findens eines möglichst kurzen Pfades von einem gegebenen Startknoten zu einem gegebenen Zielknoten.

6.1.2 Online Routing

Beim Routing in Graphen ist nicht immer zu jedem Zeitpunkt die vollständige Information über den Graphen gegeben. Oft sind nur *lokale* Informationen vorhanden, d.h. die Koordinaten des Knoten, an dem sich die zu routende Nachricht zum aktuellen Zeitpunkt befindet und die Koordinaten der Nachbarknoten. Das Routing wird entsprechend Online-Routing genannt. Da in diesem Fall dem Routing-Algorithmus weniger Informationen zur Verfügung stehen, kann es sein, dass für eine Familie von Graphen zwar eine Spanning Ratio existiert, allerdings kein Online-Algorithmus, der den entsprechenden Pfad findet. Eine Klasse von Graphen, für die Online-Algorithmen existieren, ist die Klasse der Delaunay-Trinangulationen.

6.1.3 Bewertung von Routing-Algorithmen

Hat man einen Routing-Algorithmus gefunden, muss dieser bewertet werden. Das Maß ist dabei die Länge des von dem Algorithmus produzierten Pfades von Startknoten s zu Zielknoten t (siehe Abb. 1).

Zur Bewertung werden *competitive ratio* und *routing ratio* herangezogen. Im Folgenden gelte stets $s \neq t$. Die Competitive Ratio ist definiert als

$$\text{Competitive Ratio} = \max_{(s,t)} \frac{\text{Länge des vom Algorithmus ausgegebenen Pfades von s nach t}}{\text{Länge des kürzesten Pfades von s nach t}},$$

während die Routing Ratio definiert ist als

$$\text{Routing Ratio} = \max_{(s,t)} \frac{\text{Länge des vom Algorithmus ausgegebenen Pfades von s nach t}}{\text{Euklidische Distanz zwischen s und t}}.$$

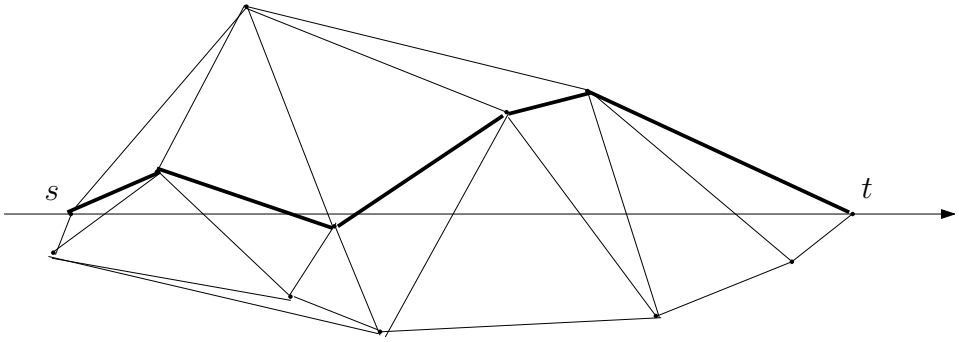


Abbildung 1 Eine Delaunay-Triangluation mit Startknoten s und Zielknoten t. Der vom Routing-Algorithmus gefundene Pfad von s nach t ist fett eingezeichnet.

Da der kürzeste Pfad zwischen s und t mindestens so groß ist wie die Euklidische Distanz zwischen s und t, gilt:

Competitive Ratio $\leq$ Routing Ratio.

Man möchte nun Routing-Algorithmen mit möglichst keiner Routing Ratio und möglichst kleiner Competitive Ratio finden. Allerdings existieren diese nicht für jede Familie von Graphen, d.h. nicht immer lässt sich eine obere Schranke für das Verhältnis aus der Länge eines Pfades zwischen zwei Knoten und der Euklidischen Distanz zwischen diesen Knoten angeben. Falls es für eine Familie von Graphen eine Schranke für dieses Verhältnis gibt, die für jedes Knotenpaar gilt, dann heißt diese Schranke *Spanning Ratio*:

$$\text{Spanning Ratio} = \max_{(s,t)} \frac{\text{Länge des kürzesten Pfades von s nach t}}{\text{Euklidische Distanz zwischen s und t}}$$

Ein Graph, für den eine Spanning Ratio existiert, wird *Spanner* genannt. Da ein von einem Algorithmus ausgegebener Pfad von s nach t immer mindestens so lang ist wie der kürzeste existierende Pfad von s nach t, gilt:

Spanning Ratio $\leq$ Routing Ratio.

Folglich möchte man idealerweise Algorithmen finden, deren Routing Ratio der Spanning Ratio entspricht. Diese haben dann die beste mögliche Routing Ratio.

6.1.4 Delaunay-Triangulationen

Delaunay-Triangulationen sind bestimmte geometrische Graphen. Man kann sie sich wie folgt entstanden vorstellen: Die konvexe Hülle der als Punkte in der Ebene interpretierten Knoten wird in Dreiecke unterteilt, so dass für jeden Kreis, auf dem die Ecken eines solchen Dreiecks liegen, gilt, dass in dessen Inneren keine Knoten liegen (siehe Abb. 2). Die Seiten der Dreiecke entsprechen dann den Kanten des Graphen.

Wenn man in der Definition der Delaunay-Triangulation „Kreis“ durch „gleichseitiges Dreieck“ ersetzt, erhält man die Definition der TD-Delaunay-Triangulation. Ersetzt man „Kreis“ durch „Quadrat“, erhält man die Definition der L_1-Delaunay-Triangulation bzw. der L_∞-Delaunay-Triangulation, abhängig von der Orientierung des Quadrates.

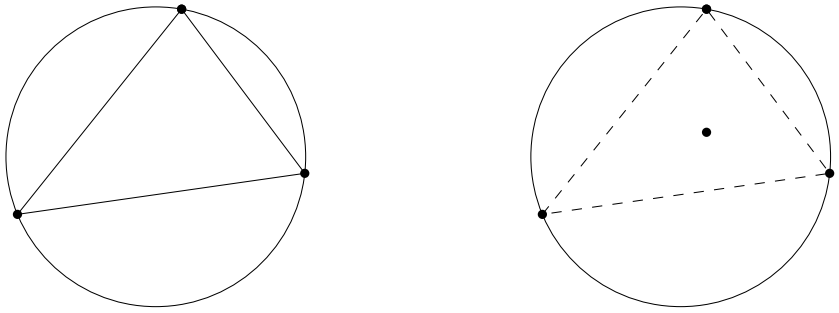


Abbildung 2 Veranschaulichung der Definition der Delaunay-Triangulation. In der linken Graphik sind die drei Punkte, die auf dem Kreis liegen, durch Kanten verbunden, da sich innerhalb des Kreises kein weiterer Punkt befindet. In der rechten Graphik befindet sich ein weiterer Punkt im Inneren des Kreises. Daher existieren in einer Delaunay-Triangulation die drei gestrichelt eingezeichneten Kanten *nicht*.

6.1.5 Vorangegangene Arbeiten

Es gibt einige vorangegangene Arbeiten, die obere bzw. untere Schranken für die Spanning Ratio, die Routing Ratio und die Competitive Ratio für die drei Typen von Triangulationen (Delaunay-Triangulation, TD-Delaunay-Triangulation und L_1- bzw. L_∞-Delaunay-Triangulation) zeigen (siehe Tab. 1).

Tabelle 1 Derzeitiger Kenntnisstand der oberen und unteren Schranken von Spanning Ratio, Routing Ratio und Competitive Ratio für die drei Arten von Delaunay-Triangulationen. Entsprechend den Definitionen der Triangulationen steht „Dreieck“ für die TD-Delaunay-Triangulation, „Quadrat“ für die L_1- bzw. L_∞-Delaunay-Triangulation und „Kreis“ für die normale Delaunay-Triangulation. „o. S.“ steht für „obere Schranke“, „u. S.“ für „untere Schranke“. Die durch die Arbeit von Bonichon et al. [1] neu hinzugekommenen und in dieser Ausarbeitung beschriebenen Ergebnisse sind fett gedruckt.

	Dreieck	**Quadrat**	**Kreis**
Spanning Ratio, o. S.	2	$\sqrt{4+2\sqrt{2}} \approx 2.61$	1.998
Spanning Ratio, u. S.	2	$\sqrt{4+2\sqrt{2}} \approx 2.61$	1.593
Routing Ratio, o. S.	$5/\sqrt{3} \approx 2.89$	$\sqrt{10} \approx 3.16$	$\mathbf{1.185 + 3\frac{\pi}{2} \approx 5.90}$
Routing Ratio, u. S.	$5/\sqrt{3} \approx 2.89$	**2.707**	**1.701**
Competitive Ratio, o. S.	$5/\sqrt{3} \approx 2.89$	$\sqrt{10} \approx 3.16$	$\mathbf{1.185 + 3\frac{\pi}{2} \approx 5.90}$
Competitive Ratio, u. S.	$5/3 \approx 1.66$	**1.1213**	**1.2327**

Dobkin et al. [6] haben gezeigt, dass die Delaunay-Triangulation ein Spanner ist. Xia hat gezeigt, dass 1.998 eine obere Schranke [7] und 1.593 eine untere Schranke [8] für die Spanning Ratio der Delaunay-Triangulation ist. Chew [5] hat gezeigt, dass 2 sowohl eine obere als auch eine untere Schranke für die Spanning Ratio der TD-Delaunay-Triangulation ist. Bonichon et al. [2] haben gezeigt, dass $\sqrt{4+2\sqrt{2}} \approx 2.61$ sowohl eine obere als auch eine untere Schranke für die Spanning Ratio der L_1- und der L_∞-Delaunay-Triangulation ist.

Es können auch allgemeine obere und untere Schranken für die Routing Ratio und die Competitive Ratio angegeben werden. Um eine obere Schranke anzugeben, wird ein konkreter Algorithmus angegeben, für den diese Schranke gilt. Für die TD-Delaunay-Triangulation haben Bose et al. [4] einen Online-Routing-Algorithmus mit einer Competitive und Routing Ratio von $5/\sqrt{(3)} \approx 2.89$ beschrieben. Außerdem haben sie die untere Schranke $5/\sqrt{(3)} \approx 2.89$

für die Routing Ratio und die untere Schranke $5/3 \approx 1.66$ für die Competitive Ratio bewiesen. Für die Routing Ratio der L_1-Delaunay-Triangulation hat Chew [5] die obere Schranke $\sqrt{10} \approx 3.162$ unter Angabe eines Online-Algorithmus gezeigt. Bose et al. [3] haben für die Competitive und die Routing Ratio der Delaunay-Triangulation die obere Schranke 15.48 gezeigt.

6.1.6 Ziel und Aufbau der Arbeit

In [1] wird ein neuer Routing-Algorithmus für Delaunay-Triangulationen vorgestellt. Es wird die oberere Schranke $1.185 + 3\frac{\pi}{2} \approx 5.90$ für die Competitive Ratio und die Routing Ratio bewiesen. Dies ist eine deutliche Verbesserung der zuvor in [3] angegebenen Schranke.

Der in [1] vorgestellte Algorithmus ist eine Anpassung des von Chew [5] für die L_1-Delaunay-Triangulation vorgestellten Algorithmus, an die Delaunay-Triangulation. Für den angepassten Algorithmus wird die obere Schranke $1.185 + 3\frac{\pi}{2}$ für die Routing Ratio - und damit für die Competitive Ratio - gezeigt. Des Weiteren werden von Bonichon et al. untere Schranken für Competitive und Routing Ratio für die Delaunay-Triangulation bewiesen. Schließlich werden auch untere Schranken für Competitive und Routing Ratio für die L_1-Delaunay-Triangulation gezeigt.

In Abschnitt 6.2.1 dieser Arbeit wird der Online-Routing-Algorithmus für die Delaunay-Triangulation beschrieben. In Abschnitt 6.2.2 wird der Beweis für die obere Schranke der Routing Ratio des Algorithmus vorgestellt. In Abschnitt 6.2.3 werden untere Schranken für Competitive und Routing Ratio von Online-Routing-Algorithmen angegeben. Die Arbeit wird mit einer Zusammenfassung in Abschnitt 6.3 abgeschlossen.

6.2 Der Routing-Algorithmus und seine Routing Ratio

6.2.1 Der Routing-Algorithmus

Es wird nun der Online-Routing-Algotihmus auf einer Delaunay-Triangulation beschrieben. Zunächst wird die Annahme getroffen, dass der Startknoten s und der Zielknoten t auf der x-Achse des Koordinatensystems liegen (siehe Abb. 3). Dies kann immer durch Rotation des Koordinatensystems erreicht werden.

Weiterhin werden bei der Durchführung des Algorithmus nur Dreiecke der Triangulation und deren Ecken betrachtet, die die x-Achse schneiden. Es wird angenommen, dass Knoten, die den Ecken der Dreiecke entsprechen, die die x-Achse nicht schneiden, so weit von s und t entfernt sind, dass sie für das Finden eines *kurzen* Pfades von s nach t nicht relevant sind. Die Dreiecke werden entsprechend ihres Schnittes mit der x-Achse entlang der x-Achse angeordnet. Das bedeutet für zwei Dreiecke, dass dasjenige, dessen Schnitt mit der x-Achse weiter rechts liegt, als das „weiter rechts“ liegende bezeichnet wird. Der Algorithmus baut den Pfad $s = p_0, p_1, ..., p_k = t$ wie folgt auf:

Der erste Knoten des Pfades entspricht dem Startknoten s: $p_0 = s$. Sobald ein Knoten p_i erreicht wurde, muss mit Hilfe der Kenntnis der Nachbarknoten (aber nicht der übrigen Knoten im Graphen) der nächste Knoten auf dem Pfad bestimmt werden. Dazu wird zunächst das *rechteste* Dreieck T_i betrachtet, zu dem p_i gehört. Sei C_i der Kreis, auf dem die Ecken des Dreiecks T_i liegen (siehe Abb. 3). (Da drei Punkte einen Kreis eindeutig bestimmen, ist C_i eindeutig bestimmt.) Sei w_i der *linkeste* (*westlichste*) Punkt auf dem Kreis und r_i der rechteste Schnitt von C_i mit der Geraden st. Die Strecke $w_i r_i$ teilt C_i in einen oberen und in

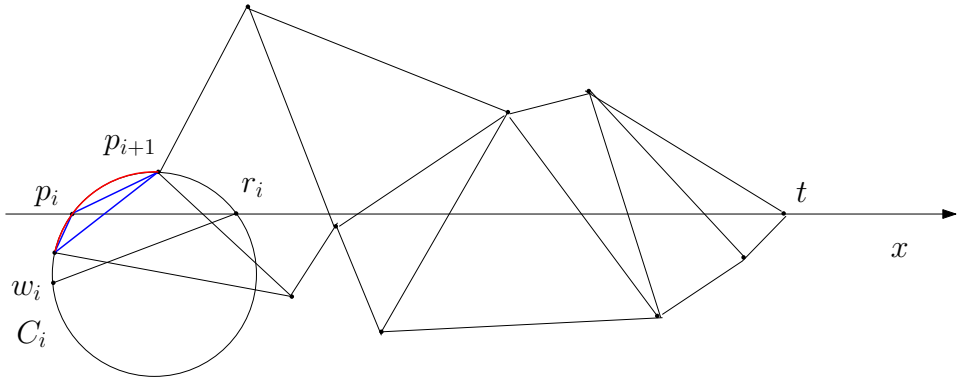


Abbildung 3 Beim Routing-Algorithmus werden nur die Dreiecke der Delaunay-Trinangulation betrachtet, die die x-Achse schneiden. Das im Schritt i betrachtete Dreieck T_i ist blau eingezeichnet. Der Kreis, auf dem die Ecken von T_i liegen, heißt C_i. In diesem Fall liegt p_i auf dem oberen Bogen, so dass im Uhrzeigersinn zum nächsten Knoten p_{i+1} gegangen wird. Der entsprechende gerichtete Kreisbogen $\mathcal{A}_i\langle p_i, p_{i+1}\rangle$ ist rot eingezeichnet.

einen unteren Bogen. Nun wird eine Fallunterscheidung durchgeführt, bei der der nächste Knoten auf dem Pfad bestimmt wird:

Falls p_i auf dem oberen Bogen liegt oder falls $p_i = w_i$, läuft man *im Uhrzeigersinn* auf C_i bis zum nächsten Knoten von T_i. Dieser ist der nächste Knoten auf dem Pfad, also p_{i+1}.

Falls p_i auf dem unteren Bogen liegt, läuft man *gegen den Uhrzeigersinn* auf C_i bis zum nächsten Knoten von T_i. In diesem Fall ist dieser Knoten p_{i+1}.

Es ist zu beachten, dass der Knoten p_{i+1} nicht in beiden Fällen gleich sein kann, da auf dem Kreis mindestens zwei Knoten außer p_i liegen. Die Fallunterscheidung bewirkt, dass die Bewegung auf dem Kreisbogen „nach rechts“ erfolgt. Der gerichtete Kreisbogen, der zur Kante (p_i, p_{i+1}) gehört, wird $\mathcal{A}_i\langle p_i, p_{i+1}\rangle$ genannt.

Da jeder Knoten des Graphen zu mindestens zwei Dreiecken gehört, und die Bewegung von links nach rechts erfolgte, ist p_{i+1} auch Knoten mindestens eines Dreiecks, das rechts von T_i liegt. Falls p_{i+1} zu mehreren solchen Dreicken gehört, wird das rechteste ausgewählt. Dieses wird T_{i+1} genannt, und das Vorgehen wird für T_{i+1} wiederholt. Dies wird so lange durchgeführt, bis der Zielknoten t erreicht ist. Es entsteht der Pfad $s = p_0, p_1, ..., p_k = t$ (siehe Abb. 4).

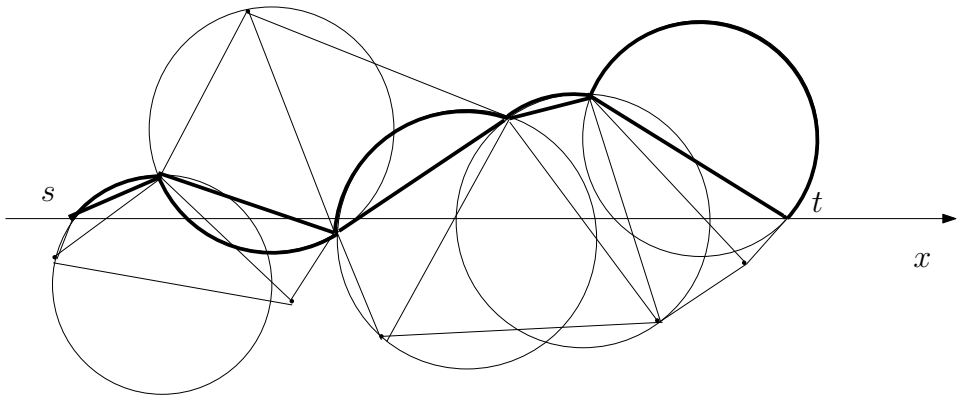


Abbildung 4 Die Delaunay-Triangulation und die beim Algorithmus benutzten Kreise C_i. Der gefundene Pfad und die zu den Kanten des Pfades gehörenden Kreisbögen sind fett eingezeichnet.

6.2.2 Obere Schranke der Routing Ratio

Das folgende Theorem gibt eine obere Schranke für die Routing Ratio des Algorithmus an. Das bedeutet, dass die Länge des von dem Algorithmus ausgegebenen Pfades zwischen zwei Punkten s und t, die einen festen Abstand haben, nach oben abgeschätzt wird. Das Theorem beschreibt eine deutliche Verbesserung der bisher besten bekannten Schranke für die Delaunay-Triangulation (15.48).

▶ **Theorem 1.** *Der Routing-Algorithmus hat eine Routing Ratio von maximal* $(1.185043874 + 3\frac{\pi}{2}) \approx 5.89743256$.

Eine untere Schranke der Routing Ratio für den Algorithmus ist 5.7281 (siehe Abschnitt 6.2.3). Daher ist die Schranke aus Theorem 1 eine sehr gute Abschätzung der Routing Ratio für diesen konkreten Algorithmus.

Die Idee, um Theorem 1 zu beweisen, ist folgende: Die Länge des vom Algorithmus ausgegebenen Pfades wird nach oben abgeschätzt durch $(1.185043874 + 3\frac{\pi}{2})|st|$, wobei $|st|$ die Länge der Strecke von s nach t bezeichnet. Die Länge des Pfades wird nicht direkt nach oben abgeschätzt. Die Vereinigung der Kreisbögen $\mathcal{A}_i\langle p_i, p_{i+1}\rangle$ - im Folgenden mit $\mathcal{A}$ bezeichnet - ist immer eine obere Schranke für die Länge des Pfades (siehe Abb. 4). Dies wird im Beweis genutzt, in dem $\mathcal{A}$ nach oben abgeschätzt wird. Diese Schranke ist dann auch eine Schranke für die Länge des Pfades.

Auch $\mathcal{A}$ wird nicht direkt nach oben abgeschätzt. $\mathcal{A}$ wird durch die Kreisbögen sogenannter *Worst-Case*-Kreise C_i' nach oben abgeschätzt (siehe Abb. 5).

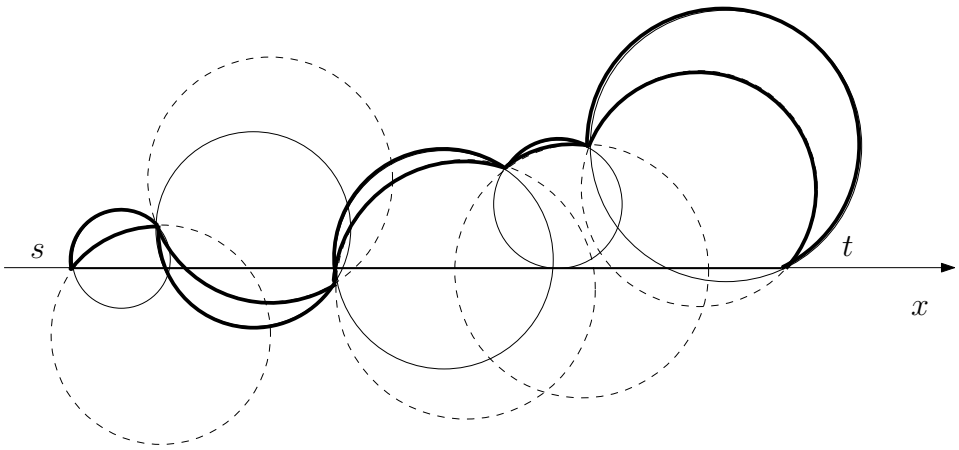


Abbildung 5 Die vom Algorithmus benutzen Kreise C_i sind gestrichelt eingezeichnet, die *Worst-Case*-Kreise C_i' mit durchgehender Linie. Die zum Pfad gehörenden Kreisbögen und die entsprechenden *Worst-Case*-Kreisbögen sind fett eingezeichnet. Die *Worst-Case*-Kreisbögen sind immer mindestens so lang wie die Kreisbögen des Pfades.

Diese Kreisbögen von einem Punkt p_i zum nächsten Punkt p_{i+1} sind durch ihre Definition immer mindestens so lang wie alle anderen möglichen Kreisbögen von p_i nach p_{i+1}. Die *Worst-Case*-Kreise C_i' sind wie folgt definiert: Sei A ein Kreis, der durch p_i und p_{i+1} geht und für den p_i der linkeste Punkt auf A ist. Falls A die Gerade st zwei Mal schneidet, definiere $C_i' = A$. Sonst definiere C_i' als Kreis, der durch p_i und p_{i+1} geht und der tangent zu st ist.

Im Folgenden bezeichnet w_i' den *linkesten* (*westlichsten*) Punkt des *Worst-Case*-Kreises C_i'. p_ip_{i+1} bezeichnet die Verbindungsstrecke zwischen p_i und p_{i+1}. Es gibt drei disjunkte Typen von *Worst-Case*-Kreisen (siehe Abb. 6):

- Typ A_1: $p_i \neq w'_i$, $p_i p_{i+1}$ schneidet nicht st und C'_i is tanget zu st.
- Typ A_2: $p_i = w'_i$ und $p_i p_{i+1}$ schneidet nicht st.
- Typ B: $p_i = w'_i$ und $p_i p_{i+1}$ schneidet st.

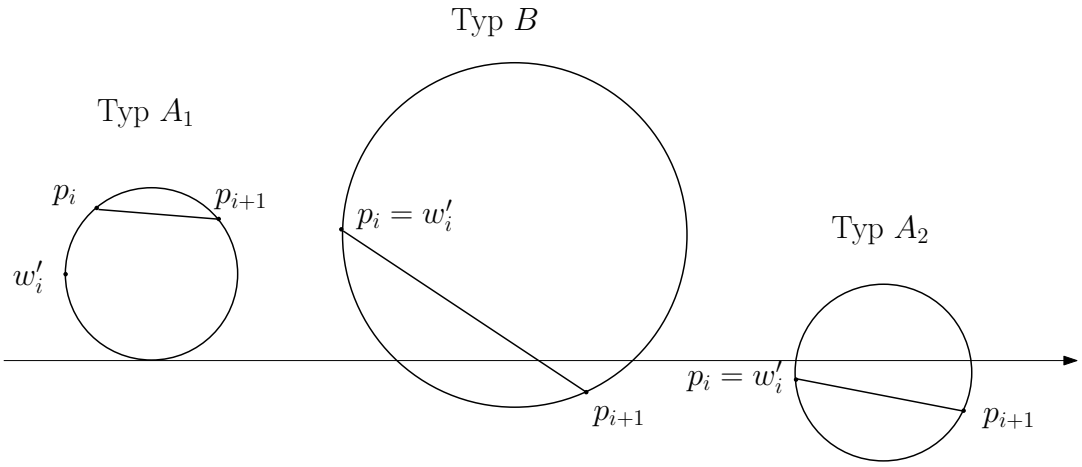


Abbildung 6 Die drei Typen von *Worst-Case*-Kreisen: A_1, A_2 und B.

Sei nun für beliebige Punkte p und q auf C'_i der Ausdruck $\mathcal{A}'_i\langle p, q\rangle$ definiert als der Kreisbogen von p nach q auf C'_i, der die gleiche Orientierung hat wie $\mathcal{A}_i\langle p_i, p_{i+1}\rangle$. Sei $\mathcal{A}'$ die Vereinigung der $\mathcal{A}'_i\langle p_i, p_{i+1}\rangle$. Es gilt $|\mathcal{A}_i\langle p_i, p_{i+1}\rangle| \leq |\mathcal{A}'_i\langle p_i, p_{i+1}\rangle|$ und damit $|\mathcal{A}| \leq |\mathcal{A}'|$. Im Beweis von Theorem 1 wird $|\mathcal{A}'|$ nach oben abgeschätzt. Die Schranke ist dann auch eine Schranke für $|\mathcal{A}|$ und damit für die Länge des vom Algorithmus ausgegebenen Pfades.

Der Beweis von Theorem 1 gliedert sich in drei Teile: Zunächst wird der letzte *Worst-Case*-Kreisbogen mit Hilfe von Lemma 3 nach oben abgeschätzt. Dann werden mit Hilfe von Lemma 4 die übrigen *Worst-Case*-Kreisbögen nach oben abgeschätzt. Zuletzt werden die Abschätzungen zur Abschätzung von $|\mathcal{A}'|$ zusammengesetzt.

Um den Beweis führen zu können, sind zunächst einige Definitionen notwendig. $\bar{w}'_i$ bezeichnet die orthogonale Projektion von w'_i auf die Gerade st. $\mathcal{P}_i$ ist die Vereinigung der senkrechten Strecke von $\bar{w}'_i$ nach w'_i mit dem Kreisbogen von w'_i nach p_i auf C'_i, der die gleiche Orientierung hat wie $\mathcal{A}'_i\langle p_i, p_{i+1}\rangle$: $\mathcal{P}_i = \bar{w}'_i w'_i + \mathcal{A}'_i\langle w'_i, p_i\rangle$ (siehe Abb. 7).

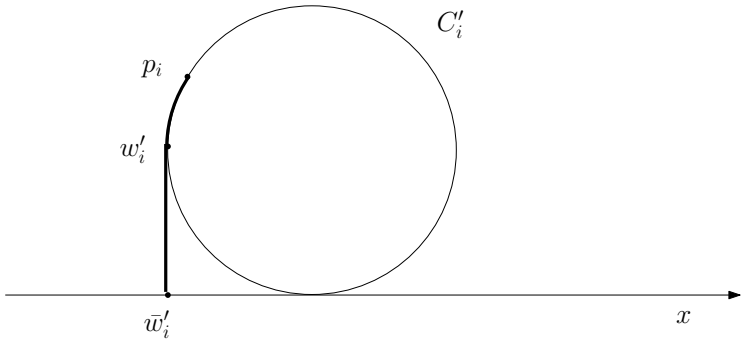


Abbildung 7 $\mathcal{P}_i = \bar{w}'_i w'_i + \mathcal{A}'_i\langle w'_i, p_i\rangle$.

Ein weiteres wichtiges Konzept ist das der *Snail Curve* (siehe Abb. 8). Ebenso wie $\mathcal{P}_i$ ist eine Snail Curve eine Vereinigung einer senkrechten Strecke mit einem Kreisbogen. $\mathcal{P}_i$ und

bestimmte Snail Curves werden später benutzt, um die *Worst-Case*-Kreisbögen nach oben abzuschätzen. Die Definition der Snail Curve ist im Folgenden gegeben.

▶ **Definition 2.** Seien p und q zwei Punkte mit $x(p) < x(q)$ und $y(p) = y(q)$. Sei S der Kreis über der Geraden pq der tangent ist zu pq am Punkt q und der tangent ist zur Geraden $x = x(p)$ an einem Punkt p'. Die *Snail Curve* $\mathcal{S}_{p,q}$ ist die Vereinigung von pp' mit dem im Uhrzeigersinn orientierten Kreisbogen von p' nach q auf S.

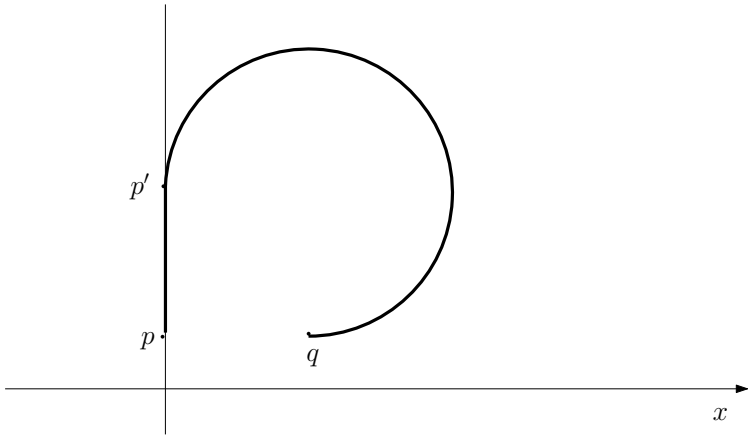


Abbildung 8 Die Snail Curve $\mathcal{S}_{p,q}$.

Die Länge der Snail Curve $\mathcal{S}_{p,q}$ ist allein durch $|q-p|$, den Abstand zwischen p und q, bestimmt. Der Kreis S aus Definition 2 hat den Umfang $|S| = 2\pi|q-p|$. Wegen $|p'-p| = |q-p|$ gilt für die Länge der Snail Curve:

$$|\mathcal{S}_{p,q}| = |q-p| + \frac{3}{4} \cdot 2\pi|q-p| = (1 + 3\frac{\pi}{2})|q-p| \tag{1}$$

Mit den obigen Definitionen kann nun Lemma 3 den letzten *Worst-Case*-Kreisbogen abschätzen (siehe Abb. 9).

▶ **Lemma 3.** $|\mathcal{P}_{k-1}| + |\mathcal{A}'_{k-1}\langle p_{k-1}, t\rangle| \leq |\mathcal{S}_{\bar{w}'_{k-1},t}|$.

Beweis. Dies folgt daraus, dass der Pfad $\mathcal{P}_{k-1} + \mathcal{A}'_{k-1}\langle p_{k-1}, t\rangle$ konvex ist und innerhalb von $\mathcal{S}_{\bar{w}'_{k-1},t}$ liegt. ◀

Sei nun f_i der erste Punkt p_j nach p_i $(i < j)$, so dass p_ip_j die Gerade st schneidet. Sei $\bar{f}_i$ die orthogonale Projektion von f_i auf die Gerade st.

Das folgende Lemma gibt eine Abschätzung für die übrigen *Worst-Case*-Kreisbögen. Es ist das Key-Lemma für den Beweis von Theorem 1.

▶ **Lemma 4** (Key-Lemma). *Für alle* $0 < i < k$ *und* $\delta = 0.185043874$ *gilt:*
$|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| + |y(f_{i-1})| \leq |\mathcal{P}_i| + |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + |y(f_i)| + \delta|\bar{f}_{i-1}\bar{f}_i|$.

