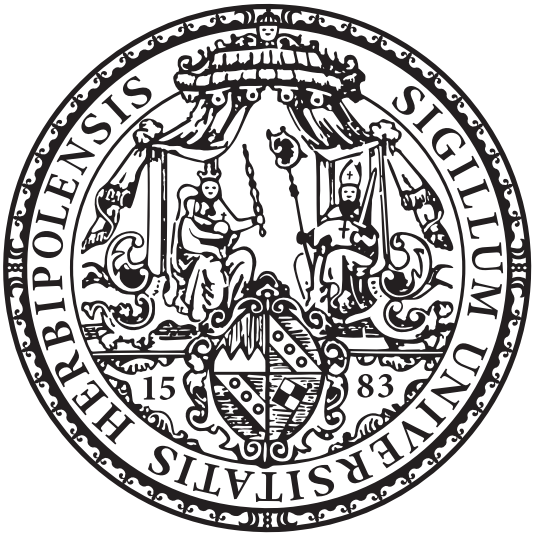
Praktikumsbericht

Kantenlängenverhältnis planarer Graphen auf dem Gitter

Michael Leeming

Abgabedatum:	12. August 2021
Betreuer:	Prof. Dr. Alexander Wolff
	Myroslav Kryven, M. Sc.



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

1 Einleitung

Wie bei vielen Themen von Praktika und Abschlussarbeiten bestehen auch zum Thema, das diesem Praktikum zugrunde liegt, schon viele Arbeiten über etwaige Teilaspekte. Es soll um das Kantenlängenverhältnis von Zeichnungen planarer Graphen auf dem Gitter gehen. Gleich zu Beginn soll für das „Kantenlängenverhältnis“ die Abkürzung KLV eingeführt werden. Das KLV in Bezug auf die Zeichnung eines Graphen beschreibt das größte Verhältnis der Längen zweier Kanten in dieser Zeichnung, wird aber im Folgenden noch definiert (Abschnitt 2). Blazej et al. [zFL20] haben sich mit dem KLV auf 2-Bäumen beschäftigt und dafür eine untere Schranke in $\Omega(\log n)$ bewiesen, sowie eine konstante obere Schranke von 4 für das lokale KLV, wenn also nur adjazente Kanten betrachtet werden. Borrazzo und Frati [BF20] fanden für planare Graphen eine untere Schranke in $\Omega(n)$. Und Lazard et al. [LLL19] konnten eine scharfe obere Schranke von $2 - \varepsilon$ für außenplanare Graphen zeigen.

Ein Problem, das bei allen drei Arbeiten aber bestand, war die Tatsache, dass das Argument der Graphästhetik und -lesbarkeit, das berechtigterweise oft im Zusammenhang mit dem KLV erwähnt wird, leider nicht mit zufriedenstellenden Ergebnissen untermauert werden konnte. In allen drei Fällen wurden nämlich Methoden verwendet, die extrem kleine Winkel (je nach erlaubtem ε) zwischen adjazenten Kanten produzierten. Das Thema des Graph Drawing Contest 2021 ist das im Titel dieses Praktikums erwähnte - das KLV von planaren Graphen, die mit einer beschränkten Anzahl erlaubter Knicke pro Kante kreuzungsfrei gezeichnet werden sollen. Sowohl Knoten als auch Knickpunkte sollen dabei auf dem Gitter liegen. Hinzu kommt eine Beschränkung der Höhe und Breite des Gitters. Im Hinblick auf ästhetische Aspekte können die Winkel also nicht beliebig klein werden. Zusammenfassend ist die Eingabe ein planarer Graph, die Dimensionen des Gitters und die erlaubte Anzahl an Knicken pro Kante. Die Ausgabe soll eine planare Zeichnung mit genannten Kriterien sein, die das KLV minimiert.

Diese Arbeit ist auf diesen Contest ausgelegt und hat daher folgende Rahmenbedingungen:

Die Laufzeit des Algorithmus ist durchaus von Belang, da im Rahmen des Graph Drawing Contests die erlaubte Zeit zum Lösen der vorgegebenen Graphinstanzen auf eine Stunde begrenzt ist. Da das Problem der Minimierung des KLV aber (wahrscheinlich) NP-schwer ist, wird man sich für größere Graphinstanzen wohl mit einer Heuristik oder Approximation zufriedengeben müssen.

Das Vorgehen bzw. die angewandten Techniken sind jedem selbst überlassen. Es stand also zu Beginn dieser Arbeit noch nicht fest, welche Technik verwendet wird. Wie bereits angeklungen, war eine Möglichkeit, mehrere Algorithmen auszuarbeiten, die Laufzeit zu analysieren und je nach Eingabegröße einen Brute-Force-Algorithmus oder einen heuristischen Algorithmus zu wählen. Zufälligerweise beschäftigt sich meine Masterarbeit ebenfalls mit dem KLV und in diesem Rahmen scheine ich aktuell beweisen zu können, dass die Minimierung des globalen und lokalen KLV NP-schwer ist durch eine Reduktion auf SAT. Ansonsten konnte ich bei einer Literaturrecherche keine Aussage über NP-Schwere von KLV finden - nur zu Fixed-Edge-Length-Drawing [EW90].

Letztendlich wurde folgender Algorithmus umgesetzt:

1. Einlesen des (planaren) Graphen
2. Erstellung einer Schnyder-Zeichnung
3. Finden der kürzesten und längsten Kante
4. Finden des größten Dreiecks, das über die kürzeste Kante in einer der beiden angrenzenden Facetten aufgespannt werden kann
5. Berechnung der Knicke innerhalb des Dreiecks, sodass die Gesamtlänge kleiner gleich der der längsten Kante ist
6. Rausschreiben des Graphen

2 Problemstellung, Schreibkonventionen, Vorbemerkungen

Die Eingabe sei ein Quadrupel (G, b, h, w) und enthalte einen planaren Graphen G mit Knotenmenge $V(G)$, Kantenmenge $E(G)$ und Kantensegmentmenge $E^*(G)$ (wichtig, wenn Laufzeiten von der Anzahl der Knickpukte abhängen) und Integerwerte b, h und w. Gesucht ist eine geradlinige, planare Zeichnung Γ von G mit bis zu b erlaubten Knicken pro Kante und deren Knoten und Knicke auf den Gitterpunkten eines Rasters der Größe $[0, w] \times [0, h]$ liegen.

