Karlsruhe Institute of Technology

Effiziente Berechnung guter Joggingrouten

Bachelorarbeit
von

Tobias Zündorf

An der Fakultät für Informatik
Institut für Theoretische Informatik

Erstgutachter:	Prof. Dr. Dorothea Wagner
Zweitgutachter:	Prof. Dr. Peter Sanders
Betreuende Mitarbeiter:	Dipl.-Inform. Andreas Gemsa
	Dipl.-Inform. Thomas Pajor

Bearbeitungszeit: 20. Juli 2012 – 19. Oktober 2012

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Kalsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Karlsruhe, 19. Oktober 2012

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Verwandte Arbeiten . . . 2
- 1.3 Gliederung der Arbeit . . . 2

2 Grundlagen **3**
- 2.1 Begriffe . . . 3
- 2.2 Komplexitätsklassen . . . 5

3 Theoretische Betrachtung **7**
- 3.1 Komplexitätsanalyse . . . 7

4 Effiziente Algorithmen **13**
- 4.1 Problemvariationen . . . 13
- 4.2 Facetten basierte Algorithmen . . . 14
  - 4.2.1 Bestimmung der Facetten . . . 16
  - 4.2.2 Hilfsdatenstruktur – FaceSet . . . 18
  - 4.2.3 Greedy-Algorithmen . . . 19
  - 4.2.4 Optimierungen . . . 21
  - 4.2.5 Glättungstrategien . . . 23
- 4.3 Teilwege-Algorithmen . . . 25
  - 4.3.1 Joggingroute durch drei Wegpunkte . . . 26
  - 4.3.2 Joggingroute durch vier Wegpunkte (TW4) . . . 29
  - 4.3.3 Joggingroute mit bidirektionaler Suche (TWB) . . . 31
  - 4.3.4 Verbesserungen . . . 32
  - 4.3.5 Implementierungsdetails . . . 34

5 Evaluierung der Algorithmen **37**
- 5.1 Datenaufbereitung . . . 37
- 5.2 Auswertung der Parameter . . . 39
  - 5.2.1 Parameter des FacesGreedy-Algorithmus . . . 39
  - 5.2.2 Parameter der Teilwege-Algorithmen . . . 41
- 5.3 Vergleich der vorgestellten Algorithmen . . . 44
- 5.4 Fallbeispiele . . . 48

6 Zusammenfassung **53**

Literaturverzeichnis **55**

1. Einleitung

1.1 Motivation

Jogging ist ein weit verbreiteter Freizeitsport, den so gut wie jeder ausüben kann. Es gibt dabei kaum Voraussetzungen, abgesehen davon, dass man geeignetes Schuhwerk tragen sollte, kann es jederzeit und an jedem Ort losgehen. Das einzige, was man nun noch wissen sollte, ist, entlang welcher Route man überhaupt joggen möchte.

Eine Joggingroute ist eine Folge von Straßen, die ein Jogger zu Trainingszwecken oder einfach nur zum Spaß abläuft. Dabei ist es für den Jogger von Interesse, wie lang die gelaufene Tour ist, damit ihre Länge zum Beispiel zu seiner körperlichen Leistungsfähigkeit passt. Natürlich ist nicht nur die Länge der Strecke ein wichtiges Kriterium. Als Jogger möchte man zusätzlich noch auf gut geeignetem Untergrund laufen und eine ansprechende Umgebung rechts und links der Strecke sehen können.

Diese verschiedenen Punkte, die zu beachten sind, machen es nicht leicht, eine geeignete Route manuell durch den Blick auf eine Karte zu finden. Aus diesem Grund gibt es bereits im Internet zahlreiche Angebote, die es den Benutzern erlauben, ihre bevorzugten Joggingrouten, mit anderen zu teilen. Jogmap[1] ist ein Beispiel für eine solche Seite und zeigt, dass durchaus bei einer größeren Personengruppe das Interesse besteht schöne Joggingrouten zu finden.

Dementsprechend wäre es schön eine Möglichkeit zu besitzen, neue Routen von einem Computer generieren zu lassen. Dies lohnt sich besonders, wenn sich die Anforderungen an eine gute oder passende Joggingroute häufig ändern. Möchte man zum Beispiel einmal eine längere Strecke als üblich laufen, so könnte man einfach den Computer eine neue Route, der gewünschten Ziellänge, berechnen lassen. Auch wenn man auf Reisen oder in einer, einem selbst, unbekannten Stadt ist, aber trotzdem trainieren will, kann es von Vorteil sein, wenn man direkt geeignete Strecken berechnen kann.

In dieser Arbeit befassen wir uns deshalb damit, Algorithmen zur effizienten Berechnung solcher Joggingrouten zu entwickeln. Dazu definieren wir formal was eine Joggingroute ist. Wir werden die Komplexität, des Problems, solche Routen zu finden analysieren und zwei verschieden heuristische Verfahren zur Berechnung solcher Joggingrouten vorstellen.