Beweis. Da in Lemma 4 die Indizes i und $i-1$ auftauchen, müssen für den Beweis zwei aufeinander folgende *Worst-Case*-Kreise C'_i und C'_{i-1} betrachtet werden. Dabei muss zwischen den unterschiedlichen Typen von *Worst-Case*-Kreisen unterschieden werden. Der Beweis wird für drei Fälle separat geführt. Diese sind:

- Fall 1a: C'_{i-1} ist vom Typ A_1 oder vom Typ A_2 und C'_i ist vom Typ A_2 oder vom Typ B

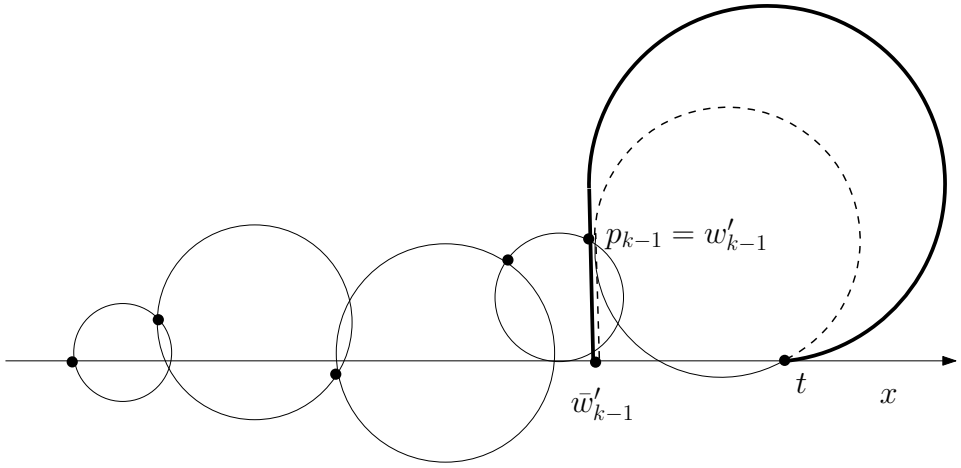


Abbildung 9 Visualisierung von Lemma 3. Die *Worst-Case*-Kreisbögen sind mit normalen Linien dargestellt. $\mathcal{P}_{k-1} + \mathcal{A}'_{k-1}\langle p_{k-1}, t\rangle$ ist mit gestrichelten Linien dargestellt, $\mathcal{S}_{\bar{w}'_{k-1},t}$ mit durchgezogenen, fetten Linien.

- Fall 1b: C'_{i-1} ist vom Typ A_1 oder vom Typ A_2 und C'_i ist vom Typ A_1
- Fall 2: C'_{i-1} ist vom Typ B.

Fall 1a und Fall 1b haben gemeinsam, dass C'_{i-1} vom Typ A_1 oder vom Typ A_2 ist. Das bedeutet, dass in diesem Fall p_ip_{i+1} nicht die Gerade st scheidet. Daraus folgt $f_{i-1} = f_i$. Dadurch vereinfacht sich die zu zeigende Ungleichung. Es bleibt zu zeigen, dass für alle $0 < i < k$ und $\delta = 0.185043874$ gilt:

$$|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| \leq |\mathcal{P}_i| + |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}|.$$

Fall 1a: Wir führen nun den Beweis für Fall 1a. Die Idee ist in Abb. 10 gezeigt.

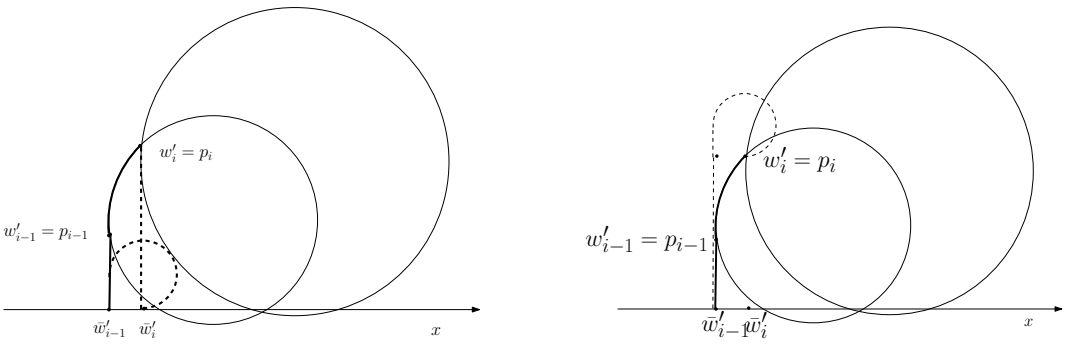


Abbildung 10 Die Idee zum Beweis von Fall 1a. $|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_i, p_{i+1}\rangle|$ ist fett eingezeichnet. Im linken Bild ist $|\mathcal{P}_i| + |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}|$ gestrichelt eingezeichnet. Im rechten Bild ist $|\bar{w}'_{i-1}x| + |\mathcal{S}_{x,p_i}|$ gestrichelt eingezeichnet.

In diesem Fall gilt, dass C'_i vom Typ A_2 oder vom Typ B ist. Daraus folgt $p_i = w'_i$. D.h. es gilt $|\mathcal{P}_i| = |\bar{w}'_iw'_i|$. Sei x die orthogonale Projektion auf die Gerade $\bar{w}'_{i-1}w'_{i-1}$. Dann gilt $|\bar{w}'_iw'_i| = |\bar{w}'_{i-1}x|$. Daraus folgt

$$|\bar{w}'_{i-1}x| + |\mathcal{S}_{x,p_i}| = |\bar{w}'_iw'_i| + |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| = |\mathcal{P}_i| + |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}|. \tag{2}$$

Da $\mathcal{P}_{i-1} + \mathcal{A}'_{i-1}\langle p_i, p_{i+1}\rangle$ konvex ist und innerhalb von $\bar{w}'_{i-1}x + \mathcal{S}_{x,p_i}$ liegt, gilt:

$$|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_i, p_{i+1}\rangle| \leq |\bar{w}'_{i-1}x| + |\mathcal{S}_{x,p_i}|. \tag{3}$$

Gleichung (2) und Ungleichung (3) zusammen ergeben

$$|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_i, p_{i+1}\rangle| \leq |\mathcal{P}_i| + |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}|. \tag{4}$$

Dies schließt den Beweis für Fall 1a ab.

Fall 1b: Für diesen Fall betrachte man Abb. 11.

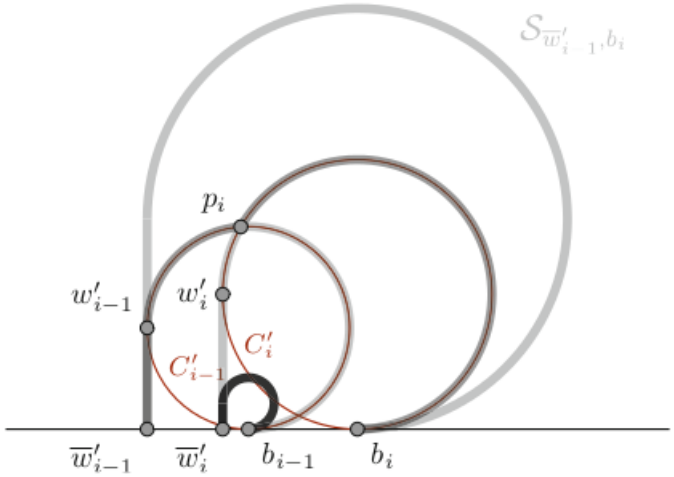


Abbildung 11 Visualisierung von Fall 1b. $\mathcal{P}_{i-1} + \mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle + \mathcal{A}'_i\langle p_i, b_i\rangle$ ist in dunkelgrau eingezeichnet, $\mathcal{S}_{\bar{w}'_{i-1},b_i}$, $\mathcal{P}_i$ und $\mathcal{A}'_{i-1}\langle p_i, b_{i-1}\rangle$ in hellgrau, $\mathcal{S}_{\bar{w}'_i,b_{i-1}}$ in schwarz.

Im Fall 1b ist C'_{i-1} vom Typ A_1 oder vom Typ A_2 und C'_i vom Typ A_1. Es gilt, dass $|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle|$ am längsten ist, wenn C'_{i-1} vom Typ A_1 ist. Es wird daher im folgenden angenommen, dass sowohl C'_{i-1} als auch C'_i vom Typ A_1 sind. Seien b_{i-1} bzw. b_i die Berührpunkte von C'_{i-1} bzw. C'_i mit der Geraden st. Dann gilt

$$|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| + |\mathcal{A}'_i\langle p_i, b_i\rangle| = |\mathcal{S}_{\bar{w}'_{i-1},b_{i-1}}| + |\mathcal{S}_{\bar{w}'_i,b_i}| - |\mathcal{A}'_{i-1}\langle p_i, b_{i-1}\rangle| - |\mathcal{P}_i| \tag{5}$$

Es werden zwei Fälle unterschieden: $x(b_{i-1}) \leq x(\bar{w}'_i)$ und $x(b_{i-1}) > x(\bar{w}'_i)$.

Sei zunächst $x(b_{i-1}) \leq x(\bar{w}'_i)$. Für die rechte Seite von Gleichung (5) gilt

$$|\mathcal{S}_{\bar{w}'_{i-1},b_{i-1}}| + |\mathcal{S}_{\bar{w}'_i,b_i}| - |\mathcal{A}'_{i-1}\langle p_i, b_{i-1}\rangle| - |\mathcal{P}_i| \leq |\mathcal{S}_{\bar{w}'_{i-1},b_{i-1}}| + |\mathcal{S}_{\bar{w}'_i,b_i}|.$$

Für $x(b_{i-1}) \leq x(\bar{w}'_i)$ gilt

$$|\bar{w}'_{i-1}b_{i-1}| + |\bar{w}'_i b_i| \leq |\bar{w}'_{i-1}b_i|.$$

$$\Rightarrow |\mathcal{S}_{\bar{w}'_{i-1},b_{i-1}}| + |\mathcal{S}_{\bar{w}'_i,b_i}| \leq |\mathcal{S}_{\bar{w}'_{i-1},b_i}|.$$

Sei nun $x(b_{i-1}) > x(\bar{w}'_i)$ (wie in Abb. 11). Es ist dann $|\mathcal{S}_{\bar{w}'_i,b_{i-1}}| \leq |\mathcal{A}'_{i-1}\langle p_i, b_{i-1}\rangle| + |\mathcal{P}_i|$. Daraus folgt für die rechte Seite von Gleichung (5)

$$|\mathcal{S}_{\bar{w}'_{i-1},b_{i-1}}| + |\mathcal{S}_{\bar{w}'_i,b_i}| - |\mathcal{A}'_{i-1}\langle p_i, b_{i-1}\rangle| - |\mathcal{P}_i| \leq |\mathcal{S}_{\bar{w}'_{i-1},b_{i-1}}| + |\mathcal{S}_{\bar{w}'_i,b_i}| - |\mathcal{S}_{\bar{w}'_i,b_{i-1}}| = |\mathcal{S}_{\bar{w}'_{i-1},b_i}|.$$

Es folgt, dass in beiden Fällen die rechte Seite von Gleichung (5) mit $|\mathcal{S}_{\bar{w}'_{i-1},b_i}|$ nach oben abgeschätzt werden kann. Somit kann auch die linke Seite nach oben abgeschätzt werden. Es gilt

$$|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| + |\mathcal{A}'_i\langle p_i, b_i\rangle| \leq |\mathcal{S}_{\bar{w}'_{i-1},b_i}| = |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + |\mathcal{S}_{\bar{w}'_i,b_i}|.$$

Subtraktion von $|\mathcal{A}'_i\langle p_i, b_i\rangle|$ von beiden Seiten der Ungleichung ergibt

$$|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| \leq |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + |\mathcal{S}_{\bar{w}'_i,b_i}| - |\mathcal{A}'_i\langle p_i, b_i\rangle| = |\mathcal{P}_i| + |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}|.$$

Somit ist Lemma 4 für den Fall 1b bewiesen.

Fall 2: Es wird nun Fall 2 betrachtet: C'_{i-1} ist vom Typ B (siehe Abb. 12).

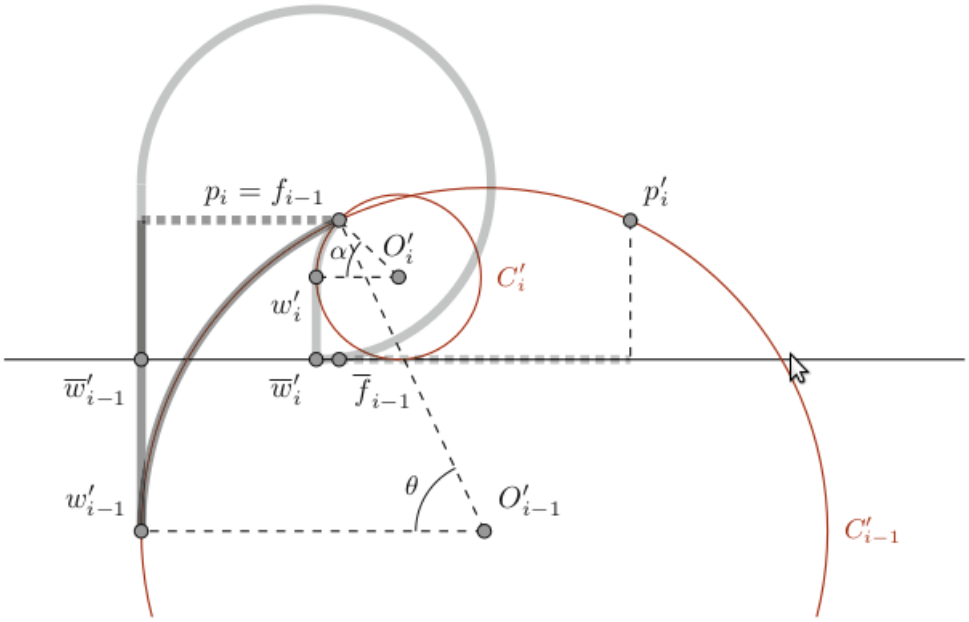


Abbildung 12 Visualisierung von Fall 2 mit der Einschränkung $|y(p_{i-1}) - y(p_i)| \geq |y(p_{i-1}) - y(f_i)|$. $|\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + |\mathcal{P}_i| + \delta|\bar{f}_{i-1}\bar{f}_i|$ ist in hellgrau eingezeichnet, $|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| + |y(f_{i-1})|$ in dunkelgrau.

Es wird hier nur der Fall betrachtet, für den gilt: $|y(p_{i-1}) - y(p_i)| \geq |y(p_{i-1}) - y(f_i)|$. Für den Fall $|y(p_{i-1}) - y(p_i)| < |y(p_{i-1}) - y(f_i)|$ siehe [1]. Sei D der Abstand zwischen rechter und linker Seite von Ungleichung 4:

$$D = |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + |\mathcal{P}_i| + \delta|\bar{f}_{i-1}\bar{f}_i| + |y(f_i)| - |\mathcal{P}_{i-1}| - |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| - |y(f_{i-1})|. \tag{6}$$

Es ist nun zu zeigen, dass $D \geq 0$.

Sei $\theta = \angle w'_{i-1}O'_{i-1}p_i$ und $\alpha = \angle w'_iO'_ip_i$, wobei O'_{i-1} bzw. O'_i die Kreismittelpunkte von C'_{i-1} bzw. C'_i sind. Zunächst gilt folgendes Lemma (ohne Beweis).

▶ **Lemma 5.** *Es gilt für alle* $i = 1, ..., k-1$*:* $0 \leq \alpha \leq \theta \leq 3\frac{\pi}{2}$.

O.B.d.A. sei der Radius von C'_{i-1} 1. Es ist

$$\theta = |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle|. \tag{7}$$

Da C'_{i-1} vom Typ B ist, ist $p_{i-1} = w'_{i-1}$, und daher

$$|\mathcal{P}_{i-1}| = |w'_{i-1}\bar{w}'_{i-1}|. \tag{8}$$

Da der Radius von C'_{i-1} 1 ist, gilt:

$$\sin(\theta) = |w'_{i-1}\bar{w}'_{i-1}| + |y(f_{i-1})|. \tag{9}$$

Aus (8) und (9) folgt

$$\sin(\theta) = |\mathcal{P}_{i-1}| + |y(f_{i-1})|. \tag{10}$$

Da der Radius von C'_{i-1} 1 ist, gilt:

$$|x(O'_{i-1}) - x(\bar{w}'_{i-1})| = 1 \text{ und} \tag{11}$$

$$|x(O'_{i-1}) - x(f_{i-1})| = \cos(\theta)|. \tag{12}$$

$$\Rightarrow |\bar{w}'_{i-1}\bar{f}_{i-1}| = |x(f_{i-1}) - x(\bar{w}'_{i-1})| = 1 - \cos(\theta). \tag{13}$$

Sei nun $r = |y(w'_i)|$. Falls C'_i vom Typ A_1 ist, dann ist C'_i tangent zu st und hat damit den Radius r. Es folgt $|\mathcal{P}_i| = (1+\alpha)r$ und $|\bar{w}'_i\bar{f}'_{i-1}| = (1-\cos(\alpha))r$.

Falls C'_i vom Typ A_2 oder B ist, dann ist $\alpha = 0$, $p_i = w'_i$ und $\bar{w}'_i = \bar{f}'_{i-1}$. Daraus folgt, dass auch in diesem Fall gilt $|\mathcal{P}_i| = r = (1+\alpha)r$ und $|\bar{w}'_i\bar{f}'_{i-1}| = 0 = (1-\cos(\alpha))r$. Es gilt also für Fall 2 immer:

$$|\mathcal{P}_i| = (1+\alpha)r \text{ und} \tag{14}$$

$$|\bar{w}'_i\bar{f}'_{i-1}| = (1-\cos(\alpha))r. \tag{15}$$

Aus (13) und (15) folgt

$$|\bar{w}'_{i-1}\bar{w}'_i| = 1 - \cos(\theta) - (1-\cos(\alpha))r.$$

Daraus folgt mit Gleichung (1)

$$|\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| = (1+3\frac{\pi}{2})(1-\cos(\theta)-(1-\cos(\alpha))r). \tag{16}$$

Mit (7), (10), (14) und (16) kann nun Gleichung (6) umformuliert werden zu:

$$\begin{aligned} D &= (1+3\frac{\pi}{2})(1-\cos(\theta)-(1-\cos(\alpha))r) + (1+\alpha)r + \delta|\bar{f}_{i-1}\bar{f}_i| + |y(f_i)| - \theta - \sin(\theta) \\ &= r(1+\alpha-(1+3\frac{\pi}{2})(1-\cos(\alpha))) + (1+3\frac{\pi}{2})(1-\cos(\theta)) \\ &\quad + \delta|\bar{f}_{i-1}\bar{f}_i| + |y(f_i)| - \theta - \sin(\theta). \end{aligned}$$

Es müssen nun zwei Fälle unterschieden werden: $\theta \leq \frac{\pi}{4}$ und $\theta > \frac{\pi}{4}$. Im folgenden wird der Beweis für den Fall $\theta \leq \frac{\pi}{4}$ geführt. Für den anderen Fall siehe [1].

Sei nun $\theta \leq \frac{\pi}{4}$. Wegen Lemma 5 gilt dann auch $\alpha \leq \frac{\pi}{4}$. Sei p'_i die Spiegelung von p_i an der senkrechten Gerade $x = x(O'_{i-1})$. Wegen $\theta \leq \frac{\pi}{4}$ gilt

$$x(p'_i) > x(O'_i) > x(p_i).$$

Es gilt

$$|y(p_{i-1}) - y(p_i)| < |y(p_{i-1}) - y(p)|$$

für alle p außerhalb von C_{i-1} mit $x(p_i) \leq x(p) \leq x(p'_i)$. Da nach Annahme $|y(p_{i-1}) - y(p_i)| \geq |y(p_{i-1}) - y(f_i)|$, ist $x(f_i) \geq x(p'_i)$. Daraus folgt

$$|\bar{f}_{i-1}\bar{f}_i| \geq |f_{i-1}p'_i| = 2\cos(\theta).$$

Daraus folgt für D:

$$D \geq r(1+\alpha-(1+3\frac{\pi}{2})(1-\cos(\alpha)))+(1+3\frac{\pi}{2})(1-\cos(\theta))+2\delta\cos(\theta)+|y(f_i)|-\theta-\sin(\theta).$$

Es gilt für $\alpha \in [0, \frac{\pi}{4}]$:

$$1+\alpha-(1+3\frac{\pi}{2})(1-\cos(\alpha) \geq 0,$$

und damit

$$r(1+\alpha-(1+3\frac{\pi}{2})(1-\cos(\alpha)) \geq 0. \tag{17}$$

Durch einfachens Einsetzen kann nachvollzogen werden, dass für $\delta = 0.185043874$ und $\alpha \in [0, \frac{\pi}{4}]$ folgende Ungleichung gilt:

$$(1+3\frac{\pi}{2})(1-\cos(\theta))+2\delta\cos(\theta)+|y(f_i)|-\theta-\sin(\theta) \geq 0. \tag{18}$$

Aus (17) und (18) folgt $D \geq 0$. Dies schließt den Beweis für Fall 2, und damit den Beweis von Lemma 4 ab. ◀

Mit Lemma 3 und Lemma 4 kann nun Theorem 1 bewiesen werden.

Beweis. (Theorem 1) Lemma 4 gibt eine Abschätzung für alle *Worst-Case*-Kreisbögen bis auf den letzten. Zur Erinnerung: Für alle $0 < i < k$ und $\delta = 0.185043874$ gilt:

$$|\mathcal{P}_{i-1}| + |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| + |y(f_{i-1})| \leq |\mathcal{P}_i| + |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + |y(f_i)| + \delta|\bar{f}_{i-1}\bar{f}_i|.$$

Subtraktion von $|\mathcal{P}_{i-1}|$ und $|y(f_{i-1})|$ auf beiden Seiten der Ungleichung ergibt:

$$|\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| \leq |\mathcal{P}_i| - |\mathcal{P}_{i-1}| + |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + |y(f_i)| - |y(f_{i-1})| + \delta|\bar{f}_{i-1}\bar{f}_i|.$$

Die rechten und linken Seiten der $k-1$ Ungleichungen werden nun jeweils aufsummiert:

$$\sum_{i=1}^{k-1} |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| \leq \sum_{i=1}^{k-1}(|\mathcal{P}_i| - |\mathcal{P}_{i-1}| + |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + |y(f_i)| - |y(f_{i-1})| + \delta|\bar{f}_{i-1}\bar{f}_i|)$$

Hineinziehen des Summenzeichens in die Klammer ergibt:

$$\begin{aligned}&\sum_{i=1}^{k-1} |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| \\ &\leq \sum_{i=1}^{k-1} |\mathcal{P}_i| - \sum_{i=1}^{k-1} |\mathcal{P}_{i-1}| + \sum_{i=1}^{k-1} |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + \sum_{i=1}^{k-1} |y(f_i)| - \sum_{i=1}^{k-1} |y(f_{i-1})| + \sum_{i=1}^{k-1} \delta|\bar{f}_{i-1}\bar{f}_i|\end{aligned} \tag{19}$$

Lemma 4 gibt eine Abschätzung für den letzten *Worst-Case*-Kreisbogen. Zur Erinnerung:

$$|\mathcal{P}_{k-1}| + |\mathcal{A}'_{k-1}\langle p_{k-1}, t\rangle| \leq |\mathcal{S}_{\bar{w}'_{k-1},t}|$$

Subtraktion von $|\mathcal{P}_{k-1}|$ auf beiden Seiten der Ungleichung ergibt:

$$|\mathcal{A}'_{k-1}\langle p_{k-1}, t\rangle| \leq |\mathcal{S}_{\bar{w}'_{k-1},t}| - |\mathcal{P}_{k-1}| \tag{20}$$

Ungleichung (20) kann nun zur Ungleichung (19) addiert werden. Dies ergibt:

$$\begin{aligned}
&\sum_{i=1}^{k} |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle| \\
&\leq \sum_{i=1}^{k-1} |\mathcal{P}_i| - \sum_{i=1}^{k} |\mathcal{P}_{i-1}| + \sum_{i=1}^{k} |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + \sum_{i=1}^{k-1} |y(f_i)| - \sum_{i=1}^{k-1} |y(f_{i-1})| + \sum_{i=1}^{k-1} \delta|\bar{f}_{i-1}\bar{f}_i| \\
&= \sum_{i=1}^{k-1} |\mathcal{P}_i| - \sum_{i=0}^{k-1} |\mathcal{P}_i| + \sum_{i=1}^{k} |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + \sum_{i=1}^{k-1} |y(f_i)| - \sum_{i=0}^{k-2} |y(f_{i-1})| + \sum_{i=1}^{k-1} \delta|\bar{f}_{i-1}\bar{f}_i| \\
&= -|\mathcal{P}_0| + \sum_{i=1}^{k} |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + |y(f_{k-1})| - |y(f_0)| + \sum_{i=1}^{k-1} \delta|\bar{f}_{i-1}\bar{f}_i| \\
&= -|\mathcal{P}_0| + \sum_{i=1}^{k} |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + |y(t)| - |y(f_0)| + \sum_{i=1}^{k-1} \delta|\bar{f}_{i-1}\bar{f}_i| \\
&= -|\mathcal{P}_0| + \sum_{i=1}^{k} |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + 0 - |y(f_0)| + \sum_{i=1}^{k-1} \delta|\bar{f}_{i-1}\bar{f}_i| \\
&\leq \sum_{i=1}^{k} |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| + \sum_{i=1}^{k-1} \delta|\bar{f}_{i-1}\bar{f}_i|.
\end{aligned} \tag{21}$$

Mit Gleichung (1) gilt

$$\sum_{i=1}^{k} |\mathcal{S}_{\bar{w}'_{i-1},\bar{w}'_i}| = \sum_{i=1}^{k} (1 + 3\frac{\pi}{2})|\bar{w}'_i - \bar{w}'_{i-1}| = (1 + 3\frac{\pi}{2}) \sum_{i=1}^{k} |\bar{w}'_i - \bar{w}'_{i-1}| = (1 + 3\frac{\pi}{2})|st|. \tag{22}$$

Außerdem gilt

$$\sum_{i=1}^{k-1} \delta|\bar{f}_{i-1}\bar{f}_i| = \delta \sum_{i=1}^{k-1} |\bar{f}_{i-1}\bar{f}_i| = \delta|st|. \tag{23}$$

Da mit $\mathcal{A}'$ die Vereinigung aller *Worst-Case*-Kreisbögen $\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle$ bezeichnet wird, gilt

$$|\mathcal{A}'| = \sum_{i=1}^{k} |\mathcal{A}'_{i-1}\langle p_{i-1}, p_i\rangle|. \tag{24}$$

Mit (22), (23) und (24) wird Ungleichung (21) zu

$$|\mathcal{A}'| \leq (1 + 3\frac{\pi}{2})|st| + \delta|st|.$$

Setzt man nun $\delta = 0.185043874$ ein (siehe Lemma 4), ergibt sich

$$|\mathcal{A}'| \leq (1.185043874 + 3\frac{\pi}{2})|st|.$$

Da $|\mathcal{A}'|$ eine obere Schranke für einen Pfad von Startknoten s nach Zielknoten t ist, gilt

$$\text{Länge des Pfades von } s \text{ nach } t \leq (1.185043874 + 3\frac{\pi}{2})|st|.$$

Division beider Seiten der Ungleichung durch $|st|$ ergibt

$$\frac{\text{Länge des Pfades von } s \text{ nach } t}{|st|} \leq (1.185043874 + 3\frac{\pi}{2}).$$

Da dies für alle s und t gilt, folgt:

$$\max_{(s,t)} \frac{\text{Länge des Pfades von } s \text{ nach } t}{|st|} \leq (1.185043874 + 3\frac{\pi}{2}).$$

Damit ist $(1.185043874 + 3\frac{\pi}{2}) \approx 5.89743256$ eine obere Schranke für die Routing Ratio des Algorithmus, was den Beweis von Theorem 1 abschließt.

◀

6.2.3 Untere Schranken

In diesem Abschnitt wird eine untere Schranke für den vorgestellten Routing-Algorithmus gegeben. Weiterhin werden werden allgemeine untere Schranken für Competitive und Routing Ratio angegeben. Für Details zu den Beweisen siehe [1].

▶ **Theorem 6.** *Der vorgestellte Routing-Algorithmus hat eine Routing Ratio von mindestens* 5.7282*.*

Dies kann mit Hilfe des in Abb. 13 gezeigten Graphen, der eine, für den gegeben Algorithmus, sehr ungünstige Anordnung an Knoten hat, eingesehen werden.

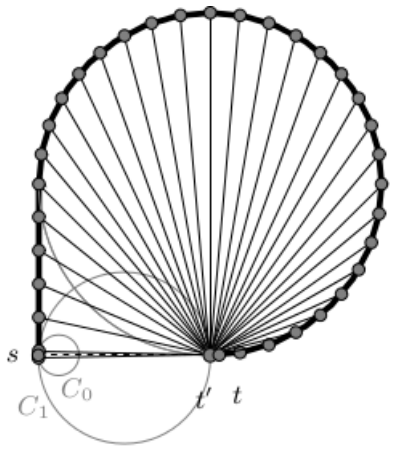


Abbildung 13 Veranschaulichung der unteren Schranke des Routing-Algorithmus. Der gezeigte Graph hat eine sehr ungünstige Anordnung der Knoten, so dass der ausgegebene Pfad von s nach t einem „großen Bogen“ entspricht.

Folgende allgemeine untere Schranken werden gezeigt.

▶ **Theorem 7.** *Es existiert kein deterministischer Online-Routing-Algorithmus auf Delaunay-Triangulationen mit einer Routing Ratio kleiner oder gleich* 1.7018 *oder mit einer Competitive Ratio kleiner oder gleich* 1.2327*.*

▶ **Theorem 8.** *Es existiert kein deterministischer Online-Routing-Algorithmus auf L_1- und L_∞-Delaunay-Triangulationen mit einer Routing Ratio kleiner $2 + \sqrt{2}/2 \approx 2.7071$ oder mit einer Competitive Ratio kleiner oder gleich $(2 + \sqrt{2}/2)/(1 + \sqrt{2}) \approx 1.1213$.*

6.3 Zusammenfassung

Es wurde ein Online-Routing-Algorithmus für Delaunay-Triangulationen, der eine Anpassung des Online-Routing-Algorithmus für L_1-Delaunay-Triangulationen von Chew ist, vorgestellt. Es wurde der Beweis für die obere Schranke der Routing Ratio vorgestellt. Dabei wurde der vom Algorithmus ausgegebene Pfad mit *Worst-Case*-Kreisbögen abgeschätzt. Für die Abschätzung der Länge der *Worst-Case*-Kreisbögen wurden unterschiedliche Typen von zwei aufeinander folgenden *Worst-Case*-Kreisen betrachtet und der Beweis für drei Fälle separat geführt. Aus der Abschätzung der einzelnen *Worst-Case*-Kreisbögen konnte eine Abschätzung für die Vereinigung der *Worst-Case*-Kreisbögen und damit für den Pfad gefolgert werden.

Weiterhin wurden eine untere Schranke für die Routing Ratio des Routing-Algorithmus und allgemeine untere Schranken für Online-Routing-Algorithmen auf Delaunay-Triangulationen und L_1- und L_∞-Delaunay-Triangulationen angegeben.

Referenzen

1 Nicolas Bonichon, Prosenjit Bose, Jean-Lou Carufel, Ljubomir Perković, and André Renssen. *Algorithms - ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, chapter Upper and Lower Bounds for Online Routing on Delaunay Triangulations, pages 203–214. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

2 Nicolas Bonichon, Cyril Gavoille, Nicolas Hanusse, and Ljubomir Perković. *Algorithms – ESA 2012: 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, chapter The Stretch Factor of L_1- and L_∞-Delaunay Triangulations, pages 205–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

3 Prosenjit Bose, Jean-Lou Carufel, Stephane Durocher, and Perouz Taslakian. *Algorithm Theory – SWAT 2014: 14th Scandinavian Symposium and Workshops, Copenhagen, Denmark, July 2-4, 2014. Proceedings*, chapter Competitive Online Routing on Delaunay Triangulations, pages 98–109. Springer International Publishing, Cham, 2014.

4 Prosenjit Bose, Rolf Fagerberg, André van Renssen, and Sander Verdonschot. Competitive routing in the half-θ_6-graph. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 1319–1328. SIAM, 2012.

5 L.Paul Chew. There are planar graphs almost as good as the complete graph. *Journal of Computer and System Sciences*, 39(2):205–219, 1989.

6 David P. Dobkin, Steven J. Friedman, and K.J. Supowit. Delaunay graphs are almost as good as complete graphs. In *28th Symposium on Foundations of Computer Science, 1987*, pages 20–26. IEEE, 1987.

7 Ge Xia. The Stretch Factor of the Delaunay Triangulation Is Less than 1.998. *SIAM Journal on Computing*, 42(4):1620–1659, 2013.