Das KLV einer Zeichnung Γ eines Graphen G ist definiert als $\rho(\Gamma) = \max_{e_1,e_2 \in E(G)} |e_1|/|e_2|$. Diese stellt die zu minimierende Zielfunktion dar.

GDC sei die Abkürzung für Graph Drawing Contest (und wie bereits erwähnt *KLV* für Kantenlängenverhältnis). Gemäß der üblichen Konvention sei n die Anzahl der Knoten des im Zusammenhang genannten Graphen und m die Anzahl der Kanten. n_f bezeichne die Anzahl der Knoten und Knicke, die inzident zu einer Facette f sind, n_f wächst also mit dem Legen der Knicke.

Im *GDC* wird der Graph in Form eines JSON-Files übergeben. Im gleichen Format soll der bearbeitete Graph auch wieder ausgegeben werden. Der JSON-File enthält folgende Einträge:

- *nodes*: Der Eintrag *nodes* enthält für jeden Knoten wiederum einen Eintrag *id*, *x* und *y*. Diese enthalten die ID des Knotens als Integer zwischen 0 und $n-1$, dessen x-Koordinate als Integer zwischen 0 und w und dessen y-Koordinate als Integer zwischen 0 und h.
- *edges*: Der Eintrag *edges* enthält für jede (gerichtete) Kante wiederum einen Eintrag *source*, *target* und optional *bends*. Diese enthalten die ID des Quellknotens, die ID des Zielknotens und optional die Positionen der Knicke dieser Kante. Ist *bends* gesetzt, so enthält dieser Eintrag jeweils einen Eintrag für jeden Knick bestehend aus x- und y-Koordinate.

- *width*: Der optionale Eintrag *width* enthält die Breite des Gitters und entspricht damit w. Ist *width* nicht gesetzt, so gilt $w = 1\,000\,000$.
- *height*: Der optionale Eintrag *height* enthält die Höhe des Gitters und entspricht damit h. Ist *height* nicht gesetzt, so gilt $h = 1\,000\,000$.
- *bends*: Der Eintrag *bends* enthält die maximale Anzahl Knicke, die pro Kante erlaubt ist.

Nachfolgend steht der JSON-File eines Beispielgraphs und die entsprechende Zeichnung dazu. Der Beispielgraph ist der Internetseite des *GDC* entnommen. Es ist allerdings zu beachten, dass die Zeichnung hier horizontal spiegelverkehrt zu der des *GDC* ist, da im Rahmen dieses Praktikums zur Darstellung der Graphen das SVG-File-Format genutzt wurde, bei dem die y-Koordinate nach unten wächst (wo also der Nullpunkt oben links im Bild liegt), was aus Gründen der Konsistenz bereits hier entsprechend umgesetzt ist.

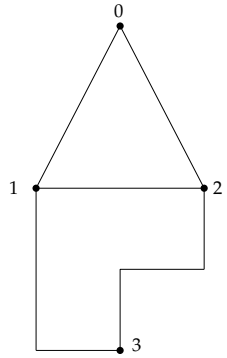
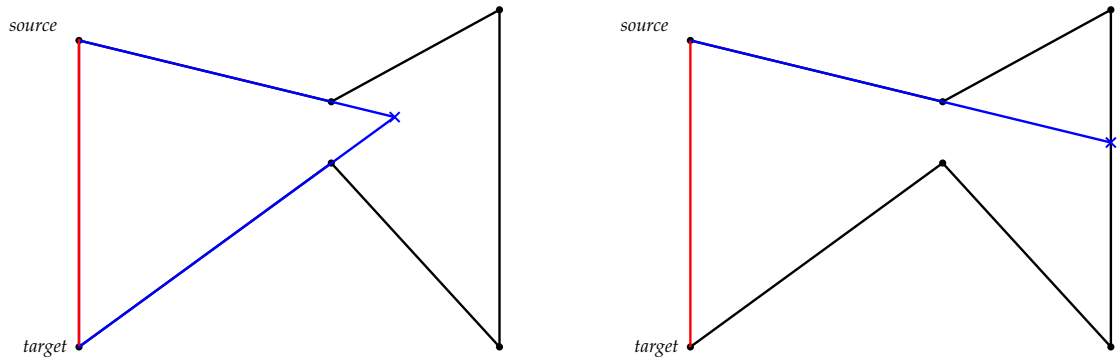


Abb. 1: Zeichnung zum Beipspielgraphen des JSON-Files

```
{ "nodes": [
      { "id": 0, "x": 1, "y": 0 },
      { "id": 1, "x": 0, "y": 2 },
      { "id": 2, "x": 2, "y": 2 },
      { "id": 3, "x": 1, "y": 4 }],
   "edges": [
      { "source": 0, "target": 1 },
      { "source": 0, "target": 2 },
      { "source": 1, "target": 2 },
      { "source": 1, "target": 3,
        "bends": [
          { "x": 0, "y": 4 }]},
      { "source": 2, "target": 3,
        "bends": [
```



(a) Schnitt zweier Geraden

(b) Schnitt einer Gerade mit Kante

Abb. 2: Unterschiedliche Kandidatenpunkte für größtes Dreieck

{ “x”: 2, “y”: 3 },
{ “x”: 1, “y”: 3 }]}],
“width”: 2,
“height”: 4,
“bends”: 2 }

3 Algorithmus

In diesem Abschnitt wird detaillierter auf den umgesetzten Algorithmus eingegangen, vor allem auf das Finden des größten Dreiecks und das Setzen der entsprechenden Knickpunkte und anschließend auf ein paar Aspekte zu Vorteilen dieses Algorithmus gegenüber alternativen Algorithmen. Der Algorithmus setzt also an, nachdem der Graph eingelesen wurde. Die für diesen Abschnitt relevante Implementierung erfolgte hauptsächlich in der Klasse Layouter.cpp (siehe Abschnitt 4.5).