[1] http://www.jogmap.de/civic4/?q=streckensuche

1.2 Verwandte Arbeiten

Ein erster trivialer Ansatz, solche Routen zu berechnen wäre, zunächst alle Kreise in einem gegebenen Straßengraphen zu berechnen und dann unter diesen Kreisen solche Routen zu suchen, die verschiedene Randbedingungen optimieren. Über das Auffinden aller Kreise in einem Gegebenem Graphen existieren schon einige wissenschaftliche Arbeiten. H. Liu und J. Wang haben einen Algorithmus vorgeschlagen, der alle Kreise der Länge k in $\mathcal{O}((n + m)(c + 1))$ berechnet, wobei c die Anzahl der Kreise mit Länge $\leq k$ und k die Anzahl der Knoten im Kreis ist [LW06].

Derartiges Vorgehen ist aber für unser Problem nicht praktikabel, da die Anzahl der in einem Graphen existierenden Kreise zu hoch ist, womit auch die Benötigte Zeit um alle Kreise zu betrachten zu hoch ist. Betrachten wir sogar nur eine eingeschränkte Familie von Graphen wie planare Graphen, so ist die Anzahl der Möglichen Kreise schon exponentiell in der Größe des Graphen. K. Buchin et al. haben gezeigt, das Familien von planaren Graphen existieren, die für Graphen mit n Knoten mindestens $2{,}4262^n$ Kreise enthalten [BKK+07].

Die Anzahl der Kreise, die wir dementsprechend in einem Straßengraphen (welcher noch nicht einmal planar ist) erwarten können ist also viel zu hoch, als das es effizient möglich wäre alle Kreise zu betrachten. Hier ist es sinnvoll Verfahren zu betrachten, die direkt einzelne Kreise berechnen. R. Yuster und U. Zwick haben eine Algorithmus vorgestellt, der einen Kreis der Länge k, sofern dieser existiert, in erwartet $\mathcal{O}(2^k m)$ und deterministisch $\mathcal{O}(2^k m \log n)$ findet [YZ95]. Hierbei bezeichnet aber die Länge eine Pfades die Anzahl der in ihm enthaltenen Kanten. Darüber hinaus wird, falls mehrere Kreise der Länge k existieren nur einer von diesen gewählt.

Ließe sich dieser Algorithmus für allgemeine Längenfunktionen Adaptieren, und könnten wir bei mehreren gleichwertigen Kreisen beeinflussen welcher Kreis tatsächlich zurückgegeben wird, so wäre die Laufzeit immer noch mindestens exponentiell in der Gesuchten Gesamtlänge des Kreises, was kein effizienter Algorithmus ist.

Wir suchen aber einen Kreis der bezüglich einer bestimmten Metrik eine feste Länge besitzt, zusätzlich einen ausgezeichneten Knoten enthält, und verschiedene Nebenbedingungen optimiert. Zu dieser Problemstellung gibt es bisher keine Vorarbeiten.

1.3 Gliederung der Arbeit

Wir werden zunächst in Kapitel 2 einige grundlegende Begriffe und Konzepte aus der Theoretischen Informatik einführen, auf die wir im weiteren Verlauf der Arbeit zurückgreifen werden. Anschließend werden wir in Kapitel 3 das Problem, eine Joggingroute zu finden, formal genau definieren und die Komplexität des Problems analysieren. Wir werden auch untersuchen was genau eine schöne oder ansprechende Joggingroute auszeichnet. Im Laufe der Arbeit werden wir drei Kriterien beschreiben, mit denen sich die Qualität einer Joggingroute messen lässt.

In Kapitel 4 stellen wir zwei heuristische Verfahren vor, die entsprechend den von uns vorgestellten Kriterien schöne Joggingrouten in effizienter Weise berechnen. Dabei werden wir grundsätzlich zwei verschiedene Ansätze verfolgen. Zum einen werden wir einen Ansatz präsentieren, der Joggingrouten auf der Grundlage von Facetten zusammenstellt. Zum anderen werden wir Joggingrouten auf der Grundlage von kürzesten Wegen berechnen.

In Kapitel 5 werden wir die verschiedenen Algorithmen evaluieren. Dabei werden wir schauen, wo die Stärken und schwächen der einzelnen Algorithmen liegen, und wir werden die Leistung der einzelnen Algorithmen miteinander vergleichen.

Wir schließen die Arbeit in Kapitel 6 mit einer kurzen Zusammenfassung unserer Ergebnisse und einem Ausblick auf weiterführende Problemstellungen ab.

2. Grundlagen

In diesem Kapitel werden einige grundlegende Konzepte, die für das Verständnis der Arbeit nötig sind, vorgestellt.

2.1 Begriffe

Die folgenden Begriffe werden wir im Laufe der Arbeit durchgängig unter den hier beschriebenen Bedeutungen verwenden.

Graphen

Ein *einfacher Graph* $G = (V, E)$ ist ein Tupel bestehend aus einer Knotenmenge V und einer Kantenmenge E. Ein Graph heißt *ungerichtet*, wenn seine Kanten in beide Richtungen verfolgt werden können. In dieser Arbeit werden, sofern nicht explizit anders gefordert, Graphen immer als ungerichtet vorausgesetzt. Dementsprechend ist E also eine Menge von zweielementigen Teilmengen von V. Ein solcher Graph kann keine Schlaufen enthalten.

