Praktikumsbericht

Geometrische Cluster

Jakob Geiger

Abgabedatum:	24. September 2020
Betreuer:	Prof. Dr. Alexander Wolff
	Dr. Philipp Kindermann

Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen, Komplexität und wissensbasierte Systeme

Inhaltsverzeichnis

1 Einführung

Im Kontext geographischer Analysen ist es häufig sinnvoll, große Datenmengen zur besseren Visualisierung zu vereinfachen, indem ähnliche Objekte zu einer übergeordneten Einheit zusammengefasst werden. Dies erleichtert es einem Betrachter, überspannende Zusammenhänge leicht zu erfassen, da die Komplexität der Darstellung stark reduziert wird. Dabei gibt es mehrere Kriterien, die berücksichtigt werden können. Besonders wichtig ist in der Geographie natürlich die räumliche Distanz, aber auch andere, anwendungsspezifische Kriterien, die angeben, wie ähnlich sich zwei Datenpunkte sind, können nicht außer Acht gelassen werden. Die Beispiele in dieser Arbeit beziehen sich hauptsächlich auf abstrahierte Stadtkarten, in der die Objekte verschiedene Orte von Interesse repräsentieren, beispielsweise Geschäfte oder Restaurants. In diesem Zusammenhang erscheint die Zugehörigkeit zur gleichen Klasse wie eine sinnvolle Metrik für die Ähnlichkeit zweier Datenpunkte. In anderen Anwendungen könnten zum Beispiel Vögel ihrer Spezies nach in Gruppen eingeteilt werden.
Ein erster Ansatz zum Clustern von Datenpunkten ist, die Delaunay-Triangulierung des Graphen zu berechnen, alle Kanten zwischen Knoten verschiedener Kategorien zu löschen, und jede übrige Zusammenhangskomponente als ein eigenes Cluster zu betrachten. Diese Herangehensweise liefert zwar ein eindeutiges Ergebnis, weist jedoch auch einige Schwächen auf. Die entstehenden Cluster können beispielsweise verhältnismäßig dünn ausfallen. Aus diesem Grund wird in dieser Arbeit eine andere Methode betrachtet.
Wir beginnen mit deutlich dichteren Nachbarschaftsgraphen. In dieser Arbeit wurden zum Generieren dieser die sogenannten *β-Skelette* (siehe 2.1) verwendet. β-Skelette sind im allgemeinen nicht planar, daher ist es nötig, den Umgang mit Kantenkreuzungen festzulegen. Um zu vermeiden, dass die Ausgabe Cluster enthält, die sich überlappen, fordern wir, dass im Resultat keine Kantenkreuzungen vorhanden sein dürfen. Daraus leitet sich folgende Problemdefinition ab:

Cluster-Minimierung Sei $G = (V, E)$ ein nicht notwendigerweise planarer, geometrischer Graph. Finde einen Subgraphen $H = (V, E')$ von G mit $E' \subseteq E$, so dass H jeden Knoten aus V enthält, sich keine zwei Kanten in E' kreuzen und die Anzahl der Zusammenhangskomponenten in H minimiert wird.

Man beachte, dass in dieser Definition die Zugehörigkeit der Knoten zu Kategorien nicht gefordert ist. Die Bedingung, dass keine Datenpunkte aus verschiedenen Kategorien in einem gemeinsamen Cluster liegen können, ist eine Spezialisierung und kann dadurch erreicht werden, dass mögliche Kanten zwischen solchen Punkten einfach gelöscht werden. Durch die Verwendung von Kategorien, die wir im Folgenden als *Farben* bezeichnen, ergeben sich andererseits weitere interessante Fragestellungen.
In einem geometrischen Graphen G, in dem jedem Knoten genau eine von k Farben zugewiesen wird, nennen wir eine Kante *gefärbt*, wenn beide ihrer Endpunkte die gleiche Farbe besitzen. Die Farbe dieser Kante ist gleich der Farbe ihrer Endpunkte. Kanten, bei denen die Farben der Endpunkte nicht übereinstimmen, nennen wir *ungefärbt*. Kreuzungen zwischen zwei Kanten nennen wir *einfarbig*, wenn beide Kanten gefärbt sind und

die gleiche Farbe haben. Andere Kreuzungen bezeichnen wir als *zweifarbig*. Ein geometrischer Graph ist *1-eben*, wenn jede seiner Kanten von höchstens einer anderen Kante gekreuzt wird. In 1-ebenen Graphen können wir die Bedingung, die Kantenkreuzungen verbietet, relaxieren, indem einfarbige Kreuzungen *planarisiert* werden. Unter Planarisierung verstehen wir die Ersetzung einer Kreuzung zwischen zwei Kanten $\{u, v\}, \{p, q\}$ durch ihren Schnittpunkt s sowie die Kanten $\{u, s\}, \{v, s\}, \{p, s\}$ und $\{q, s\}$. Es ergibt sich das folgende Problem:

Cluster-Minimierung in 1-ebenen Graphen Sei $G = (V, E)$ ein 1-ebener Graph, in dem jedem Knoten genau eine von k Farben zugewiesen ist. Löse Cluster-Minimierung in demjenigen Graphen, der sich ergibt, wenn in G einfarbige Kreuzungen planarisiert werden.