3.1 Beschreibung des Algorithmus

Zunächst muss für den Graphen G in $O(E^*(G))$ die längste und die kürzeste Kante ermittelt werden. Für diese kürzeste Kante $shortest = (source, target)$ wird dann die Kontur der beiden inzidenten Facetten als *List<edge>* ermittelt. Dabei wird von *source* beginnend (je nach linker oder rechter Facette) die nächste adjazente Kante im oder gegen den Uhrzeigersinn an die Ausgabeliste angefügt, bis man bei *target* angekommen ist. Auf entsprechende Spezialfälle ist zu achten, z.B. überhaupt keine adjazenten Kanten zu *shortest* oder keine inzidenten zu *target*.

Anschließend wird für die beiden inzidenten Facetten f_1 und f_2 der Punkt p innerhalb der Facette gesucht, der mit *shortest* das (flächenmäßig) größtmögliche Dreieck aufspannt, welches keine Kante schneidet oder beinhaltet. Andere Knoten und Kanten-

segmente dürfen aber auf den Schenkeln des Dreiecks liegen. Die Knickpunkte müssen dann also im echten Innern des Dreiecks liegen. Dieser Schritt erfolgt in $O(n_{f_1}^3 + n_{f_2}^3)$, da in den Facetten als Kandidaten für diesen Dreieckspunkt p entweder die Schnittpunkte aus der Geraden durch *source* und einen beliebigen Knoten oder Knickpunkt mit einer Geraden durch *target* und einem anderen (oder dem selben) Knoten oder Knickpunkt infrage kommt (siehe Abbildung 2a) oder mit einem Kantensegment (Abbildung 2b). Jeder dieser $O(n_{f_i}^2), i \in \{1, 2\}$ Kandidatenpunkte muss dann noch auf Kreuzung mit $O(n_{f_i})$ Kantensegmenten überprüft werden.

Zuletzt werden die Knicke innerhalb des Dreiecks gelegt, indem zunächst die längere der beiden Linien $(source, p)$ und $(target, p)$ gewählt wird (bezeichne diese als *line*) und die Knickepunkte abwechselnd ins Dreieck möglichst nah an *line* und *shortest* gelegt werden. Es werden so lange weitere Knicke gelegt, solange 1. noch nicht b Knicke gelegt wurden, 2. noch Platz auf dem Gitter ist, wenn man sich auf beiden Linien in Richtung *source* bzw. *target* annähert und 3. die Gesamtlänge der Kante kleiner gleich der längsten Kante ist. Vgl. dazu Abbildung 6, wäre die längste Kante länger und b groß genug, würden sich die Kicke auf beiden Seiten dem oberen mittleren Knoten annähern.

3.2 Alternative: Größtes konvexes Polygon

Eine andere mögliche Implementierung wäre, anstatt des größten Dreiecks das größte konvexe Polygon zu ermitteln, das im Innern der Facette liegt und das auf seinem Rand *shortest* enthält. Das Polygon muss konvex sein, damit zwei neu gelegte Knickpunkte immer durch ein geradliniges Kantensegment erreichbar sind. Es existiert auch bereits ein Linearzeitalgorithmus, um das größte konvexe Polygon in einer Facette zu berechnen, dieser ist aber kompliziert zu implementieren und beinhaltet nicht zwangsläufig die Endpunkte der kürzesten Kante. Entsprechend Abbildung 3 war die Idee für eine Implementierung, von beiden Endpunkten der kürzesten Kante beginnend jeweils im bzw. gegen den Uhrzeigersinn aufeinander zuzulaufen und den größten Schnitt dieser beiden Linienzüge zu wählen, ähnlich einem Gift-Wrapping-Algorithmus, nur über das Innere statt das Äußere der Facette. Für die grüne Linie in Abbildung 3 hieße das z.B., bei einem Rechtsknick in gleichem Winkel nach rechts abzuknicken, bei einem Linksknick aber bis auf die nächste Kante weiterzulaufen.

Dies erwies sich allerdings als sehr komplex, da auch die Schnitte dieser beiden Linienzüge nicht zwangsläufig konvex sind. Das Finden eines größten Dreiecks hatte außerdem den Vorteil, dass für das Legen der Knicke in einem Dreieck Sweep-Lines benutzt werden konnten, die Knicke konnten so also einfach in $O(\min(b, l_1 + l_2))$ gefunden werden für Längen l_1 und l_2 der beiden entsprechenden Dreieckskanten der Sweep-Lines.

3.3 Alternative: Visibility-Polygon

Des Weiteren gäbe es auch die Möglichkeit, statt des größten Dreiecks ein Visibility-Polygon über *shortest* aufzuspannen und für die Knickpunkte abwechselnd den weitentferntesten Punkt in diesem Polygon zu wählen und anschließend mit dem nächsten Knick wieder zu *shortest* zurückzukehren. Wie Abbildung 5 zeigt, kann das auch zu besseren

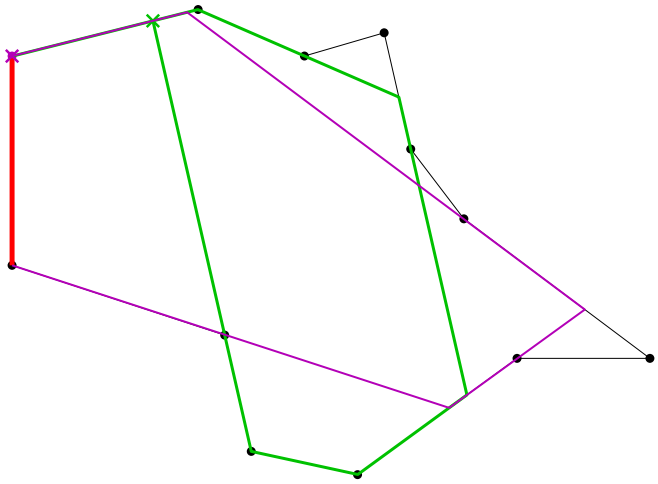


Abb. 3: “Gift-Wrapping” für größtes konvexes Polygon; rot: Kante, für die Polygon bestimmt wird (“kürzeste”), grün: Wrapping im Uhrzeigersinn, lila: Wrapping gegen Uhrzeigersinn

Ergebnissen führen.