Wenn für zwei Knoten $v, w \in V$ die Kante $\{v, w\} \in E$ existiert, so heißen die beiden Knoten v und w zueinander *adjazent*. Ist $v \in V$ ein Knoten und $e \in E$ eine Kante mit der Eigenschaft $v \in e$, so heißen der Knoten v und die Kante e zueinander *inzident*.

Die Anzahl der Knoten in einem Graph wird im allgemeinen mit $n := |V|$ bezeichnet. Die Anzahl der Kanten in einem Graph wird mit $m := |E|$ bezeichnet. Der *Grad* eines Knoten $v \in V$ ist die Anzahl der zu ihm inzidenten Kanten, es gilt: $\deg(v) := |\{e \in E \mid v \in e\}|$.

Eine *Einbettung* eines Graphen ist eine Zeichnung des Graphen in der zweidimensionalen Ebene. Hierbei werden Knoten als Punkte und Kanten als Linien zwischen den Punkten, die ihre Endpunkte repräsentieren, gezeichnet. Eine Einbettung kann dementsprechend als eine Abbildung $\varphi\colon V \to \mathbb{R}^2$ beschrieben werden. Eine *kombinatorische Einbettung* eines Graphen ist gegeben durch zyklische Ordnungen der ausgehenden Kanten für jeden Knoten des Graphen.

Ein Graph heißt *planar*, wenn eine Einbettung des Graphen existiert, bei der alle Kanten durch Strecken repräsentiert werden und bei der sich keine zwei Kanten schneiden.

Ein *Multigraph* ist, im Gegensatz zu einem einfachen Graphen, ein Graph in dem zwei Knoten durch mehrere verschiedene Kanten verbunden sein können. Ist $G = (V, E)$ eine Multigraph, so ist E eine Multimenge über der Menge der zweielementigen Teilmengen von V.

Pfade

Ein *Pfad* P in einem Graphen $G = (V, E)$ ist eine Folge von Knoten aus V, mit der Eigenschaft, dass zwei in dem Pfad aufeinanderfolgende Knoten im Graphen durch eine Kante verbunden sind. Also ist $P = (p_1, \ldots, p_k)$ genau dann ein Pfad in G, wenn für alle $i \in \{1, \ldots, k\}$ gilt, dass p_i ein Element aus V ist und für alle $j \in \{1, \ldots, k-1\}$ gilt, dass $\{p_j, p_{j+1}\}$ ein Element aus E ist. Der Knoten p_1 wird in diesem Fall der *Startknoten* des Pfades genannt und p_k der *Endknoten*. Eine Kante $\{v, w\} \in E$ ist im Pfad P enthalten, wenn ein i existiert, so dass $v = p_i$ und $w = p_{i+1}$ gelten.

Ist zusätzlich zu dem Graphen noch eine Gewichtsfunktion $c\colon E \to \mathbb{R}_+$ auf den Kanten gegeben, so lässt sich diese auf Pfade erweitern. Das Gewicht des Pfades ist dann definiert durch $c(P) := \sum_{i=1}^{k-1} c(\{p_i, p_{i+1}\})$.

Sind $P = (p_1, \ldots, p_k)$ und $Q = (q_1, \ldots, q_j)$ zwei Pfade und der Endknoten von P ist gleich dem Startknoten von Q so können die beiden Pfade zu einem längeren Pfad *konkateniert* werden. Es gilt: $P \circ Q := (p_1, \ldots, p_k = q_1, \ldots, q_j)$.

Einen Pfad $P = (p_1, \ldots, p_k)$ bezeichnet man als *Kreis*, wenn zusätzlich die Bedingung $p_1 = p_k$ gilt.

Der Pfad P heißt *knotendisjunkt*, wenn in ihm kein Knoten mehrfach enthalten ist. Für knotendisjunkte Pfade gilt also: $\forall i, j \in \{1, \ldots, k\} : i \neq j \Rightarrow p_i \neq p_j$. Ein Kreis heißt knotendisjunkt, falls dies für alle Knoten mit Ausnahme des letzten gilt. Ein Pfad heißt *kantendisjunkt*, wenn keine Kante in ihm mehrfach enthalten ist, es gilt also $\forall i, j \in \{1, \ldots, k-1\} : i \neq j \Rightarrow \{p_i, p_{i+1}\} \neq \{p_j, p_{j+1}\}$

Sind $u, v \in V$ zwei Knoten, dann definieren wir $\mathrm{SP}_c(u, v)$ als den bezüglich einer Gewichtsfunktion c *kürzesten Pfad* von u nach v. Unter allen Pfaden mit Startkonten u und Endknoten v ist $\mathrm{SP}_c(u, v)$ also derjenige Pfad P für den $c(P)$ minimal ist.

Der *Abstand* zweier Knoten $u, v \in V$ ist definiert als $\mathrm{dist}(u, v) := c(\mathrm{SP}_c(u, v))$.