Ein 1-ebener Graph könnte sich zum Beispiel ergeben, indem man zu einer gegebenen Punktmenge die Vereinigung aller Delaunay-Triangulierungen der Ordnung 1 als Kantenmenge verwendet. Dieser Ansatz bietet eine höhere Kantendichte als die normale Delaunay-Triangulierung, bildet aber dennoch gut ab, ob zwei Punkte in gegenseitiger Nähe liegen. In dieser Arbeit wurde er jedoch nicht experimentell betrachtet, da sich Cluster-Minimierung in 1-ebenen Graphen durch einen simplen Greedy-Algorithmus bereits optimal lösen lässt (siehe 3.2).

In dieser Arbeit wird zunächst ein einfacher Ansatz zur Lösung von Cluster-Minimierung in 1-ebenen Graphen gegeben. Im Anschluss werden drei verschiedene Varianten eines Greedy-Algorithmus für den allgemeinen Fall vorgestellt und experimentell miteinander verglichen.

2 Grundlagen

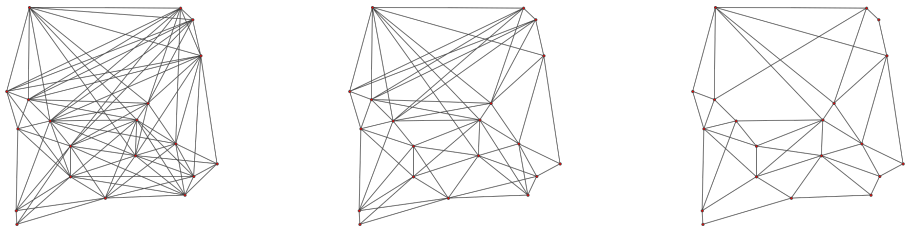
In diesem Abschnitt sollen einige Konzepte, die zum Verständnis der Arbeit relevant sind, kurz beschrieben werden.

2.1 β-Skelette

Ein β-Skelett ist ein ungerichteter Graph G, der durch eine Punktmenge V und einen Parameter $\beta \geq 0$ eindeutig definiert wird. Dabei bildet V die Knotenmenge von G, die Kantenmenge E wird abhängig von β wie folgt definiert. Zunächst wird mit der folgenden Formel ein Winkelmaß θ berechnet.

$$\theta = \begin{cases} \sin^{-1}(\frac{1}{\beta}) & \text{für } \beta \geq 1 \\ \pi - \sin^{-1}(\beta) & \text{für } \beta \leq 1 \end{cases}$$

Eine Kante $e = \{p, q\}$ liegt genau dann in E, wenn kein Punkt $r \in V$ existiert mit $\angle prq > \theta$. Anschaulich gesprochen wird durch den Winkel θ ein Bereich definiert, der im Fall $\beta \leq 1$ als Schnitt zweier Kreisscheiben verstanden werden kann, im Fall $\beta \geq$



(a) $\beta = 0,5$ (b) $\beta = 0,7$ (c) $\beta = 0,9$

Abb. 1: β-Skelette für verschiedene Werte von β

1 als Vereinigung dieser. Die Kante $e = \{p, q\}$ liegt genau dann in E, wenn der so definierte Bereich keine Punkte aus V enthält. Wie in Abbildung 1 zu sehen ist, nimmt mit steigendem β die Kantendichte im resultierenden Graphen ab. Aus der Definiton der Skelette folgt, dass dieses negative Wachstum sogar monoton erfolgt, bei steigendem β also keine vorher nicht vorhandenen Kanten entstehen können.

2.2 Union-Find

Für alle beschriebenen Algorithmen ist eine *Union-Find*-Datenstruktur zentral, daher soll diese hier kurz erklärt werden.
Grundsätzlich wird Union - Find benutzt, um eine disjunkte Partition einer Menge zu verwalten. Jeder Teilmenge wird dabei ein eindeutiges, in ihr enthaltenes kanonische Element zugewiesen, das den Namen dieser repräsentiert. Die Datenstruktur bietet zudem drei Funktionen: Union, Find, und Make-Set. Make-Set(x) erzeugt eine neue Teilmenge, die nur das Element x enthält, welches zugleich auch als kanonisches Element für diese fungiert. Find(x) liefert das kanonische Element derjenigen Menge, die x enthält, Union(x, y) wird benutzt, um zwei Mengen zu vereinigen, dabei wird x zum neuen Repräsentanten der entstandenen Menge.
Union-Find wurde, neben der offensichtlichen Kompatibilität mit den Algorithmen, auch wegen seiner sehr kompetitiven Laufzeiten gewählt. Wie bei Tarjan und van Leeuwen [TvL84] nachzulesen ist, kann eine Sequenz von m Find-Operationen und n Union-Operationen in $O(n + m \cdot \alpha(n))$ ausgeführt werden, wobei α die *Inverse Ackermann-funktion* bezeichnet. Diese ist für alle Eingabewerte $n < 10^{80}$, was nur wenige Größenordnungen unter der geschätzten Anzahl der Atome im beobachtbaren Universum liegt, kleiner als 5 und kann somit für alle praktischen Zwecke als konstant betrachtet werden. Die MakeSet-Operation kann trivialerweise in konstanter Zeit ausgeführt werden.