Die Umsetzung ist aber wiederum nicht trivial. Einerseits kann es sein, dass man bei einem ungeraden b nach dem letzten Knick nicht mehr ohne Kreuzung zu *target* zurückkehren kann. Außerdem ist nicht jeder Punkt im Visibility-Polygon von jedem Punkt auf *shortest* erreichbar, was wiederum zu Kreuzungen führen kann.

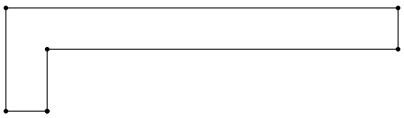
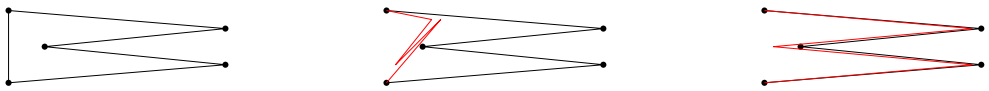


Abb. 4: schlechte Voraussetzung für Visibility-Polygon

Im Übrigen führt aber auch das Visibility-Polygon nicht zu optimalen Lösungen. Abbildung 4 zeigt eine Graphzeichnung, in der das Visibility-Polygon nur einen kleinen Teil der Facette miteinbezieht. Für $b = 4$ wäre die optimale Lösung nämlich eine Kante entlang dem gesamten Kantenzug der Facette (je nach Feinheit des Gitters).

3.4 Alternative: Größtes Dreieck im Sinne der längsten Schenkel

Zuletzt gäbe es noch die Alternative, das größte Dreieck nicht über seinen Flächeninhalt, sondern über die Distanz von p zu *source* und *target* zu definieren. Für $b = 1$ führt das auch zu einer optimalen Lösung. Ansonsten hat die Wahl des Dreiecks über seinen Flächeninhalt aber den Vorteil, dass dieses wahrscheinlich nicht zu dünn sein wird (da es ansonsten extrem lang sein müsste). Für ein dünnes Dreieck, ein grobes Gitter und ein großes b hätte man dann nämlich das Problem, dass man die Knicke nicht mehr vor- und zurücklegen könnte.



(a) ohne Knicke (b) berechnete Knicke (c) optimale Knicke

Abb. 5: Alternativ Finden des Visibility-Polygons

4 Programmierumgebung, OGDF, Klassenaufbau

Für diese Aufgabe fiel die Wahl für die Programmiersprache und -umgebung auf Microsoft Visual Studio (2019) und C++. Somit konnte der Framework OGDF genutzt werden. Dieser bietet Klassen bzw. Objekte zum Verwalten und Bearbeiten von Grapheigenschaften (Knotenpositionen, Kanten, Knicke) sowie geometrische Funktionalitäten (z.B. Erstellen von Punkten und Polygonen, Berechnung von Schnittpunkten oder ob ein Punkt im Innern eines Polygons liegt). Außerdem bietet OGDF die Möglichkeit, für einen Graphen zu Testzwecken eine Zeichnung in Form eines SVG-Files zu speichern. Hauptsächlich wurden folgende Klassen von OGDF genutzt:

- *Graph* ist ein Graph im Sinne einer Adjazenzliste und stellt die Funktionen *node newNode()* und *edge newEdge(node source, node target)* zur Verfügung.
- *GraphAttributes* bezieht sich auf ein Objekt der Klasse *Graph* und verwaltet Attribute wie z.B. Knotenpositionen, Knicke und (nur für die SVG-Zeichnungen relevant) Labels, Dicke und Farbe von Knoten und Kanten.
- *SchnyderLayout* stellt die Funktion *call(GraphAttributes GA)* zur Verfügung, um die Positionen von *GA* entsprechend einem Schnyder-Layout zu bearbeiten.
- *GraphIO* kann für ein *GraphAttributes*-Objekt eine Zeichnung z.B. im SVG-Format erstellen und abspeichern.
- Geometrische Klassen wie *DPoint* oder *DPolyline*.

Der größte Nutzen im Zusammenhang dieses Praktikums war, dass OGDF eine Schnyder-Zeichnung für einen Graphen erstellen kann, was in zweierlei Hinsicht wichtig war: Erstens war nicht ganz klar, ob laut Aufgabenstellung des *GDC* die Eingabe in Form eines JSON-Files ein Graph wäre, der bereits Koordinaten für die Punkte enthält (was sich aus der Aufgabenstellung so interpretieren ließe). Andernfalls wäre zunächst (irgend-)eine planare Zeichnung nötig, bevor Knicke gelegt werden könnten. In einer Schnyder-

Zeichnung sind außerdem die drei Kanten der äußeren Facette (wahrscheinlich) die längsten Kanten. Dies bewirkt, dass man sich beim Berechnen der größten Dreiecke und Legen der Knicke keine Gedanken darum machen muss, ob die entsprechenden beiden Facetten der aktuell kürzesten Kante die äußere Facette sein könnten. Man muss lediglich aufpassen, dass die aktualisierte Länge der vorher kürzesten Kante immer kleiner gleich der drittlängsten Kante bleibt.

Es musste allerdings besonders auf die Benutzung von Pointern geachtet werden, wenn man beispielsweise nach dem korrekten Einlesen eines JSON-Files den *Graph* zusammen mit seinen *GraphAttributes* in einem eigenen Objekt speichert, in dem beide als Pointer referenziert werden, die bereits in der Leserklasse als solche definiert und übergeben werden und am Ende wieder alle Pointer zu löschen, um Zugriffsverletzungen zu vermeiden.

Es folgen nun die Klassen, die für dieses Praktikum in der VisualStudio-Projektmappe erstellt wurden und deren Funktionalitäten.