8 Ge Xia and Liang Zhang. Toward the Tight Bound of the Stretch Factor of Delaunay Triangulations. In *CCCG*, 2011.

7 Verteilte Algorithmen für maximal unabhängige Mengen

David Vogelbacher

Zusammenfassung

Ein Maximal Independent Set (MIS, deutsch: Inklusionsmaximale unabhängige Menge) zu berechnen ist ein grundlegendes Problem der Graphentheorie. Die zentralisierte Berechnung ist mittels eines Greedy-Algorithmus in linearer Zeit möglich. Interessant ist das Problem auch im Kontext des verteilten Rechnens. Es stellt sich die Frage, wie man ein MIS eines verteilten Netzwerks möglichst effizient (z.B. bezüglich Laufzeit, Kommunikationsaufwand) berechnen kann. In dieser Seminararbeit werden wir zunächst den Algorithmus von Luby (1985) [6] einführen, der eine erwartete Laufzeit von $O(\log n)$ Runden hat. Danach werden wir den Algorithmus von Ghaffari (2015) [4] detailliert betrachten. Dieser Algorithmus basiert auf dem Algorithmus von Luby und hat eine verbesserte Laufzeit von $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$. Damit kommt er nahe heran an die untere Schranke von $\Omega(\log \Delta + \sqrt{\log n})$ von *Kuhn et al.* [5].

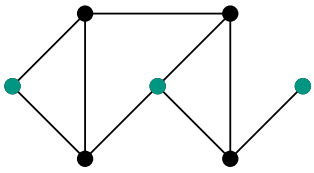
7.1 Einführung

Im Folgenden werden wir zuerst einige wichtige Begriffe einführen und das Thema motivieren. Im nächsten Kapitel werden wir dann auf den Stand der Forschung eingehen. Danach schauen wir uns den Algorithmus von Luby [6] und den Algorithmus von Ghaffari [4] an.

7.1.1 Inklusionsmaximale unabhängige Menge

▶ **Definition 1** (Inklusionsmaximale unabhängige Menge)**.** Eine Teilmenge $S \subseteq V$ der Knotenmenge eines Graphen $G = (V, E)$ nennen wir *unabhängig*, wenn keine zwei Knoten aus S zueinander adjazent sind. Ist S inklusionsmaximal (d.h., das Hinzufügen eines weiteren Knotens verletzt die Unabhängigkeit), dann ist S eine Inklusionsmaximale unabhängige Menge (MIS).

Abbildung 1 Die grünen Knoten bilden ein MIS des Graphen.



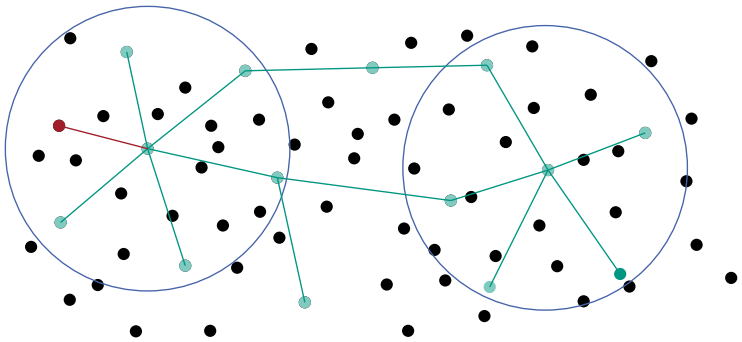
Beispielsweise bilden die grünen Knoten in Abbildung 1 ein MIS. Zu beachten ist, dass wir nur Inklusionsmaximale unabhängige Mengen betrachten und nicht Kardinalitätsmaximale unabhängige Mengen. Das zweite Problem ist NP-schwer und wird im Folgenden nicht weiter betrachtet.

7.1.2 Motivation

MIS ist ein grundlegendes Problem der Graphentheorie und des verteilten Rechnens [4]. Praktisch gesehen kann es in Netzwerken sinnvoll sein bestimmte Leader-Knoten zu bestimmen, die bestimmte Aufgaben übernehmen. Um solche Leader-Knoten zu bestimmen

kann man beispielsweise ein MIS des Netzwerkes zu Hilfe nehmen. Ein etwas spezielleres Anwendungsproblem ist das Routing in Sensornetzwerken bei Broadcasts [3] (ein Knoten will eine Nachricht an alle anderen Knoten schicken). Der naive Ansatz wäre die Nachricht von einem Knoten an alle seine Nachbarn zu schicken. Diese schicken die Nachricht dann wieder weiter an ihre Nachbarn. Dies wird so lange fortgesetzt bis jeder Knoten die Nachricht empfangen hat. Dies ist allerdings sehr ineffizient und gerade bei drahtlosen Sensornetzwerken möchte man den Kommunikationsaufwand gering halten um Energie zu sparen. Ein besserer Ansatz ist es die Nachrichten über ein *Backbone* (Connected Dominating Set) zu routen [3] (siehe Abbildung 2): Das Backbone sind eine Reihe von verbunden Knoten im Netzwerk, zu dem jeder Knoten adjazent ist. Um nun einen Broadcast durchzuführen, schickt der Knoten die Nachricht an den ihm nächstgelegenen Knoten des Backbones. Die Nachricht wird dann zuerst im Backbone verteilt. Schlussendlich leiten die Knoten des Backbones die Nachricht an alle anderen Knoten weiter. Dies spart viel Kommunikation und Energie. Zwar ist ein Backbone kein MIS, aber es gibt Verfahren um aus einem MIS ein Backbone zu berechnen [3].
Schließlich gibt es auch noch andere Probleme der Graphentheorie wie *Maximal Matching* und *Graphfärben* die sich auf das MIS-Problem reduzieren lassen [4].

Abbildung 2 Die grünen Knoten bilden ein Backbone des Graphen. Der rote Knoten schickt seine Nachricht an den nächstgelegenen Backbone-Knoten und von da aus wird sie über das Backbone an alle Knoten verteilt.



7.2 Modell von verteiltem Rechnen

Es gibt viele verschiedene Arten von verteilten Systemen. Beispielsweise Systeme mit gemeinsamem Speicher (z.B. PRAM, Multi-Core Rechner), das Internet oder drahtlose Sensornetzwerke. Wir werden uns hier auf ein bestimmtes Modell beschränken, das Modell LOCAL [8, 4]:

▶ **Definition 2** (LOCAL)**.** Das verteilte System wird durch einen Graph G modelliert. Die Prozessoren des Systems entsprechen den Knoten von G. Die Kanten entsprechen Kommunikationsleitungen. Ein Prozessor kann nur mit seinen Nachbarn in G kommunizieren. Die Kommunikation erfolgt hierbei in synchronen Runden: Pro Runde kann ein Prozessor mit all seinen Nachbarn einmal kommunizieren. Die Laufzeit, die uns interessiert ist die Anzahl an Kommunikationsrunden. Die lokale Rechenzeit hingegen wird als frei angesehen[1].

[1] Für die Algorithmen, die wir im Folgenden anschauen werden, ist die lokale Laufzeit pro Runde konstant.

Abbildung 3 Beispiele von Netzwerken mit acht Prozessoren im LOCAL-Modell: Links kann jeder Prozessor pro Runde nur mit zwei anderen Prozessoren kommunizieren. Das Netzwerk ist ein Kreis. Rechts kann jeder Prozessor pro Runde mit jedem anderen Prozessor kommunizieren.

In Abbildung 3 finden sich zwei Beispiele von im LOCAL-Modell modellierten Netzwerken. Das Ziel der von uns betrachteten, verteilten MIS Algorithmen ist es nun ein MIS für ein gegebenes System im LOCAL-Modell zu bestimmen.

7.3 Wie man ein MIS zentralisiert findet

Zuerst wollen wir uns überlegen wie man ein MIS zentralisiert finden kann. Da wir eine maximale Menge suchen, liegt es nahe greedy vorzugehen: Pro Iteration löschen wir einen Knoten und seine Nachbarn aus dem Graphen, bis der Graph leer ist. Der gelöschte Knoten wird Teil des MIS, seine Nachbarn nicht. Dieser Algorithmus hat lineare Laufzeit in der Anzahl an Knoten und Kanten. Der Pseudocode ist dargestellt in Algorithmus 1. $N(v)$ ist hierbei die Nachbarschaft des Knotens v.

Input : $G = (V, E)$.
Output : Maximal Unabhängige Menge S.
while $V \neq \emptyset$ **do**
  Sei $v \in V$;
  $S := S \cup \{v\}$;
  $V := V \backslash (\{v\} \cup N(v))$;
end

Algorithmus 1 : Greedy Algorithmus für zentralisiertes MIS.

7.4 Stand der Forschung

Von *Luby* [6] und *Alon et al.* [1] stammen Algorithmen um ein MIS in Zeit $O(\log n)$ zu finden. Auf den Algorithmus von Luby werden wir noch genauer eingehen, da der Algorithmus von *Ghaffari* [4] auf ihm aufbaut. *Barenboim et al.* [2] bieten eine sehr gute Übersicht über verschiedene MIS-Algorithmen und deren Laufzeit. Von Ihnen kommt auch ein Algorithmus mit Laufzeitkomplexität $O(\log \Delta \log \log n) + 2^{O(\sqrt{\log \log n})}$ [2]. Hierbei ist Δ der maximale Knotengrad des Graphen. Außerdem stellen sie eine Algorithmus mit Laufzeit $O(\log \Delta \sqrt{\log n})$ vor. *Schneider und Wattenhofer* [9] haben einen deterministischen Algorithmus vorgestellt, der das Problem in Zeit $O(\log^* n)$ löst ($\log^* n$ ist der iterierte Logarithmus, der angibt wie oft der Logarithmus auf n angewandt werden muss, so dass das Ergebnis kleiner oder gleich eins ist), allerdings nur für wachstumsbeschränkte Graphen. Wachstumsbeschränkt bedeutet, dass die Größe einer Maximalen Unabhängigen Menge in der r-Nachbarschaft (Knoten die über höchstens r Kanten erreicht werden können) eines Knoten mit einer polynomiellen Funktion (in Abhängigkeit von r) beschränkt werden kann [9]. Für wachstumsbeschränkte Graphen ist diese Laufzeit asymptotisch optimal [9]. Ein schneller deterministischer Algorithmus

für generelle Graphen kommt von *Panconesi und Srinivasan* [7] mit einer Laufzeit von $O(n^{d\sqrt{\frac{1}{\log n}}})$ (d ist eine Konstante). Von *Kuhn et al.* [5] kommen untere Schranken für das verteilte Finden eines MIS von $\Omega(\log \Delta)$ und $\Omega(\sqrt{\log n})$.

7.5 Algorithmus von Luby (1985) [6]

Dieser Abschnitt basiert auf Michael Luby's *A simple parallel algorithm for the maximal independent set problem* [6]. Dabei halten wir uns an die Darstellung des Algorithmus aus der Vorlesung „Algorithmen für Ad-hoc- und Sensornetze“ von *Fuchs et al.* [3]. Alle algorithmischen Techniken und theoretischen Resultate kommen von diesen Quellen, wenn keine andere Quellenangabe vorliegt.

7.5.1 Idee

Da wir uns nun im LOCAL-Modell befinden, läuft der Algorithmus parallel auf jedem Knoten ab. Dabei muss jeder Knoten selbst entscheiden, ob er Teil des MIS wird (alle Knoten parallel) oder nicht. Da jeder Knoten bei dieser Entscheidung mit seinen Nachbarn konkurriert, ist es sehr nützlich Zufall zu verwenden, um Symmetrien zu brechen. Der Algorithmus findet in Runden statt. Pro Runde wird jeder Knoten versuchen dem MIS mit einer bestimmten Wahrscheinlichkeit beizutreten. Wenn zwei benachbarte Knoten in der gleichen Runde versuchen dem MIS beizutreten, gewinnt derjenige mit dem höheren Knotengrad (wenn beide den gleichen Grad haben keiner) und tritt dem MIS bei. Knoten, die dem MIS beigetreten sind können terminieren. Weiterhin können auch ihre Nachbarn terminieren, da sie nicht mehr Teil des MIS sein können. Außerdem kann ein Knoten sofort dem MIS beitreten und terminieren, wenn er keine aktiven Nachbarn mehr hat. In Abbildung 4 wird dargestellt wie zwei benachbarte Knoten dem MIS beitreten wollen und derjenige mit dem höheren Grad gewinnt.

Abbildung 4 Links: Zwei benachbarte Knoten versuchen dem MIS beizutreten (blau). Rechts: Der Knoten mit dem höheren Grad gewinnt und wird Teil des MIS (grün). Seine Nachbarn terminieren auch (grau).

7.5.2 Der Algorithmus

Die Wahrscheinlichkeit mit der ein Knoten v in einer Runde versucht dem MIS beizutreten ist $1/(2d_v)$, wobei d_v der Grad von v ist. Der Algorithmus wird aus Sicht eines einzelnen Knoten beschrieben und läuft parallel auf allen Knoten synchron ab. Der Kommunikationsaufwand pro Iteration der While-Schleife ist konstant, daher werden wir im Folgenden die Anzahl an Iteration der While-Schleife analysieren. Dies wird uns die Anzahl an Kommunikationsrunden im O-Kalkül liefern. In Algorithmus 2 ist der Pseudocode des Algorithmus dargestellt.

Output : Entscheidung, ob $v \in$ MIS
while *Nicht entschieden* **do**
  $x_v = \begin{cases} \text{blau} & \text{, mit Wahrscheinlichkeit } 1/(2d_v) \\ \text{schwarz} & \text{, sonst} \end{cases}$;
  Sende x_v zu $N(v)$ und empfange x_u von allen $u \in N(v)$;
  if $(x_v = \mathit{blau} \wedge x_u = \mathit{schwarz}\ \forall u \in N(v)\ \mathit{mit}\ d_u \geq d_v) \vee d_v = 0$ **then**
    Trete MIS bei;
    $x_v =$ grün;
    Informiere Nachbarn;
    Terminiere;
  end
  if *Nachbar tritt MIS bei* **then**
    Trete MIS nicht bei;
    $x_v =$ grau;
    Terminiere ;
  end
end

Algorithmus 2 : MIS nach Luby, aus Sicht des Knotens $v \in V$

7.5.3 Laufzeitanalyse

Im Folgenden werden wir die Laufzeit des Algorithmus untersuchen. Pro Runde (Iteration der While-Schleife) hat ein Knoten zwei Möglichkeiten zu terminieren: Entweder er tritt dem MIS selber bei (wird grün) oder einer seiner Nachbarn tritt dem MIS bei (Knoten selber wird grau). Wir wollen nun abschätzen wie groß die Wahrscheinlichkeiten für diese Ereignisse sind. Schlussendlich werden wir dann zeigen, dass es pro Runde einen konstanten Anteil an sogenannten *guten* Kanten gibt, die mit konstanter Wahrscheinlichkeit terminieren (Wir sagen eine Kante terminiert, wenn einer ihrer inzidenten Knoten terminiert). Dadurch wissen wir dann, dass pro Runde erwartet ein konstanter Anteil an Kanten wegfällt. Deshalb ist die erwartete Laufzeit des Algorithmus in $O(\log m) = O(\log n)$. Nun zum ersten Schritt:

▶ **Lemma 3** (Die Wahrscheinlichkeit eines Knotens grün zu werden.)**.** *Ein Knoten v wird in einer Runde mit Wahrscheinlichkeit mindestens $\frac{1}{4d_v}$ grün.*

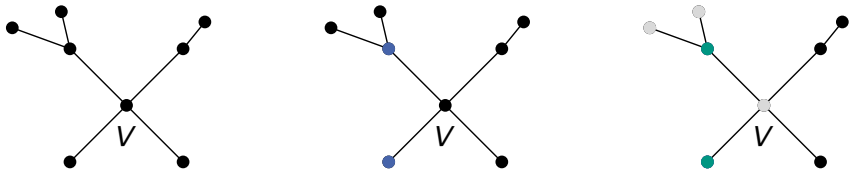
Beweis. Sei $v \in V$ ein Knoten der noch nicht terminiert ist. Per Definition wird v mit Wahrscheinlichkeit $\frac{1}{2d_v}$ blau. Nur ein Nachbar u mit höherem Grad kann die Grünfärbung nun noch verhindern. Die Wahrscheinlichkeit, dass solch ein Nachbar blau wird ist $\frac{1}{2d_u} \leq \frac{1}{2d_v}$. Weiterhin gibt es höchstens d_v viele solche Knoten. Die Wahrscheinlichkeit, dass mindestens einer von den Nachbarn mit höherem Grad blau wird, ist daher höchstens $d_v \cdot \frac{1}{2d_v} = \frac{1}{2}$. Daher ist die Wahrscheinlichkeit, dass v blau und dann grün wird mindestens $\frac{1}{2d_v} \cdot \frac{1}{2} = \frac{1}{4d_v}$. ◀

Nun werden wir zeigen, dass es *gute* Knoten gibt, die mit konstanter Wahrscheinlichkeit terminieren.

▶ **Definition 4** (Guter Knoten)**.** Ein Knoten v ist *gut*, wenn $\sum_{u \in N(v)} \frac{1}{2d_u} \geq \frac{1}{6}$.

Ein Knoten ist also gut, wenn die Wahrscheinlichkeit, dass ein Nachbar sich blau färbt (und danach auch grün) hoch ist. Dadurch ist die Wahrscheinlichkeit hoch, dass der Knoten selber sich grau färben kann, wie beispielsweise zu sehen in Abbildung 5. Dies wird genauer spezifiziert in dem folgenden Lemma:

Abbildung 5 v ist ein guter Knoten. Er hat gute Chancen zu terminieren, indem Nachbarn sich blau und dann grün färben (von links nach rechts).



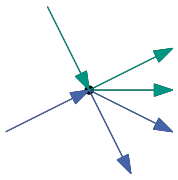
▶ **Lemma 5.** *Ein guter Knoten v wird mit Wahrscheinlichkeit mindestens $\frac{1}{36}$ grau.*

Bewiesen werden kann dies, in dem man Lemma 3 benutzt um abzuschätzen, dass mindestens einer der Nachbarn von v grün wird [3].
Nun zum letzten Schritt der Analyse. Wir nennen eine Kante *gut*, wenn sie zu mindestens einem guten Knoten inzident ist. Aus Lemma 5 wissen wir schon, dass eine gute Kante mit konstanter Wahrscheinlichkeit terminiert (indem ein zu ihr inzidenter Knoten terminiert). Um zu zeigen, dass pro Runde erwartet ein konstanter Anteil an Kanten terminiert, zeigen wir nun noch, dass in jeder Runde ein konstanter Anteil der verbliebenen Kanten gut ist.

▶ **Lemma 6.** *In einer Runde gibt es mindestens $\frac{m}{2}$ (m ist hier die Anzahl an Kanten im Restgraphen der aktuellen Runde) gute Kanten.*

Beweis. Dieser Beweis richtet sich nach [10]. Wir werden zeigen, dass es höchstens $\frac{m}{2}$ nicht-gute Kanten gibt. Zuerst richten wir alle Kanten von Knoten mit niedrigem zu Knoten mit hohem Grad aus. Nun gilt, dass der Ausgangsgrad von nicht-guten Knoten mindestens doppelt so hoch wie der Eingangsgrad ist. Ansonsten wäre der Knoten gut, denn mindestens $\frac{d_v}{3}$ Nachbarn hätten kleineren Grad als v und $\sum_{u \in N(v)} \frac{1}{2d_u} \geq \frac{d_v}{3} \cdot \frac{1}{2d_v} = \frac{1}{6}$. Daher können wir jeder nicht-guten eingehenden Kante injektiv zwei ausgehende Kanten zuordnen. In Abbildung 6 kann man beispielhaft eine solche Zuordnung sehen. Da die Zuordnung injektiv ist, gibt es höchstens $\frac{m}{2}$ schlechte Kanten. ◀

Abbildung 6 Injektive Zuordnung von zwei eingehenden, schlechten Kanten zu ausgehenden Kanten.



Wir haben gezeigt, dass pro Runde erwartet ein konstanter Anteil der Kanten wegfällt. Damit haben wir das folgende Theorem gezeigt:

▶ **Theorem 7.** *Die Laufzeit des Algorithmus von Luby ist erwartet $O(\log n)$.*

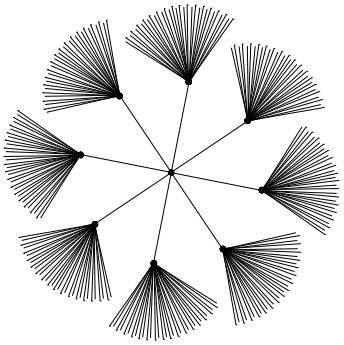
7.6 Algorithmus von Ghaffari (2015) [4]

Dieser Abschnitt basiert auf Mohsen Ghaffari's *An Improved Distributed Algorithm for Maximal Independent Set* [4]. Alle algorithmischen Techniken und theoretischen Resultate kommen von dieser Quelle, wenn keine andere Quellenangabe vorliegt.

7.6.1 Idee

Bei Luby's Algorithmus ist die Wahrscheinlichkeit, dass ein Knoten in einer Runde versucht dem MIS beizutreten fix gewählt als $1/(2d_v)$. Bei bestimmten Fällen klappt das nicht immer so gut (z.B. Fächer-Struktur, siehe Abbildung 7). Daher wird bei Ghaffari stattdessen die Wahrscheinlichkeit dynamisch angepasst, je nachdem wie die Nachbarschaft des Knotens aussieht. Dabei wird aktiv versucht zu erreichen, dass jeder Knoten entweder eine hohe Wahrscheinlichkeit hat grün zu werden oder eine hohe Wahrscheinlichkeit grau zu werden (das waren die guten Knoten bei Luby's Algorithmus). In der Grundstruktur ist der Algorithmus ansonsten gleich wie bei Luby.

Abbildung 7 Bei dieser Fächerstruktur hat der mittlere Knoten einen hohen Grad. Daher hat er bei Luby eine sehr kleine Wahrscheinlichkeit blau und dann grün zu werden. Sein Nachbarn haben einen noch größeren Grad und damit eine noch kleinere Wahrscheinlichkeit. Dies könnte man noch weiter fortsetzen. In diesem Fall wäre es besser, wenn der Knoten in der Mitte aktiv versucht dem MIS beizutreten, da er keine Konkurrenz von seinen Nachbarn hat



7.7 Definitionen

▶ **Definition 8** (Beitrittswahrscheinlichkeit)**.** In jeder Runde t hat jeder Knoten v eine *Beitrittswahrscheinlichkeit* $p_t(v)$ mit welcher er dem MIS beitreten will. Zu Beginn ist $p_0(v) = \frac{1}{2}$.

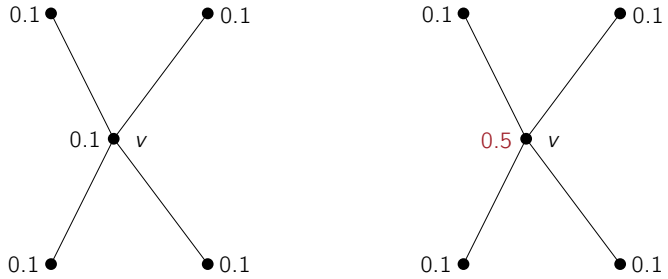
Die Beitrittswahrscheinlichkeit eines Knotens v entspricht bei Luby's Algorithmus also der fixen Wahrscheinlichkeit von $\frac{1}{2d_v}$, mit der ein Knoten versucht in einer Runde dem MIS beizutreten. Zu Beginn wird die Beitrittswahrscheinlichkeit initialisiert zu $\frac{1}{2}$ für alle Knoten. Dies kann durchaus schlechter sein als die Wahrscheinlichkeit von $\frac{1}{2d_v}$ bei Luby. Allerdings wird die Beitrittswahrscheinlichkeit im Laufe des Algorithmus dynamisch angepasst und dadurch haben die einzelnen Knoten gute Chancen zu terminieren.

▶ **Definition 9** (Effektiver Grad)**.** Die Summe der Beitrittswahrscheinlichkeiten der Nachbarn eines Knoten v ist sein *effektiver Grad* $d_t(v) = \sum_{u \in N(v)} p_t(u)$. Wir sagen, dass ein Knoten v einen *niedrigen* effektiven Grad hat, wenn $d_t(v) < 2$ ist.

▶ **Definition 10** (Goldene Runde Typ 1)**.** Eine *Goldene Runde Typ 1* liegt vor für einen Knoten v, wenn er dem MIS beitreten will und wenig Konkurrenz hat. Dies ist der Fall wenn $p_t(v)$ hoch ist und $d_t(v)$ niedrig. Formal (für die Analyse) fordern wir, dass $p_t(v) = \frac{1}{2}$ und $d_t(v) < 2$ ist.

Um zu erreichen, dass ein Knoten oft in einer Goldenen Runde Typ 1 ist, erhöhen wir daher $p_t(v)$ wenn $d_t(v)$ niedrig ist. Ein Beispiel hierzu findet sich in Abbildung 8.

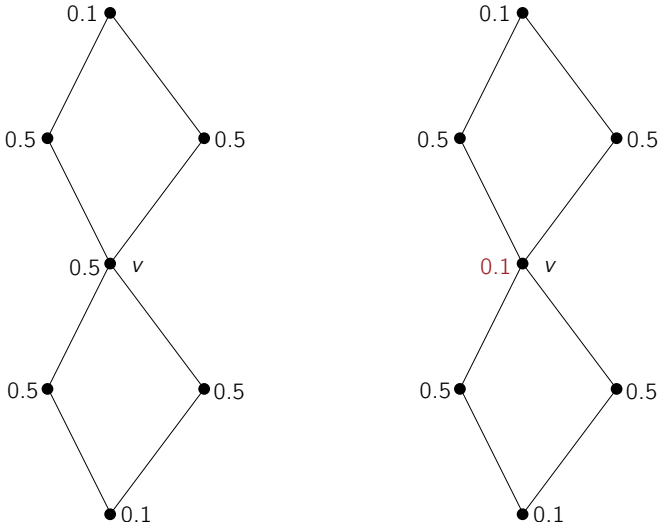
Abbildung 8 Erhöhe $p_t(v)$ (Zahl neben Knoten) um zu einer Goldenen Runde Typ 1 zu kommen (von links nach rechts).



▶ **Definition 11** (Goldene Runde Typ 2)**.** Eine *Goldene Runde Typ 2* liegt vor für einen Knoten v, wenn v viel Konkurrenz hat und seine Nachbarn wenig Konkurrenz haben. Dies ist der Fall, wenn $d_t(v)$ hoch ist und $d_t(u)$ niedrig ist für viele Nachbarn von v. Formal (für die Analyse) fordern wir, dass $d_t(v) > 1$ ist und mindestens $\frac{d_t(v)}{10}$ davon von Nachbarn mit niedrigem effektivem Grad kommt.

Um zu erreichen, dass ein Knoten oft in einer Goldenen Runde Typ 2 ist, verkleinern wir daher $p_t(v)$ (und damit auch $d_t(v)$ für alle Nachbarn u von v), wenn $d_t(v)$ hoch ist. Wenn ein Knoten in einer Goldenen Runde Typ 2 ist, hat er gute Chancen grau zu werden (wie die guten Knoten bei Luby). Ein Beispiel hierzu findet sich in Abbildung 9.

Abbildung 9 Verkleinere $p_t(v)$ (Zahl neben Knoten) um zu einer Goldenen Runde Typ 2 zu kommen (von links nach rechts).



7.7.1 Algorithmus

Der Algorithmus ist ähnlich wie Luby's. Anders ist, dass die Beitrittswahrscheinlichkeiten jede Runde angepasst werden, um zu erreichen, dass der Knoten oft in einer Goldenen Runde Typ 1 oder 2 ist. Außerdem gibt es nicht mehr die Regel, dass der Knoten mit höherem Grad gewinnt, wenn zwei benachbarte Knoten in der gleichen Runde dem MIS beitreten wollen, sondern es ist dann einfach keiner der Knoten erfolgreich. Die Regel gibt es nicht mehr, da sie für die Analyse hier nicht notwendig ist. In der Praxis würde es natürlich trotzdem Sinn ergeben

dies gleich zu handhaben wie bei Luby. Der Pseudocode ist dargestellt in Algorithmus 3.

Output : Entscheidung, ob $v \in$ MIS
$p_0(v) := \frac{1}{2};\ t := 0;$
while *Nicht entschieden* **do**
  $x_v = \begin{cases} \text{blau} & \text{, mit Wahrscheinlichkeit } p_t(v) \\ \text{schwarz} & \text{, sonst} \end{cases};$
  Sende x_v zu $N(v)$ und empfange x_u von allen $u \in N(v)$;
  if $x_v = blau \wedge x_u = schwarz\ \forall u \in N(v)$ **then**
    Trete MIS bei;
    $x_v =$ grün;
    Informiere Nachbarn;
    Terminiere;
  end
  if *Nachbar tritt MIS bei* **then**
    Trete MIS nicht bei;
    $x_v =$ grau;
    Terminiere;
  end
  $p_{t+1} = \begin{cases} \min\{2p_t(v), 1/2\} & \text{für } d_t(v) < 2 \\ p_t/2 & \text{für } d_t(v) \geq 2 \end{cases};$
  $t \leftarrow t + 1;$
end

Algorithmus 3 : MIS nach Ghaffari, aus Sicht des Knotens $v \in V$

7.7.2 Laufzeitanalyse

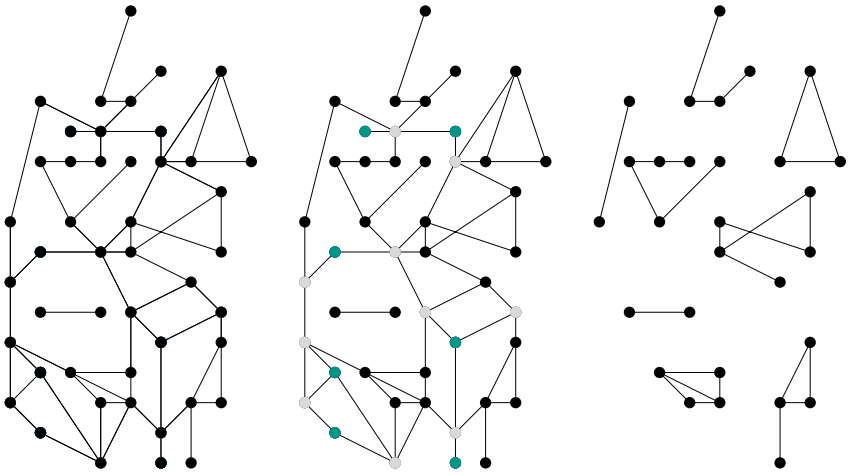
Die Laufzeitanalyse läuft etwas anders ab als bei Luby. *Ghaffari* [4] benutzt Techniken von *Barenboim et al.* [2]: Um die Analyse zu vereinfachen, wird der Algorithmus nur solange eingesetzt bis der Graph in viele hinreichend kleine verbundene Komponenten zerfallen ist. Danach wird ein deterministischer Algorithmus eingesetzt um diese kleinen Komponenten noch zu erledigen. In Abbildung 10 wird diese Vorgehensweise veranschaulicht. Deterministische MIS Algorithmen sind vergleichsweise langsam. Allerdings ist das nicht allzu schlimm, da der deterministische Algorithmus nur auf den gesplitterten, kleinen Komponenten eingesetzt wird.

Die Analyse ist lang und wir werden aus Platzgründen hier nicht alles genau wiedergeben können. Insbesondere ist die Analyse von Ghaffari unterteilt in eine lokale und globale Anaylse. Die Zeit, nach der ein bestimmter Knoten terminiert, wird *Lokale Laufzeit* genannt. Im Gegensatz dazu steht die *Globale Laufzeit*, die Zeit in der alle Knoten terminieren. Bei Luby hatten wir nur die Globale Laufzeit betrachtet. Hier bei Ghaffari betrachten wir nun auch die lokale Laufzeit. Diese lokale Laufzeit wird nur vom Knotengrad und dem Parameter ε abhängen, aber nicht von der Größe des Graphen. Selbst wenn der Graph unendlich groß ist, garantiert uns diese Laufzeit, dass ein bestimmter Knoten schnell terminiert.

Im Folgenden zeigen wir zuerst kurz, wie man von der lokalen Laufzeit auf die globale Laufzeit des Algorithmus schließen kann. Danach gehen wir genauer auf die Analyse der lokalen Laufzeit ein. Weitere Details können in der Arbeit von Ghaffari [4] nachgelesen werden.

Zuerst definieren wir, was gesplittert bedeutet:

Abbildung 10 Auf dem großen Graph (links) wird eine Weile der Algorithmus von Ghaffari ausgeführt um einen Teil-MIS zu finden (mitte). Die grünen Knoten sind schon sicher Teil des MIS und die grauen nicht. Die restlichen Knoten sind noch unentschieden. Der Restgraph ist gesplittert (rechts) und wird durch einen deterministischen Algorithmus bearbeitet.



▶ **Definition 12** (Gesplittert). Ein Graph ist *gesplittert*, wenn folgende zwei Bedingungen erfüllt sind:

- Alle verbundenen Komponenten des Graphen haben Größe höchstens $O(\log_\Delta(n) \cdot \Delta^4)$.
- Für alle Teilmengen $S \subseteq V'$ (wobei V' die verbliebenen, noch nicht terminierten Knoten des Graphen sind) mit $|S| \geq \log_\Delta n$ gilt, dass es Knoten gibt, die Abstand kleiner als 5 oder Abstand größer als 9 haben.