3 Algorithmen und Heuristiken

In diesem Abschnitt wenden wir uns konkreten Algorithmen für das Problem Cluster-Minimierung zu. Wie in der zu diesem Praktikum passenden Masterarbeit [Gei20] bewiesen wurde, ist das Problem im Allgemeinen NP-vollständig, die Existenz eines effizi-

enten Lösungsalgorithmus ist daher unwahrscheinlich. Wir betrachten also Heuristiken für Cluster-Minimierung. Dabei wird neben ihrer Motivation und Formulierung auch die Laufzeit der Heuristiken untersucht. Im Anschluss betrachten wir die Einschränkung von Cluster-Minimierung auf 1-ebene Graphen, dieses Problem ist nach Akitaya et al. [HAC+19] effizient lösbar.

3.1 Das allgemeine Problem

Aufgrund der NP-Schwere von Cluster-Minimierung [Gei20] ist die Existenz eines effizienten Lösungsalgorithmus unwahrscheinlich. In diesem Abschnitt werden daher einige Heuristiken für dieses Problem vorgestellt.

Greedy-Algorithmus Der natürlichste Ansatz zur Lösung dieses Problems ist sicherlich der Greedy-Algorithmus. Dabei wählen wir iterativ immer diejenige Kante, die die wenigsten Kreuzungen aufweist, und löschen alle Kanten, die diese Kante kreuzen. Dabei achten wir darauf, keine Kanten zu wählen, die innerhalb bereits bestehender Cluster verlaufen. Diese Überprüfung realisieren wir mittels einer Union-Find Datenstruktur.

Algorithmus 1 : Greedy Cluster Minimierung(Graph $G = (V, E)$)

Eingabe : Graph $G = (V, E)$
Ausgabe : $E' \subseteq E$, so dass E' keine Kreuzungen enthält und möglichst wenige Zusammenhangskomponenten in $G' = (V, E')$ verbleiben

```
1  E' ← ∅
2  foreach v ∈ V do MakeSet(v)
3  foreach e = (v, w) aus E ohne Kreuzungen do
4      Verschiebe e von E nach E'
5      Union(Find(v),Find(w))
6  while E ≠ ∅ do
7      Wähle beliebige Kante e = (v, w) mit minimaler Kreuzungszahl
8      if find(v) ≠ find(w) then
9          Verschiebe e von E nach E' und lösche alle Kanten aus E, die e kreuzen
10         Union(Find(v),Find(w))
11     else
12         Lösche e.
13 return E'
```

Laufzeit Die Laufzeit des Greedy-Algorithmus wird von drei Komponenten des Verfahrens dominiert. Bei diesen handelt es sich um die Berechnung der Kantenkreuzungen, die Auswahl der minimal gekreuzten Kanten, sowie die Verwendung von Union-Find.
Betrachten wir zunächst die Komplexität des ersten Bausteins, der Ermittlung aller Paare von gekreuzten Kanten. Balaban [Bal95] entwickelte 1995 ein Verfahren, mit dem zu

einer Menge von m Segmenten in der Ebene, die k Schnitte zwischen Kanten aufweist, mit einer Laufzeit von $O(k+m\log m)$ das gewünschte Resultat liefert. Wie Chazelle und Edelsbrunner [CE92] zeigten, ist diese Laufzeit aus komplexitätstheoretischer Sicht optimal. In der Praxis ist die Verwendung des asymptotisch langsameren *Bentley-Ottmann*-Algorithmus aufgrund dessen einfacherer Implementierung jedoch weiterhin üblich.
Bei der nächsten Komponente des Greedy-Algorithmus, die wir untersuchen, handelt es sich um die Auswahl der nächsten zu betrachtenden Kante. Nach Definition des Verfahrens wird in jedem Schritt die aktuell minimal gekreuzte Kante gewählt. Zur Realisierung dieser Auswahl bietet sich die Verwendung einer Prioritätswarteschlange an. Aus den Anforderungen des Algorithmus folgt, dass die Laufzeiten der Operationen Remove, ExtractMin und DecreaseKey ausschlaggebend für die Kosten der Verwendung der gesamten Datenstruktur sind. Eine Realisierung einer Prioritätswarteschlange, die kompetitive Laufzeiten für diese drei Operationen aufweist, ist der *Fibonacci-Heap*. Fibonacci-Heaps sind eine Erfindung von Fredman und Tarjan [FT87] und versprechen amortisierte Laufzeiten von $O(\log n)$ für Remove und ExtractMin sowie $O(1)$ für DecreaseKey. Die Initialisierung der Datenstruktur, ein weiterer potenzieller Kostenpunkt, ist in linearer Zeit durchführbar. Mit Feststellung der Kosten für die einzelnen Operation bleibt zu diskutieren, wie oft jede der Operationen durchgeführt wird. Bei Analyse des Greedy-Algorithmus fällt auf, dass jede Kante aus E genau einmal gelöscht wird, entweder wenn sie selektiert wird, oder wenn eine sie kreuzende Kante selektiert wird. Diese beiden Ereignisse korrespondieren zu den Operationen ExtractMin beziehungsweise Remove. Da beide dieser Operationen eine amortisierte Laufzeit von $O(\log n)$ aufweisen, ergeben sich für einen Graphen mit m Kanten Kosten von $O(n\log m)$. Die Operation DecreaseKey wird durchgeführt, wenn eine Kante gelöscht wird und sich damit die Kreuzungszahlen aller sie schneidenden Kanten verringert. Die Anzahl der Aufrufe der Funktion ist damit jedoch durch die Anzahl der Kantenkreuzungen k beschränkt. Mit den amortisiert konstanten Kosten für die Operation DecreaseKey ergibt sich eine Laufzeit von $O(k)$. Insgesamt kann für die Kantenauswahl bei Verwundung eines Fibonacci-Heaps also eine Laufzeit von $O(k+m\log m)$ erreicht werden.
Als letztes betrachten wir die Kosten für die Verwendung von Union-Find. Wie in Abschnitt 2.2 beschrieben, wird für die Abarbeitung von $n-1$ Union- sowie m Find-Operationen eine Laufzeit von $O(n+m\alpha(m))$ benötigt. Es ist leicht zu sehen, dass beide Operationen zur Selektion der Kanten korrespondieren. Damit ergibt sich, dass beide Operationen höchstens $O(m)$ mal durchgeführt werden müssen. Es folgt eine Gesamtlaufzeit von $O(m+m\alpha(m)) \subseteq O(m\alpha(m))$. Zusammen mit den linearen Initialisierungskosten ergeben sich für die Verwendung von Union-Find Kosten von $O(n+m\alpha(m))$.
Die Laufzeit des Greedy-Algorithmus ergibt sich aus der Summe der Laufzeiten der drei analysierten Bausteine. Da $O(m\alpha(m))$ von $O(m\log m)$ dominiert wird, beläuft sich diese insgesamt auf $O(n+k+m\log m)$.