Facetten und Dualgraph

Für eine gegebene planare Einbettung eines planaren Graphen G unterteilen die Kanten des Graphen die Ebene in mehrere Flächen. Diese Flächen heißen *Facetten*. Es kann nun eine neuer Multigraph G^* konstruiert werden, dessen Knoten gerade die Facetten von G sind. Dieser Graph enthält genau dann eine Kante zwischen zwei Knoten, wenn die entsprechenden Facetten in G von einer gemeinsamen Kanten begrenzt werden. Der Graph G^* wird *Dualgraph* von G genannt. Der Dualgraph eines Graphen ist bereits durch eine kombinatorische Einbettung des Graphen eindeutig festgelegt.

Der *Rand* einer Facette ist der Kreis im Graphen, der alle Knoten enthält, welche an die Fläche der Facette angrenzen.

Ist auf dem zugrundeliegenden Graphen auch noch eine Gewichtsfunktion $c\colon E \to \mathbb{R}_+$ definiert, so lässt sich diese auf Facetten erweitern. Ist R der Rand der Facette f, so gilt $c(f) := c(R)$. In dem Fall wird $c(f)$ auch *Umfang* von f genannt.

Konvexe Hülle

Die *konvexe Hülle* einer endlichen Menge $M \subseteq \mathbb{R}$ von Punkten im zweidimensionalem Raum, ist das Polygon P mit der kleinsten Fläche, für das alle Punkte aus M und ihre Verbindungsstrecken in P enthalten sind. Es gilt also:

$$\forall p, q \in M : \overline{pq} \in P$$

2.2 Komplexitätsklassen

Die folgenden Komplexitätsklassen und Begriffe werden wir in der theoretischen Analyse verwenden.

Die Komplexitätsklasse $\mathcal{P}$

Die Komplexitätsklasse $\mathcal{P}$ umfasst alle Probleme, die von einer deterministischen Turingmaschine in polynomieller Zeit bezüglich der Eingabegröße n gelöst werden können. Für jedes Problem $\Pi \in \mathcal{P}$ existiert also ein Algorithmus, der das Problem löst, und es existiert ein Polynom p, so dass der Algorithmus eine Laufzeit von $\mathcal{O}(p(n))$ hat.

Die Komplexitätsklasse $\mathcal{NP}$

Die Komplexitätsklasse $\mathcal{NP}$ umfasst alle Probleme, die von einer *nichtdeterministischen* Turingmaschine in polynomieller Zeit, bezüglich der Eingabegröße, gelöst werden können. Es gilt $\mathcal{P} \subseteq \mathcal{NP}$, allerdings ist nicht klar, ob es sich hierbei um eine echte Teilmengenbeziehung handelt oder ob die beiden Komplexitätsklassen identisch sind [For09].

$\mathcal{NP}$-Vollständigkeit

Ein Problem Π heißt $\mathcal{NP}$-vollständig, wenn es in der Komplexitätsklasse $\mathcal{NP}$ liegt, also $\Pi \in \mathcal{NP}$ gilt, und sich alle anderen Probleme aus $\mathcal{NP}$ in polynomieller Zeit auf Π reduzieren lassen. Bis jetzt wurde noch kein deterministischer Algorithmus gefunden, der ein $\mathcal{NP}$-schweres Problem in polynomieller Zeit lösen kann. Würde nur für ein einziges $\mathcal{NP}$-vollständiges Problem ein polynomieller und deterministischer Lösungsalgorithmus gefunden, so wäre $\mathcal{P} = \mathcal{NP}$ [GJ90].

Um für ein neues Problem Π zu beweisen, dass dieses $\mathcal{NP}$-vollständig ist, müssen aber nicht alle Probleme der Klasse $\mathcal{NP}$ tatsächlich auf dieses reduziert werden. Es ist ausreichend, ein Problem, für das bekannt ist, dass es $\mathcal{NP}$-vollständig ist, auf Π zu reduzieren. Wegen der Transitivität polynomieller Reduktionen lassen sich dann automatisch auch alle anderen Probleme aus $\mathcal{NP}$ auf Π reduzieren.

Ein Problem heißt *stark* $\mathcal{NP}$-vollständig, wenn es unabhängig von der Größe der in der Eingabe vorkommenden Zahlen $\mathcal{NP}$-vollständig ist. Ist dies nicht der Fall, so heißt das Problem *schwach* $\mathcal{NP}$-vollständig. Dies ist äquivalent dazu, dass ein pseudopolynomieller Lösungsalgorithmus für das Problem existiert [GJ78]. Das bedeutet das der Algorithmus polynomiell in den Werte der numerischen Parameter ist und polynomiell in der Eingabgröße der restlichen Paramter ist.

3. Theoretische Betrachtung

Ausgehend von der Situation des Joggers, der für sein Training eine schöne Strecke sucht, wollen wir nun das Problem exakt definieren. Dabei ist zunächst unklar, wie eine *schöne* Strecke charakterisiert ist. Deshalb beschränken wir uns in der ersten Version der Problemdefinition darauf, die gewünschte Länge der Trainingsstrecke einzuhalten. Da es für einen Jogger keine Wege gibt, die er nur in eine Richtung passieren kann, modellieren wir das Straßennetz als ungerichteten Graphen. Die Knoten stellen hierbei Punkte auf der Karte dar und zwei Knoten sind genau dann durch eine Kante verbunden, wenn eine Straße existiert, welche die entsprechenden Punkte verbindet. Darüber hinaus benötigen wir noch eine Gewichtsfunktion auf den Kanten, welche die Länge der entsprechenden Straßen modelliert, und zwei Parameter für den Ausgangspunkt der Trainingsstrecke sowie deren gewünschte Länge. Diese Überlegungen führen uns zur folgenden Definition.