Um zu zeigen, dass ein Graph schon nach kurzer Zeit gesplittert ist, benötigen wir die lokale Laufzeit des Algorithmus von Ghaffari:

▶ **Theorem 13** (Lokale Laufzeit). *Ein Knoten v ist nach Zeit $O(\log d_v + \log \frac{1}{\varepsilon})$ mit Wahrscheinlichkeit mindestens $1 - \varepsilon$ terminiert. Dies gilt auch, wenn Knoten außerhalb der 2-Nachbarschaft (Knoten die über höchstens 2 Kanten erreicht werden können) sich störend (englisch: adversarial) verhalten.*

Der Beweis dieses Theorems folgt weiter unten. Hier gehen wir zuerst kurz auf die Analyse der globalen Laufzeit mittels der lokalen Laufzeit aus Theorem 13 ein. Zuerst zeigen wir, dass ein Graph schon nach wenigen Runden des Algorithmus gesplittert ist.

▶ **Lemma 14.** *Nach $O(c \log \Delta)$ Runden ist der Graph mit Wahrscheinlichkeit mindestens $1 - \frac{1}{n^c}$ gesplittert für eine groß genuge Konstante c.*

Beweis-Skizze. Aus der lokalen Laufzeit aus Theorem 13 folgt, dass die Wahrscheinlichkeit, dass ein bestimmter Knoten nach $\Theta(c \log \Delta)$ Runden noch nicht terminiert ist, höchstens Δ^{c-1} ist. Für eine Teilmenge S der Knotenmenge, in der die Knoten Abstand mindestens fünf haben, gilt dann weiterhin, dass die Wahrscheinlichkeit, dass keiner der Knoten terminiert ist, höchstens $\Delta^{(c-1)|S|}$ ist. Die Forderung, dass der Abstand zwischen Knoten von S mindestens fünf ist, ist erforderlich, da der Beweis von Theorem 13 die 2-Nachbarschaft (alle Knoten die über höchstens zwei Kanten erreicht werden können) des lokal betrachteten Knoten benutzt. Knoten mit Abstand mindestens fünf sind daher bei der Analyse der Wahrscheinlichkeit unabhängig voneinander. Durch Abzählen der Möglichkeiten, bei denen der Graph nicht

gesplittert ist, kann schlussendlich die Wahrscheinlichkeit, dass der Graph nach $O(c \log n)$ Runden noch nicht gesplittert ist, abgeschätzt werden. Für den genauen Beweis dieses Lemmas verweisen wir auf die Originalarbeit von *Ghaffari* [4]. ◀

▶ **Lemma 15.** *Es gibt einen deterministischen Algorithmus, der auf einem gesplitterten Graphen in Zeit* $2^{O(\sqrt{\log \log n})}$ *ein MIS findet.*

Für den Beweis dieses Lemmas verweisen wir auf die Originalarbeit von *Ghaffari* [4], die Techniken von *Barenboim et al.* [2] benutzt. Mit Lemma 14 und Lemma 15 lässt sich das folgende Theorem beweisen:

▶ **Theorem 16.** *Mit hoher Wahrscheinlichkeit (Wahrscheinlichkeit mindestens* $1 - \frac{1}{n}$*) wird in Zeit* $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$ *ein MIS gefunden.*

Beweis. Die Aussage folgt direkt aus Lemma 14 und Lemma 15. ◀

In Folgenden wollen wir nun noch etwas genauer auf den Teil der Analyse eingehen der sich mit der Lokalen Laufzeit und dem Beweis von Theorem 13 befasst. Dazu werden wir zuerst zeigen, dass sich ein Knoten v sehr oft in einer der zwei Goldenen Runden befindet. Dann zeigen wir, dass die Wahrscheinlichkeit in einer Goldenen Runde beliebigen Typs zu terminieren größer gleich einer Konstanten ist. Schließlich schätzen wir damit die Wahrscheinlichkeit ab, dass v nach $O(\log d_v + \log 1/\varepsilon)$ Runden noch nicht terminiert ist.

▶ **Lemma 17.** *Sei* $\varepsilon > 0$ *beliebig gewählt und* $v \in V$*. Dann gilt:*
Nach $O(\log d_v + \log 1/\varepsilon)$ *Runden ist* v *terminiert oder ist in mindestens* $200(\log d_v + \log 1/\varepsilon)$ *goldenen Runden von entweder Typ 1 oder Typ 2 gewesen. Hierbei ist* d_v *der Grad von* v *zu Beginn des Algorithmus.*

Beweis. Dieser Beweis richtet sich eng nach [4].
Wir wollen zeigen, dass nach $2300(\log d_v + \log 1/\varepsilon)$ Runden die Aussagen des Lemmas erfüllt sind (dann stimmt die Aussage auch im O-Kalkül). Dazu nehmen wir an, dass v nicht terminiert ist und in weniger als $200(\log d_v + \log 1/\varepsilon)$ Goldenen Runden Typ 1 war. Wir müssen nun zeigen, dass v dann in mindestens $200(\log d_v + \log 1/\varepsilon)$ Goldenen Runden Typ 2 war.

Sei h die Anzahl an Runden mit $d_t(v) \geq 2$. In diesen Runden wird $p_{t+1}(v) = \frac{p_t}{2}$ gesetzt. $p_t(v)$ kann höchstens so oft verdoppelt wie halbiert werden. Weiterhin kann $p_t(v)$ nur dann gleich bleiben, wenn $p(v) = \frac{1}{2}$. Daher muss $p_t(v) = \frac{1}{2}$ in mindestens $2300(\log d_v + \log 1/\varepsilon) - 2h$ Runden gelten. Diese Runden sind Goldene Runden von Typ 1, falls der effektive Grad des Knotens niedrig ist. Der effektive Grad ist in allen außer den h Runden mit $d_t(v) \geq 2$ niedrig. Daher gibt es mindestens $2300(\log d_v + \log 1/\varepsilon) - 3h$ Goldene Runden Typ 1. Damit die Forderung des Lemmas noch nicht erfüllt ist muss gelten:

$$2300(\log d_v + \log 1/\varepsilon) - 3h < 200(\log d_v + \log 1/\varepsilon)$$

Umstellen nach h ergibt: $h > 700(\log d_v + \log 1/\varepsilon)$. Runden mit $d_t(v) \geq 2$ sind potentielle Goldene Runden von Typ 2. Wir wollen nun zeigen, dass es mindestens $200(\log d_v + \log 1/\varepsilon)$ viele solche Runden gibt.

Falls $d_t(v) > 1$ ist und wir keine Goldene Runde von Typ 2 haben, dann hat v weniger als $\frac{d_t(v)}{10}$ Nachbarn mit niedrigem effektivem Grad. Diese Nachbarn können Ihre Beitrittswahrscheinlichkeit eventuell verdoppeln, alle anderen Nachbarn halbieren sie. Daher gilt dann:

$$d_{t+1}(v) \leq 2 \cdot \frac{d_t(v)}{10} + \frac{1}{2} \cdot \frac{9 \cdot d_t(v)}{10} = \frac{13}{20} d_t(v) < \frac{2}{3} d_t(v)$$

In zwei Runden mit $d_t(v) > 1$, die keine Goldenen Runden Typ 2 sind, sinkt der effektive Grad also auf weniger als $\frac{4}{9} < \frac{1}{2}$ des Ursprungswertes. Für zwei Runden mit $d_t(v) > 1$, die keine Goldenen Runden von Typ 2 sind, brauchen wir also mindestens eine Goldene Runde von Typ 2 um den effektiven Grad des Knoten zu erhalten.

Sei nun g_2 die Anzahl an Goldenen Runden von Typ 2. Dann gibt es mindestens $h - 3g_2$ Runden mit $d_t(v) \geq 2$, bei denen der effektive Grad mindestens um den Faktor $\frac{2}{3}$ sinkt ($2g_2$ Runden, die nicht Goldene Runden von Typ 2 sind, können durch die g_2 Goldene Runden von Typ 2 eventuell ausgeglichen werden, aber mehr nicht). Es kann maximal $\log_{(3/2)} d_v$ solche Runden geben, sonst würde der effektive Grad durch diese Runden unter 2 fallen. Also gilt $h - 3g_2 \leq \log_{(3/2)} d_v$. Daraus folgt:

$$3g_2 \geq h - \log_{(3/2)} d_v > 700(\log d_v + \log 1/\varepsilon) - \log_{(3/2)} d_v$$

Es gilt $\log_{(3/2)} d_v = \frac{\log d_v}{\log 3/2} \leq 100 \log d_v$. Damit ergibt sich:

$$3g_2 > 700(\log d_v + \log 1/\varepsilon) - 100 \log d_v = 600(\log d_v + \log 1/\varepsilon)$$

Schlussendlich gilt also $g_2 > 200(\log d_v + \log 1/\varepsilon)$. Dies ist genau das, was wir zeigen wollten. ◀

▶ **Lemma 18.** *In jeder Goldenen Runde von Typ 1 tritt v mit Wahrscheinlichkeit mindestens $\frac{1}{200}$ dem MIS bei. In jeder Goldenen Runde von Typ 2 tritt mit Wahrscheinlichkeit mindestens $\frac{1}{200}$ ein Nachbar von v dem MIS bei. Zusammenfassend terminiert v in einer Goldenen Runde beliebigen Typs also mit Wahrscheinlichkeit mindestens $\frac{1}{200}$. Dies gilt auch, wenn Knoten außerhalb der 2-Nachbarschaft sich störend verhalten.*

Beweis. Wir werden die zwei verschiedenen Typen separat analysieren.
Goldene Runde Typ 1:
Wenn v in einer Goldenen Runde Typ 1 ist, gilt per Definition $p_t(v) = \frac{1}{2}$. Daher färbt sich v mit Wahrscheinlichkeit $\frac{1}{2}$ blau. Weiterhin gilt $\sum_{u \in N(v)} p_t(u) = d_t(v) < 2$. Die Wahrscheinlichkeit, dass sich keiner der Nachbarn blau färbt ist $\prod_{u \in N(v)} (1 - p_t(u)) \geq \frac{1}{16}$ (siehe [4] für die Details der Abschätzung). Damit färbt sich v mit Wahrscheinlichkeit mindestens $\frac{1}{32} > \frac{1}{200}$ grün und terminiert.
Goldene Runde Typ 2: Per Definition ist $d_t(v) > 1$ und mindestens $\frac{d_t(v)}{10} > \frac{1}{10}$ davon kommt von Nachbarn mit niedrigem effektiven Grad. Man kann nun zeigen (Details siehe [4]), dass mit Wahrscheinlichkeit mindestens $\frac{2}{25}$ ein Nachbar von v mit niedrigem effektivem Grad sich blau färbt. Analog zur Goldenen Runde Typ 1 färbt sich mit Wahrscheinlichkeit mindestens $\frac{1}{16}$ keiner der Nachbarn dieses Knoten blau und der Knoten selber färbt sich grün. Daher kann v sich mit Wahrscheinlichkeit mindestens $\frac{2}{25} \cdot \frac{1}{16} = \frac{1}{200}$ grau färben und terminieren.
Beide Fälle hängen nur von der 2-Nachbarschaft des Knotens ab. Daher gilt das Resultat auch dann, wenn Knoten außerhalb der 2-Nachbarschaft sich störend verhalten. ◀

Nun können wir Lemma 17 und Lemma 18 kombinieren, um die Wahrscheinlichkeit, dass v nach $O(\log d_v + \log 1/\varepsilon)$ Runden terminiert ist, abzuschätzen und damit Theorem 13 über die Lokale Laufzeit beweisen:

Beweis von Theorem 13. Aus Lemma 17 wissen wir: Falls v noch nicht terminiert ist, hat es mindestens $200(\log d_v + \log \frac{1}{\varepsilon})$ goldene Runden von Typ 1 oder 2 durchlaufen. Nach Lemma 18 ist die Wahrscheinlichkeit in jeder dieser Runden nicht zu terminieren höchstens

$(1-\frac{1}{200})$. Die Wahrscheinlichkeit nach all diesen Runden noch nicht terminiert zu sein ist also:

$$(1-\frac{1}{200})^{200(\log d_v+\log 1/\varepsilon)} \leq e^{-(\log d_v+\log 1/\varepsilon)} = e^{-(\log d_v/\varepsilon)} \leq 2^{-\log d_v/\varepsilon} = \frac{\varepsilon}{d_v} \leq \varepsilon$$

Daher ist der Knoten mit Wahrscheinlichkeit höchstens ε noch nicht terminiert. ◀

Damit haben wir die lokale Laufzeit des Algorithmus bewiesen.

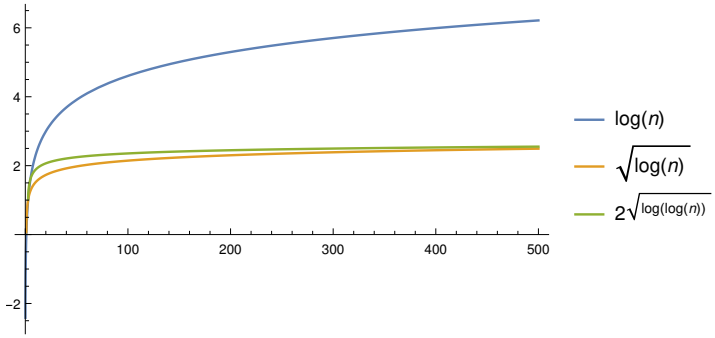
7.7.3 Genauerer Blick auf die Laufzeit

Die globale Laufzeit des Algorithmus ist $O(\log \Delta)+2^{O(\sqrt{\log\log n})}$. Es gibt eine untere Schranke von *Kuhn et al.* [5, 4] von $\Omega(\log \Delta + \sqrt{\log n})$. Der Term $\log \Delta$ taucht in beiden Laufzeiten auf und kann also nicht mehr verbessert werden. Nun wollen wir genauer anschauen wie sich $2^{O(\sqrt{\log\log n})}$ zu $\sqrt{\log n}$ verhält. Es gilt:

$$2^{\sqrt{\log\log n}} < 2^{\log\log n} = \log n$$

Wir können uns unter $2^{O(\sqrt{\log\log n})}$ also etwas vorstellen, dass (für große n) zwischen $\sqrt{\log n}$ und $\log n$ (die Laufzeit von Luby's Algorithmus) liegt. In Abbildung 11 sind die Funktionswerte der Funktionen $\log n$, $\sqrt{\log n}$ und $2^{\sqrt{\log\log n}}$ geplottet.

Abbildung 11 Laufzeit-Plot von $\log n$, $\sqrt{\log n}$ und $2^{\sqrt{\log\log n}}$



7.8 Fazit

Wir haben den Algorithmus von Ghaffari zur verteilten Bestimmung eines MIS im Detail kennengelernt. Weiterhin haben wir uns auch den Algorithmus von Luby angeschaut, auf dem der Algorithmus von Ghaffari basiert. Der Algorithmus von Ghaffari verbessert den Algorithmus von Luby. Dies wird dadurch erreicht, dass die Wahrscheinlichkeit, mit der ein Knoten versucht dem MIS beizutreten, nicht mehr direkt vom Grad des Knotens abhängt, sondern im Laufe der Runden dynamisch verändert wird. Dadurch kommt der Algorithmus nahe heran an die untere Laufzeitschranke von Kuhn et al. [5, 4]. Weiterhin hat er eine von der Netzwerkgröße unabhängige lokale Laufzeit, die auch dann noch gilt, wenn Knoten außerhalb der 2-Nachbarschaft sich störend verhalten.

Tabelle 1 Übersicht von Laufzeiten.

	Lokale Laufzeit	Globale Laufzeit
Laufzeit-Schranke	-	$\Omega(\log \Delta + \sqrt{\log n})$
Luby	-	$O(\log n)$
Ghaffari	$O(\log d_v + \log 1/\varepsilon)$	$O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$

Referenzen

1. Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of algorithms*, 7(4):567–583, 1986.
2. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In *Proceedings of the Fifty-Third Annual IEEE Symposium on Foundations of Computer Science*, pages 321–330. IEEE, 2012.
3. Fabian Fuchs, Roman Prutkin, and Dorothea Wagner. Algorithmen für Ad-hoc- und Sensornetze. http://i11www.iti.uni-karlsruhe.de/teaching/winter2015/sensornetze/index, 2015.
4. Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 270–277. ACM-SIAM, 2016.
5. Fabian Kuhn, Thomas Moscibroda, and Rogert Wattenhofer. What cannot be computed locally! In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, pages 300–309. ACM, 2004.
6. Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.
7. Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 581–592, New York, NY, USA, 1992. ACM.
8. David Peleg. Distributed computing. *SIAM Monographs on discrete mathematics and applications*, 5, 2000.
9. Johannes Schneider and Roger Wattenhofer. A log-star distributed maximal independent set algorithm for growth-bounded graphs. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 35–44, New York, NY, USA, 2008. ACM.
10. Eric Vigoda. Luby's Alg. for Maximal Independent Sets using Pairwise Independence. http://www.cc.gatech.edu/~vigoda/RandAlgs/MIS.pdf, 2006.

8 2-Vertex Connectivity in Directed Graphs

Niklas Baumstark

Abstract

This report serves as an exposition of the paper [3] by Georgiadis *et al.* Our goal is to present the core results of their publication while also providing the reader with an intuition behind the different decisions made and techniques used during the development of their algorithm.

We will end up with an algorithm to compute the vertex-resilient blocks of a directed graph G in linear time. Combined with previous results, this leads to the first linear-time algorithm to compute the 2-vertex-connected blocks of G, and to prepare a data structure to answer 2-vertex connectivity queries in constant time.

8.1 Introduction

We are given a directed graph $G = (V, E)$ with a vertex set V of size n and edge set $E \subseteq V \times V$ of size m. For two vertices u and v, we say there *exists a path from u to v in G* if there is a sequence of vertices $(u = x_0, x_1, x_2, \ldots, x_{k-1}, x_k = v)$ with $(x_i, x_{i+1}) \in E$ for all $0 \leq i \leq k$.

We call a pair u and v *strongly connected* if there exist paths from u to v as well as from v to u. The graph G is strongly connected if all pairs of vertices are. Strong connectivity is an equivalence relation and its equivalency classes are called *strongly connected components* or SCCs in short. In the following we assume that our input graph G is strongly connected. If it is not, we can process the SCCs separately, because the vertex relation that we are interested in computing implies strong connectivity.

We call u a *strong articulation point* of G if $G \setminus u$ (i.e. the graph with u removed) is not strongly connected. Similarly, we call an edge (u, v) with the same property a *strong bridge*. Figure 1 shows an example of a strongly connected graph with strong articulation points and bridges highlighted.

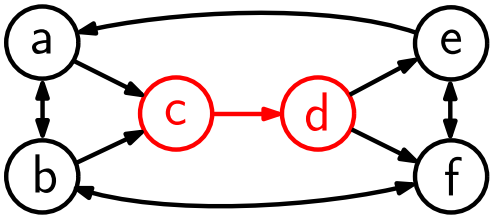


Figure 1 A strongly connected graph. Strong articulation points and bridges are highlighted in red.

Two vertices u and v are *2-vertex-connected* if there exist two paths from u to v that do not share a common vertex except for u and v and the same is true for two paths from v to u. We denote this relation by $u \leftrightarrow_{2v} v$. Figure 2 shows an example of a graph where u and v are 2-vertex-connected.

We can decompose G into *2-vertex-connected blocks*, which are inclusion-maximal subsets B such that each vertex pair $u, v \in B$ is 2-vertex-connected. This decomposition is not disjoint, there can be vertices that are part of multiple 2-vertex-connected blocks, as shown by Figure 3. Analogously, u and v are *2-edge-connected* if there exist such paths that do not share common edges. This yields the *2-edge-connected blocks* of G.

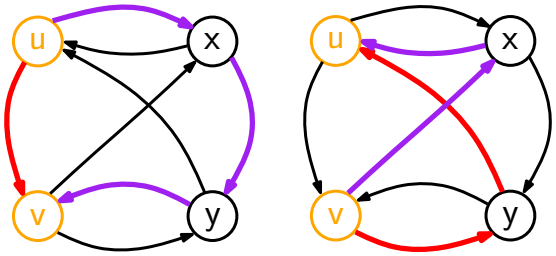


Figure 2 u and v are 2-vertex-connected as evidenced by the fact that the highlighted paths are vertex-disjoint.

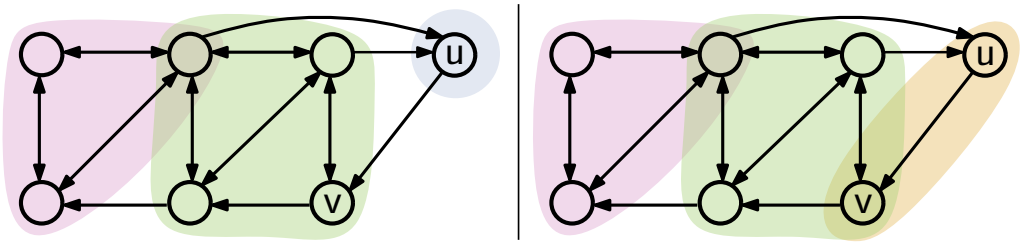


Figure 3 An example graph with 2-vertex-connected blocks (left) and vertex-resilient blocks (right) highlighted. Note that while u and v are vertex-resilient, they are not 2-vertex-connected because the connecting edge is a strong bridge (see Lemma 1).

Our final goal is to compute the 2-vertex-connected blocks of G in linear time. In order to achieve this, we will compute the *vertex-resilient blocks* first, which is a related, albeit not equivalent concept: Two vertices u and v are *vertex-resilient*, denoted as $u \leftrightarrow_{vr} v$, if for any other vertex $x \in V \setminus \{u, v\}$, u and v are strongly connected in the graph $G \setminus x$. A vertex-resilient block is an inclusion-maximal subset B of V, where each pair of vertices $u, v \in B$ is vertex-resilient. Figure 3 contrasts the 2-vertex-connected and vertex-resilient blocks of an example graph.

Obviously, $u \leftrightarrow_{2v} v$ implies $u \leftrightarrow_{vr} v$. The converse does not hold, but Menger's theorem [4] implies the following relation between 2-vertex connectivity and vertex resiliency:

▶ **Lemma 1.** *If $u \leftrightarrow_{vr} v$, then $u \leftrightarrow_{2v} v$ if and only if u and v are not connected by a strong bridge.*

Using this lemma, it can be shown that the 2-vertex-connected blocks are the intersections of vertex-resilient blocks and 2-edge-connected blocks. The latter can be computed in linear time using previous results from the same authors [2]. This means that if we can compute the vertex-resilient blocks of G and intersect them with the 2-edge-connected blocks in linear time, we have a linear-time algorithm for computing 2-vertex-connected blocks.

The rest of this paper thus deals with the development of a linear-time algorithm for vertex-resilient blocks. We first introduce a simple algorithm to compute vertex-resilient blocks that is obviously correct, but which has quadratic running time in the size of the graph. After that we introduce the concept of *domination* that allows us to analyse vertex resiliency and its properties better. Using the dominator relation we will eventually be able to formulate a second algorithm for vertex-resilient blocks which runs in linear time.

The data structures built during the execution of the second algorithm have only size $\mathcal{O}(n)$ and allow answering vertex-resiliency and 2-vertex connectivity queries in constant time. Furthermore, in the case where two vertices are *not* vertex-resilient/2-vertex-connected, we

can compute in $\mathcal{O}(1)$ a witness of this fact, i.e., a strong articulation point or strong bridge that separates them.

8.2 A Simple but Inefficient Algorithm for Vertex-Resilient Blocks

It will be helpful for the remainder of this paper to consider the inverse relation of vertex resiliency. If $u \not\leftrightarrow_{vr} v$, then there exists a vertex x such that u and v lie in different SCCs of $G \setminus x$. We call a vertex x with this property a *separator* of u and v. It is easy to show that x either lies on all paths from u to v or on all paths from v to u, which we will use in several of the proofs below. Also, in the following we are not really interested in vertex-resilient blocks of size one, so we will not consider these *trivial blocks* from here on when we refer to vertex-resilient blocks.

We now present a simple algorithm to compute vertex-resilient blocks, by using the observation from above: We start out with an initial, coarse set of blocks, initialized as $\mathcal{B} = \{V\}$. We proceed to compute the SCCs $S_1, S_2, \dots S_m$ of $G \setminus x$ for each possible $x \in V$. For each SCC S_i, we know that all vertex-resilient blocks are subsets of $S_i \cup \{x\}$, so we can replace each intermediate block in $\mathcal{B}$ by its intersections with the sets $S_i \cup \{x\}$. In the end, we output the final set of blocks $\mathcal{B}$. The pseudocode of this algorithm can be seen in Algorithm 1.

```
Input: A directed, strongly connected graph G = (V, E)
Output: The set B = {B_1, B_2, ..., B_k} of (non-trivial) vertex-resilient blocks of G
Initialize B ← {V};
for each vertex x ∈ V do
    Compute the SCCs S_1, S_2, ..., S_m of G \ x;
    for each B ∈ B do
        Remove B from B;
        for each S_i with |B ∩ (S_i ∪ {x})| ≥ 2 do
            Add B ∩ (S_i ∪ {x}) to B;
        end
    end
end
```

Algorithm 1: `SimpleVRB`, a straightforward algorithm to compute vertex-resilient blocks

Intuitively, for every vertex x, we use the SCCs of $G \setminus x$ to refine the set of vertex-resilient blocks we have so far. In the end we have separated all the pairs of vertices that are *not* vertex-resilient, and leave the others unseparated in their corresponding blocks.

▶ **Lemma 2.** *The algorithm `SimpleVRB` is correct*

Proof. Let $u, v \in V$ and $u \neq v$. If $u \leftrightarrow_{vr} v$, then for each vertex x of G, u and v will be part of the same SCC of $G \setminus x$ and thus they will never be separated in a block $B \in \mathcal{B}$ during the execution of `SimpleVRB`.

If, on the other hand, $u \not\leftrightarrow_{vr} v$, then there exists a vertex x that separates them in G and thus there is an iteration of `SimpleVRB` where they will be separated in all the blocks $B \in \mathcal{B}$ that contains both of them. ◀

When trying to analyse the time complexity of `SimpleVRB`, two questions remain: The first is technical: How can we compute the intersections between blocks and SCCs efficiently?

This will be addressed briefly in subsubsection 8.2.1. The second is that of the output size. How many vertex-resilient blocks can there be, and what is their combined size? In order to answer the latter, we introduce the concept of a block graph that represents the association of vertices with the vertex-resilient blocks containing them, and bound its size in subsubsection 8.2.2.

The results are that the combined size of all vertex-resilient blocks is in $\mathcal{O}(n+m)$ and that we can compute the intersections in linear time. Of course we can also compute SCCs in linear time, for example using Tarjan's algorithm [5]. Thus, `SimpleVRB` can be implemented with time complexity $\mathcal{O}(n \cdot (n+m))$.

In fact, we only need to consider strong articulation points x, because for all other vertices, $G \setminus x$ has only one SCC. However, there can be up to n strong articulation points, for example in the case where G is a directed cycle.

8.2.1 Efficient Set Intersections

As part of the algorithm `SimpleVRB`, we iterated over all current blocks B and replaced each of them with their intersections with the sets $S_i \cup \{x\}$. We will call this operation `refine`$(\mathcal{B}, \mathcal{S} = \{S_1, \dots S_{|\mathcal{S}|}\}, x)$. We will show that `refine` can be implemented with running time $\mathcal{O}(N+K)$ where N is the combined size of $\mathcal{B}$ and K is the combined size of $\mathcal{S}$.

First realize that the sets $S_i \cup \{x\}$ are disjoint except for the common element x. Let U be the subset of vertices $v \in V$ that appear in any of the sets S_i. The set U has cardinality K, so we can number its elements using the integers 1 to K and map between the elements and their integer representations in $\mathcal{O}(1)$ using a lookup table, which has to be set up once before any `refine` operations are performed. Using a second table of size K, we can store for each element $v \in U \setminus x$ the unique index i such that $v \in S_i$. Now in order to refine a block $B \in \mathcal{B}$, we group its members by their corresponding set index i, using a linear-time sort such as counting sort. The element x can be handled as a special case. The pseudocode description of the `refine` operation can be found in Algorithm 2.

Counting sort is linear in the length of the arrays and the range of their values, hence the overall time complexity of $\mathcal{O}(N+K)$.

8.2.2 The Block Graph and its Size

To analyse the size of `SimpleVRB`'s output, we define the *block graph* F of G. The vertex set of F is $V_F = \mathcal{B} \cup V$, where $\mathcal{B}$ is the set of vertex-resilient blocks of G. There is an edge $\{v, B\}$ in F for each vertex v in each block B. In other words, each vertex-resilient block B is connected to exactly all the vertices $v \in B$. Figure 4 shows a graph G with its block graph F drawn next to it.

From the example, we can already suspect that F is in fact a forest, and this is indeed the case as we will show here, after establishing a basic corollary first.

▶ **Corollary 3.** *For two vertex-resilient blocks A and B we have $|A \cap B| \leq 1$.*

Proof. Assume this is not the case. Than there exist vertex-resilient blocks A, B that share two vertices u and v. Because of inclusion maximality of A and B, there must be vertices $x \in A \setminus B$ and $y \in B \setminus A$ with $x \not\leftrightarrow_{vr} y$. The situation can be summarized by the picture in Figure 5.

Let w be any vertex in $V \setminus \{x, y\}$. We will show that there exist paths from x to y and back that do not contain w. This is a contradiction to the assumption that $x \not\leftrightarrow_{vr} y$.

Input: A set of blocks $\mathcal{B} = \{B_1, \dots B_{|\mathcal{B}|}\}$, a disjoint set of sets $\mathcal{S} = \{S_1, \dots, S_{|\mathcal{S}|}\}$ and an element x
Let $U = \{u_1, \dots, u_K\} = \bigcup_{i=1}^{|\mathcal{S}|} S_i$;
Set up the previously created global lookup L table to map between elements of U and their index in U;
Set up a table T of size K such that $u_j \in S_{A[j]}$ for each $j = 1, 2, \dots, K$;
for *each block* $B \in \mathcal{B}$ **do**
    Create array A of size $|B|$;
    for *each element* $v \in B$ **do**
        if $v \in U$ **then**
            Look up j with $v = u_j$ in L;
            Append the tuple $(T[j], v)$ to A;
        end
    end
    Sort A by the first tuple component, using counting sort;
    Group A by the first tuple component;
    Add x to each group;
    Replace B with the resulting groups;
end
Clear all the modified entries in the global lookup table L, so it can be reused;

Algorithm 2: The `refine` operation

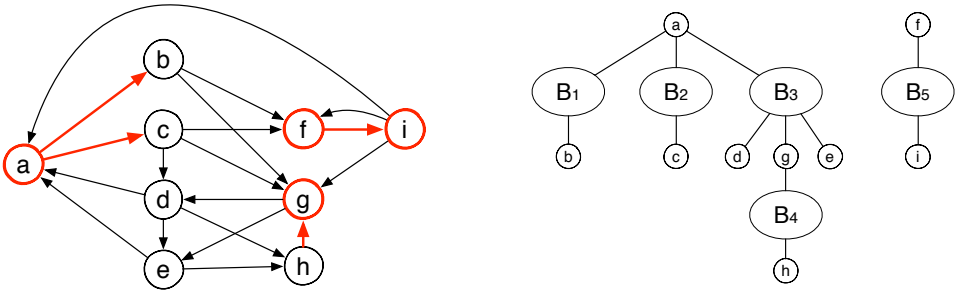


Figure 4 On the left: An example graph with strong articulation points and bridges marked in red. On the right: The associated block graph F. Source: [3]

Without loss of generality, we can assume $w \neq v$, otherwise swap the roles of u and v. Because $x, v \in A$, we have $x \leftrightarrow_{vr} u$, so there exists a path P from x to v that avoids w, otherwise w would be a separator of x and v.

Similarly, since $v, y \in B$, there exists a path Q from v to y that avoids w. The concatenation of P and Q is a path from x to y that avoids w. The same argument can be used to construct a path from y to x that avoids w, which concludes the proof. ◀

▶ **Lemma 4.** *F is acyclic (i.e. F is a forest).*

Proof. Assume F contains a cycle. Let C be a cycle of minimal length. Then C has size larger than 4 due to Corollary 3. Say $C = (v, B, u, B', w, \underbrace{\dots}_{\text{subpath } P})$ with $v, u \in B$ and $u, w \in B'$. Then we have $w \notin B$ due to Corollary 3, so there must be a $z \in B$ with $z \not\leftrightarrow_{vr} w$. For now we assume $z \neq v$, but the case $z = v$ has a very similar proof. Figure 6 depicts the situation visually.

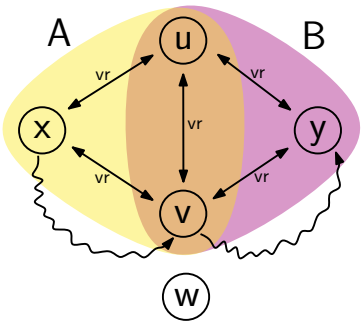


Figure 5 The situation in the proof for Corollary 3.