Reverse Greedy-Algorithmus Bei dieser Variante entfernen wir aus dem Input-Graphen iterativ immer diejenige Kante, die die höchste Kreuzungszahl aufweist, bis der übrige Graph planar ist. Dieser wird dann zurückgegeben. Im Verlauf dieser Methode kann es

passieren, dass ungekreuzte Kanten entstehen. Darauf überprüfen wir vor jeder Iteration und selektieren alle ungekreuzten Kanten. Auch hier achten wir wie beim Greedy-Algorithmus mittels Union-Find darauf, Kanten, die innerhalb bestehender Cluster verlaufen, nicht mehr zu berücksichtigen.

Algorithmus 2 : Reverse Greedy Cluster Minimierung(Graph $G = (V, E)$)

Eingabe : Graph $G = (V, E)$
Ausgabe : $E' \subseteq E$, so dass E' keine Kreuzungen enthält und möglichst wenige Zusammenhangskomponenten in $G' = (V, E')$ verbleiben

```
1 E' ← ∅
2 foreach v ∈ V do MakeSet(v)
3 while E ≠ ∅ do
4     foreach e = (v, w) aus E ohne Kreuzungen do
5         Verschiebe e von E nach E'
6         Union(Find(v),Find(w))
7     Lösche beliebige Kante mit maximaler Kreuzungszahl
8 return E'
```

Laufzeit Die Laufzeit dieser Variante setzt sich im Wesentlichen aus den gleichen drei Bausteinen zusammen wie die Laufzeit des Greedy-Algorithmus: das Ermitteln der Kantenkreuzungen, die Auswahl der nächsten zu betrachtenden Kante, und die Verwendung von Union-Find.
Die Kosten für Berechnung der Kantenkreuzungen und die Verwendung von Union-Find decken sich unter Verwendung der obigen Argumente mit den Kosten der entsprechenden Komponenten des Greedy-Algorithmus. Anders verhält es sich jedoch mit der Auswahl der jeweils nächsten zu betrachtenden Kante.
Während beim Greedy-Algorithmus jeweils die aktuell minimal gekreuzte Kante im Fokus liegt, orientiert sich der Reverse Greedy-Algorithmus an der aktuell maximal gekreuzten Kante. Die verwendete Datenstruktur muss also die Operation ExtractMax effizient unterstützen, eine Anforderung, die der Fibonacci-Heap nicht erfüllt. Stattdessen greifen wir für den Reverse Greedy-Algorithmus auf die Verwendung eines *binären Suchbaums* zurück. Dieser liefert logarithmische Laufzeiten für die benötigten Operationen ExtractMax, Remove und DecreaseKey. Da die Aufrufe jeder Operation im Reverse Greedy-Algorithmus den Aufrufen der entsprechenden Operationen im Greedy-Algorithmus entsprechen ergibt sich für die Verwendung des Binärbaums eine Laufzeit von $O(k \log k)$.
Insgesamt benötigt der Reverse Greedy-Algorithmus also bei einem Graphen mit n Knoten, m Kanten und k Kantenkreuzungen eine Laufzeit von $O(n + k \log k + m \log m)$.

Reverse 1-plane Greedy-Algorithmus Da ein effizienter Algorithmus existiert, um 1-ebene Instanzen des Problems exakt zu lösen, liegt es nahe, den Reverse Greedy-Algorith-

mus so zu modifizieren, dass er ausgeführt wird, bis die restliche Instanz 1-eben ist und diese dann exakt zu lösen. Dabei ergibt sich der *Reverse 1-plane Greedy*-Algorithmus. Wie sich allerdings herausstellt, liefern beide Varianten des Reverse Greedy-Ansatzes die gleichen Ergebnisse. Dies ist damit zu begründen, dass beide Ansätze bis zum Erreichen einer 1-ebenen Subinstanz gleich operieren, danach wählt der exakte Algorithmus aus einem Paar gekreuzter Kanten willkürlich eine aus, beim Reverse Greedy-Algorithmus wird stattdessen willkürlich eine dieser Kanten gelöscht. Da jedoch die nicht gelöschte Kante nun zwingend ausgewählt wird unterscheiden sich beide Ansätze höchstens in der zufälligen Wahl einer Kante. Da beide Varianten im Grunde gleich sind, betrachten wir im Folgenden nur noch den regulären Reverse Greedy-Algorithmus.