Definition 3.1. Jogger-Problem (JP)

Gegeben: *Ein Graph $G = (V, E)$, eine Gewichtsfunktion $c\colon E \to \mathbb{N}$, ein Startknoten $s \in V$ sowie die Ziellänge $l \in \mathbb{N}$.*

Gesucht: *Ein Pfad $P = (p_1, \ldots, p_k)$ in G mit $p_1 = p_k = s$ und $\sum_{i=1}^{k-1} c(\{p_i, p_{i+1}\}) = l$.*

3.1 Komplexitätsanalyse

Auf Grund der Ähnlichkeit des Jogger-Problems zu einigen anderen kombinatorischen Problemen wie Undirected Hamiltonian Circuit oder Subset Sum, für die bekannt ist, dass sie $\mathcal{NP}$-vollständig sind, vermuten wir, dass auch JP $\mathcal{NP}$-vollständig ist.

Theorem 3.2. *Das Jogger-Problem ist $\mathcal{NP}$-vollständig.*

Um die $\mathcal{NP}$-Vollständigkeit zu beweisen, benutzen wir das Hamiltonkreisproblem, da die Eigenschaften eines Hamiltonkreises denen einer Lösung für das Jogger-Problem sehr ähnlich sind.

Definition 3.3. Undirected Hamiltonian Circuit (UHC)

Gegeben: *Ein Graph $G = (V, E)$.*

Gesucht: *Ein Pfad $P = (p_1, \ldots, p_k)$ in G mit $p_1 = p_k$, $k = |V| + 1$ und $\forall v \in V\ \exists i \in \{1, \ldots, k\} : v = p_i$.*

Eine Lösung P des UHC Problems wird auch *Hamiltonkreis* genannt.

Das Undirected Hamiltonian Circuit Problem gehört zu Karps 21 $\mathcal{NP}$-vollständigen Problemen [Kar72]. Eine Reduktion von UHC auf das Jogger-Problem eignet sich somit, um die $\mathcal{NP}$-Vollständigkeit des Jogger-Problems zu zeigen.

Beweis zu Theorem 3.2. Wir führen im folgenden den $\mathcal{NP}$-Vollständigkeitsbeweis in zwei Teilen. Zunächst zeigen wir die Zugehörigkeit des Jogger-Problems zur Komplexitätsklasse $\mathcal{NP}$. Anschließend reduzieren wir UHC auf das Jogger-Problem.

1. Es gilt Jogger-Problem $\in \mathcal{NP}$, denn für einen gegebenen Pfad $P = (p_1, \ldots, p_k)$ kann in linearer Zeit überprüft werden, ob P ein gültiger Pfad in G ist. Auch die Summe $\sum_{i=1}^{k} c(\{p_i, p_{i+1}\})$ kann in linearer Zeit berechnet und mit l verglichen werden. Somit lässt sich auch in linearer Zeit überprüfen, ob P eine gültige Lösung ist.

2. Der Grundgedanke der folgenden Reduktion ist es, die Kantengewichte und die Länge so zu wählen, dass jede korrekte Lösung des Jogger-Problems auf einen Hamiltonkreis zurückgeführt werden kann. Wir versuchen dies zu erreichen, indem wir zu einem gegebenen Graphen $G = (V, E)$ die Kantengewichte so wählen, dass sich aus der Länge eines beliebigen Pfades, die in ihm enthaltenen Knoten rekonstruieren lassen. Da aber die enthaltenen Knoten nicht direkt zur Länge des Pfades beitragen, konstruieren wir einen neuen Graphen G', der für jeden Knoten $v \in G$ zwei neue Knoten (*in* und *out*) enthält, die jeweils durch eine Kante verbunden sind. Diese Kante und das Gewicht, das wir ihr noch zuordnen werden, repräsentiert nun den Knoten v. Wie in Abbildung 3.1 enthält der Graph G' zusätzlich für jede Kante $\{v_i, v_j\}$ des ursprünglichen Graphen zwei neue Kanten $\{\text{in}_i, \text{out}_j\}$ und $\{\text{in}_j, \text{out}_i\}$, welche für die Kanten in G stehen.
   Wir wählen nun die Kantengewichte so, dass in einem Stellenwertsystem mit geeigneter Basis b jede Kante, die einen Knoten repräsentiert, das Gewicht $c(\{\text{in}_i, \text{out}_i\}) = b^i$ besitzt. Enthält ein Pfad nun weniger als b Kanten, so kommt es bei der Summierung ihrer Längen zu keinerlei Überträgen. Deshalb kann aus der Darstellung der Länge des Pfades zur Basis b genau abgelesen werden, welche Kante wie oft im Pfad enthalten ist. Für einen Graphen mit n Knoten gilt, dass ein gültiger Hamiltonkreis genau n Kanten enthält. Wir können also als Basis $b = n+1$ wählen. Nun müssen wir noch sicherstellen, dass ein Pfad, der das Jogger-Problem löst, auch maximal n Kanten enthält. Dazu erhöhen wir das Gewicht jeder Kante um b^n und wählen $l < b^{n+1}$.
   Für eine konkrete UHC Instanz $G = (V, E)$ mit $V = \{v_0, \ldots, v_{n-1}\}$ wählen wir nach unseren Vorüberlegungen die Basis $b := 2n + 1$, des weiteren definieren wir $c_{\text{vertex}}(i) := b^n + b^i$ und $c_{\text{edge}} := b^n$. Die zugehörige Jogger-Problem Instanz ist nun gegeben durch:

$$
\begin{aligned}
G' &:= (V', E') \\
V' &:= \{\text{in}_0, \ldots, \text{in}_{n-1}, \text{out}_0 \ldots, \text{out}_{n-1}\} \\
E' &:= \{\{\text{in}_i, \text{out}_i\} \mid i \in \{0, \ldots, n-1\}\} \cup \{\{\text{in}_i, \text{out}_j\} \mid \{v_i, v_j\} \in E\} \\
c(\{\text{in}_i, \text{out}_j\}) &:= \begin{cases} c_{\text{vertex}}(i) & \text{, falls } i = j \\ c_{\text{edge}} & \text{, falls } i \neq j \end{cases} \\
l &:= n \cdot c_{\text{edge}} + \sum_{i=0}^{n-1} c_{\text{vertex}}(i) = 2n(2n+1)^n + \sum_{i=0}^{n-1} (2n+1)^i \\
s &:= \text{out}_0
\end{aligned}
$$

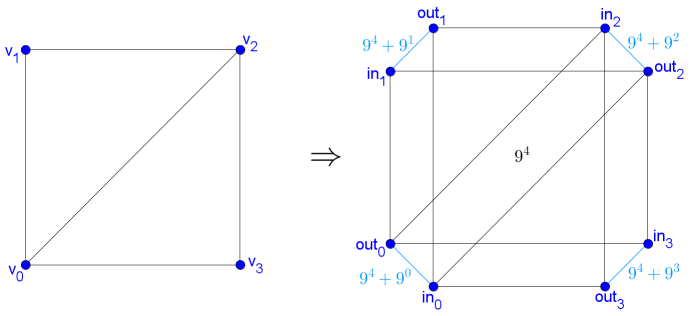


Abbildung 3.1: Transformation eines Beispielgraphen.

Es ist leicht ersichtlich, dass aus der Existenz eines Hamiltonkreises in G auch die Lösbarkeit des Jogger-Problems folgt. Denn für einen gegebenen Hamiltonkreis $H = (v_{i_0}, \ldots, v_{i_n})$ mit $i_0 = i_n = 0$ hat der Pfad $P := (\text{out}_{i_0}, \text{in}_{i_1}, \text{out}_{i_1} \ldots, \text{in}_{i_n}, \text{out}_{i_n})$ die Länge:

$$c(P) = \sum_{j=1}^{n} (c(\{\text{out}_{i_{(j-1)}}, \text{in}_{i_j}\}) + c(\{\text{in}_{i_j}, \text{out}_{i_j}\})) = \sum_{j=1}^{n} (c_{\text{edge}} + c_{\text{vertex}}(i_j)) = l$$

und ist somit eine Lösung des Jogger-Problems.
Sei nun umgekehrt $P = (p_0, \ldots, p_k)$ eine Lösung des Jogger-Problems, dann müssen wir zeigen, dass ein Hamiltonkreis in G existiert. Zunächst einmal gilt $k = 2n$, denn der längste Pfad mit $k = 2n - 1$ ist kürzer als l (3.1), und der kürzeste Pfad mit $k = 2n + 1$ ist länger als l (3.2).

$$\begin{aligned} \max_{P=(p_0,\ldots,p_{(2n-1)})} c(P) &= (2n-1) \cdot \max_{e \in E'} c(e) \\ &= (2n-1) \cdot ((2n+1)^n + (2n+1)^{(n-1)}) \\ &< 2n(2n+1)^n \\ &< 2n(2n+1)^n + \sum_{i=0}^{n-1} (2n+1)^i = l \end{aligned} \tag{3.1}$$

$$\begin{aligned} \min_{P=(p_0,\ldots,p_{(2n+1)})} c(P) &= (2n+1) \cdot \min_{e \in E'} c(e) \\ &= (2n+1) \cdot (2n+1)^n \\ &= 2n(2n+1)^n + (2n+1)^n \\ &> 2n(2n+1)^n + \sum_{i=0}^{n-1} (2n+1)^i = l \end{aligned} \tag{3.2}$$

Da der Pfad P also genau $2n$ Knoten enthält, kann bei der Berechnung der Länge zur Basis $b = 2n + 1$ kein Übertrag aufgetreten sein. Andererseits hat P die Länge l, welche in der Darstellung zur Basis b an jeder Stelle den Wert 1 hat. Die Summe der im Pfad enthaltenen Kanten kann somit nur gleich l sein, wenn jede Kante aus G', welche einen Knoten aus G repräsentiert, genau ein Mal in P enthalten ist. Somit existiert in G ein Hamiltonkreis, der die Knoten aus V in genau der Reihenfolge enthält, wie die ihnen entsprechenden Kanten aus E' in P enthalten sind.
Die Reduktion lässt sich in polynomieller Zeit berechnen, da der konstruierte Graph $2n$ Knoten und $2m + n$ Kanten enthält. Auch die Kantengewichte und die Länge lassen sich in polynomieller Zeit berechnen.