4.1 GraphWA

Die Klasse *GraphWA* steht für "Graph with attributes" und dient dazu, nach dem Einlesen *Graph*, *GraphAttributes*, *width*, *height* und *bends* zusammen in einem Objekt zu speichern. Es sind ein entsprechender Konstruktor und Getter-Methoden implementiert. Wichtig war in dieser Klasse wie bereits erwähnt, *Graph* und *GraphAttributes* nur als Pointer zu speichern und auch *GraphWA* stets als Pointer zu übergeben.

4.2 EdgeTri

Die Klasse *EdgeTri* wurde zunächst implementiert, erwies sich dann aber als unnötig. Die Idee war, für eine Kante den Punkt zu speichern, über den das größte Dreieck in einer angrenzenden Facette aufgespannt wurde, was sich aber natürlich erledigt hatte, da der Punkt für das größte Dreieck sowieso nochmal neu berechnet werden müsste, sollten die Knicke einer Kante mehrfach berechnet werden müssen. Außerdem wächst die Laufzeit zum Finden des größten Dreiecks in einer Facette f kubisch in n_f, sodass es aus Laufzeitgründen wohl ohnehin kaum dazu kommen würde, dass eine Kante mehrfach berechnet wird, da sich n_f mit jedem Knick für jeden Knoten, den man in f legt, erhöht.

4.3 jsoncpp

Die Klasse *jsoncpp* diente zum Auslesen von Einträgen in JSON-Files. Diese Klasse habe ich nicht selbst geschrieben, sondern importiert. Ein Eintrag in einem JSON-File entspricht der Klasse *Json::Value* und kann weitere *Value*s enthalten. Zum besseren Verständnis der Implementierung steht nachfolgend der entsprechende C++-Code zum Einlesen der Knoten aus einem JSON-File *file* in ein *GraphAttributes**-Objekt *GA* des Graphen *G*:

```
std::ifstream graphFile(file, std::ifstream::binary);
Json::Value value;
```

```
graphFile >> value;

for (Json::Value next : value["nodes"])
{
        ogdf::node n = G->newNode();
        GA->x(n) = next["x"].asDouble();
        GA->y(n) = next["y"].asDouble();
        GA->idNode(n) = next["id"].asInt();
}
```

4.4 ReaderWriter

ReaderWriter ist eine Klasse zum Einlesen und Rausschreiben von JSON-Files in oder aus einem *GraphWA*-Objekt. Bei der Methode *read* ist über boolean-Parameter anzugeben, ob nur das *GraphWA*-Objekt zurückgegeben wird oder ob zusätzlich noch eine SVG- und GML-Datei erstellt und abgespeichert wird und ob die Knoten und Kanten möglichst dünn gezeichnet werden sollen (Kantendicken auf 0.01, Knotendicken auf 0.5), was bei dichten Graphen zu besserer Leserlichkeit führt.

Für diesen und den folgenden Unterabschnitt (Layouter) ist im Anhang noch eine Auflistung der implementierten Methoden angegeben.

4.5 Layouter

Diese Klasse ist für die Umsetzung des bereits in Abschnitt 3 ausführlich beschriebenen Algorithmus zuständig. Es wird u.a. die kürzeste Kante und das größte dazu inzidente Dreieck ermittelt und die Knicke darin gelegt.

4.6 main

Zu dieser Klasse bleibt nicht mehr viel zu sagen. Sie vereint alle Funktionalitäten, die bis jetzt implementiert wurden (wie am Ende der Einleitung beschrieben). Letztendlich wird an keiner Stelle das *KLV* berechnet, in einer *while*-Schleife werden so lange Knicke für die aktuell kürzeste Kante berechnet, bis sich nichts mehr ändert oder eine vorher festgelegte Anzahl an Schleifendurchläufen erreicht wird.

5 Tests

Für die Tests wurden ein 5×5-Grid-Graph erstellt, ein 5×5-Dreieck-Graph (ein Grid-Graph, bei dem die einzelnen Zellen nochmal mit einer diagonalen Kante in zwei Dreiecke geteilt werden) und einer, der auf Pentagonen basiert (Instanzen von C_5, die miteinander verknüpft werden, siehe Abbildung 7). Die Tests wurden auf einem Acer Aspire 3 mit 8GB Arbeitsspeicher und Intel Core i3-8130U Prozessor mit 2,2 GHz durchgeführt. Leider haben beim 5×5-Grid-Graphen bereits zehn Schleifendurchläufe 44,52 Sekunden gedauert, da die Laufzeit zur Berechnung der Knicke in einer Facette f kubisch in

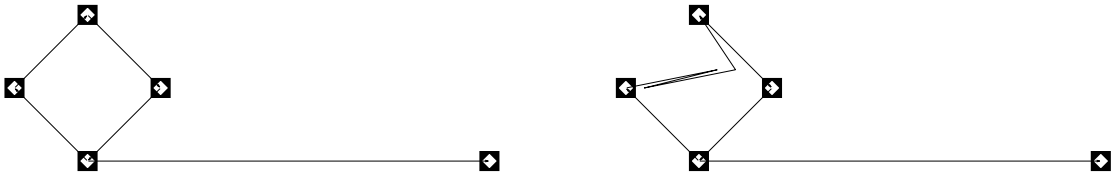
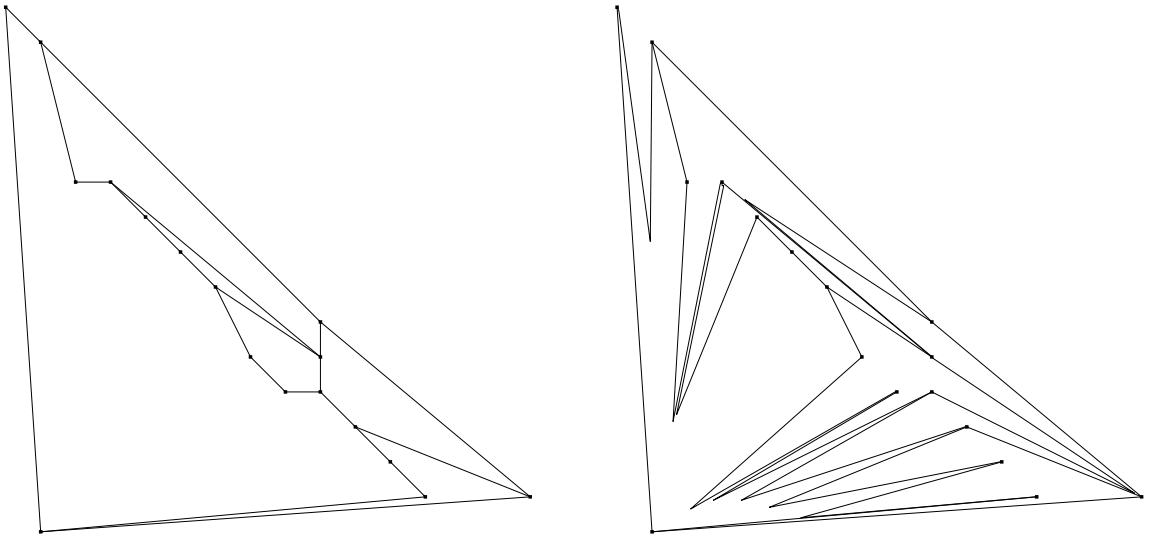