Algorithmus 3 : Reverse 1-plane Greedy Cluster Minimierung (Graph $G = (V, E)$)

Eingabe : Graph $G = (V, E)$
Ausgabe : $E' \subseteq E$, so dass E' keine Kreuzungen enthält und möglichst wenige Zusammenhangskomponenten in $G' = (V, E')$ verbleiben

```
1 E' ← ∅
2 foreach v ∈ V do MakeSet(v)
3 while (G, E) ist nicht 1-eben do
4     foreach e = (v, w) aus E ohne Kreuzungen do
5         Verschiebe e von E nach E'
6         Union(Find(v),Find(w))
7     Lösche beliebige Kante mit maximaler Kreuzungszahl
8 Löse die übrige Instanz exakt
9 return E'
```

Laufzeit Die Laufzeit des Reverse 1-plane Greedy-Algorithmus entspricht im Wesentlichen der Laufzeit des Reverse Greedy-Algorithmus. Der einzige Unterschied liegt in der Verwendung des exakten Algorithmus nach erreichen von 1-Ebenheit. Da der exakte Algorithmus jedoch einer einfachen Greedy-Heuristik entspricht, ist seine Laufzeit sicherlich nicht größer als die Laufzeit des oben beschriebenen Greedy-Algorithmus und wird somit von der Laufzeit des Reverse Greedy-Algorithmus dominiert. Es ergeben sich für den Reverse 1-plane Greedy-Algorithmus somit Kosten von $O(n + k \log k + m \log m)$.

3.2 Der 1-ebene Fall

Wie von Akitaya et al. [HAC+19] beschrieben, kann Cluster-Minimierung für 1-ebene Graphen effizient gelöst werden. Dabei lassen wir einfarbige Kreuzungen zu, diese werden wie oben beschrieben planarisiert. Es ergibt sich die folgende Formulierung des Algorithmus.

Algorithmus 4 : Cluster Minimierung in gefärbten 1-ebenen Graphen

Eingabe : gefärbter 1-ebener Graph $G = (V, E)$

Ausgabe : $E' \subseteq E$, so dass E' keine Kreuzungen enthält und möglichst wenige Zusammenhangskomponenten in $G' = (V, E')$ verbleiben

```
1  begin
2      E' ← ∅
3      foreach e ∈ E ungefärbt do lösche e aus E
4      foreach einfarbige Kreuzung zwischen Kanten e, e' do
5          planarisiere die Kreuzung
6      foreach v ∈ V do MakeSet(v)
7      foreach e = {v, w} ∈ E ungekreuzt do
8          if Find(v) ≠ Find(w) then
9              Union(Find(v), Find(w))
10             verschiebe e nach E'
11     foreach e = {v, w} ∈ E do
12         if Find(v) ≠ Find(w) then
13             Union(Find(v), Find(w))
14             lösche die Kante, die e kreuzt, aus E
15             verschiebe e nach E'
```

Laufzeit Zur Analyse der Laufzeit dieses Algorithmus betrachten wir seine Unterschiede zum oben beschrieben allgemeinen Greedy-Algorithmus. Offensichtlich ist die einzige relevante Änderung die Planarisierung einfarbiger Kantenkreuzungen. Zu betrachten ist hier, dass sich durch die Planarisierung einer Kreuzung sowohl die Anzahl der Knoten als auch die Anzahl der Kanten im Graphen erhöht. Weiterhin ist eine Analyse der Kosten für die Planarisierung selbst vonnöten.

Wir beginnen mit der Beobachtung, dass ein 1-ebener Graph nur eine lineare Anzahl an Kanten besitzen kann. Diese folgt aus der Tatsache, dass, würde man aus jedem gekreuzten Kantenpaar eine Kante entfernen, eine Zeichnung eines planaren Graphen mit mindestens halb so vielen Kanten wie der ursprüngliche Graph entstünde. Da die Kantendichte planarer Graphen jedoch linear beschränkt ist, folgt die gleiche Aussage für 1-ebene Graphen. Aus dieser Beobachtung folgt sofort, dass auch die Anzahl an Kantenkreuzungen, und damit die Anzahl der durchzuführenden Planarisierungen, linear beschränkt ist. Da bei jeder Planarisierung genau ein neuer Knoten generiert wird, ist die Anzahl der Knoten n' im planarisierten Graphen linear in der Anzahl der Knoten im ursprünglichen Graphen, es gilt also $n' \in O(n)$.