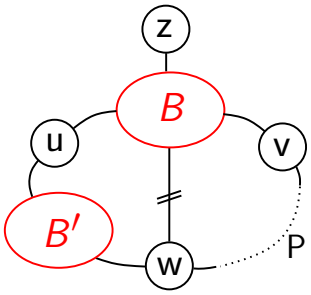


Figure 6 The situation in the proof for Lemma 4.

Let x be an arbitrary vertex from $V \setminus \{z, w\}$. We will show that there are paths from z to w and back that avoid x, a contradiction to $z \not\leftrightarrow_{vr} w$.

Case 1. $x \neq u$: Since $z, u \in B$ and $u, w \in B'$, we have $z \leftrightarrow_{vr} u \leftrightarrow_{vr} w$ and thus there exist paths between z and u as well as between u and w that avoid x. The concatenation of those paths are paths between z and w that avoid x.

Case 2. $x = u$: First, note that due to minimality of C the subpath P of C cannot contain u. By using the vertices from V on P, we can make the same argument from case 1 to find paths between z and w that avoid u. ◀

Now that we have established the structure of F (it is a set of trees), we can easily bound its size: Root each tree in F individually at a vertex from V, as was done in Figure 4. Since $|B| \geq 2$ for each (non-trivial) vertex-resilient block B, none of the blocks appears as a leaf in F. This means that we can assign a distinct vertex v to each block B, namely its first child in the rooted tree. Thus, we have $|\mathcal{B}| \leq n$ and hence $|V_F| = |\mathcal{B}| + n \leq 2n$. Furthermore, the edges in F correspond exactly to the membership of vertices from V in their containing blocks from $\mathcal{B}$. Thus, and because F is a forest, we get

$$\sum_{B \in \mathcal{B}} |B| = |E_F| < |E_V| \leq 2n$$

This yields the following lemma about the cumulative size of all vertex-resilient blocks:

▶ **Lemma 5.** *The number of vertex-resilient blocks of G and their combined size is in $\mathcal{O}(n)$.*

This result shows that we can afford to represent and store $\mathcal{B}$ explicitly and concludes our analysis of the algorithm `SimpleVRB`.

8.3 A Linear-Time Algorithm

So far for the development of `SimpleVRB` we used only very basic properties of the vertex resiliency relation. In order to design a more efficient algorithm to compute the vertex-resilient blocks, we will use the concept of a dominance relation from the domain of static data flow analysis. It captures the notion of some vertices being on all paths from one vertex to another, so intuitively this seems related to our concept of vertex resiliency:

First we select a start vertex $s \in V$. We say a vertex u *dominates* a vertex v if u lies on every path from s to v. In the example shown in Figure 7, the vertex a dominates b, c, d and e. However, b does not dominate d, because the path (s, a, c, d) is a path from s to d that avoids b.

It is easy to show that dominance is reflexive and transitive. Also, its transitive reduction is a tree, which we call the *dominator tree* $D(s)$ and which is also visualized in Figure 7. The *immediate dominator* of a vertex v is its parent in the dominator tree and is denoted as $d(v)$. The dominator tree can be computed in linear time [1] and it will turn out to be a powerful tool for our problem.

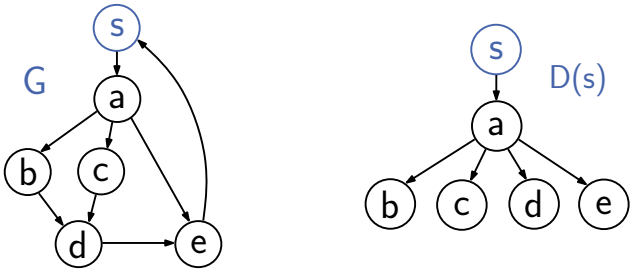


Figure 7 A graph G with designated start vertex s and its dominator tree $D(s)$.

Given the dominator tree $D(s)$, we can immediately make some statements about vertex resiliency: For example, in order for a vertex v to be vertex-resilient with s, it has to be a child of s in $D(s)$. Otherwise, $d(v)$ is a separator of s and v, because, by definition, it lies on every path from s to v.

Of course we cannot afford to compute dominator trees for each possible start vertex s, so this observation alone is not yet very helpful. However, we can generalize it into a useful condition that is necessary for arbitrary vertices u, v to be vertex-resilient:

▶ **Lemma 6.** *Let $u \leftrightarrow_{vr} v$. Then u and v are either adjacent in $D(s)$ (i.e., one is the child of the other) or they are siblings in $D(s)$ (i.e., they have a common parent).*

Proof. Let u and v be neither adjacent nor siblings. Then we have one of two different situations, for each of which we will construct a separator of u and v.

Case 1. *u is an ancestor of v, but $d(v) \neq u$ (or vice versa)*: Without loss of generality we can assume that u is an ancestor of v, otherwise swap the roles of u and v. The situation can be seen on the left side of Figure 8.

Let P be a path from s to u that does not contain $x := d(v)$. It exists because x is not a dominator of u. Let Q be a arbitrary path from u to v. Then $P \cdot Q$ is a path from s to v, so it contains x by definition of dominance. Because $x \notin P$, we have must have $x \in Q$. This holds for every possible Q, so x is a separator of u and v.

Case 2. *u and v are not ancestors of each other, but also not siblings*:

Let w be the least common ancestor of u and v and $x = d(v)$. Since u and v are not siblings, we have without loss of generality $w \neq x$ (otherwise swap the roles of u and v). The

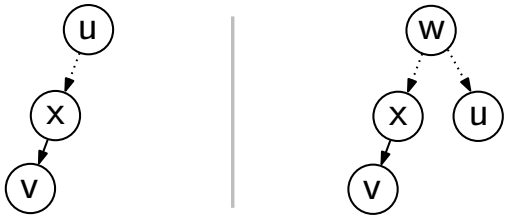


Figure 8 The situation in the proof for Lemma 6.

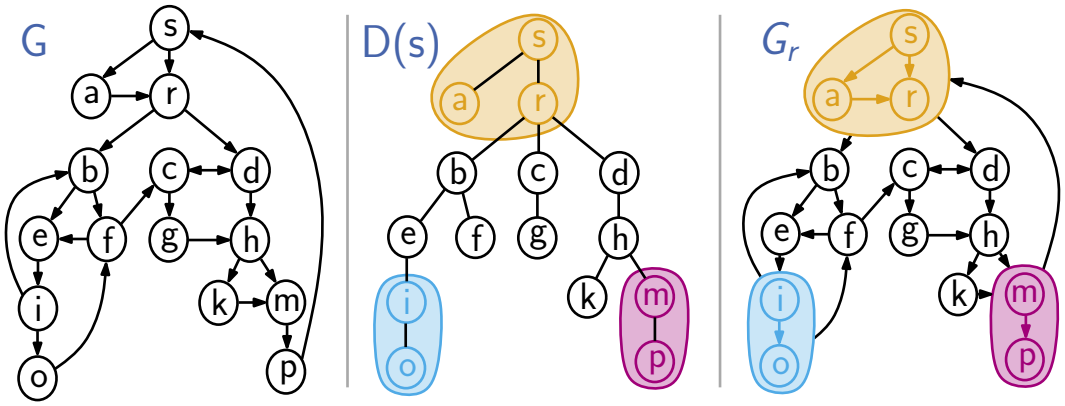


Figure 9 A graph G with designated start vertex s and its dominator tree $D(s)$. If we are only interested in analysing the vertex resiliency of vertices in the middle two layers, we can contract the vertices above the root r of the subtree and the ones in the lower layers and obtain the auxiliary graph G_r.

situation is depicted in detail on the right side of Figure 8.

By a very similar argument as above, there can be no path Q from u to v that avoids x. Otherwise the concatenation of any path P from s to u and Q would be a path from s to v that avoids x, a contradiction. So again, x is a separator of u and v.

In conclusion, for both cases we showed that $u \not\leftrightarrow_{vr} v$. ◀

Unfortunately, the converse of Lemma 6 does not hold, i.e., there can be vertices that are siblings or adjacent in $D(s)$ but that are not not vertex-resilient. Still, we have established that vertex-resilient vertices are "close together" in $D(s)$.

8.3.1 Auxiliary graphs

A consequence of Lemma 6 is that vertex-resilient pairs are within two layers of each other in the dominator tree $D(s)$. In the case where they are siblings, they are even on the same layer. This makes us wonder whether we can somehow analyse vertex resiliency of pairs that fulfill the necessary condition in a localized way, i.e., in a subgraph of G that represents only the interesting two layers of $D(s)$.

And indeed this is the case. Figure 9 shows a more complex example of a graph G, along with its dominator tree. If we want to find all the vertices that are vertex-resilient with b, then there are only a few candidates according to Lemma 6: e, f, c, d and r. For every other vertex x we already know that $b \not\leftrightarrow_{vr} x$. We pick f as an example and we now want to know whether $b \leftrightarrow_{vr} f$ or not.

We observe that the precise structure of the vertices above r in the dominator tree does not really interest us: There can be no paths from any of those vertices to b or f that do not contain r. Similarly, the vertices that are deeper than f in $D(s)$, in this case i, o, k, m and p, are also not very interesting for similar reasons, which we will state precisely later.

The key idea now is to contract the vertices in subtrees below the two layers we are interested in and the vertices that are not in the subtree rooted at r and obtain the *auxiliary graph* G_r. More precisely, if $T(x)$ is the subtree of $D(s)$ rooted at x, and $C(X)$ is the set of children of vertices $x \in X$ in $D(s)$, we perform the following transformations in G to obtain G_r:

- Contract the vertices $Top(r) := (V \setminus T(r)) \cup \{r\}$
- For each vertex $x \in Bot(r) := C(C(C(\{r\})))$, contract the vertices $T(x)$

The contraction of a vertex set X is defined as the process of removing the vertices X from G and instead inserting a new vertex c and edges (c, v) for each edge $(x, v) \in E$ with $x \in X$, as well as edges (v, c) for each edge $(v, x) \in E$ with $x \in X$. The rightmost graph shown in Figure 9 is the auxiliary graph G_r with contracted sets marked in color.

We call the vertices in $G_r \cap V$, i.e. the ones that were not contracted, *regular vertices* of G_r. We constructed G_r in the hope that it would preserve the vertex resiliency properties of regular vertices, and indeed it does, as Lemma 7 claims:

▶ **Lemma 7.** *For regular vertices u, v in G_r, we have $u \leftrightarrow_{vr} v$ in G if and only if $u \leftrightarrow_{vr} v$ in G_r.*

Proof. We will prove an equivalent statement: $u \not\leftrightarrow_{vr} v$ in G if and only if $u \not\leftrightarrow_{vr} v$ in G_r.

Let $u \not\leftrightarrow_{vr} v$ in G. We refer you to the case distinction in the proof of [3, Lemma 4.6] from the original paper for details on this case. On the other hand, let $u \not\leftrightarrow_{vr} v$ in G_r. Then there exists a separator x that (without loss of generality) lies on every path from u to v. We distinguish the three cases, depending on the type of x. Either x is a regular vertex, or $x = Top(r)$, or $x = T(w)$ for one of the $w \in Bot(r)$.

Case 1. *x is a regular vertex, i.e., $x \in V$*: Due to the nature of the contraction we used to obtain G_r, the SCCs of $G_r \setminus x$ are supersets of the SCCs of G. The reason for this is that for every path in G, there exists a corresponding path in G_r, after replacing every non-regular node by its contracted counterpart.

Because of this, if x is a separator of u and v in G_r, it is also a separator of u and v in G.

Case 2. $x = Top(r)$: We will show that if this is the case, then r lies on every path from u to v in G. Let $P = (u, x_1, \ldots, x_{k-1}, v)$ be any path from u to v in G. Let $P' = (u, x'_1, \ldots, x'_{k-1}, v)$ be the corresponding path in G_r, after replacing every non-regular vertex on P by its contracted version in G_r. The lefthand side of Figure 10 shows the situation.

Let $1 \leq i \leq k-1$ be the position of the last occurence of x in P', i.e. the largest i such that $x'_i = x$. We claim that $x_i = r$. Otherwise, there exists a path $(x_i, x_{i+1}, \ldots, x_{k-1}, v)$ in G from x_i to v that does not contain r, which violates the assumption that r is a dominator for v. So r lies on P, which was chosen arbitrarily. Therefore r is a separator of u and v in G.

Case 3. *$x = T(w)$ for some $w \in Bot(r)$*: Let P be a path from u to v in G. By a very similar argument as above, we can show that the first occurence of a vertex $y \in T(w)$ on P is w. So w separates u and v in G.

In all the cases, we have found a separator of u and v in G, so $u \not\leftrightarrow_{vr} v$. ◀

Figure 10 On the left: A path P in G that visits any vertex in $Top(r)$ contains r as the last vertex from $Top(r)$. On the right: A path P in G that visits any vertex in $T(w)$ for $w \in Bot(r)$ contains w as the first vertex from $T(w)$

.

8.3.2 Computing Vertex-Resilient Blocks of G_r

We introduced the concept of auxiliary graphs to allow us to examine vertex resiliency locally (inside an auxiliary graph) rather than globally: For each auxiliary graph, we can find its vertex-resilient blocks and use those to refine a preliminary set of blocks. The basic outline for an algorithm can be seen in Algorithm 3.

Input: A directed, strongly connected graph $G = (V, E)$
Output: The set $\mathcal{B} = \{B_1, B_2, \ldots, B_k\}$ of (non-trivial) vertex-resilient blocks of G
Initialize $\mathcal{B} \leftarrow \{V\}$;
Choose $s \in V$ arbitrarily;
Compute the dominator tree $D(s)$;
for *each vertex* $r \in V$ **do**
  Intersect each block $B \in \mathcal{B}$ with the set $\{r\} \cup C(\{r\})$;
  Compute the auxiliary graph G_r;
  TODO: Compute vertex-resilient blocks of G_r;
  Intersect each $B \in \mathcal{B}$ with the vertex-resilient blocks of G_r;
end

Algorithm 3: `FastVRB` basic outline

Apart from the so far unknown process of computing the vertex-resilient blocks of an the auxiliary graphs G_r, we can do all the operations in linear time. This is because of the following lemma:

▶ **Lemma 8.** *All the auxiliary graphs G_r combined have $\mathcal{O}(n)$ vertices and $\mathcal{O}(n+m)$ edges. All of them can be computed together in time $\mathcal{O}(n+m)$.*

Proof. See [3, Lemma 4.5]. Intuitively, for each r, we can restrict the number of edges from vertices in $Top(r)$ to vertices in $Bot(r)$ and vice versa, due to the structure of the dominance relation. The algorithm to compute the auxiliary graphs is then just a standard depth-first search, with a modification similar to that used by Tarjan's SCC algorithm [5] in order to find the edges from vertices in $Top(r)$ and $Bot(r)$ fast.

◀

What remains is to show how we can compute the vertex-resilient blocks of an auxiliary graph G_r. Consider the case of two vertices u and v that are adjacent or siblings of each other in $D(s)$, so they fulfill the necessary condition of Lemma 6. Still, assume they are not vertex-resilient, i.e. there exists a separator in G that separates them.

Select r such that either $d(v) = r$ or $d(u) = r$ (or both). Then both u and v are regular vertices in the auxiliary graph $H := G_r$. The auxiliary graph and the corresponding subtree of $D(s)$ gives us a tool to analyse the paths **from** r to u and v. We will, however, need a second tool to examine paths from u or v **to** r. And we can in fact reuse the existing framework to do that: We simply compute the dominator tree $D_H^R(r)$ of H^R, the reverse auxiliary graph, with starting point r. Paths in H^R starting at r correspond to paths in H ending in r, so this will turn out to be a useful construct. $D_H^R(r)$ intuitively captures the notion of vertices lying on every path from a vertex x in G_r to r. G_r and $D_H^R(r)$ together allow us to formulate the following lemma, which gives an explicit characterization of the separators of u and v in G_r.

▶ **Lemma 9.** *Let $u \not\leftrightarrow_{vr} v$. Choose r such that $d(v) = r$ or $d(u) = r$ or both (Lemma 6 implies this is always possible). Let $H = G_r$ and let $D_H^R(r)$ be the dominator tree of H^R with starting point r. Then either the parent of u or the parent of v in $D_H^R(r)$ separates u and v in H.*

Proof. Without loss of generality, there exists a separator x that lies on every path from u to v in G_r (otherwise swap the roles of u and v). Hence we know that $d(v) \neq u$, otherwise this would be impossible. Thus, $d(v) = r$. There are still two possible configurations for u: It could be either the child of r or the child of v.

Because x is on every path from u to v and there certainly is a path from r to v that avoids it, we have that x is already on every path from u to r. Thus it is a dominator of u in $D_H^R(r)$ The situation is depicted in Figure 11.

Figure 11 The situation in the proof of Lemma 9.

Let q be the immediate dominator of u in $D_H^R(r)$. We claim that q also separates u and v. Let P be an arbitrary path from u to v. Then it contains a subpath Q from u to x, because x is on every path from u to v. Because q is a descendant of x, but an ancestor of u in $D_H^R(r)$, every path from u to x contains q, including Q. So $q \in Q$ and thus $q \in P$.

Removing q thus leads to u and v being in different SCCs of $H \setminus q$. ◀

There is one more idea to introduce, after which we will end up with the complete algorithm. We now know what the separators are for non-vertex-resilient pairs of vertices in $H = G_r$. We can also afford to compute $D_H^R(r)$. However, we cannot afford to compute for each possible separator vertex q the strongly connected components of $H \setminus q$.

The idea now is the following: Lemma 6 holds just as well in $D_H^R(r)$ as it does in $D(s)$. So for a pair $u \not\leftrightarrow_{vr} v$, we know that they are siblings or parents of each other in $D_H^R(r)$. Lemma 9 established that the parent of one of the two separates them, let it be called q. We can now compute the *second-level auxiliary graph* H_q^R, which is the auxiliary graph of $D_H^R(r)$ representing the subtree rooted at q. Both u and v are regular vertices in H_q^R, and removing q separates them into different SCCs. This single SCC computation we can actually afford to do, because the combined size of all the H_q^R is only linear in the size of G_r.

This leads to the final formulation of the algorithm `FastVRB`, shown in Algorithm 4.

Input: A directed, strongly connected graph $G = (V, E)$
Output: The set $\mathcal{B} = \{B_1, B_2, \ldots, B_k\}$ of (non-trivial) vertex-resilient blocks of G
Initialize $\mathcal{B} \leftarrow \{V\}$;
Choose $s \in V$ arbitrarily;
Compute the dominator tree $D(s)$;
for *each vertex* $r \in V$ **do**
  Intersect each block $B \in \mathcal{B}$ with the set $\{r\} \cup C(\{r\})$;
  Compute the auxiliary graph $H = G_r$;
  Compute the dominator tree of H^R with start vertex r, $D_H^R(r)$;
  Initialize $\mathcal{B}_r \leftarrow \{G_r\}$, the set of vertex-resilient blocks of G_r;
  for *each vertex* $q \in H$ **do**
    Intersect each block $B \in \mathcal{B}_r$ with the set $\{q\} \cup C_H^R(\{q\})$;
    Compute the second-level auxiliary graph H_q^R of $D_H^R(r)$;
    Compute the strongly connected components $S = \{S_1, S_2, \ldots, S_k\}$ of $H_q^R \setminus q$;
    Intersect each $B \in \mathcal{B}_q$ with the sets $S_i \cup \{q\}$
  end
  Intersect each $B \in \mathcal{B}$ with the sets in $\mathcal{B}_r$;
end

Algorithm 4: The complete algorithm `FastVRB`

The combined size of all first- and second-level auxiliary graphs is in $\mathcal{O}(n+m)$, so `FastVRB` can be implemented in linear time. We have already established correctness by our analysis above that led us to the final algorithm, but here is a short summary: `FastVRB` separates vertex pairs that are either not siblings or adjacent in $D(s)$ or any $D_H^R(r)$, which is correct due to Lemma 6. It also separates pairs that are regular vertices in a H_q^R, but are not in the same SCC of $H_q^R \setminus q$. This is correct due to Lemma 7, because they being in different SCCs means that q separates them.

If on the other hand a vertex pair u, v is not vertex-resilient, and they do not violate Lemma 6 in any of the dominator trees, then there exist vertices $r \in V$ and $q \in G_r$ such that u and v are regular vertices in both the first-level auxiliary graph G_r and the second-level auxiliary graph H_q^R, and q is a separator of u and v in H_q^R, due to Lemma 9. Thus, the algorithm will correctly separate them because they lie in different SCCs of $H_q^R \setminus q$. This concludes our argument for the correctness of `FastVRB`.

We still have to show that all the set intersections used by `FastVRB` can be performed in linear time. For the cases where we intersect with the strongly connected components of a second-level auxiliary graph, we can use the `refine` operation from subsubsection 8.2.1. For the cases where we intersect with the tree subsets $\{q\} \cup C_H^R(\{q\})$, we can use the `split` operation that can also be implemented with running time linear in the size of the tree [3, Lemma 4.10]. Note that `split` is quite different from `refine` because the intersecting sets are not disjoint in this scenario. Overall we obtain our central theorem

▶ **Theorem 10.** *`FastVRB` runs in time $\mathcal{O}(n + m)$ and outputs exactly the non-trivial vertex-resilient blocks of G.*

8.4 Queries and 2-Vertex-Connected Blocks

At this point, we have an algorithm at our hands that computes vertex-resilient blocks efficiently. However, we were originally interested in computing the 2-vertex connectivity relation. There is a simple reduction that allows us to compute the latter when knowing the former. `FastVRB` can be modified to return not only the set of vertex-resilient blocks, but also the block graph F that represents membership of a vertex v in a vertex-resilient block B by an edge $\{v, B\}$.

Since F is a tree, it is easy to preprocess it in a way that enables answering vertex resiliency queries for two vertices u and v in constant time: Just root F arbitrarily. Then there exists a path of length two between u and v if and only if the parents of u and v in F are either the same, or if u is grandparent of v, or the other way around. The parent of a vertex in a tree can be checked in $\mathcal{O}(1)$, so this leads to constant query time.

If we also store the dominator tree $D(s)$ of G and the reverse dominator trees $D_H^R(r)$ of all auxiliary graphs G_r, we can even compute a separator of u and v in constant time. For the details of this algorithm please refer to [3, Section 4.4]. The combined size of these dominator trees is only $\mathcal{O}(n)$ due to Lemma 8 (and because we do not need to represent the edges of the auxiliary graphs).

Combined with the data structure from [2], we can also query the 2-vertex connectivity of two vertices u and v in constant time, by checking if the connecting edge is a strong bridge in case it exists. As a witness of non-connectivity, we report either a witness for non-vertex resiliency, or the strong bridge that connects them.

We can also compute the 2-vertex-connected blocks explicitly, by intersecting the vertex-resilient blocks computed by `FastVRB` with the 2-edge-connected blocks computed by [2], in linear time.

References

1 Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in Linear Time, 1999.

2 Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2-Edge Connectivity in Directed Graphs. In *SODA 2015*, pages 1988–2005, 2015.

3 Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2-Vertex Connectivity in Directed Graphs. In *ICALP 2015*, pages 605–616, 2015.

4 Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

5 Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

9 Berechnung des Greedy-Spanner bei linearem Speicherplatzverbrauch

Tobias Müller

Zusammenfassung

Diese Ausarbeitung beschäftigt sich mit der Konstruktion eins Greedy-Spanner-Algorithmus, welcher einen t-Spanner mit nur linearem Speicherplatzverbrauch berechnet. Der vorgestellte Algorithmus bleibt mit der Laufzeit $O(n^2 \log^2 n)$ nur um einen logarithmischen Faktor hinter dem derzeit besten bekannten theoretischen Algorithmus zurück und liefert gleichzeitig die für Greedy-Spanner üblichen asymtotisch optimalen Ergebnisse in Größe, Gewicht und maximalem Knotengrad des t-Spanner.

9.1 Einleitung

Sucht man für eine gegebene Menge von Punkten ein Verbindungssystem, sodass jeder Punkt von jedem anderen aus erreichbar ist, bieten sich trivial zwei extreme Möglichkeiten - die Vollvermaschung und die minimale Verspannung. Ein Beispiel hierfür könnten Sehenswürdigkeiten einer Stadt sein, welche über ein U-Bahn-Netz verbunden werden sollen. Beide Extrema hätten hier jedoch große Nachteile. Die Vollvermaschung wäre aufgrund der großen Anzahl an zu grabenden Tunnel zu aufwändig und nicht effizient nutzbar. Beim minimalen Ansatz kann es allerdings passieren, dass zwei Sehenswürdigkeiten, die räumlich recht nahe beieinander liegen, über eine relativ lange Fahrstrecke verbunden sind. Man ist hier also interessiert an einem Kompromiss aus geringer Anzahl an Verbindungen und kurzen Fahrtstrecken. Einen solchen Kompromiss bieten die sogenannten *Geometric Spanner* (engl: „geometrische Aufspanner“), auch *t-Spanner* genannt. Diese sind eine t-Approximation eines vollständigen Graphen. Dabei ist $t > 1$ ein Parameter, welcher die Länge des kürzesten Pfades zwischen zwei Punkten auf das t-fache ihrer Distanz im Raum nach oben beschränkt.

Zur Berechnung eines t-Spanner gibt es mehrere Vorgehen, welche verschiedene Garantien beispielsweise in der Laufzeit oder dem maximalen Knotengrad bieten. Die in dieser Arbeit betrachteten *Greedy-Spanner-Algorithmen* liefern in Bezug auf den maximalen Knotengrad, die Größe sowie das Kantengewicht asymptotisch optimale Werte. Aufgrund dieser Eigenschaften sind Greedy-Spanner von besonderem Interesse beispielsweise bei Netzwerkentwürfen, in welchen der geringe maximale Knotengrad Vorteile beim Routing erbringt.

Der schnellste derzeit bekannte Algorithmus benötigt $O(n^2 \log n)$ Zeit [BC+10]. Nachteil aller bisher implementierten Greedy-Spanner-Algorithmen ist jedoch, dass ihr Platzverbrauch quadratisch mit der Anzahl der zu verbindenden Knoten wächst. Dies hat zur Folge, dass Greedy-Spanner bereits für Instanzen mit mehr als 13000 Knoten kaum mehr verwendbar sind [AB+15]. Inhalt dieser Arbeit ist ein 2015 von Alewijnse et al. vorgestellter Greedy-Spanner-Algorithmus, dessen Platzverbrauch nur linear in der Eingabegröße ist, gleichzeitig aber nur geringe Einbußen in der Berechnungszeit mit sich bringt.

9.2 Aufbau der Arbeit

Zunächst werden in Kapitel 2 die grundlegenden Begriffe und Schreibweisen definiert. Kapitel 3 erläutert das grundlegende Prinzip der Greedy-Spanner-Algorithmen. Kapitel 4 beschreibt die Idee sowie die Umsetzung eines Algorithmus mit linearem Speicherplatzverbrauch. Anschließend werden für den in Kapitel 4 entwickelten Algorithmus im 5. Kapitel einige Optimierungen eingeführt. Abschließend werden im 6. Kapitel experimentelle Ergebnisse des entwickelten Algorithmus vorgestellt.

10 Grundlagen

In diesem Kapitel werden einige Definitionen sowie grundlegende Begriffe eingeführt, welche in der Ausarbeitung verwendet werden.

Metrischer Raum

Ein *metrischer Raum* ist eine Menge X, auf welcher eine Metrik definiert ist. Eine *Metrik* ist eine Abbildung $d : X \times X \to \mathbb{R}$, wenn $\forall x, y, z \in X$:

- $d(x, y) \leq 0 \wedge d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) = d(y, x)$
- $d(x, y) \geq d(x, z) + d(z, y)$ (Dreiecksungleichung)

In metrischen Räumen können also Abstände berechnet werden. Für den Abstand zwischen zwei Knoten $u, v \in X$ schreiben wir |uv|. Diesen Abstand nennen wir im Folgenden auch „direkte Distanz“

Beispiele für metrische Räume mit euklidischer Distanzfunktion sind die $\mathbb{R}^d$, wobei $d \in \mathbb{N}$ die Dimension ist. Diese Arbeit beschränkt sich in alle Beispielen aus Gründen der Übersichtlichkeit auf den $\mathbb{R}^2$. Alle Algorithmen funktionieren jedoch auch auf metrischen Räumen höherer Dimension.

t-Spanner

Ein *t-Spanner* für eine in einem metrischen Raum gegebene Punktmenge P ist ein ungerichteter, gewichteter Graph G(P, E) mit Knotenmenge P und Kantenmenge E, sodass die Länge des kürzesten Pfades zwischen zwei Knoten $u, v \in P$ maximal t mal länger ist, als ihre direkte Distanz. Es muss also gelten:

$$\delta_G(u, v) \leq t \cdot |uv|$$

Dabei bezeichnet $\delta_G(u, v)$ die Länge des kürzesten Pfades zwischen u und v in G. Das Kantengewicht entspricht dabei jeweils dem Abstand zwischen den beiden Endknoten. Einen solchen Pfad nennt man t-Pfad (engl.: *t-path*). Üblich ist $1 < t \leq 2$.
Für t-Spanner gibt es drei Qualitätskriterien:

1. Größe, also die Anzahl der Kanten
2. Summe der Kantengewichte
3. Maximaler Knotengrad

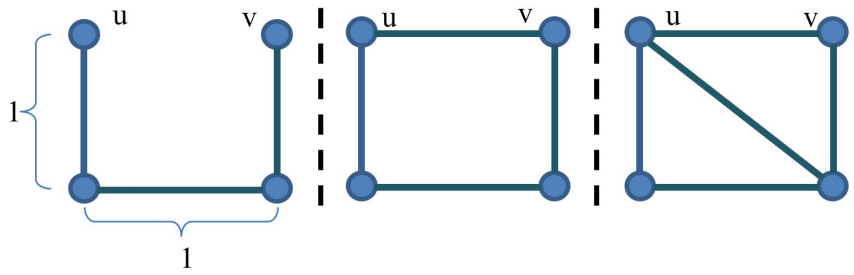


Abbildung 1 Drei Graphen für dieselbe Punktmenge. Links: kein gültiger 2-Spanner. Mitte: gültiger 2-Spanner mit optimalem maximalen Knotengrad, Kantenanzahl und Kantengewicht. Rechts: gültiger, nicht optimaler 2-Spanner.

Abbildung 1 zeigt drei verschiedene Graphen für dieselbe Punktmenge, für welche ein 2-Spanner gefunden werden soll. Hierbei ist der Graph links kein gültiger 2-Spanner, da der Pfad zwischen den Knoten u und v die Länge 3 hat, maximal jedoch die Länge 2 haben darf. Der Mittlere Graph ist ein gültiger 2-Spanner und optimal in den eben genannten Kriterien. Der Graph auf der rechten Seite ist ebenfalls ein gültiger 2-Spanner, er hat allerdings im Vergleich zum mittleren eine schlechtere Qualität, da mehr Kanten verwendet wurden und der maximale Knotengrad höher ist.

Umschließender Kreis

Der *umschließende Kreis* (hier engl.: *bounding box*) einer Punktmenge Y ist ein Kreis mit minimalem Durchmesser, sodass alle Punkte aus Y im inneren des Kreises liegen. Dabei bezeichnet *diam(Y)* den Durchmesser (von engl. *diamter*) des umschließenden Kreises von Y. Ein Beispiel wird in Abbildung 2 gegeben.

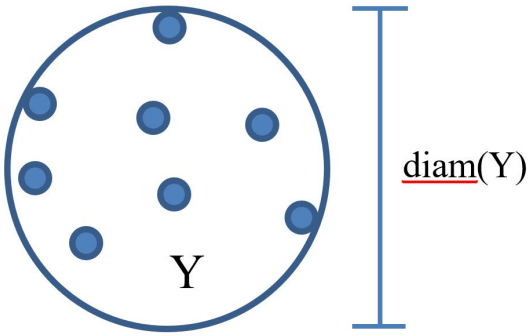


Abbildung 2 Umschließender Kreis einer Punktmenge. Dabei ist diam(Y) der Durchmesser.

Für zwei disjunkte Knotenmengen X, Y bezeichnet $min(X, Y)$ den minimalen Abstand zwischen den jeweiligen umschließenden Kreisen.

s-Wohlseparierheit

Zwei disjunkte Mengen von Knoten A, B heißen *s-wohlsepariert* (engl.: *s-well-separated*) für ein $s \in \mathbb{R}, s > 0$, wenn die umschließenden Kreise von A und B mindestens den Abstand $s \cdot max(diam(A), diam(B))$ haben. Der Abstand zwischen einem Knotenpaar aus $A \times B$

ist dabei stets mindestens so groß wie der Abstand der umschließenden Kreise A und B. Abbildung 3 verdeutlicht dies.

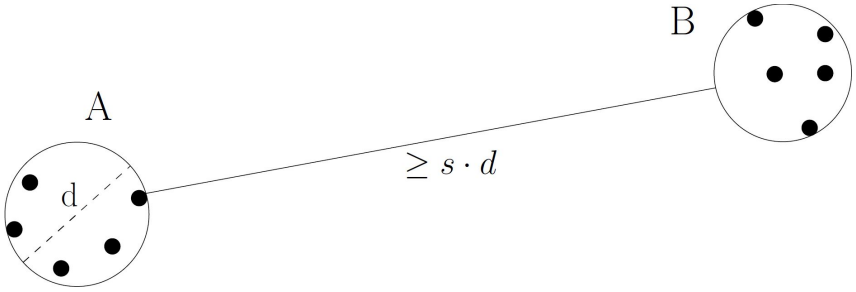


Abbildung 3 Zwei wohlseparierte Knotenmengen A und B.