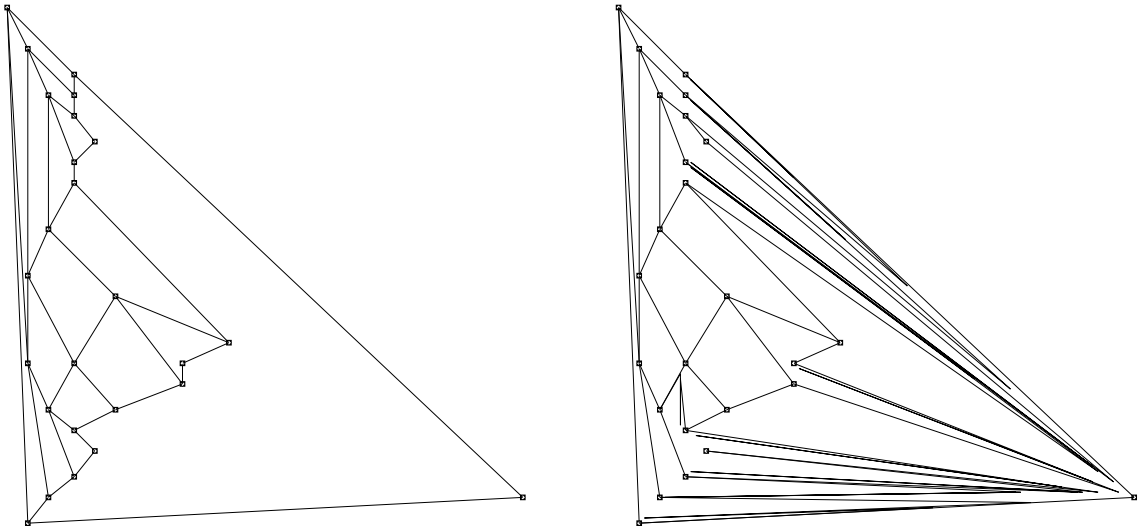
Abb. 6: Einfacher Graph



(a) KLV: 15.03 (b) KLV: 12.56

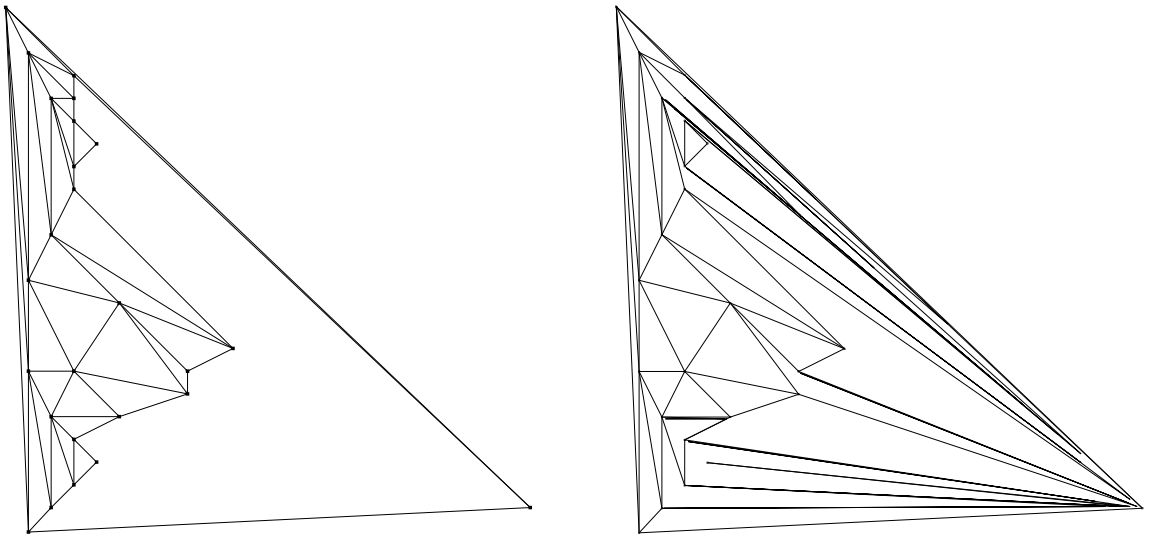
Abb. 7: Pentagon-Graph

n_f wächst, da für die Kreuzung für jedes Paar von Geraden durch Knoten oder Knickpunkte der Schnitt mit jedem Kantensegment überprüft werden muss. Zudem haben Grid-Graphen eine sehr lange äußere Facette, was zu einer entsprechend großen Facette im Innern der Schnyder-Zeichnung führt. Liegen nun viele kurze Kanten an dieser großen Facette, wächst sie außerdem um bis zu b Knicke in jedem Schleifendurchlauf. In Abbildung 6 bis Abbildung 9 sind die Ergebnisse der drei Tests bei zehn Schleifendurchläufen abgebildet und für einen kleineren Graphen, wo die Knicke besser sichtbar sind, für einen Schleifendurchlauf.