Wenden wir uns nun der Kantenanzahl zu. Bei einer Planarisierung werden die beiden sich kreuzenden Kanten durch vier Kanten ersetzt. Da jede Kante höchstens einmal gekreuzt ist, kann sich durch den Planarisierungsvorgang die Anzahl der Kanten höchstens verdoppeln. Für die Anzahl an Kanten m' im planarisierten Graphen ergibt sich also eine

lineare Beschränkung in der Anzahl der ursprünglichen Kanten, $m' \in O(m) \subseteq O(n)$.
Da die Berechnung der Kantenkreuzungen bereits in der Laufzeit des ursprünglichen Greedy-Algorithmus enthalten ist, erfordert eine Planarisierung nur noch wenig Aufwand und ist in konstanter Zeit durchführbar. Mit den obigen Argumenten zur Anzahl der durchgeführten Planarisierungen werden deren Kosten von der übrigen Laufzeit dominiert.
Insgesamt stellen wir fest, dass $O(n+k+m \log m)$ eine gültige obere Schranke für die Laufzeit dieses Algorithmus ist. Die Analyse lässt sich jedoch noch etwas verfeinern. Wie oben gezeigt wurde, ist sowohl die Anzahl der Kanten als auch die Anzahl der Kantenkreuzungen linear in der Anzahl der Knoten beschränkt, oder, formell ausgedrückt: $k, m \in O(n)$. Mit dieser Überlegung reduziert sich die ursprüngliche Laufzeit von $O(n + k + m \log m)$ auf $O(n \log n)$.

4 Experimente

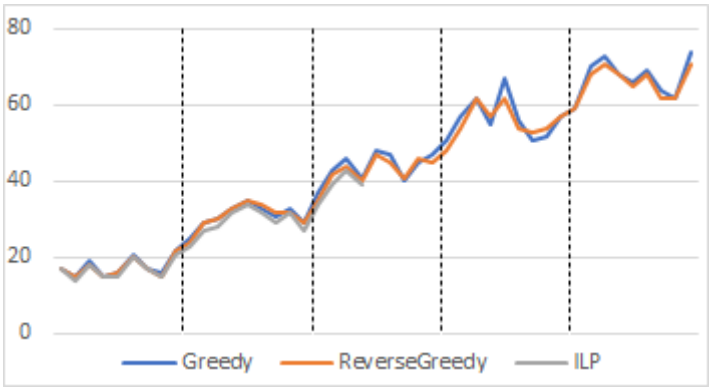
Um die Leistung der verschiedenen Algorithmen einschätzen zu können bietet es sich an, sie auf verschiedenen Probleminstanzen auszuführen und die Ergebnisse miteinander zu vergleichen. Dabei betrachten wir hauptsächlich die Qualität der jeweiligen Lösungen, da die Laufzeit je nach Implementierung stark variieren kann. Da die Algorithmen nur bezüglich der Anzahl der Cluster in einer Lösung optimiert sind, liegt das Augenmerk bei der Auswertung auf dieser Größe.

4.1 Aufbau der Experimente

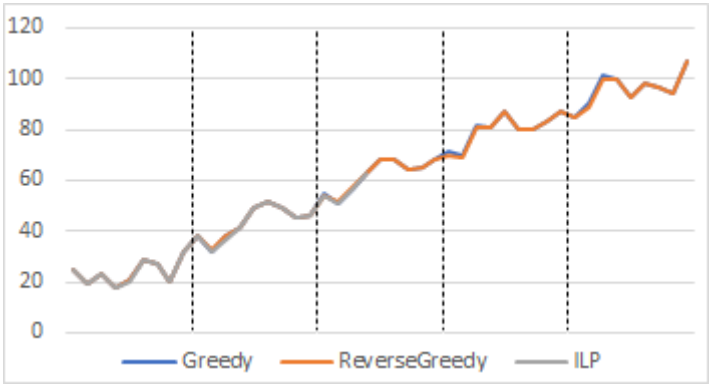
Zur Generierung der verwendeten Probleminstanzen wurde ein Datensatz aus der realen Welt verwendet, nämlich eine Karte, in der die Orte von Interesse in Bad Godesberg verzeichnet sind. Diese sind dabei in Kategorien gegliedert. Um mehrere verschiedene Graphen zu erhalten, wurde die Karte zunächst in neun Quadranten untergliedert und zu jedem Quadranten das Zentrum bestimmt. Zu jedem dieser Zentren wurden die nächsten 50 bis 250 Punkte, in 50er-Schritten, berechnet. Auf diese Weise entstanden 45 verschiedene Punktmengen. Zu jeder dieser Punktmengen wurden neun Graphen generiert, indem für diverse Werte jeweils β-Skelette generiert wurden. Auf den so entstandenen Instanzen wurden nun die Algorithmen ausgeführt und die Ergebnisse verglichen.

4.2 Ergebnisse

Wie in Abbildung 2 zu sehen ist, finden beide Algorithmen häufig Lösungen, die nur in geringem Maße vom Optimum abweichen. Die prozentuale Abweichung der jeweiligen Lösungen vom Optimum ist in Abbildung **??** dargestellt. Weiterhin verdeutlichen beide Grafiken die Tatsache, dass die Performanz beider Algorithmen nur wenig voneinander Abweicht. Ein Vergleich der Performanz ist in Abbildung 3 dargestellt. In dieser Grafik lässt sich erkennen, dass der Greedy-Algorithmus tendenziell eine leicht schlechtere Performanz aufweist als der Reverse Greedy Ansatz. Zwar existieren auch Instanzen, auf