Wohlseparierte Paarzerlegung

Für eine gegebene Punktmenge P und ein $s > 0$ ist eine w*ohlseparierte Paarzerlegung* (engl.: *well-separated pair decomposition, WSPD*) eine Menge von p Knotenmengenpaaren $A_i, B_i, 1 \leq i \leq p$, sodass gilt:

1. jedes Paar ist s-well-separated
2. für jedes Knotenpaar $u, v \in P$ gibt es genau ein i:
   - $u \in A_i \wedge v \in B_i$ oder
   - $u \in B_i \wedge v \in A_i$

Wenn im Folgenden von (Knoten-)Gruppen oder (Knoten-)Gruppenpaaren gesprochen wird, beziehen sich diese Begriffe immer auf die Mengen A und B der WSPD.

ClosestPair

Das *ClosestPair* (engl.: *nächstes Paar* (räumlich)) zweier s-wohlseparierter Knotenmengen A und B ist das Knotenpaar $(u, v,) \in A \times B$, welches die kürzeste direkte Distanz hat und für das im aktuellen Graph noch kein t-Pfad existiert.

11 Das Prinzip des Greedy-Spanner

In diesem Kapitel wird zunächst die grundlegende Arbeitsweise eines Greedy-Spanner-Algorithmus beschrieben. Anschließend wird die Korrektheit überprüft sowie Laufzeit und Speicherplatzverbrauch analysiert.

Grundprinzip Greedy-Spanner-Algorithmen

Der folgende in Abbildung 4 skizzierte Algorithmus, welcher von Keil vorgestellt wurde, beschreibt die Grundidee der Greedy-Spanner-Algorithmen [K88].

Alle Knotenpaare werden zunächst nach ihrer direkten Distanz sortiert. Anschließend wird in aufsteigender Reihenfolge für jedes Paar überprüft, ob zwischen dem aktuellen Paar bereits ein t-Pfad existiert. Falls ja, so wird keine direkte Kante zwischen diesen beiden Knoten benötigt und es kann unmittelbar das nächste Knotenpaar betrachtet werden. Falls nein, fügen wir dem Graphen die Kante zwischen dem aktuellen Knotenpaar hinzu.

Algorithm *GreedySpannerOriginal*(V, t)
1. $E \leftarrow \emptyset$
2. **for** every pair of distinct points (u, v) in ascending order of $|uv|$
3. **do if** $\delta_{(V,E)}(u, v) > t \cdot |uv|$
4. **then** $add(u, v)$ to E
5. **return** E

Abbildung 4 Grundlegender Algorithmus zur Berechnung eines Greedy-Spanner.

Greedy-Spanner-Algorithmen fügen also in jedem Iterationsschritt genau dann eine Kante zwischen den aktuell betrachteten beiden Knoten ein, wenn dies die geringste direkte Distanz haben und zwischen ihnen noch kein t-Pfad existiert.

Korrektheit

Dieses Verfahren ist offensichtlich korrekt. In einem ungültigen t-Spanner existiert mindestens ein Knotenpaar, welches nicht über einen t-Pfad verbunden ist. Da der Algorithmus im Verlauf jedoch jedes Knotenpaar genau einmal betrachtet und ein Knotenpaar verbindet, falls es noch keinen t-Pfad gibt, kann ein solches Paar nach Terminierung nicht vorhanden sein.

Laufzeit

Bei n gegebenen Knoten existieren $O(n^2)$ viele Knotenpaare. Für jedes der Knotenpaare wird einmal die Länge des kürzesten Pfades im aktuellen Graph mittels des Dijkstra-Algorithmus ermittelt, welcher eine Laufzeit von $O(n \log n + m)$ hat. Aufgrund seiner asymptotisch optimalen Größe enthält ein Greedy-Spanner nur linear viele Kanten [BC+10]. Die Laufzeit des Algorithmus ist demnach asymptotisch, neben weiter additiver Terme beispielsweise für die Sortierung, $n^2 \cdot n \log n \in O(n^3 \log n)$.

Speicherplatzverbrauch

Der Algorithmus sortiert und verwaltet $O(n^2)$ viele Knotenpaare, demensprechend ist der Speicherplatzverbrauch $O(n^2)$.

12 Greedy-Spanner mit linearem Speicherplatzverbrauch

Der in Kapitel 3 vorgestellte Algorithmus liefert einen gültigen Greedy-Spanner, allerdings ist der Speicherplatzverbrauch quadratisch in der Eingabegröße. Dies hat zur Folge, dass bereits Instanzen mit 13000 Knoten kaum mehr handhabbar sind [AB+15]. Dieses Kapitel beschreibt die Idee und Konstruktion eines Greedy-Spanner-Algorithmus, welcher mit linearem Speicherplatzverbrauch auskommt.

12.1 Idee des Algorithmus

Der quadratische Speicherplatzverbrauch resultiert aus der Betrachtung aller möglichen Knotenpaare. Der Linearspeicheralgorithmus setzt an dieser Stelle an und reduziert die Anzahl der nötigen Knotenpaarbetrachtungen. Die zugrundeliegende Idee von Alewijnse et al. ist eine Knotenzerlegung in Gruppenpaare, sodass zwischen den entstehenden Knotengruppen nur genau eine Kante eingefügt werden muss, welche von jedem t-Pfad zwischen Knoten der beiden Gruppen verwendet wird. Hat man demnach eine Kante zwischen zwei Knotengruppen eingefügt, so muss man alle weiteren Knotenpaare innerhalb dieser Gruppen nicht weiter betrachten. Ziel ist es also, eine Knotenzerlegung mit dieser Eigenschaft zu finden, die gleichzeitig linearen Speicherplatz nicht überschreitet.

Anforderungen an die Knotenzerlegung

An die Knotenzerlegung stellen wir die folgenden Anforderungen:
A1 Linearer Speicherplatzverbrauch darf nicht überschritten werden
A2 Jedes Knotenpaar wird durch genau ein Gruppierung getrennt
A3 Zwischen zwei Knotengruppen darf maximal eine Kante Teil des Greedy-Spanner sein
Anforderung A1 ist für einen Linearspeicheralgorithmus trivialerweise notwendig. Die zweite Anforderung liefert uns zwei Garantien. Zunächst stellt sie sicher, dass jeder Knoten Teil der Zerlegung ist. Sie schränkt jedoch noch weiter ein und fordert, dass jedes Knotenpaar durch genau eine Gruppierung getrennt wird. Ist dies erfüllt, so muss jede Gruppierung nur genau einmal angesehen werden, damit alle möglichen Knotenpaare abgedeckt sind. A3 garantiert, dass die Anzahl der Kanten zwischen je zwei Gruppen konstant (sogar 1) ist. In Kombination erzwingen A1, A2 und A3 also, dass wir linear viele Gruppenpaare jeweils genau einmal betrachten müssen und maximal eine Kante einfügen müssen. Insgesamt folgt so linearer Speicherplatzverbrauch.

12.2 Die Knotenzerlegung

Der Algorithmus verwendet die WSPD mit einem von t abhängig geeignet gewählten Parameter s. Diese Zerlegung erfüllt per Definition bereits Anforderung A2. Callahan hat gezeigt, dass man für jedes $s > 0$ eine WSDP der Größenordnung $O(s^d n)$ in $O(n \log n + s^d n)$ Zeit finden kann, welche mit $O(s^d n)$ Speicherplatz verwaltete werden kann [C95]. Da s nur in Abhängigkeit von t gewählt ist, gibt es eine WSPD, die linear in der Eingabegröße ist. Somit ist auch Anforderung A1 erfüllt.

Nun müssen wir uns noch davon überzeugen, dass A3 ebenfalls gilt. Hierfür machen wir zunächst folgende *Beobachtung 1*:

Für Knoten $a_1, a_2 \in A, b \in B$ zweier s-wohlseparierter Knotenmengen A, B gilt:

$$|a_1 a_2| \leq \frac{1}{s} \cdot |a_2, b|$$

Die direkte Distanz zwischen zwei Knoten a_1, a_2 einer Knotengruppe ist mindestens s-mal kürzer als die direkte Distanz zwischen einem beliebigen Knotenpaar aus $A \times B$. Der Extremfall, in welchem bei Beobachtung 1 Gleichheit gilt, ist in Abbildung 5 dargestellt und tritt dann auf, wenn die direkte Distanz der beiden Knoten aus verschiedenen Knotengruppen gleicher der minimalen Distanz der beiden Knotengruppen ist.

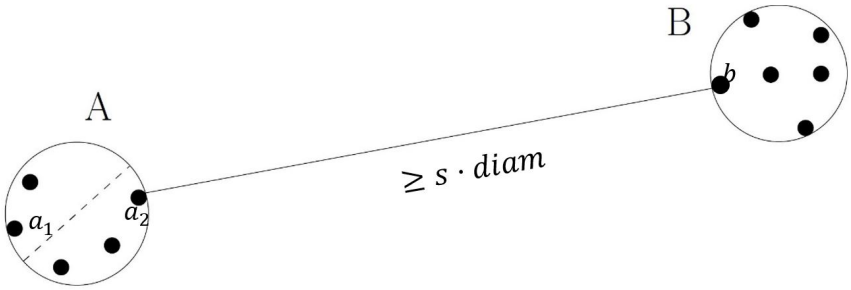


Abbildung 5 Zwei Knotengruppen mit minimal möglichem Abstand s zwischen Knoten a_2 und b.

Ohne den Algorithmus bereits vorgestellt zu haben wissen wir aus Kapitel 3, dass Greedy-Spanner-Algorithmen pro Schritt genau dann eine Kante zwischen zwei Knoten einfügen, wenn diese beiden Knoten die geringste direkte Distanz haben und zwischen ihnen noch kein t-Pfad existiert (ClosestPair). Um uns davon zu überzeugen, dass Anforderung A3 gilt, betrachten wir nun den Zeitpunkt im Algorithmus, an dem zwischen einem Knotenmengenpaar der WSPD ein zweites Knotenpaar betrachtet werden würde und zeigen, dass dieses nicht zur Kantenmenge hinzugefügt werden kann.

Es seien $a_1, a_2 \in A$ und $b_1, b_2 \in B$. Im Verlauf sei die Kante $\{a_1, b_1\}$ bereits eingefügt worden. Das nun betrachtete Knotenpaar sei a_2, b_2. Die Situation wird in Abbildung 6 verdeutlicht. Kriterium, ob eine Kante zwischen zwei Knoten eingefügt wird, ist, ob bereits ein

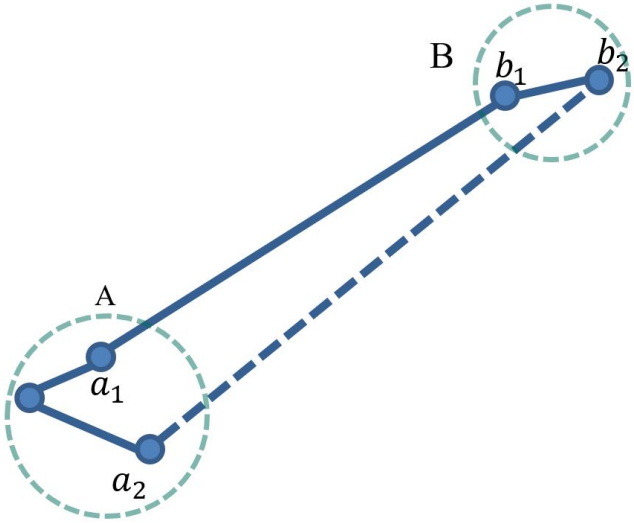


Abbildung 6 Situation bei Betrachtung eines weiteren Knotenpaarens zwischen A und B.

t-Pfad existiert oder nicht. Bevor wir die Kante $\{a_2, b_2\}$ einfügen, müssen wir also $\delta(a_2, b_2)$ betrachten. Da Greedy-Spanner-Algorithmen die Knotenpaare aufsteigend nach ihren direkten Distanzen betrachten wissen wir mit Beobachtung 1, dass alle Knotenpaare innerhalb von A und alle Paare innerhalb von B bereits betrachtet wurden und dementsprechend bereits einen t-Pfad besitzen. Es existierit somit im Graph bereits ein Pfad von a_2 nach b, nämlich

von a_2 über a_1 über b_1 zu b_2 . Wir können $\delta(a_2, b_2)$ also nach oben abschätzen mit

$$\delta(a_2, b_2) \leq \delta(a_2, a_1) + |a_1 b_1| + \delta(b_1, b_2)$$

Mit der Definition eines t-Pfad folgt direkt

$$\delta(a_2, b_2) \leq t \cdot |a_2 a_1| + |a_1 b_1| + t \cdot |b_1 b_2|$$

Mit Beobachtung 1 und $|a_1 b_1| \leq |a_2 b_2|$ (da $\{a_1, b_1\}$ früher betrachtet wurde) gilt

$$\delta(a_2, b_2) \leq \frac{t}{s} \cdot |a_2 b_2| + |a_2 b_2| + \frac{t}{s} \cdot |a_2 b_2|$$

Wählt man nun ***s=2t/(t-1)*** folgt insgesamt zu dem Zeitpunkt, an welchem wir $\{a_2, b_2\}$ betrachten

$$\delta(a_2, b_2) \leq t \cdot |a_2 b_2|$$

Es existiert demnach bereits ein t-Pfad, die Kante würde also nicht eingefügt werden. Somit steht fest, dass sobald zwischen zwei Knotengruppen der WSPD eine Kante existiert, die beiden Knotengruppen nicht weiter betrachtet werden müssen. Anforderung A3 ist somit erfüllt und wir können die WSDP als Grundlage unseres Linearspeicheralgorithmus verwenden.

12.3 Der Linearspeicheralgorithmus – erster Entwurf

Im Folgenden wird die Arbeitsweise des Linearspeicheralgorithmus skizzenhaft in einer nicht optimierten Version beschrieben. Der konkrete Pseudocode des Algorithmus wird nach den Optimierungen in Kapitel 5 vorgestellt.

Greedy Spanner(V,t)

1. WSPD berechnen für geeignetes s
2. Prioritätswarteschlange Q der Knotengruppenpaare füllen
3. Solange Q nicht leer
    4. Minimum der Schlange entfernen und zu E hinzufügen
    5. Warteschlange anpassen

Abbildung 7 Skizzierung der Arbeitsweise des Linearspeicheralgorithmus.

Der Algorithmus arbeitet nach der in Abbildung 7 skizzierten Vorgehensweise. Zunächst wird die WSPD mit $s = 2t/(t-1)$ berechnet. Diese liefert jedoch nur die minimalen Abstände zwischen den umschließenden Kreisen der Gruppenpaare. Wie Abbildung 8 zeigt, ist das ClosestPair jedoch nicht notwendigerweise Teil des Gruppenpaars mit minimalem Abstand. Im zweiten Schritt des Algorithmus wird deshalb eine Prioritätswarteschlange der Knotenpaare mit minimaler direkter Distanz pro Gruppenpaar aufgebaut.
Um pro Gruppenpaar $\{A, B\}$ das ClosestPair zu finden, wird auf allen Knoten der kleineren Gruppe der Dijkstra-Algorithmus durchgeführt. Existiert zwischen zwei Gruppenpaaren kein ClosestPair mehr, weil schon alle Paare über einen t-Pfad verbunden sind, so kann dieses Paar von der weiteren Berechnung ausgenommen werden.

Anschließend fügt der Algorithmus solange das Minimum der Prioritätswarteschlange zu E hinzu, bis diese leer ist. Durch das Hinzufügen einer Kante im 4. Schritt können sich neue t-Pfade im Graphen ergeben. Für ein vormaliges ClosestPair eines Gruppenpaars kann nun

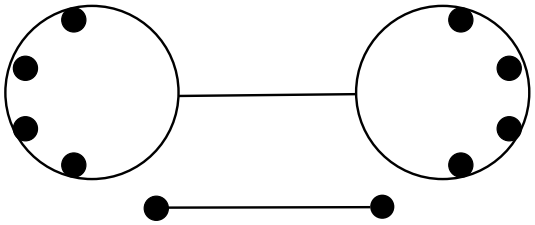


Abbildung 8 Knotenpaar mit minimaler direkten Distanz liegt nicht notwendig in Gruppenpaar mit minimalem Abstand.

ein t-Pfad existieren, weshalb es nicht mehr betrachtet werden soll. Nach dem Hinzufügen einer Kante muss die Warteschlange demnach stets neu angepasst werden, indem auf allen der p Gruppenpaare das ClosestPair neu berechnet wird.

Korrektheit

Dieser Greedy-Spanner-Algorithmus liefert ein korrektes Ergebnis, da er in jedem Schritt genau das Knotenpaar betrachtet, für welches noch kein t-Pfad existiert und dessen direkte Distanz am geringsten ist. Dies folgt aus Anforderung A2 und der Tatsache, dass für jedes der Gruppenpaare das ClosestPair in die Warteschlange aufgenommen wird.

Laufzeit

Für jedes Gruppenpaar $\{A, B\}$ wird das ClosestPair berechnet, indem auf jedem Element der kleineren Gruppe eine Dijkstra-Berechnung durchgeführt wird. Die Laufzeit des Dijkstra-Algorithmus ist $O(n \log n + m)$. In [BC+10] wurde gezeigt, dass ein Greedy-Spanner maximal $O(\frac{1}{(t-1)^d} n)$ viele Kanten besitzt, weshalb jede Dijkstra-Berechnung in $O(n \log n + \frac{1}{(t-1)^d} n)$ laufen. Insgesamt benötigt die Berechnung eines ClosestPair $O(min(|A|, |B|) \cdot (n \log n + \frac{1}{(t-1)^d} n))$. Die Summe der jeweils kleineren Knotengruppen kann nach oben abgeschätzt werden durch [AB+15]:

$$\sum_{i=1}^{p} min(|A|, |B|) \in O(s^d n \log n)$$

wobei s^d eine von t abhängige Konstante ist. Diese Berechnung führen wir nach jedem Einführen einer Kante, also $O(n)$ mal, durch. Die WSPD kann in $O(n \log n + s^d n)$ berechnet werden [C95]. Asymptotisch wird die Laufzeit also durch die ClosestPair-Berechnungen bestimmt und ist demnach $O(n^3 \log^2 n)$.

Speicherplatzverbrauch

Eine WSPD lässt sich mit linearem Speichererbrauch verwalten (Anforderung A1). Für die Dijkstra-Berechnungen wird pro Durchlauf $O(n)$ Speicher zur Verwaltung der Warteschlange benötigt, dieser kann jedoch wiederverwendet werden. Insgesamt benötigt der Algorithmus nur $O(n)$ Speicherplatz und erfüllt so das gewünschte Ziel.

13 Optimierungen

Die asymptotische Laufzeit des in Kapitel 4 vorgestellten Algorithmus soll verbessert werden. Da diese hauptsächlich von den Dijkstra-Berechnungen bestimmt ist wird versucht, die

Anzahl der Berechnungen zu verringern. Hierfür werden zwei Optimierungen angegeben.

13.1 Füllen der Prioritätswarteschlange

Für das Füllen der Prioritätswarteschlange Q wird eine Abbruchbedingung eingeführt. Wenn zum Zeitpunkt der Berechnung für ein Gruppenpaar A, B $min(A, B) > min(Q)$ gilt, muss für dieses Gruppenpaar das ClosestPair nicht berechnet werden, da es aktuell das Minimum der Prioritätswarteschlange verändern kann. Zur Vereinfachung dieser Überprüfung, werden alle Gruppenpaare zuvor aufsteigend nach ihrer Distanz sortiert. Diese Optimierung verändert zwar die Laufzeit asymptotisch nicht, erbringt aber dennoch eine Beschleunigung, da Dijkstra-Berechnungen gespart werden. Da das Sortieren linear vieler Gruppenpaare in $O(n \log n)$ läuft, fällt dies asymptotisch auch nicht weiter ins Gewicht.

13.2 Anpassen der Prioritätswarteschlange

Bisher würde, nachdem eine Kante eingefügt wurde, für alle noch verbleibenden Gruppenpaare das ClosestPair neu berechnet werden. Tatsächlich hat das Hinzufügen einer Kante nur auf eine beschränkte Anzahl der Gruppenpaare direkten Einfluss. Sind die Gruppenpaare hinreichend weit entfernt, so kann sich das ClosestPair nicht verändern. Dies kann man sich in Abbildung 9 gegebenen Beispiel klar machen.

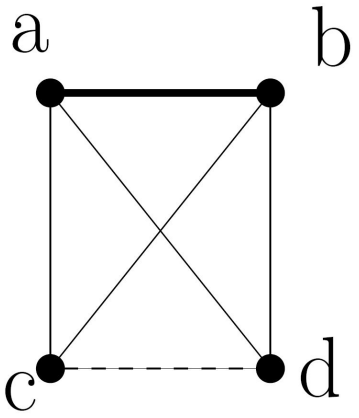


Abbildung 9 Beispiel der Auswirkung einer hinzugefügten Kante.

a,b,c,d sind Knoten, welche über Kanten verbunden sind. Die Kante $\{a, b\}$ wird hinzugefügt und wir betrachten deren Auswirkungen auf einen t-Pfad zwischen c und d. Gilt für die Abstände der Knoten

$$|ac|, |ad|, |bc|, |bd| > t \cdot |cd|$$

so kann kein t-Pfad zwischen c und d die Kante $\{a, b\}$ verwenden, da diese zu weit entfernt liegen. Angewandt auf unseren Algorithmus bedeutet dies, dass nach dem Einfügen einer Kante nur für diejenigen Gruppenpaare das ClosestPair neu berechnet werden muss, von denen mindestens einer der umschließenden Kreise maximal $t \cdot min(A, B)$ von einem der Endknoten der eingefügten Kante entfernt ist. Alle anderen Gruppenpaare sind so weit von der eingefügten Kante entfernt, dass sie in keinem der t-Pfade vorkommen kann.

Dank dieser Einschränkung ist die Anzahl an ClosestPair-Berechnungen pro Gruppenpaar nach oben durch $O(1 + s^d(1 + ts)^d)$ beschränkt [AB+15].

13.3 Der Linearspeicheralgorithmus

Der in Abbildung 10 dargestellte Pseudocode beschreibt das Füllen der Prioritätswarteschlange, also das Ermitteln des Knotenpaars jedes Gruppenpaars, welches die kürzeste direkte Distanz hat und noch nicht durch einen t-Pfad verbunden ist. Die in Kapitel 5.1

```
Algorithm FillQueue(Q, i )
1. while i ≤ p and either min(A_i , B_i ) ≤ min(Q) or Q is empty
2.       do p ←ClosestPair(i)
3.             if p is not nil , but a pair (u, v)
4.                   then add (u, v) to Q with key |uv|, and associate this entry with {A_i , B_i }
5.             i ←i + 1
```

Abbildung 10 Pseudocode – Füllen der Prioritätsarteschlange mit den jeweiligen ClosestPair.

```
Algorithm GreedySpanner(V, t)
1. Compute the WSPD W for V with s = 2t/(t−1) and let {A_i, B_i} be the resulting pairs, 1 ≤ i ≤ p
2. Sort the pairs {A_i, B_i} increasing according to min(A_i , B_i)
3. E ← ∅
4. Q ← empty priority queue
5. i ← 1
6. FillQueue(Q, i)
7. while Q is not empty
8.       do extract the minimum from Q, let this be (u, v)
9.             add (u, v) to E
10.            for all pairs { A_i , B_i } with an entry in Q for which either bounding box is
               at most t · min(A_j , B_j) away from either u or v
11.                  do p ←ClosestPair(j)
12.                        if p is nil , remove the entry in Q associated with { A_i ,B_i} from Q
13.                        if p is a pair (u', v'), update the entry in Q associated with { A_i  B_i}
                           to contain (u', v') and increase its key to |u'v'|
14.            FillQueue(Q, i)
15. return E
```

Abbildung 11 Pseudocode des Linearspeicheralgorithmus zur Berechnung des Greedy-Spanner.

eingeführte Optimierung findet sich in der ersten Zeile. In Abbildung 11 ist schließlich der Pseudocode des Linearzeitalgorithmus, inklusive der Optimierung aus Kapitel 5.2 in Zeile 10, angegeben. Die Variable i ist global. Der erneute Aufruf von *FillQueue* in Zeile 14 ist für den Fall notwendig, dass alle der j in Zeile 11 bis 13 betrachteten Paare aus der Warteschlange entfernt würden. Die Warteschlange könnte so leer sein (Abbruchbedingung Zeile 7), obwohl wegen der Optimierung in Zeile 1 der *FillQueue*-Operation noch gar nicht alle Gruppenpaare betrachtet wurden. Zeile 14 verhindert diese ungewollte Terminierung.

Korrektheit

Der optimierte Algorithmus ist nach wie vor korrekt, da die Optimierungen nur die Elemente der Prioritätswarteschlange betreffen, welche garantiert nicht das minimale Element sein können. Der Algorithmus wählt so in jedem Schritt noch immer das korrekte Knotenpaar.

Laufzeit

Wie schon erwähnt, hat die Optimierung aus Kapitel 5.1 keinen Einfluss auf die asymptotische Laufzeit. Die Optimierung in Kapitel 5.2 führt auf eine obere Schranke für die Anzahl an ClosestPair-Berechnungen pro Gruppenpaar von $O(1 + s^d(1 + ts)^d)$ [AB+15]. Mit der Laufzeitanalyse zur ClosestPair-Berechnung aus Kapitel 4.3 gilt für die Gesamtlaufzeit

$$O(\sum_{i=1}^{p}(1 + s^d(1 + ts)^d) \cdot min(|A_i|, |B_i|) \cdot n \log n + \frac{1}{(t-1)^d}n)$$

was wieder mit der Abschätzung $\sum_{i=1}^{p} min(|A|, |B|) \in O(s^d n \log n)$ vereinfachet werden kann zu

$$O(n^2 \log^2 n \cdot \frac{1}{(t-1)^3 d} + n^2 \log n \frac{1}{(t-1)^{4d}})$$

Die Laufzeit ist mit $O(n^2 \log^2 n)$ demnach nur um einen logarithmischen Faktor schlechter als die des derzeit besten bekannten theoretischen Algorithmus von $O(n^2 \log n)$ [BC+10].

Speicherplatzverbrauch

Das Sortieren der $O(s^d n)$ Gruppenpaare benötigt lediglich $O(n)$ Speicherplatz. Alle anderen Optimierungen reduzieren nur die Anzahl an Dijkstra-Berechnungen und benötigen deshalb keinen zusätzlichen Speicherplatz, weshalb der Gesamtspeicherplatzverbrauch nach wie vor $O(n)$ ist.

14 Experimente

Dieses Kapitel beschäftigt sich mit dem praktischen Verhalten des Algorithmus. In mehreren Experimenten wurde der von Alewijnse entwickelte Algorithmus mit anderen verglichen.

14.1 Vorbedingungen

Algorithmen

Der Algorithmus wurde mit zwei weiteren Algorithmen verglichen:

- FG-Greedy [FG07]
  Dieser von Farshi et al. vorgestellte Greedy-Spanner-Algorithmus hat eine Laufzeit von $O(n^3 \log n)$, weist in der Praxis jedoch nahezu quadratisches Laufzeitverhalten auf.
- Θ-Graph [K88]
  Dieser Algorithmus bietet mit einer Laufzeit von $O(n \log n)$ eine deutlich schnellere Konstruktion eines t-Spanner. Allerdings ist die Qualität dieses Spanner schlechter als beim Greedy-Ansatz.

Maschine

Die Algorithmen wurden auf einem Intel Core i5-3470 (Quad-Core, 3,2 GHz) bei 4GB Arbeitsspeicher durchgeführt. Das verwendete Betriebssystem war Debian 6.0.7. Die Algorithmen wurden in C++ implementiert.

Punktmengen

Die Algorithmen wurden auf drei verschiedenen Punktmengen getestet
1. Gleichverteile Punktmenge (engl.: *uniform*)
2. Gruppierte Punktmengen (engl.: *clustered*) auf $\sqrt{n}$ Clustern der Größe $\sqrt{n}$
3. Punktdatensatz mit geografischen Datenpunkten der USA

14.2 Variation von t

Der vorgestellte Algorithmus wird mit dem FG-Greedy-Algorithmus bezüglich Laufzeit und Speicherplatzverbrauch bei Variationen von t zwischen 1,1 und 2 auf einer Punktmenge mit $n = 10000$ Knoten verglichen. Zunächst wurden die Algorithmen für verschiedene Werte von

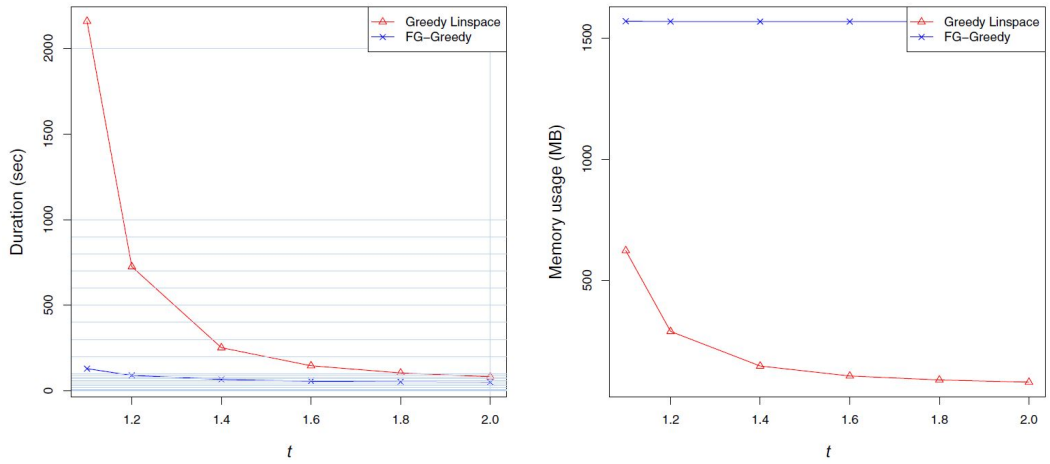


Abbildung 12 Vergleich der Laufzeiten (links) sowie des Speicherverbrauchs (rechts) des vorgestellten Algorithmus und FG-Greedy bei gleichverteilter Punktmenge. Diagramme von Alewijnse et al. übernommen [AB+15].

t auf einer gleichverteilten Punktmenge ausgeführt. Die Ergebnisse sind in Abbildung 12 aufgetragen. Die linken Kurven zeigen die Ausführungsdauer (engl.: *duration*) in Sekunden in Abhängigkeit von t. Die rechten Kurven zeigen den Speicherplatzbedarf (engl.: *memory usage*) in MB, ebenfalls in Abhängigkeit von t.

Es fällt auf, dass beide Algorithmen für größere Werte von t sehr ähnliche Laufzeiten haben. Für kleine Werte von t steigt die Laufzeit des vorgestellten Algorithmus jedoch deutlich an. Dies lässt sich vermutlich über eine durch den größeren Dehnungsfaktor s stark erhöhte Anzahl an Gruppenpaaren erklären. Dies deckt sich auch mit dem Anstieg des Speicherplatzbedarfs.

In einem weiteren Versuch wurden die Algorithmen auf geclusterten Punktmenge mit ebenfalls $n = 10000$ Knoten ausgeführt. Abbildung 13 zeigt wieder links die Laufzeit und rechts den Speicherplatzbedarf, jeweils in Abhängigkeit von t.

Zunächst fällt auf, dass beide Algorithmen im Vergleich zu gleichverteilten Punktmenge deutlich schneller (Faktor größer 20) sind. Greedy-Spanner scheinen also stark von Clusterung

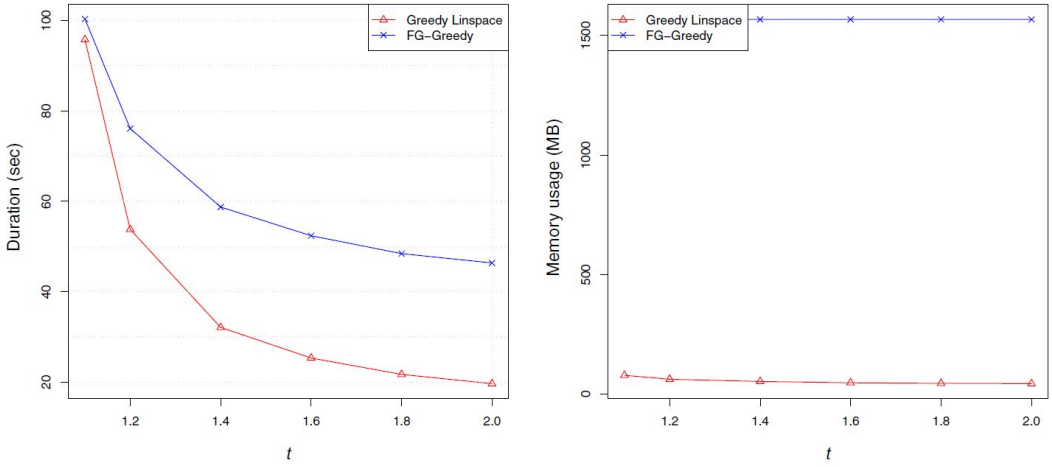


Abbildung 13 Vergleich der Laufzeiten (links) sowie der Speicherverbrauchs (rechts) des vorgestellten Algorithmus und FG-Greedy bei geclusterter Punktmenge. Diagramme von Alewijnse et al. übernommen [AB+15].

zu profitieren. Die zweite Auffälligkeit ist, dass der Linearspeicheralgorithmus dieses Mal durchgehend schneller ist, als FG-Greedy. Für große Werte von t sogar um den Faktor 2. Die WSPD profitiert hier stark von der Clusterung, sodass nur wenige Paare gebildet werden müssen. Dies deckt sich wieder mit den Beobachtungen beim Speicherplatzbedarf, der nun auch für kleine Werte von t kaum ansteigt.