(a) KLV: 29.88 (b) KLV: 13.02

Abb. 8: Grid-Graph



(a) KLV: 34.48 (b) KLV: 16.78

Abb. 9: Dreiecks-Graph

6 Vergleich Greedy-Algorithmus

Zuletzt soll noch der Vergleich zu einem Greedy-Algorithmus gezogen werden, der im Rahmen meiner Masterarbeit entstand. Diese beschäftigte sich mit dem lokalen Kantenlängenverhältnis von 3-Bäumen. Der erste Unterschied zwischen den beiden Problemstellungen war, dass das lokale Kantenlängenverhältnis λ im Gegensatz zum globalen ρ als das maximale Längenverhältnis von ausschließlich zueinander adjazenten Kanten definiert ist: $\lambda(\Gamma) =_{def} \max_{uv,vw \in E(G)} |uv|/|vw|$ (für eine Zeichnung Γ eines Graphen G und Kanten $uv = \{u, v\}$, $vw = \{v, w\}$). Der zweite Unterschied liegt darin, dass in meiner Masterarbeit die Zeichnung nicht auf ein Gitter begrenzt war, wodurch in Bezug auf das *KLV* bessere Ergebnisse erzielt werden konnten. Trotz der leicht unterschiedlichen Zielfunktionen erzielte der Greedy-Algorithmus auch gute Ergebnisse in Bezug auf das globale Kantenlängenverhältnis.

Die Idee des Greedy-Algorithmus ist, für einen in eine Facette neu zu stapelnden Knoten die Position innerhalb des Dreiecks so zu wählen, dass das Maximum über die sechs Funktionen minimiert wird, die ein neues lokales *KLV* bestimmen könnten - für jeden der drei Eckknoten jeweils das Verhältnis zwischen der bisher längsten Kante dieses Knotens und der Distanz zum neuen Knoten und das Verhältnis zwischen der neuen Distanz und der bisher kürzesten Kante.

Der Greedy-Algorithmus war auf 3-Bäume ausgelegt. Auf diesen soll nun der Vergleich der beiden Algorithmen durchgeführt werden. K_3 ist ein 3-Baum. Einen 3-Baum kann man wiederum zu einem 3-Baum erweitern, indem man einen neuen Knoten ins Innere eine Facette legt und für ihn Adjazenz zu den drei ihn umschließenden Knoten herstellt.

Der *Greedy*-Algorithmus fing für die oberste Facette mit einem gleichseitigen Dreieck an. Für den hier präsentierten Algorithmus (*Schnyder-and-bends*) ist das initiale Layout egal, da ohnehin ein Schnyder-Layout berechnet wird. Die Tests konnten nur bis zu einer Tiefe von 9 für vollständige 3-Bäume durchgeführt werden. Bei dieser Tiefe besitzt der Graph 9844 Knoten. Alleine das Einlesen des Graphs mittels *JSON* dauerte dafür ca. 32 Minuten. Bei geringer Baumtiefe kann das *KLV* nahezu auf 1 optimiert werden, je nach Feinheit des Gitters (bei den Tests $w = h = 1000000$). Wenn aber mehr Kanten enthalten sind als die im Vornherein für *Schnyder-and-bends* festgelegte Anzahl an Schleifendurchläufen, ist das resultierende *KLV* nahezu das gleiche wie das des zuvor erstellten Schnyder-Layouts. Die Anzahl der Knoten steigt hier exponentiell mit der Baumtiefe. Diese Grenze wird also recht „schlagartig“ erreicht.

Wie Abb. 10 zeigt, ist *Greedy* zwar besser als *Schnyder-and-bends*, im Vergleich zur Knotenzahl, die exponentiell mit der Baumtiefe zunimmt, aber mutmaßlich linear in der Baumtiefe.

Tiefe	1	2	3	4	5	6	7	8	9
Greedy	1.7	3.5	6.0	9.8	26.2	42.7	112.3	361.0	917.6
S-a-b	1.0	1.0	1.0	1.0	168.4	515.3	1546.6	4651.3	14001.4
Schnyder	2.2	6.4	19.1	57.3	171.8	515.6	1546.6	4651.3	14001.4
$\frac{Schnyder}{Greedy}$	1.3	1.8	3.2	5.8	6.6	12.1	13.8	12.9	15.3
Anz. Knoten	4	7	16	43	124	367	1096	3283	9844

Abb. 10: Kantenlängenverhältnisse auf vollständigem 3-Baum für *Greedy* und *Schnyder-and-bends*