(a) $\beta = 0,5$



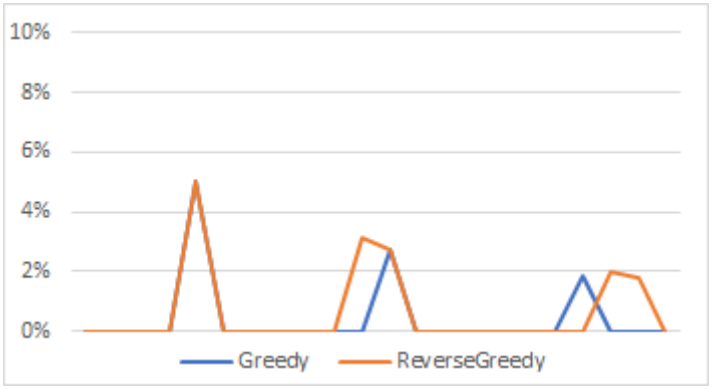
(b) $\beta = 0,9$

Abb. 2: Vergleich der Lösungen für verschiedene Werte von β, man beachte dabei die unterschiedlichen Skalen auf der y-Achse. Jeder Datenpunkt repräsentiert eine Probleminstanz. Auf der y-Achse ist jeweils die erreichte Clusteranzahl aufgetragen. Die Abschnitte der x-Achse repräsentieren die behandelten Instanzgrößen.

denen er eine Lösung mit weniger Clustern liefert, diese Fälle sind jedoch dem umgekehrten Fall zahlenmäßig unterlegen. Weiterhin lässt sich erkennen, dass sich bei steigendem β-Wert die Qualität der Algorithmen immer weiter angleicht. Dies ist damit zu erklären, dass die Anzahl der Kanten in einem β-Skelett bei steigendem β sinkt, und damit den Algorithmen deutlich weniger Entscheidungsfreiheit verbleibt.

Für $\beta = 0,9$ ist klar zu sehen, dass die Performanz der Algorithmen wenig Abweichung untereinander aufweist. Abbildung 4b verdeutlicht den Grund dafür. Hier ist ein Skelett für diesen Wert abgebildet, dieses weist keine einzige Kantenkreuzung auf. Offensichtlich finden für diesen Graphen alle Algorithmen die optimale Lösung. Zwar gilt diese Eigenschaft nicht für alle $0,9$-Skelette, die Kantendichte wird jedoch für alle Instanzen mit diesem β-Wert verhältnismäßig gering ausfallen und so den Algorithmen nicht viel Raum für Variation lassen. Im Gegensatz dazu verfügt die gleiche Knotenmenge mit $\beta = 0,5$ über eine deutlich größere Anzahl an Kanten (siehe Abbildung 4a). Wie deutlich zu sehen ist, sind nahezu alle Kanten mindestens einmal gekreuzt, im Gegensatz zum vorher

(a) $\beta = 0,5$



(b) $\beta = 0,9$

Abb. 3: Vergleich der prozentualen Abweichung vom Optimum für verschieden Werte von Beta. Jeder Datenpunkt repräsentiert eine Probleminstanz.

besprochenen Fall kann die Wahl des Algorithmus hier eine größere Rolle spielen.

Insgesamt lässt sich festhalten, dass im Allgemeinen der Reverse Greedy-Algorithmus leicht bessere Ergebnisse aufweist als der intuitive Greedy-Ansatz. Auch die asymptotische Laufzeit der Algorithmen weicht nicht wesentlich voneinander ab, der Reverse-Greedy-Algorithmus scheint also die bessere Wahl zu sein.

Für alle Instanzen wurde die Laufzeit der verschiedenen Algorithmen gemessen, dabei überschritt die Lösungszeit beider Heuristiken für keine Instanz eine Sekunde. Die in Abschnitt 3 beschriebene Ähnlichkeit der Laufzeiten bestätigt sich damit also auch experimentell.

5 Fazit

In dieser Arbeit wurde das Problem Cluster-Minimierung hergeleitet und definiert. Im Anschluss wurden einige Heuristiken für das Problem vorgeschlagen und auf ihre Laufzeit hin untersucht. Weiterhin wurde ein effizienter Algorithmus angegeben, der das Problem, auf 1-ebene Graphen eingeschränkt, optimal löst. Schließlich wurden Experimente