□

Im vorangegangenen Beweis fällt allerdings auf, dass die Werte der Kantengewichte mit b^n exponentiell in der Anzahl der Knoten n sind. Ergänzen wir die Problemstellung des Jogger-Problems um die Zusatzbedingung, dass alle Kantengewichte c sowie die Ziellänge l polynomiell in n sind, so lässt sich das Problem effizient lösen. Um die Existenz eines entsprechenden Algorithmus zu zeigen, orientieren wir uns an dem ähnlichen Problem Subset Sum, welches mittels dynamischer Programmierung pseudopolynomiell gelöst werden kann [GJ90].

Definition 3.4. Subset Sum

Gegeben: *Eine Menge $S \subseteq \mathbb{N}$ und ein Wert $v \in \mathbb{N}$.*

Gesucht: *Eine Teilmenge $A \subseteq S$ mit der Eigenschaft $\sum_{a \in A} a = v$.*

Die Ähnlichkeit zum Jogger-Problem wird deutlich, wenn man als Menge S die Menge der verschiedenen Kantengewichte wählt und $v = l$ setzt. Ein wichtiger Unterschied zum Jogger-Problem ist, dass zur Lösung A von Subset Sum jedes Element aus S nur einmal beitragen kann. Im Gegensatz dazu kann beim Jogger-Problem jede Kante beliebig oft genutzt werden. Ein weiterer Unterschied ist, dass bei Subset Sum die Elemente aus S beliebig kombiniert werden können, wohingegen beim Jogger-Problem die wählbaren Kanten durch Tatsache, dass wir einen Zusammenhängenden Kreis suchen, eingeschränkt werden. Trotzdem lässt sich das Jogger-Problem mit einem ähnlichen pseudopolynomiellen Algorithmus wie Subset Sum lösen.

Sei dazu $G = (V, E), c, s, l$ eine Jogger-Problem Instanz, dann definieren wir eine Funktion $Q \colon V \times \mathbb{Z} \to \{\text{true}, \text{false}\}$, die angibt, ob für einen Knoten $v \in V$ und eine Länge $d \in \mathbb{Z}$ ein Pfad der Länge d von s nach v existiert.

$$Q(v, d) := \exists P = (p_1, \ldots, p_k) : p_1 = s \wedge p_k = v \wedge c(P) = d$$

Eine Lösung für das Jogger-Problem existiert dementsprechend genau dann, wenn $Q(s, l) = \mathit{true}$ gilt. Da alle Kantenlängen positiv sind, ist klar, dass $Q(v, d) = \mathit{false}$ gilt, falls $d < 0$ ist. Auch für $d = 0$ lässt sich Q leicht berechnen, denn nur der Pfad, der nur aus dem Startknoten besteht, hat Länge 0 und endet in s. Sei nun $d > 0$ und $Q(v, d) = \text{true}$, es existiert also ein Pfad $P = (s, p_2, \ldots, p_{k-1}, v)$ mit Länge d. Dann existiert aber auch ein Pfad $P' = (s, p_2, \ldots, p_{k-1})$ mit Länge $d' = d - c(\{p_{k-1}, v\}) < d$ und somit gilt $Q(v, d) = Q(p_k, d - c(\{p_{k-1}, v\}))$. Die möglichen Werte von p_k sind hierbei eingeschränkt auf die zu v adjazenten Knoten. Dies führt uns zu einer rekursiven Darstellung von Q.

$$Q(v, d) = \begin{cases} \text{false} & , \text{falls } d < 0 \\ s = v & , \text{falls } d = 0 \\ \exists \{v, w\} \in E : Q(w, d - c(\{v, w\})) & , \text{falls } d > 0 \end{cases}$$

Im Algorithmus 3.1 benutzen wir nun eine Tabelle mit n Zeilen für die Knoten in G und $l + 1$ Spalten für die Längen von 0 bis l, um die Werte von Q zu berechnen. Entsprechend der rekursiven Formel können wir diese Tabelle nun Spalte für Spalte mit den Werten für $Q(v, d)$ füllen. Wir erhalten somit einen pseudopolynomiellen Lösungsalgorithmus mit einer Laufzeit von $\mathcal{O}(l \cdot n \cdot m)$ für das Jogger-Problem. Da das Jogger-Problem $\mathcal{NP}$-vollständig ist und ein pseudopolynomieller Lösungsalgorithmus existiert, ist das Jogger-Problem nur schwach $\mathcal{NP}$-vollständig.

Wir wollen als nächstes eine Modifikation des ursprünglichen Jogger-Problems betrachten. Ein Pfad, der JP löst, soll nun zusätzlich kantendisjunkt sein. Übertragen auf den Jogger bedeutet dies, dass er keine Teilstrecke zweimal laufen muss.