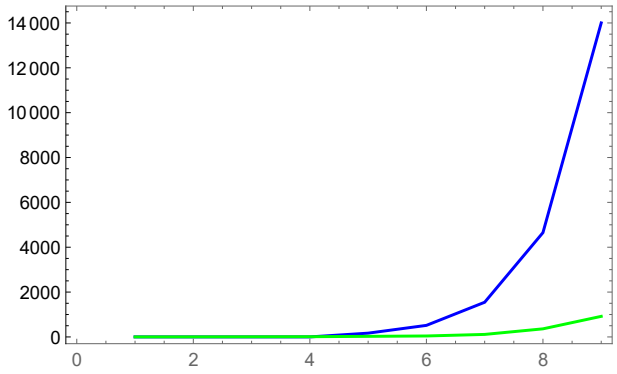


Abb. 11: *KLV* für Schnyder-and-bends und Greedy nach Baumtiefe

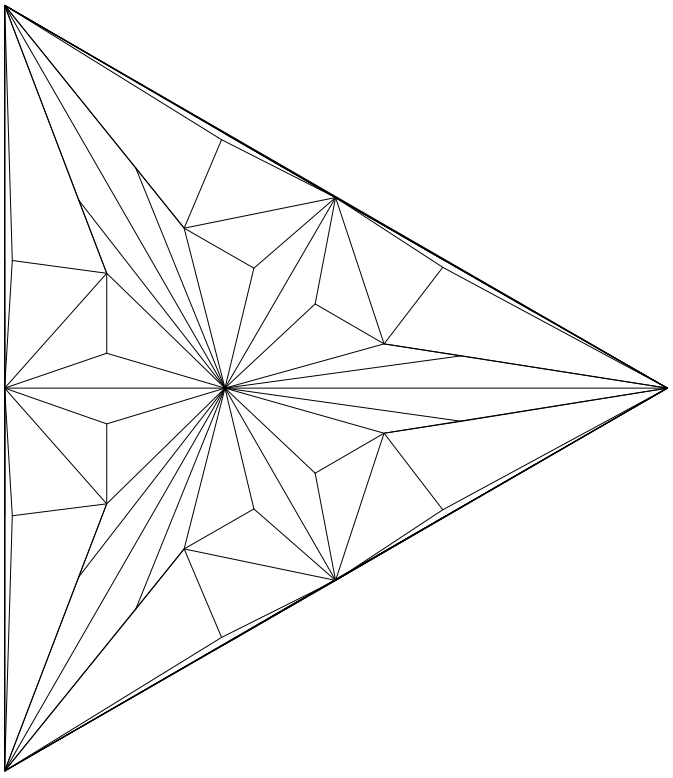


Abb. 12: vollständiger 3-Baum der Tiefe 4 (nach *Greedy*)

Literaturverzeichnis

[BF20] Manuel Borrazzo und Fabrizio Frati: On the planar edge-length ratio of planar graphs. *Journal of Computational Geometry*, 11:137–155, 2020, 10.20382/jocg.v11i1a6.

[EW90] Peter Eades und Nicholas Wormald: Fixed edge-length graph drawing is NP-hard. *Discrete Applied Mathematics*, 28:111–134, 1990, 10.1016/0166-218X(90)90110-X.

[LLL19] Sylvain Lazard, William Lenhart und Giuseppe Liotta: On the edge-length ratio of outerplanar graphs. *Theoretical Compututer Science*, 770:88–94, 2019, 10.1016/j.tcs.2018.10.002.

[zFL20] Václav Bla˘zej, Jiří Fiala und Giuseppe Liotta: On the edge-length ratio of 2-trees. *International Symposium on Graph Drawing and Network Visualization*, 28:85–98, 2020, 10.1007/978-3-030-68766-3.

Anhang

Es folgt eine Auflistung der Funktionen, die in *ReaderWriter* und *Layouter* implementiert wurden mit jeweils kurzer Erklärung. Manche Funktionen wurden am Ende nicht mehr gebraucht, diese sind mit "obsolet" gekennzeichnet.

ReaderWriter

- *GraphWA* read(std::string file, bool thin, bool withDrawing)*: Einlesen aus einem JSON-File in ein *GraphWA**-Objekt
- *int writeJson(GraphAttributes GA, std::string file, int w, int h, int b)*: Abspeichern eines *GraphAttributes*-Objekts in einen JSON-File
- *ogdf::node idToNode(ogdf::GraphAttributes GA, int id)*: selbsterklärend, Unterroutine für *read*.
- *ogdf::edge idsToEdge(ogdf::GraphAttributes GA, int source, int target)*: selbsterklärend, obsolet
- *void addBend(ogdf::GraphAttributes* GA, double x, double y, ogdf::edge e, bool reverse)*: setzt Knick in (x, y), Unterroutine für *read*.

Layouter

- Funktionen zur Konvertierung zwischen verschiedenen Datenstrukturen: *edgeToPolyline*, *edgeToLine*, *edgeListToPolyline*, *listPureToList*, *doubleToString*, *invertList*, *faceToPolygon*.
- Geometrische Funktionen (Funktionsparameter gekürzt): *getLength(DPoint p1, DPoint p2)*, *getLength(edge e)*, *getAngle(edge e, bool fromSource)*, *getNeighbourEdge(edge shortest, bool leftNeighbour, bool fromSource)*, *getBoundingRect(DPolygon polygon)*, *samePoints(DPoint p1, DPoint p2)*, *closePoints(DPoint p1, DPoint p2, double epsilon)*, *isPointXCentral(DPoint linePoint1, DPoint linePoint2, DPoint point)*, *bool isPointYCentral(DPoint linePoint1, DPoint linePoint2, DPoint point)*, *isPointOnLine(DPoint linePoint1, DPoint linePoint2, DPoint point)*, *isPointCentralBelowLine(DPoint linePoint1, DPoint linePoint2, DPoint point)*, *getTriangleArea(double a, double b, double c)*, *contains(DPoint point, DPolyline face)*, *intersect(DPoint source1, DPoint target1, DPoint source2, DPoint target2)*
- *void scale(GraphAttributes* GAPointer, double xBound, double yBound)*: Skaliert alle Knotenpositionen so hoch (oder runter), dass das vorgegebene Grid ausgenutzt wird.
- *edge getExtremeEdge(GraphAttributes* GA, bool shortest)*: Gibt längste oder kürzeste Kante zurück.

- *double getExtremeIncidentEdgeLength(GraphAttributes* GA, node v, bool shortest)*: Gibt Länge der längsten oder kürzesten Kante zurück, die zu v inzident ist.
- *List<edge> getEdgeFaceContour(GraphAttributes* GA, Graph* G, edge e, bool* isSourceCircle, bool* isTargetCircle, List<edge>* leftContour, List<edge>* rightContour, bool sourceCircle, bool targetCircle)*: Gibt Liste der Kanten der beiden zu e inzidenten Facetten zurück.
- *bool getBiggestTriangle(GraphAttributes* GA, DPolyline line, DPoint* point, edge shortest, bool leftContour)*: Speichert den Punkt, über den das größte Dreieck aufgespannt wird in *point*.
- *bool checkEdgeIntersection(DPolyline line, DPoint nextPoint1, DPoint from, DPoint pS, DPoint pT, bool pTPassedForCandidate, double* maxArea, DPoint* point, bool leftContour)*: Unterroutine für *getBiggestTriangle* für Schnitt einer Geraden mit einem Kantensegment (siehe Abbildung 2b).
- *bool makeBends(GraphAttributes* GA, edge shortest, double longest, DPoint triPoint, int bendsAllowed)*: Legt die Knicke in *shortest*.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt zu haben.

Würzburg, den 12. August 2021

. . . . . .
Michael Leeming