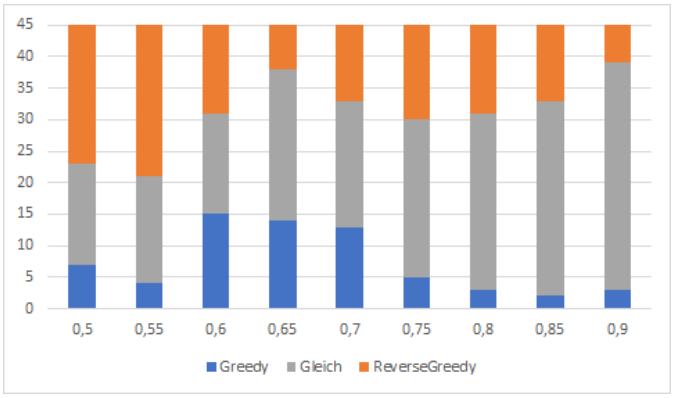
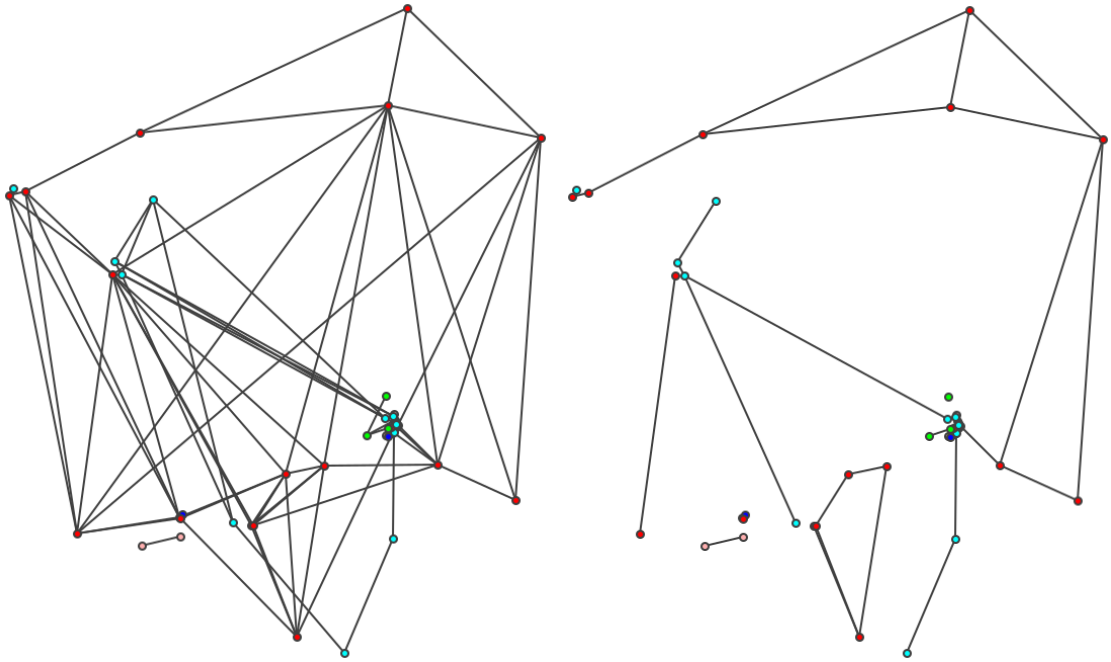


Abb. 4: Anzahl an besser gelösten Instanzen für verschiedene β - Werte

durchgeführt und beschrieben, in welchen die Performanz der vorgestellten Heuristiken miteinander verglichen wurde. Dabei wurde deutlich, dass von allen betrachteten Algorithmen auf den getesteten Instanzen Lösungen nah am Optimum gefunden werden konnten. Zusammen mit der effizienten Laufzeit der Algorithmen bescheinigt dies die praktische Relevanz der vorgestellten Heuristiken.

Obwohl das Problem Cluster-Minimierung NP-schwer ist haben unsere Experimente gezeigt, dass optimale Lösungen durch einfache Heuristiken gut approximiert werden können. Daher stellt sich die Frage, ob die Kantengenerierung durch β-Skelette die Probleminstanzen bereits zu weit vereinfacht und damit die Menge potenzieller Clusterings einer Punktmenge zu stark einschränkt. Es erscheint sinnvoll, Alternativen zu explorieren, zum Beispiel könnten bei der Kantengenerierung weitere Faktoren außer der physischen Nachbarschaft einfließen. Da wir uns in dieser Arbeit auf Cluster unter Punkten gleichen Typs beschränkt haben, könnte ein besonderes Gewicht auf die Generierung von Kanten zwischen Punkten gleichen Typs gelegt werden.



(a) $\beta = 0,5$ (b) $\beta = 0,9$

Abb. 5: Vergleich der Anzahl an einfarbigen Kanten für verschiedene Werte von β auf einer Menge von 50 Punkten.

Literatur

[Bal95] Ivan J. Balaban: An optimal algorithm for finding segments intersections. In: *Proceedings of the 11th Annual ACM Symposium on Computational Geometry*, Seiten 211–219, 1995. https://doi.org/10.1145/220279.220302.

[CE92] Bernard Chazelle und Herbert Edelsbrunner: An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39(1):1–54, 1992. https://doi.org/10.1145/147508.147511.

[FT87] Michael L. Fredman und Robert Endre Tarjan: Fibonacci Heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. https://doi.org/10.1145/28869.28874.

[Gei20] Jakob Geiger: Cluster-Minimierung in Geometrischen Graphen, 2020. http://www1.pub.informatik.uni-wuerzburg.de/pub/theses/2020-geiger-master.pdf.

[HAC+19] Jan Henrik Haunert, Hugo Akitaya, Sabine Cornelsen, Philipp Kindermann, Tamara Mchedlidze, Martin Nöllenburg, Yoshio Okamoto und Alexander Wolff: Clustering colored points in the plane. In: Sara Irina Fabrikant, Silvia Miksch und Alexander Wolff (Herausgeber): *Visual Analytics for Sets over Time and Space (Dagstuhl Seminar 19192)*, Band 9 der Reihe *Dagstuhl Reports*, Seiten 53–56. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. http://drops.dagstuhl.de/opus/volltexte/2019/11380/.

[TvL84] Robert E. Tarjan und Jan van Leeuwen: Worst-case analysis of Set Union algorithms. *J. ACM*, 31(2):245–281, 1984. https://doi.org/10.1145/62.2160.