14.3 Qualität des t-Spanner

Das Ergebnis des Greedy-Spanner wurde mit dem des Θ-Graph hinsichtlich der Qualitätsmerkmale bei Berechnung eines 2-Spanner verglichen. Die Werte sind in untenstehender Tabelle aufgelistet. Die Spalten der Tabelle beinhalten von links nach rechts:

- den verwendeten Algorithmus
- die Punktmenge, auf welcher der Algorithmus ausgeführt wird
- die Größe der Punkmenge
- die Anzahl der Kanten im Greedy-Spanner nach Ausführung des Algorithmus
- den maximalen Knotengrad im Greedy-Spanner nach Ausführung des Algorithmus
- die Summe der Kantengewischte im Greedy-Spanner nach Ausführung des Algorithmus

Der *Theta*-Graph bietet zwar deutlich schnellere Laufzeit, der Greedy-Spanner schneidet jedoch in allen Qualitätspunkten deutlich besser ab. Zwar liefern beide Algorithmen asymptotisch optimale Größe, der konstante Faktor beim Greedy-Ansatz ist jedoch mit $\sim$1,4 deutlich besser als der Faktor $\sim$4 des *Theta*-Graph.

Die geclusterte Punktmenge besteht aus 1 Mio. Knoten. Ein Greedy-Spanner konnte zuvor auf Instanzen dieser Größe aufgrund des benötigten Speicherplatzbedarfs nicht berechnet werden.

Spanner	Daten	$\|V\|$	$\|E\|$	Max Grad	Gewicht
Greedy	USA	115 475	171 456	5	11 086 417
Θ-Graph	USA	115 475	465 230	62	53 341 205
Greedy	Uniform	500 000	720 850	6	9 104 690
Θ-Graph	Uniform	500 000	2 063 164	22	39 153 380
Greedy	Clustered	1 000 000	1 409 946	6	4 236 016
Θ-Graph	Clustered	1 000 000	4 157 016	135	59 643 264

Abbildung 14 Qualitätsmerkmale des Greedy-Spanner und des Θ-Graph im Vergleich bei Berechnung eines 2-Spanner.

15 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Algorithmus präsentiert, welcher den Greedy-Spanner mit linearem Speicherplatzverbrauch berechnet. Nach einigen Optimierungen des Algorithmus hat dieser eine Laufzeit von $O(n^2 \log^2 n)$ und ist damit lediglich um einen logarithmischen Faktor langsamer als der derzeit schnellste bekannte theoretische Algorithmus, was ihn nach Einschätzung von Alewijnse et al. zum derzeitigen Mittel der Wahl zur Berechnung eines t-Spanner macht. Die Einsparungen im Platzverbrauch erreicht der Algorithmus, indem er nicht alle mögliche Knotenpaare miteinander vergleicht. Stattdessen werden die Knoten so in hinreichend weit entfernte Gruppen zerlegt, dass es ausreichend ist, nur wenige der Knotenpaare zu betrachten.

In mehreren Experimenten wurde gezeigt, dass der Algorithmus praktikabel ist. Greedy-Spanner-Algorithmen mit quadratischem Speicherplatzverbrauch sind bereits bei 13000 Knoten nicht mehr verwendbar. Der vorgestellte Linearspeicheralgorithmus konnte noch mit 1 Mio. Knoten problemlos umgehen. Gleichzeitig bietet der Algorithmus alle Qualitäten eines Greedy-Spanner: asymptotische Optimalität in Größe, Gewicht und maximalen Knotengrad.

Flaschenhals der Laufzeit ist die permanente Neuberechnung der kürzesten Pfade im Graphen. Findet man hier eine effiziente Lösung, so könnten sogar subquadratische Laufzeiten erreicht werden.

Anhänge

Variablenverzeichnis

V	Knotenmenge
E	Kantenmenge
t	Dehnungsfaktor des Geometric-Spanner, $t \leq 1$
s	Dehnungsfaktor der WSPD. $s = 2t/(t-1)$
n	Eingabegröße. Mächtigkeit der gegebenen Punktmenge
m	Anzahl der Kanten
d	Dimension des metrischen Raums
p	Anzahl an Knotenmengenpaaren der WSPD

Referenzen

K88 Keil, J.M. Approximating the complete euclidean graph. In *1st Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 318 of LNCS, pp. 208–213, Springer (1988)

C95 Callahan, P.B. Dealing with higher dimensions: the well-separated pair decomposition and its applications. PhD thesis, Johns Hopkins University, Baltimore, Maryland (1995)

BC+10 Bose, P., Carmi, P., Farshi, M., Maheshwari, A., Smid, M. Computing the greedy spanner in nearquadratic time. *Algorithmica*, 58(3), 711–729 (2010)

AB+15 Sander P. A. Alewijnse, Quirijn W. Bouts, Alex P. ten Brink, Kevin Buchin Computing the Greedy Spanner in Linear Space. *Algorithmica*, 73(3), 589–606 (2015)

FG07 Farshi, M., Gudmundsson, J. Experimental study of geometric t-spanners. *Experimental Algorithms: 6th International Workshop*, 270–284, Springer (2007)

10 Shortest Path to a Segment and Quickest Visibility Queries

Johannes Bader

Zusammenfassung

Das Finden von kürzesten Wegen zwischen einem Start- und Zielpunkt in einem Polygon (mit Löchern) ist ein bekanntes Thema der algorithmischen Geometrie. Im wissenschaftlichen Artikel "Shortest Path to a Segment and Quickest Visibility Queries" [1] von E. Arkin et al. werden zwei Varianten dieses Problems betrachtet und in Zusammenhang gebracht. Zum einen geht es um das Finden von Wegen zu einer *Zielstrecke* statt einem Zielpunkt, zum anderen geht es darum, einen Zielpunkt möglichst schnell zu *sehen* statt zu erreichen. Auch diese beiden Problemvarianten wurden in der Vergangenheit bereits analysiert, allerdings sind nur wenige Algorithmen bekannt, die sich Vorberechnungen zu Nutze machen. In diesem Sinne werden zahlreiche neue Verfahren mit teils optimaler Laufzeit vorgestellt.

10.1 Einleitung

Im Folgenden werden zwei Problemstellungen aus der algorithmischen Geometrie betrachtet und neue Lösungsverfahren vorgestellt und mit bestehenden verglichen. Es handelt sich um Modifikationen des Kürzeste-Wege-Problems in Polygonen. Im klassischen Kürzeste-Wege-Problem ist ein Polygon, ein Startpunkt sowie ein Zielpunkt gegeben, wobei der kürzeste Weg vom Start- zum Zielpunkt gesucht wird (welcher die Polygonfläche nicht verlässt).

Die eine Variante sucht den kürzesten Weg vom Startpunkt zu einer *Zielstrecke* statt einem Zielpunkt (engl. Shortest Path to a Segment) und wird in Abschnitt 10.2 behandelt. Die andere Variante sucht nicht den kürzesten Weg zum Zielpunkt, sondern den kürzesten Weg zu einem beliebigen Punkt, von dem aus der Zielpunkt zu *sehen* ist (engl. Quickest Visibility). Diese Variante wird in Abschnitt 10.3 behandelt.

Abschnitte 10.2 und 10.3 sind im Wesentlichen gleich aufgebaut: Zunächst wird die jeweilige Problemstellung nochmals formal angegeben und erläutert. Daraufhin werden die Verfahren, die in dieser Ausarbeitung (bzw. [1]) vorgestellt werden, mit bereits bekannten Verfahren verglichen und in Zusammenhang gebracht. Im Anschluss werden die neuen Verfahren dann detailliert vorgestellt.

10.1.1 Motivation

Das Finden von kürzesten Wegen zu Strecken ist integraler Bestandteil zum Berechnen von kürzesten Wegen zu *beliebig* geformten Bereichen, da deren Umriss entweder aus Strecken zusammengesetzt oder zumindest approximiert werden kann. Von dieser Tatsache wird auch in Abschnitt 10.3.4 Gebrauch gemacht.

Einen Punkt möglichst schnell zu sehen spielt zum Beispiel in Überwachungs-Szenarien eine Rolle. Statt physischem Kontakt mit dem Ziel reicht dort zumeist Sichtkontakt aus, um zum Beispiel Diebstahl eines Objekts auszuschließen. Eine darauf aufbauende bekannte Problemstellung ist es, in möglichst kurzer Zeit (also mit möglichst wenig zurückgelegtem Weg) Sichtkontakt mit dem *gesamten* zu überwachenden Bereich zu erlangen. Dies ist eine auf Sichtbarkeit basierende Variante der Routenplanung, die in [4, 5, 6, 15, 16] behandelt wird.

Genauso gibt es aber auch Anwendungen in der Robotik oder drahtlosen Sensornetzen, zum Beispiel wenn es darum geht, möglichst schnell in Funkkontakt zu treten (wobei Sichtbarkeit als Approximation von Störungsfreiheit dient).

10.1.2 Vorberechnungen

Die Idee von Vorberechnungen basiert auf der Annahme, dass Algorithmen einen Teil ihrer Eingabe schon im Voraus mitgeteilt bekommen und anhand dessen beliebige Vorbereitungen treffen dürfen.

Es ist eine naheliegende Annahme, dass Zielpunkt/-strecke häufig variieren (zum Beispiel unterschiedliche Exponate in einem Museum), während Polygon und Startpunkt gleich bleiben (zum Beispiel Lageplan und Eingang des Museums). Dadurch kann es sich lohnen, basierend auf Polygon und Startpunkt (genannt "Probleminstanz") Vorberechnungen anzustellen. Werden dann zu einem späteren Zeitpunkt die noch fehlenden Informationen (genannt "Anfrage", hier: Zielpunkt/-strecke) bereitgestellt, so kann das Problem potentiell schneller gelöst werden.

Erlaubt man Vorberechnungen auf der Probleminstanz spielen bei der Bewertung der Verfahren zwei Laufzeiten eine Rolle: Zum einen die Laufzeit T_v der Vorberechnungen, zum anderen die Laufzeit T_a zur Beantwortung einer Anfrage (engl. query). Die Idee ist es also, eine möglichst geringere Anfragezeit T_a zu erreichen, wofür potentiell hohe Vorberechnungszeit T_v in Kauf genommen wird. T_a sollte damit stets geringer als die Laufzeit des besten bekannten Algorithmus ohne Vorberechnungen sein, sonst haben sich die Vorberechnungen nicht ausgezahlt.

10.2 Shortest Path to a Segment Queries

Bei Shortest Path to a Segment Queries (kurz: SPSQ) geht es darum, den kürzesten Weg zwischen einem Punkt s und einer Strecke $\bar{q}$ in einer Polygonfläche $D \subset \mathbb{R}$ zu bestimmen. Polygonfläche D und Punkt s seien von vornherein bekannt und dürfen für Vorberechnungen genutzt werden.

10.2.1 Notation

Um die Problemstellung nachfolgend zu formalisieren, vereinbaren wir folgende Notationen. Beachte, dass sowohl von "Knoten" als auch "Punkten" von D die Rede ist. Mit Knoten von D sind die (endlich vielen) Knoten gemeint, die das Polygon *definieren*. Mit Punkten von D sind dagegen beliebige Punkte im Inneren oder auf dem Rand von D gemeint. Knoten von D sind also auch Punkte, aber nicht umgekehrt.

Der kürzeste Weg zwischen Punkten $v_1 \in D$ und $v_2 \in D$ sei $\pi(v_1, v_2)$. Kürzeste Wege dürfen das Polygon D nicht verlassen. Zwei Punkte $v_1, v_2 \in D$ "sehen" sich, wenn deren direkte Verbindungsstrecke komplett in D liegt. Man betrachtet das Innere des Polygons also als durchsichtiges Medium, dessen Rand allerdings als undurchsichtiges und undurchdringbares Hindernis. Die Länge einer Kante oder eines Pfades p sei $|p|$. Damit ist zum Beispiel $|\pi(v_1, v_2)|$ die Länge des kürzesten Weges zwischen v_1 und v_2. Für zwei Punkte $v_1, v_2 \in D$, welche sich gegenseitig sehen sei $\text{line}(v_1, v_2)$ die maximale Verlängerung der Verbindungsstrecke von v_1 nach v_2 innerhalb von D. Es handelt sich bei $\text{line}(v_1, v_2)$ also um eine Strecke, die durch v_1 und v_2 verläuft (kurz: $v_1, v_2 \in \text{line}(v_1, v_2)$) und deren Endpunkte auf dem Rand von D liegen.

Für einen beliebigen Punkt $v \in D$ sei $\text{pred}(v)$ ("Vorgänger") der letzte Knoten von D auf dem kürzesten Weg $\pi(s, v)$ von s nach v. Ist $\pi(s, v)$ lediglich eine einzige Kante von s nach v (d.h. kein Knoten von D ist involviert), so sei $\text{pred}(v) = s$. Für beliebige Punkte $v_1, v_2 \in D$ sei $\text{lca}(v_1, v_2)$ der letzte Knoten von D, der sowohl auf dem kürzesten Weg $\pi(s, v_1)$ als auch $\pi(s, v_2)$ liegt. Es handelt sich also um den letzten *gemeinsamen* Knoten der beiden Wege. Analog zur Definition von pred sei $\text{lca}(v_1, v_2) = s$, falls kein solcher Knoten von D existiert (zum Beispiel weil die beiden Pfade von s aus sofort in verschiedene Richtungen starten).

Ein Shortest Path Tree (kurz: SPT) bezüglich s ist ein Baum, der alle kürzesten Wege von s zu beliebigen *Knoten* von D enthält. Die Wurzel des Baumes ist also s, die übrigen Knoten des Baumes sind die Knoten von D. Durch Zeiger auf die Vaterknoten kann ein SPT die pred-Funktion für Knoten von D in konstanter Zeit berechnen.

Eine Shortest Path Map (kurz: SPM) bezüglich s ist die Vorberechnung der pred-Funktion zu jedem *Punkt* in D. Diese Vorberechnung ist möglich, weil stets ganze Bereiche von D (Zellen) genau den selben Vorgängerknoten besitzen, pred ist also zellenweise konstant. Die SPM berechnet diese Zellen vor und speichert zu jeder den Vorgängerknoten. Die SPM kann die pred-Funktion im Gegensatz zum SPT also für beliebige Punkte von D berechnen, benötigt hierfür aber logarithmische Laufzeit.

10.2.2 Problemstellung

Der gesuchte kürzeste Weg sei mit $\bar{\pi}(s \in D, \bar{q} \subset D)$ bezeichnet, wobei die Zielstrecke $\bar{q}$ ein Segment in D ist. Der Punkt auf dem er endet sei $q^* \in \bar{q}$. Umgekehrt gilt: Ist q^* bekannt, so kann der kürzeste Weg anhand einer SPM bezüglich s rekonstruiert werden. Formal besteht also folgender Zusammenhang:

$$\bar{\pi}(s, \bar{q}) = \pi(s, q^*) \tag{1}$$

$$q^* = \arg\min_{q \in \bar{q}} |\pi(s, q)| \tag{2}$$

Wir suchen im Folgenden daher nur nach q^*, das Problem ist dann auf ein klassisches Kürzeste-Wege-Problem reduziert. Bei Bedarf kann der Lösungspfad durch wiederholtes anwenden von pred von q^* aus rekonstruiert werden.

10.2.3 Ergebnisse und Einordnung

Für Aussagen über Laufzeit sind zwei Größen relevant: Zum einen ist das die Anzahl der Löcher des Polygons h (Löcher sind polygon-förmige Lücken in der Polygonfläche; siehe 6, dort sind 2 Löcher in grau eingezeichnet). Polygone ohne Löcher (also mit $h = 0$) werden auch als "einfache" Polygone bezeichnet. Zum anderen sei n die Anzahl der Knoten von D.

Bemerkung zu Löchern: Löcher liegen im Inneren von D, sonst wären sie einfach Teil des Polygonrandes. Passiert ein Pfad in D ein Loch, so kann er es also entweder links oder rechts von sich liegen lassen. Für einen kürzesten Pfad ist es also äußerst relevant, auf welcher Seite man ein Loch passiert (eine der beiden Varianten sorgt potentiell dafür, dass der Pfad nicht mehr optimal sein kann). Mit der Anzahl der Löcher erhöht sich also auch die Anzahl kombinatorisch unterschiedlicher Möglichkeiten, diese zu passieren, was sich unweigerlich auf die Komplexität der Algorithmen auswirkt. Verfahren für einfache Polygone können im Umkehrschluss schneller sein, da die Komplexität potentieller Lösungspfade geringer ist.

Ohne Vorberechnungen ($\mathsf{T}_v = 0$) kann ein SPSQ mit Hilfe des kontinuierlichen Dijkstra Verfahrens in $\mathsf{T}_a = O(n \log n)$ Laufzeit beantwortet werden [14]. Bei dem Verfahren wird ausgehend von s eine Art Wellenfront durch D propagiert, welche $\bar{q}$ als erstes im Punkt q^* erreichen wird. Dieses Konzept wurde auch in [1] aufgegriffen, allerdings um Vorberechnungen

anzustellen. Das resultierende Verfahren erreicht Laufzeiten von $\mathsf{T}_v = O(n^2 \log^2 n)$ und $\mathsf{T}_a = O(\log^2 n)$. Ein weiteres vorgestelltes Verfahren erreicht $\mathsf{T}_v = O(n^3 \log n)$ und $\mathsf{T}_a = O(\log n)$.

Der Spezialfall, dass D ein einfaches Polygon ist, erlaubt effizientere Algorithmen. Im Folgenden wird ein optimales Verfahren mit $\mathsf{T}_v = O(n)$ und $\mathsf{T}_a = O(\log n)$ vorgestellt.

10.2.4 Verfahren für einfache Polygone

Vorberechnet werden ein SPT sowie eine SPM bezüglich Startpunkt s. Zudem wird eine Datenstruktur gebaut, die beliebige LCA (lowest common ancestor) Anfragen auf dem SPT in konstanter Laufzeit beantworten kann [8]. All diese Vorberechnungen sind mit bekannten Algorithmen in $O(n)$ möglich. Außerdem nehmen wir an, dass uns die n Knoten des Polygons (zum Beispiel in Form von Koordinatenpaaren) in einem Array $\vec{D}$ vorliegen. Sie seien darin *im Uhrzeigersinn* sortiert.

Dank SPM können die Funktionen pred und lca dann auf beliebigen Punkten von D in $O(\log n)$ berechnet werden. Der SPT berechnet die beiden Funktionen dagegen in $O(1)$, aber nur für Knoten von D. Man beachte dass die lca-Funktion der SPM nicht mittels einer weiteren Datenstruktur, sondern anhand von pred und dem SPT umgesetzt wird:

$$\mathrm{lca}_{\mathrm{SPM}}(a, b) := \mathrm{lca}_{\mathrm{SPT}}(\mathrm{pred}_{\mathrm{SPM}}(a), \mathrm{pred}_{\mathrm{SPM}}(b)) \tag{3}$$

Wie schon weiter oben (10.2.2) erwähnt lassen sich anhand von pred kürzeste Wege von s zu einem beliebigen Punkt rückwärts aufzählen. Dies allein benötigt $\Omega(n)$ Laufzeit, allerdings wird $\mathsf{T}_a = O(\log n)$ angestrebt. Erst dadurch, dass lediglich der Punkt q^* ermittelt werden soll und gerade *nicht* der gesamte Weg dorthin, ist logarithmische Laufzeit möglich.

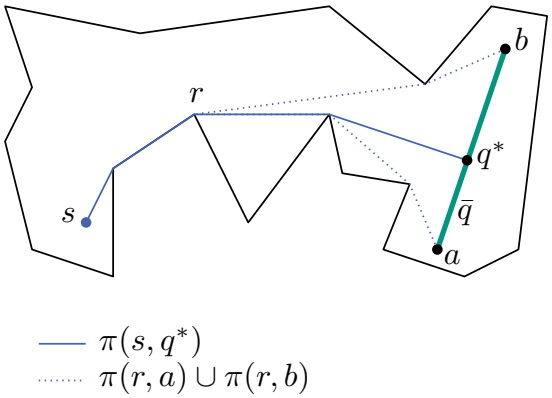


Abbildung 1 Beispiel einer SPSQ-Instanz. Zu sehen ist Polygon D, Startpunkt s, Anfrage $\bar{q}$, Lösungspfad $\pi(s, q^*)$, sowie der Trichter (angedeutet durch gepunktete Linien).

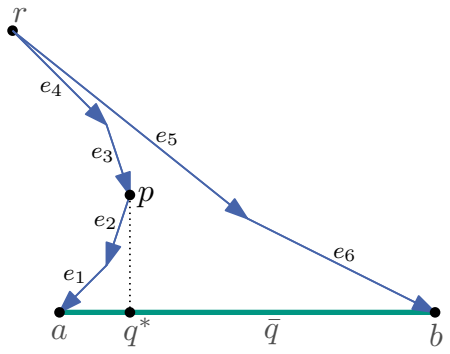


Abbildung 2 Genauere Betrachtung eines Trichters, inklusive Kanten E und Punkt p, der orthogonal über q^* steht.

Im Folgenden wird das Beantworten einer Anfrage anhand obiger Vorberechnungen beleuchtet. Die Anfrage $\bar{q}$ sei die Strecke zwischen zwei Punkten a und b (siehe Abbildung 1). Die kürzesten Wege $\pi(s, a)$ und $\pi(s, b)$ zu den beiden Endpunkten trennen sich im Punkt $r = \mathrm{lca}(a, b)$. Die drei Punkte a, b und r seien o.B.d.A. gegen den Uhrzeigersinn angeordnet (vertausche ansonsten a und b). Die von $\pi(r, a)$, $\pi(r, b)$ und $\bar{q}$ eingeschlossene Fläche wird im Folgenden als "Trichter" bezeichnet. Der kürzeste Weg von s zu $\bar{q}$ bzw. q^* geht dann ebenfalls durch Punkt r, betritt dort den Trichter und verlässt diesen auch nicht mehr.

Die Kanten E des Kantenzugs $\pi(a, r)$, $\pi(r, b)$, seien im Folgenden mit $e_1, ..., e_t$ bezeichnet. Sie seien zudem allesamt von r weg gerichtet (siehe Abbildung 2), die korrespondierenden Richtungsvektoren seien auch mit $e_1, ..., e_t$ bezeichnet.

Die binäre Relation $\times_{\text{ccw}}$ gebe an, ob zwei Vektoren durch Rotation gegen den Uhrzeigersinn aufeinander abgebildet werden können. Genauer gelte $u \times_{\text{ccw}} v$ genau dann, wenn u durch Rotation um 0° bis 180° gegen den Uhrzeigersinn auf v abgebildet werden kann (Länge vernachlässigt).

▶ **Theorem 1.** *Für zwei aufeinander folgende Vektoren e_i und e_{i+1} gilt stets $e_i \times_{ccw} e_{i+1}$.*

Beweis. Annahme: Für e_i und e_{i+1} gelte nicht $e_i \times_{\text{ccw}} e_{i+1}$.

Falls beide Kanten in r starten ("Spitze" des Trichters, in Abbildung 2 wäre $i = 4$), so ist per Definition $e_i \in \pi(r, a)$ und $e_{i+1} \in \pi(r, b)$. Aufgrund unserer Annahme dass a, b und r gegen den Uhrzeigersinn angeordnet sind, müssen $\pi(r, a)$ und $\pi(r, b)$ irgendwo kreuzen. Damit kann es sich nicht mehr kürzeste Wege handeln. Dies steht im Widerspruch zur Definition.

Falls beide Kanten dagegen zu $\pi(r, a)$ gehören, so lässt sich nun ein kürzerer Weg von r nach a konstruieren: Der Startknoten von Kante e_{i+1} sei v_1, der Endknoten von Kante e_i sei v_2. Dann liegt die Kante von v_1 zu v_2 im Inneren des Trichters. Dank der Dreiecksungleichung lässt sich $\pi(r, a)$ also verkürzen, wenn e_i und e_{i+1} durch jene Kante ersetzt werden. Dies steht im Widerspruch zur Definition von $\pi(r, a)$.

Der Beweis für $e_i, e_{i+1} \in \pi(r, b)$ verläuft analog. ◀

▶ **Theorem 2.** *Es gilt auch $e_i \times_{ccw} \bar{q}$, wobei $\bar{q}$ der Richtungsvektor von a nach b ist.*

Beweis. Alle $e_i \in E$ zeigen per Definition in Richtung $\bar{q}$ und befinden sich auch auf derselben Seite von $\bar{q}$, nämlich "links davon", wenn man auf Punkt a steht und Richtung b blickt, also die Perspektive von $\bar{q}$ annimmt. Dies ist direkte Konsequenz aus der Annahme, dass a, b und r gegen den Uhrzeigersinn angeordnet (siehe Abbildungen 1 und 2).

Demnach zeigen aus der Perspektive von $\bar{q}$ alle $e_i \in E$ nach rechts. Dreht man einen solchen Vektor also um maximal 180° gegen den Uhrzeigersinn, so wird dieser irgendwann deckungsgleich mit $\bar{q}$ (bis auf Länge). Das ist gerade die Behauptung. ◀

Folgende Beobachtungen lassen Rückschlüsse zu, wie der kürzeste Weg genau durch den Trichter verläuft: Sucht man den kürzesten Weg zu einer Gerade (auf einer Fläche ohne Hindernisse), so verläuft dieser stets exakt orthogonal zu der Gerade. Genauso läuft deshalb auch ein SPSQ-Lösungspfad $\pi(s, q^*)$ orthogonal auf $\bar{q}$ zu, sobald diese orthogonale Strecke frei von Hindernissen ist. Der Knoten, bei dem diese (letzte) Kante des Lösungspfads beginnt, sei p (siehe Abbildung 2).

▶ **Theorem 3.** *Der Punkt p liegt auf dem Rand des Trichters.*

Beweis. Annahme: p liegt im Inneren des Trichters (siehe Abbildung 3).

Die Strecke von p nach q^* berührt dann irgendwo den Rand des Trichters (ansonsten wäre p im Widerspruch zu seiner Definition nicht der erste Punkt mit besagter Eigenschaft, es bestand schon vorher freie Sicht auf $\bar{q}$; siehe Punkt p_x in Abbildung 3). Dieser Berührpunkt sei p'. Der Pfad $\pi(s, p')$ ist dann ein Teilpfad von $\pi(s, a)$ oder $\pi(s, b)$, verläuft also *nicht* im Inneren des Trichters, sondern auf dessen Rand.

Der Pfad $\pi(s, q^*)$ wiederum verläuft per Definition über p, dann über p' und endet in q^*. Insbesondere ist $\pi(s, p')$ also ein Präfix von $\pi(s, q^*)$, verläuft also auch über p. Da p im Inneren des Trichters liegt, verläuft auch $\pi(s, p')$ durch das Innere des Trichters. Das ist ein Widerspruch, es wurde soeben schon das Gegenteil bewiesen. ◀

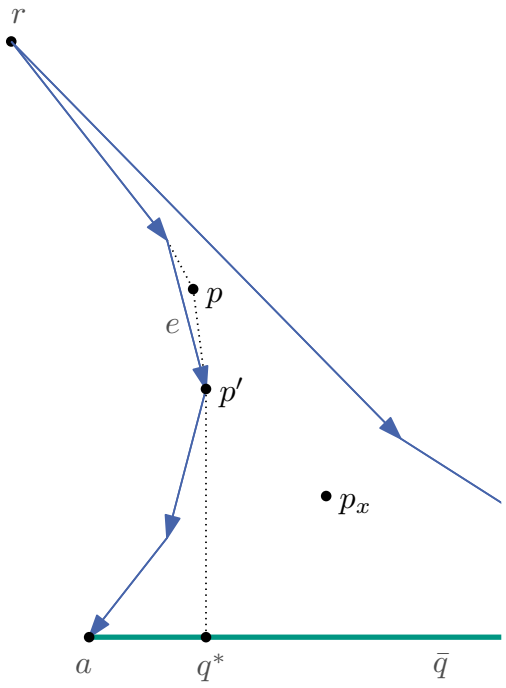


Abbildung 3 Darstellung zu Theorem 3. Der kürzeste Weg zu $\bar{q}$ verläuft offensichtlich nicht über p_x. Genauso lässt sich ein Weg über p verkürzen, hier anhand der Kante e.

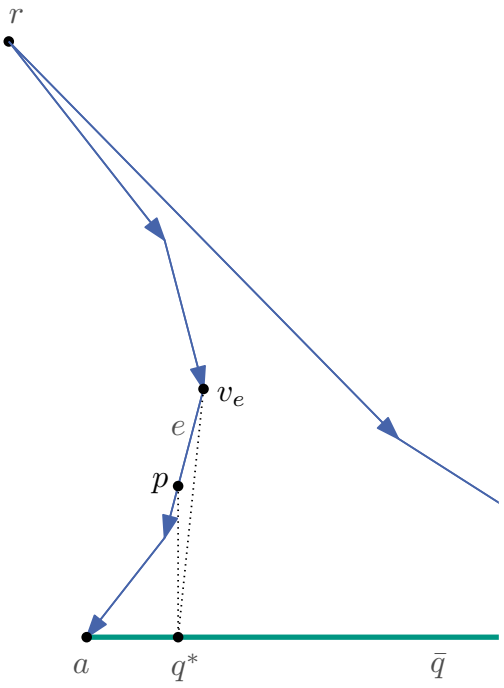


Abbildung 4 Darstellung zu Korollar 4. Wäre p innerer Knoten von e, so könnte der Weg mit Hilfe der Strecke von v_e nach q^* verkürzt werden.

▶ **Korollar 4.** *Der Punkt p ist Endpunkt von zwei Kanten aus E, außer es handelt sich gerade um a oder b (Endknoten des Kantenzugs).*

Beweis. Angenommen das Korollar gelte nicht, d.h. p ist innerer Punkt einer Kante e aus E (siehe Abbildung 4). Der Pfad $\pi(s,p)$ verläuft über einen der Endpunkte von e. Dieser Endpunkt sei v_e und es gilt $v_e = \text{pred}(p)$. Die Strecke von v_e nach q^* verläuft dank Lochfreiheit von D komplett innerhalb des Trichters. Aufgrund der Dreiecksungleichung ist diese Strecke auch kürzer als der Weg von v_e über p nach q^*. Der angeblich über p verlaufende Lösungsweg $\pi(s,q^*)$ lässt sich also verkürzen und verläuft dann nicht mehr über p. Dies steht im Widerspruch zur Definition von p, d.h. obige Annahme ist falsch. ◀

Wir nehmen zunächst an, dass p kein Endknoten des Kantenzugs ist, also seien e_j und e_{j+1} die beiden angrenzenden Kanten. Wir legen eine zu $\bar{q}$ orthogonale Gerade g durch p (und damit auch q^*).

▶ **Theorem 5.** *Die Kanten e_j und e_{j+1} befinden sich auf derselben Seite von g (siehe Abbildung 1 und 2)*

Beweis. Zunächst weitere Veranschaulichungen unter der Annahme, dass das Theorem stimmt. Mit anderen Worten: g berührt den Trichter in Knoten p. Da p Knoten von D ist, berührt g in p also nicht nur den Trichter, sondern sogar den Rand des Polygons D. Dieser befindet sich auf derselben Seite wie e_j und e_{j+1} (sonst wäre der "Knick" im Trichter ja gar nicht erst nötig gewesen). Sei $p' \neq p$ ein Punkt auf der Kante zwischen p und $\text{pred}(p)$, der

beliebig nah an p liegen darf. Blickt man von p' aus orthogonal Richtung $\bar{q}$, so hat man daher keine freie Sicht auf $\bar{q}$, die entsprechende Strecke verläuft teilweise *außerhalb* von D. Genau deshalb ist p der *erste* Punkt mit dieser Eigenschaft.

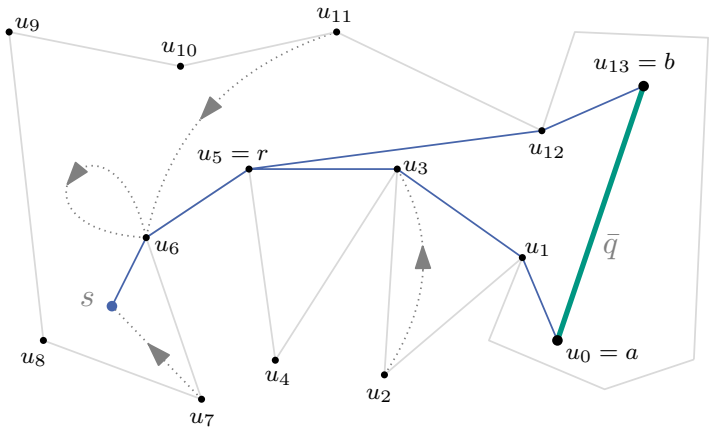
Nehmen wir nun also an, das Theorem sei falsch, d.h. g *schneidet* den Trichter in Punkt p. Bewegt man sich erneut von p aus Richtung $\text{pred}(p)$, so schiebt sich nicht der Rand des Trichters bzw. Polygons ins Sichtfeld. Auch von dort verläuft eine orthogonale Projektion Richtung $\bar{q}$ vollständig innerhalb des Trichters und damit D. Damit ist p *nicht* der erste Punkt, mit dieser Eigenschaft, im Widerspruch zur Definition. ◀

Außerdem sind e_j und e_{j+1} die *einzigen* konsekutiven Kanten, mit dieser Eigenschaft, was aus Theorem 1 und 2 folgt. Dank dieser Eindeutigkeit lässt sich also umgekehrt p anhand von e_j und e_{j+1} bestimmen. Sei g nun auch ein Richtungsvektor längs zu Gerade g, also orthogonal zu $\bar{q}$. Die folgenden Aussagen sind dann äquivalent zu Theorem 5:

$$\text{entweder } e_j \times_{\text{ccw}} g \text{ oder } e_{j+1} \times_{\text{ccw}} g \tag{4}$$

$$\text{entweder } \langle e_j, \bar{q} \rangle \leq 0 \text{ oder } \langle e_{j+1}, \bar{q} \rangle \leq 0 \tag{5}$$

d.h. die Skalarprodukte der beiden Kanten mit $\bar{q}$ haben unterschiedliches Vorzeichen. Aufgrund von Theorem 2 findet in der Folge der Skalarprodukte $\langle e_1, \bar{q} \rangle, \langle e_2, \bar{q} \rangle, ..., \langle e_t, \bar{q} \rangle$ auch nur (höchstens) ein einziger Vorzeichenwechsel statt. Gibt es keinen Vorzeichenwechsel, liegt der Spezialfall vor, dass p mit a oder b zusammenfällt. Ansonsten lässt sich mittels binärer Suche nach diesem Vorzeichenwechsel und damit auch p suchen.



$\mathsf{lca}(a, u_2) = u_3 \in \pi(s, a)$ $\quad$ $\mathsf{lca}(b, u_7) = s \in \pi(s, b)$
$\mathsf{lca}(b, u_6) = u_6 \in \pi(s, b)$ $\quad$ $\mathsf{lca}(b, u_{11}) = u_6 \in \pi(s, b)$

Abbildung 5 Approximation des Kantenzugs durch Intervall U der Polygon-Knoten $\vec{D}$. Exemplarisch sind auch vier Abbildungen von U zurück auf $\pi(s, a) \cup \pi(s, b)$ dargestellt (gepunktet). Hier gilt $s_1 = 4$ und $s_2 = 11$.

Zur Implementierung dieser binären Suche müssten allerdings die Kanten E berechnet werden, wovon es $O(n)$ gibt, was auch lineare Laufzeit erfordert. Statt den Kantenzug $\pi(a, r)$, $\pi(r, b)$ also explizit zu berechnen, aus dem E hervorgeht, wollen wir diesen approximieren. Wie in Abbildung 5 gezeigt, wählen wir hierzu den Intervall U von Polygonknoten $\vec{D}$, beginnend bei $\text{pred}(a)$ bis einschließlich $\text{pred}(b)$. Die Knoten $\text{pred}(a)$ und $\text{pred}(b)$ lassen sich dank

SPM in logarithmischer Laufzeit berechnen, U selbst wird lediglich durch zwei Zeiger in $\vec{D}$ implementiert, benötigt also nur konstanten Aufwand. Die Knoten von U seien u_1 bis u_{s-1} (demnach ebenfalls im Uhrzeigersinn). Wir definieren zusätzlich $u_0 := a$ und $u_s := b$. Es fällt auf, dass r im Intervall U liegt, es gelte daher ab sofort $r = u_r$, d.h. r bezeichne auch den Index, mit dem Knoten r unter den $u_0, ..., u_s$ zu finden ist.

Es wird nun gezeigt, wie man in konstanter Zeit ein u_i auf eine Kante e_j abbilden kann. Außerdem ist diese Abbildung f monoton, d.h.

$$\forall i_1, i_2 \in 0, ..., s, i_1 \leq i_2 : f(u_{i_1}) = e_{j_1} \wedge f(u_{i_2}) = e_{j_2} \Rightarrow j_1 \leq j_2 \tag{6}$$

Dies macht binäre Suche auf $u_0, ..., u_s$ möglich!

Beobachtungen: Betrachtet man ein u_i mit $0 \leq i \leq r$, so ist $\text{lca}(a, u_i) \in \pi(s, a)$ (siehe Abbildung 5). Analog gilt für ein u_i mit $r \leq i \leq s$, dass $\text{lca}(b, u_i) \in \pi(s, b)$. Mittels lca lässt sich ein u_i also bereits auf einen Punkt $u_i' \in \pi(s, a) \cup \pi(s, b)$ abbilden, der sich mittels pred problemlos zu einer Kante erweitern lässt. Derart ermittelte Kanten sind also entweder in E oder befinden sich auf $\pi(s, r)$ (nämlich genau dann wenn $\text{lca}(\text{pred}(u_i'), r) \neq r$, was mittels SPT in $O(1)$ erkennbar ist).

Den Intervall $u_{s_1}, ..., u_{s_2}$, der nicht auf E sondern auf $\pi(s, r)$ abbildet, kann man mittels binärer Suche in $O(\log n)$ vorbestimmen. Der Knoten r bzw. u_r befindet sich in diesem Intervall (wegen $\text{lca}(\text{pred}(r), r) = \text{pred}(r) \neq r$), d.h. es gilt $s_1 \leq r \leq s_2$. Punkte u_i aus besagtem Intervall kann man z.B. gleich behandeln wie einen angrenzenden Punkt (u_{s_1-1} oder u_{s_2+1}), sodass sich die Abbildung auf diesem Intervall konstant verhält. Insgesamt ergibt sich folgende Abbildung:

$$f : u_0, ..., u_s \to E \tag{7}$$

$$u_i \mapsto \begin{cases} (\text{pred}(\text{lca}_a), \text{lca}_a) & \text{für } 0 \leq i < s_1 \\ (\text{pred}(\text{lca}_{mid}), \text{lca}_{mid}) & \text{für } s_1 \leq i \leq s_2 \\ (\text{pred}(\text{lca}_b), \text{lca}_b) & \text{für } s_2 < i \leq s \end{cases} \tag{8}$$

wobei

$$\text{lca}_a = \text{lca}(a, u_i) \tag{9}$$

$$\text{lca}_b = \text{lca}(b, u_i) \tag{10}$$

$$\text{lca}_{mid} = \text{lca}(b, u_{s_2+1}) \tag{11}$$

▶ **Theorem 6.** *Die Abbildung f ist monoton.*

Beweis. Wähle ein u_i mit $0 \leq i < s_1$. Nehmen wir an u_i liegt auf dem Trichter (in Abbildung 5 ist das für u_0, u_1 und u_3 der Fall, beachte dass u_5 bereits nicht mehr $0 \leq i < s_1$ erfüllt). Dann ist $\text{lca}_a = \text{lca}(a, u_i) = u_i$, d.h. f bildet u_i direkt auf $\text{pred}(u_i)$ ab. Lägen alle u_i auf dem Trichter wäre f zumindest für $0 \leq i < s_1$ offensichtlich monoton.

Nehmen wir nun an, u_i läge nicht auf dem Trichter (in Abbildung 5 ist das für u_2 der Fall, u_4 erfüllt nicht $0 \leq i < s_1$). Dann ist $\text{lca}_a = \text{lca}(a, u_i) = u_{i'}$ wobei $u_{i'}$ der nächste Knoten in der Folge $u_{i+1}, u_{i+2}, ...$ ist, der sich auf dem Trichter befindet (in Abbildung 5 ist wie eingezeichnet Knoten $\text{lca}(a, u_2) = u_3$).

Die Abbildung f behandelt also alle Knoten, die nicht auf dem Trichter liegen genauso, wie der nächstgrößere Knoten, der sich dort befindet. Damit ist f stückweise konstant, aber vor allem weiterhin monoton.

Für $s_2 < i \leq s$ verläuft der Beweis analog. Auf $s_1 \leq i \leq s_2$ ist f ohnehin konstant (und per Konstruktion monoton). ◀

10.3 Quickest Visibility Queries

Bei Quickest Visibility Queries (kurz: QVQ) geht es darum, von einem Startpunkt s aus möglichst schnell einen Punkt q zu *sehen*.

Es gelte dieselbe Notation wie schon in Abschnitt 10.2.1 vereinbart. Zudem sei $V(d) \subseteq D$ der von einem beliebigen Punkt $d \in D$ aus sichtbare Bereich.

10.3.1 Problemstellung

Da Sichtbarkeit eine symmetrische Relation ist, gilt

$$\forall d_1, d_2 \in D : d_2 \in V(q_1) \Leftrightarrow d_1 \in V(q_2) \tag{12}$$

Das Ziel von QVQs ist es also, den kürzesten Weg von s zu einem beliebigen Punkt t^* zu finden, von dem aus q sichtbar ist, also $q \in V(t^*)$ gilt. Aufgrund der Symmetrie ist dies genau dann der Fall, wenn auch $t^* \in V(q)$ gilt, was eine wesentlich knappere Formulierung zulässt: Das Ziel von QVQs ist die Bestimmung des kürzesten Weges von s zu $V(q)$.

Der gesuchte kürzeste Weg sei mit $\tau(s \in D, q \in D)$ bezeichnet und kann anhand einer SPM bezüglich s rekonstruiert werden, sobald der Punkt t^* bekannt ist. Formal besteht folgender Zusammenhang:

$$\tau(s, q) = \pi(s, t^*) \tag{13}$$

$$t^* = \arg\min_{t \in V(q)} |\pi(s, t)| \tag{14}$$

Wir suchen im Folgenden daher nur nach t^*, der Pfad ist dann bei Bedarf mittels SPM rekonstruierbar.

Wie schon bei Shortest Path to a Segment Queries (Abschnitt 10.2) seien Polygonfläche D und Punkt s bereits vorher bekannt und dürfen für Vorberechnungen genutzt werden. Die zu beantwortende Anfrage besteht bei QVQ offensichtlich aus dem zu sehenden Punkt q.

10.3.2 Ergebnisse und Einordnung

Ohne Vorberechnungen ($\mathsf{T}_v = 0$) kann auch ein QVQ mit Hilfe des kontinuierlichen Dijkstra Verfahrens in $\mathsf{T}_a = O(n \log n)$ Laufzeit beantwortet werden [14]. Die Wellenfront wird $V(q)$ als erstes im Punkt t^* erreichen.

Den Autoren ist dagegen noch keinerlei QVQ-Verfahren bekannt, bei dem Vorberechnungen angestellt werden. Hierfür werden zwei Ansätze vorgestellt: Zum einen können SPSQ verwendet werden, um kürzeste Wege zu $V(q)$ zu finden. In Einklang zu den beiden in 10.2.3 vorgestellten Laufzeiten von SPSQ erreicht man dann Laufzeiten $\mathsf{T}_v = O(n^2 \log^2 n)$ und $\mathsf{T}_a = O(K \log^2 n)$ oder $\mathsf{T}_v = O(n^3 \log n)$ und $\mathsf{T}_a = O(K \log n)$ (K ist dabei die Anzahl der Kanten von $V(q)$). Zum anderen kann eine so genannte Quickest Visibility Map (kurz: QVM) vorberechnet werden, welche die Idee der Raumpartitionierung auf Sichtbarkeits-Analyse anwendet. Dieses Verfahren erreicht $\mathsf{T}_v = O(n^8 \log n)$ und $\mathsf{T}_a = O(\log n)$. Beide Ansätze werden in den folgenden Abschnitten vorgestellt.

Der Spezialfall, dass D einfach ist ($h = 0$) wurde von Khosravi und Ghodsi [13] untersucht, die einen Algorithmus mit $\mathsf{T}_v = O(n^2)$ und $\mathsf{T}_a = O(\log n)$ vorstellen. Das Problem kann allerdings auch mit Hilfe des in Abschnitt 10.2.4 vorgestellten SPSQ-Algorithmus gelöst werden und erbt dessen optimale Laufzeit von $\mathsf{T}_v = O(n)$ und $\mathsf{T}_a = O(\log n)$.

10.3.3 Berechnung von $V(q)$

Den sichtbaren Bereich $V(q)$ eines Knotens q in einer Polygonfläche zu finden, ist ein gut untersuchtes Problem. Optimale Verfahren ohne Vorberechnung sind bekannt, Heffernan und Mitchell [9] erreichen eine Laufzeit von $O(n + h \log h)$. Für einfache Polygone erreichten Joe und Simpson [12] bereits 1987 lineare Laufzeit.

Es existieren auch zahlreiche Verfahren die Vorberechnungen anstellen. In beliebigen Polygonen erreichen Zarei und Ghodsi [17] $\mathsf{T}_v = O(n^3 \log n)$ und $\mathsf{T}_a = O(K + \min(h, K) \log n)$. Inkulu und Kapoor [11] erreichen durch Kombination mehrerer Verfahren $\mathsf{T}_v = O(n^2 \log n)$ und $\mathsf{T}_a = O(K \log^2 n)$. Für einfache Polygone erreichen Guibas, Motwani und Raghavan [7] sowie Bose, Lubiw und Munro [3] $\mathsf{T}_v = O(n^3 \log n)$ und $\mathsf{T}_a = O(\log n + K)$, Aronov, Guibas, Teichmann und Zhang [2] erreichen $\mathsf{T}_v = O(n^2 \log n)$ und $\mathsf{T}_a = O(\log^2 n + K)$.

10.3.4 Verfahren mittels SPSQ

Es fällt auf, dass manche QVQ-Anfragen triviale Lösungen haben: Besteht Sichtkontakt zwischen Startpunkt s und dem Anfragepunkt q ($s \in V(q)$), so muss man sich von s aus überhaupt nicht bewegen, um q zu sehen. Die Lösung ist dann offensichtlich $t^* = s$.

Der anspruchsvollere Fall liegt also vor, wenn $s \notin V(q)$.

▶ **Theorem 7.** *Ist $s \notin V(q)$, so liegt t^* auf dem Rand von $V(q)$.*

Beweis. Angenommen t^* liegt im Inneren von $V(q)$. Da $s \notin V(q)$, verläuft ein gewisser Teil des Weges von s nach t^* also durch $V(q)$, t'^* sei der Schnittpunkt des Weges mit dem Rand von $V(q)$. Im Widerspruch zur Definition von t^* wäre $\pi(s, t'^*)$ ein kürzerer Weg von s zu $V(q)$. ◀

Der Rand von $V(q)$ bestehe aus K Kanten. Der kürzeste Weg zum Rand von $V(q)$ ist das Minimum der kürzesten Wege zu den K Kanten. Dieser kann also durch K SPSQs zu den Kanten und anschließendes Wählen der kürzesten Lösung ermittelt werden (siehe Abbildung 6).

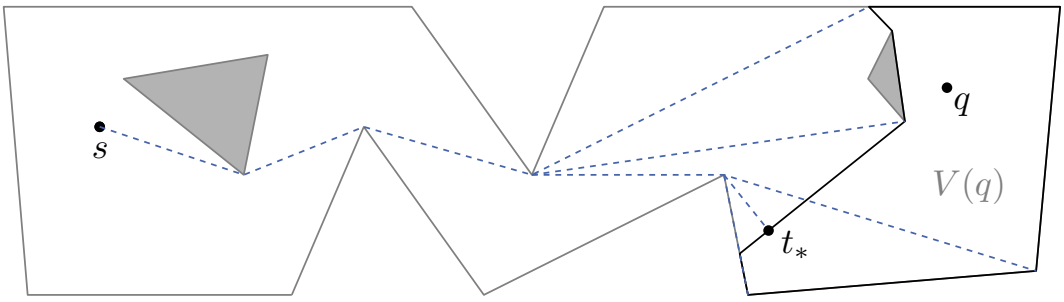


Abbildung 6 Dargestellt ist beispielhaft ein Anfragepunkt q mit dessen sichtbarem Bereich $V(q)$, welcher von $K = 7$ Kanten begrenzt ist. Zu all diesen Kanten werden SPSQs berechnet (Lösungspfade gestrichelt, fallen teilweise aufeinander). Der kürzeste dieser Pfade wurde ermittelt und dessen Endpunkt zur Lösung t^* deklariert.

Benötigt werden für dieses Verfahren eine SPM für die Fallunterscheidung ($s \in V(q) \Leftrightarrow \text{pred}(q) = s$), eine Datenstruktur zur schnellen Bestimmung von $V(q)$, sowie ein SPSQ-Algorithmus. Für die Berechnung von pred in logarithmischer Laufzeit wird eine SPM

verwendet, welche in Laufzeit $O(n \log n)$ erstellt werden kann (siehe [10]). Zur Beantwortung der SPSQ kann zum Beispiel eine der beiden in 10.2.3 erwähnten Datenstrukturen verwendet werden. Diese müssen natürlich nur einmal aufgebaut werden, werden je QVQ-Anfrage allerdings K mal verwendet. Das verwendete SPSQ-Verfahren dominiert Vorberechnungs- und Anfragezeit, sofern ein geeignetes Verfahren zur Bestimmung von $V(q)$ (siehe 10.3.3) gewählt wird. Entsprechend sind Laufzeiten von $\mathsf{T}_v = O(n^2 \log^2 n)$ und $\mathsf{T}_a = O(K \log^2 n)$ oder $\mathsf{T}_v = O(n^3 \log n)$ und $\mathsf{T}_a = O(K \log n)$ möglich.

10.3.5 Verfahren mittels QVM

Ein gänzlich anderer Ansatz basiert auf Raumpartitionierung, d.h. das Polygon D soll in Zellen mit ähnlicher Lösung eingeteilt werden. Ähnlichkeit bedeutet, dass die Lösungen kombinatorisch gleich sind, also bis auf wenige, direkt aus dem Anfragepunkt q ermittelbaren Details komplett identisch sind. Die zur Rekonstruktion der Lösung benötigten Informationen werden zu jeder Zelle gespeichert. Zur Beantwortung einer QVQ mittels QVM muss also nur die Zelle ermittelt werden, welche q enthält und t^* kann mit Hilfe von mit der Zelle gespeicherten Informationen schnell bestimmt werden.

Es stellt sich heraus, dass sich zwei charakteristische Punkte beim Verschieben des Anfragepunkts q nur manchmal sprunghaft ändern, ansonsten aber unverändert bleiben. Diese beiden Punkte sind (siehe Abbildung 7 für ein Beispiel):

$$g(q) := \text{pred}(t^*) \tag{15}$$

$$w(q) := \text{Knoten von } D \text{ auf Strecke von } t^* \text{ nach } q \text{ (oder } s\text{, falls } t^* = s) \tag{16}$$

wobei

$$g, w : D \to (\vec{D} \cup s) \tag{17}$$

Sind wiederum nur q, $g = g(q)$ und $w = w(q)$ (aber gerade nicht t^*) gegeben, so lässt sich t^* leicht explizit bestimmen:

$$t^* = t^*(g, w, q) := \arg\min_{t \in \text{line}(w,q)} |\text{dist}(g, t)| \tag{18}$$

wobei dist die euklidische Distanz zwischen zwei Punkten berechnet (also vollkommen unabhängig von D). Die Funktionen g und w teilen D in Äquivalenzklassen ein, die gerade die Zellen Z der QVM sind (siehe gestrichelte Linien in Abbildung 7).

Die mit einer Zelle $z_{g,w} \in Z$ assoziierten Werte g und w werden im Folgenden als "(Lösungs-)Strategie" $(g, w) \in (\vec{D} \cup s)^2$ bezeichnet, da sich mit deren Hilfe (wie gezeigt) die Lösung t^* konstruieren lässt. Es ist erwähnenswert, dass durchaus mehrere Lösungsstrategien existieren können, anhand derer ein Anfragepunkt q sichtbar wird. Diese müssen aber nicht unbedingt optimal (bzgl. zurückgelegter Strecke) sein. Optimal ist für jeden Punkt $q \in D$ (per Definition von g und w) stets die Strategie $(g(q), w(q))$.

Abbildung 8 zeigt einen Bereich, der zwar gänzlich durch zwei verschiedene Strategien überblickt wird, aber dennoch in unterschiedliche Äquivalenzklassen zerfällt.

Zum Vergleich zweier Lösungsstrategien zu gegebenem Punkt $q \in D$ wird die Relation $\preceq_q$ eingeführt (gesprochen "besser als"):

▶ **Definition 8.** $(g_1, w_1) \preceq_q (g_2, w_2) :\Leftrightarrow |\pi(s, t^*(g_1, w_1, q))| \leq |\pi(s, t^*(g_2, w_2, q))|$

Per Definition der Funktionen g und w als zur optimalen Strategie gehörend, gilt dann

$$\forall q \in D, (g', w') \in (\vec{D} \cup s)^2 : (g(q), w(q)) \preceq_q (g', w') \tag{19}$$

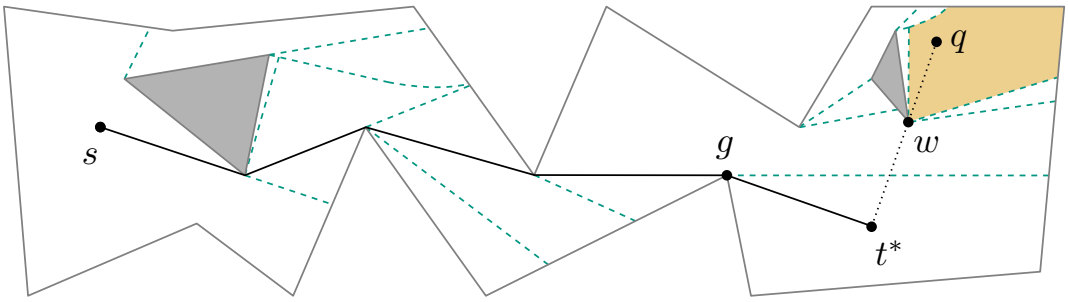


Abbildung 7 Einteilung von D in Äquivalenzklassen Z (getrennt von gestrichelten Linien) anhand von g und w. Zur Zelle rechts oben sind die Punkte g und w eingezeichnet. Zudem ist ein exemplarischer Anfragepunkt q mit Lösungsweg $\pi(s, t^*)$ dargestellt. Man kann sich leicht vorstellen, wie $g = g(q)$ und $w = w(q)$ unverändert bleiben, wenn q innerhalb der Zelle verschoben wird.

Es bleibt zu klären, wie die QVM-Zellen Z bestimmt werden können. Der Lösungsansatz sieht vor, alle $O(n^2)$ Strategien $(g, w) \in (\vec{D} \cup s)^2$ einzeln zu betrachten und jeweils die zugehörige Zelle

$$z_{g,w} = \{\ q \in D \mid g = g(q) \wedge w = w(q)\ \} \tag{20}$$

$$= \{\ q \in D \mid \forall (g', w') \in (\vec{D} \cup s)^2 : (g, w) \preceq_q (g', w')\ \} \tag{21}$$

zu bestimmen. Leider ist diese Menge nur sehr schwer zu bestimmen. Zwar ist es kein Problem, einen Punkt q auf Zugehörigkeit zur Menge $z_{g,w}$ zu testen (das wäre ein einfacher QVQ). Benötigt werden zum Zwecke einer QVM aber scharfe Grenzen zwischen den Zellen von Z. Wir begnügen uns daher zunächst mit einer (leichter berechenbaren) konservativen Approximation an die Zellen:

$$z'_{g,w} = \{\ q \in D \mid \forall g' \in (\vec{D} \cup s) : (g, w) \preceq_q (g', w)\ \} \tag{22}$$

Der entscheidende Unterschied ist, dass die Strategie der Zelle nur noch unter allen Strategien mit gleichem w optimal sein muss, damit ein $q \in D$ zur Zelle zählt. Aufgrund dieser Lockerung gilt

$$\forall (g, w) \in (\vec{D} \cup s)^2 : z_{g,w} \subseteq z'_{g,w} \tag{23}$$

Zu festem w sind alle $z'_{g,w}$ disjunkte Teilmengen von $V(w)$, welche sich leicht konstruieren lassen. Es bleibt nun Z aus Z' abzuleiten.

Knoten der tatsächlichen QVM-Zellen Z (“QVM-Knoten”) können an drei verschiedenen Stellen auftreten. Zum einen *innerhalb* von Zellen von Z', nämlich an so genannten Tripelpunkten, wo (mindestens) drei Lösungsstrategien gleich gut sind (wenn zum Beispiel die Trennlinie aus Abbildung 8 auf eine weitere träfe). Da es $O(n^2)$ Lösungsstrategien gibt, kann es nicht mehr als $O(n^6)$ solcher Tripelpunkte geben. Zum anderen können Knoten von Z' (Schnittpunkte von Kanten in Z') auch selbst QVM-Knoten sein. Da Z' aus $O(n^2)$ Zellen mit jeweils $O(n)$ Kanten besteht, enthält Z' $O(n^3)$ Kanten. Diese können sich in $O(n^6)$ Punkten schneiden. Zu guter Letzt können QVM-Knoten auf Kanten von Z' fallen, nämlich an Stellen wo zwei Lösungsstrategien gleich gut sind (das ist in Abbildung 8 der Fall, wo die Trennlinie ganz oben auf den Rand von D trifft, der trivialerweise auch eine Kante von Z' ist). Da es

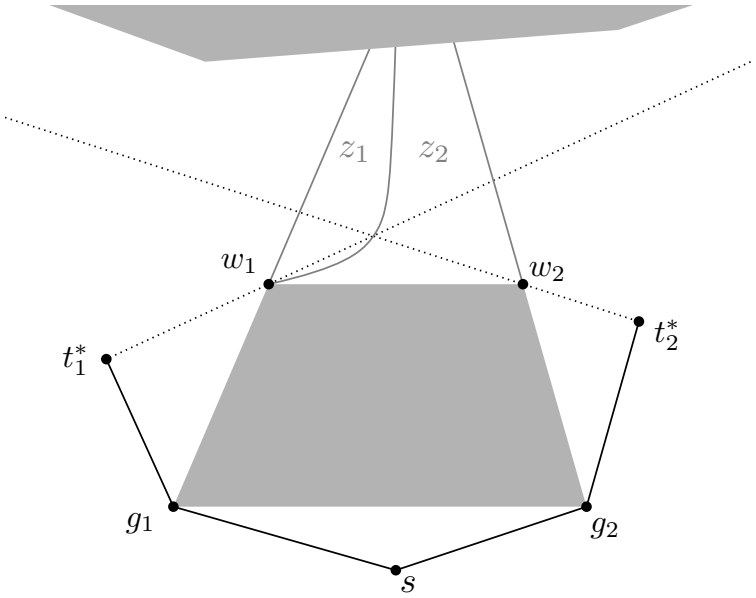


Abbildung 8 Eingezeichnet sind zwei QVM-Zellen z_1 und z_2 mit zugehörigen Strategien (q_1, w_1) und (q_2, w_2). Graue Bereiche liegen außerhalb von D. Die beiden eingezeichneten Pfade $\pi(s, t_1^*)$ und $\pi(s, t_2^*)$ sind gleich lang. Offenbar werden alle Punkte aus $z_1 \cup z_2$ mittels beider Strategien früher oder später sichtbar (Sichthorizont durch gepunktete Linien angedeutet). Die Aufteilung von $z_1 \cup z_2$ auf Zellen z_1 und z_2 liegt lediglich daran, dass die zugehörige Strategie in diesem Bereich besser ist, Punkte darin also früher zu sehen sind. Im Punkt wo sich die beiden Sichthorizonte schneiden sind offenbar beide Strategien gleich gut. Die Trennlinie zwischen z_1 und z_2 ist gerade die Menge all dieser Schnittpunkte.

$O(n^3)$ Kanten und $O(n^2)$ Lösungsstrategien gibt, gibt es $O(n^2 \cdot n^2 \cdot n^3) = O(n^7)$ solcher Punkte.

Insgesamt ergeben sich $O(n^7)$ potentielle QVM-Knoten. Dieser Term dominiert auch den Speicherbedarf der QVM. Mittels QVQs werden die potentiellen Knoten in jeweils $O(n \log n)$ Laufzeit überprüft, was zur Vorberechnungslaufzeit $\mathsf{T}_v = O(n^8 \log n)$ führt. Die erfolgreich geprüften Knoten sind dann Teil von Z, d.h. dort treffen sich tatsächlich Trennlinien zwischen QVM-Zellen. Zusammen mit den (bereits ermittelten) Trennlinien für alle Paare von Lösungsstrategien lassen sich die Knoten von Z verbinden, was zur finalen Unterteilung führt.

10.4 Fazit

Der wissenschaftliche Artikel stellt zahlreiche neue Verfahren zum Lösen von SPSQ und QVQ vor. Die Vielfalt der Verfahren und Laufzeiten erlaubt es auch abzuwägen, welches sich für einen Anwendungsfall am besten eignet.

Die Zusammenhänge zwischen SPSQ und QVQ werden ebenfalls untersucht und ausgenutzt. So kann QVQ in einem sehr einfachen Verfahren mittels SPSQ gelöst werden, die Laufzeit T_a für Anfragen hängt allerdings von der Komplexität K von $V(q)$ ab. Das daraufhin vorgestellte Verfahren mittels QVM verbessert diese Laufzeit, benötigt dafür aber wesentlich längere Vorberechnungszeit. Es basiert wie SPMs auf Raumunterteilung, was eine Laufzeit von $\mathsf{T}_a = O(\log n)$ ermöglicht.

Zusätzlich wird der Spezialfall des einfachen Polygons betrachtet, für welchen neue Algorithmen mit optimaler Laufzeit angegeben werden. Für $h = 0$ kann nämlich sowohl SPSQ als auch QVQ nach linearer Vorberechnungszeit in logarithmischer Laufzeit beantwortet werden.

Referenzen

1 Esther M Arkin, Alon Efrat, Christian Knauer, Joseph SB Mitchell, Valentin Polishchuk, Günter Rote, Lena Schlipf, and Topi Talvitie. Shortest path to a segment and quickest visibility queries. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 34. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

2 Boris Aronov, Leonidas J Guibas, Marek Teichmann, and Li Zhang. Visibility queries and maintenance in simple polygons. *Discrete & Computational Geometry*, 27(4):461–483, 2002.

3 Prosenjit Bose, Anna Lubiw, and J Ian Munro. Efficient visibility queries in simple polygons. *Computational Geometry*, 23(3):313–335, 2002.

4 Svante Carlsson, Håkan Jonsson, and Bengt J Nilsson. Finding the shortest watchman route in a simple polygon. *Discrete & Computational Geometry*, 22(3):377–402, 1999.

5 Moshe Dror, Alon Efrat, Anna Lubiw, and Joseph SB Mitchell. Touring a sequence of polygons. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 473–482. ACM, 2003.

6 Adrian Dumitrescu and Csaba D Tóth. Watchman tours for polygons with holes. *Computational Geometry*, 45(7):326–333, 2012.

7 Leonidas J Guibas, Rajeev Motwani, and Prabhakar Raghavan. The robot localization problem. *SIAM Journal on Computing*, 26(4):1120–1138, 1997.

8 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.

9 Paul J Heffernan and Joseph SB Mitchell. An optimal algorithm for computing visibility in the plane. *SIAM Journal on Computing*, 24(1):184–201, 1995.

10 John Hershberger and Jack Snoeyink. Computing minimum length paths of a given homotopy class. *Computational geometry*, 4(2):63–97, 1994.

11 Rajasekhar Inkulu and Sanjiv Kapoor. Visibility queries in a polygonal region. *Computational Geometry*, 42(9):852–864, 2009.

12 Barry Joe and Richard B Simpson. Corrections to lee's visibility polygon algorithm. *BIT Numerical Mathematics*, 27(4):458–473, 1987.

13 Ramtin Khosravi and Mohammad Ghodsi. The fastest way to view a query point in simple polygons. In *EuroCG*, pages 187–190, 2005.

14 Joseph SB Mitchell. L 1 shortest paths among polygonal obstacles in the plane. *Algorithmica*, 8(1-6):55–88, 1992.

15 Joseph SB Mitchell. Approximating watchman routes. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 844–855. SIAM, 2013.

16 Eli Packer. Computing multiple watchman routes. In *Experimental Algorithms*, pages 114–128. Springer, 2008.

17 Alireza Zarei and Mohammad Ghodsi. Query point visibility computation in polygons with holes. *Computational Geometry*, 39(2):78–90, 2008.