Algorithm 3.1: Jogger-Problem, Dynamisches Programm

Input: Graph $G = (V = \{v_0, \ldots, v_{n-1}\}, E), c, s = v_k, l$
Output: $Q(s, l)$

```
1  Q ← zweidimensionales Array von Wahrheitswerten der Länge n und Breite l + 1;
2  for i ← 0 to n − 1 do
3      Q[i][0] ← (iequalsk);
4  for d ← 1 to l do
5      for i ← 0 to n − 1 do
6          Q[i][d] ← false;
7          foreach {v_i, v_j} ∈ E do
8              d' ← d − c({v_i, v_j});
9              if d' ≥ 0 and Q[j][d'] = true then
10                 Q[i][d] ← true;
11 return Q[k][l];
```

Definition 3.5. Kantendisjunktes Jogger-Problem (KdJP)

Gegeben: *Ein Graph $G = (V, E)$, eine Gewichtsfunktion $c\colon E \to \mathbb{N}$, ein Startknoten $s \in V$ sowie die Ziellänge $l \in \mathbb{N}$.*

Gesucht: *Ein Pfad $P = (p_1, \ldots, p_k)$ in G mit $p_1 = p_k = s$, $\sum_{i=1}^{k-1} c(\{p_i, p_{i+1}\}) = l$ und $\forall i, j \in \{1, \ldots, k-1\}, i \neq j : \{p_i, p_{i+1}\} \neq \{p_j, p_{j+1}\}$.*

Es ist leicht zu zeigen, dass dieses Problem nicht nur $\mathcal{NP}$-vollständig, sondern sogar stark $\mathcal{NP}$-vollständig ist.

Theorem 3.6. *Das* Kantendisjunkte Jogger-Problem *ist stark $\mathcal{NP}$-vollständig.*

Beweis zu Theorem 3.6. Wir gehen bei diesem Beweis analog zum $\mathcal{NP}$-Vollständigkeitsbeweis von JP vor, und beschreiben wieder eine Reduktion von UHC.

1. Es gilt KdJP $\in \mathcal{NP}$, denn für einen gegebenen Pfad $P = (p_1, \ldots, p_k)$ kann in linearer Zeit überprüft werden, ob P ein gültiger, kantendisjunkter Pfad in G ist. Auch die Summe $\sum_{i=1}^{k} c(\{p_i, p_{i+1}\})$ kann in linearer Zeit berechnet und mit l verglichen werden. Somit lässt sich in linearer Zeit überprüfen, ob P eine gültige Lösung ist.
2. Zu einer gegebenen UHC Instanz $G = (V, E)$ konstruieren wir wieder einen Graphen, der für jeden Knoten $v_i \in V$ zwei neue Knoten (in_i und out_i) enthält. Auch bei der Modellierung der Kantengewichte benutzen wir wieder ein Stellenwertsystem zur Basis $b = 2n + 1$. Da wir schon wissen, dass eine Lösung immer kantendisjunkt ist, müssen wir aus der Länge eines Pfades nicht mehr ableiten können, ob eine bestimmte Kante, die einen Knoten repräsentiert, enthalten ist. Es ist ausreichend, feststellen zu können, wie viele Kanten, die Knoten repräsentieren, im Pfad enthalten sind. Wir wählen dementsprechend $c_{\text{vertex}} = b + 1$ und $c_{\text{edge}} = b$. Die restliche Konstruktion von $G' := (V', E')$ funktioniert entsprechend wie beim Beweis zu Theorem 3.2.

$$
\begin{aligned}
V' &:= \{\text{in}_0, \ldots, \text{in}_{n-1}, \text{out}_0 \ldots, \text{out}_{n-1}\} \\
E' &:= \{\{\text{in}_i, \text{out}_i\} \mid i \in \{0, \ldots, n-1\}\} \cup \{\{\text{in}_i, \text{out}_j\} \mid \{v_i, v_j\} \in E\} \\
c(\{\text{in}_i, \text{out}_j\}) &:= \begin{cases} c_{\text{vertex}} & , \text{falls } i = j \\ c_{\text{edge}} & , \text{falls } i \neq j \end{cases} \\
l &:= n \cdot (c_{\text{vertex}} + c_{\text{edge}}) = 4n^2 + 3n \\
s &:= \text{out}_0
\end{aligned}
$$

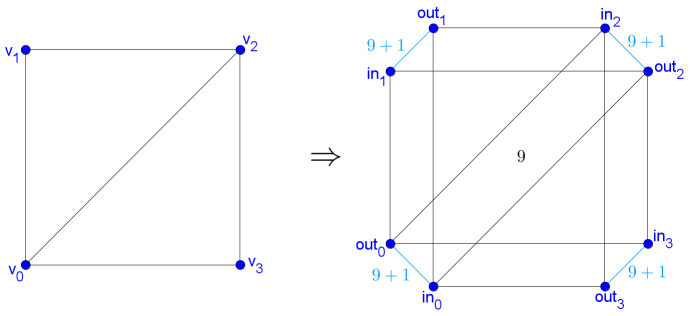


Abbildung 3.2: Transformation eines Beispielgraphen.

In Abbildung 3.2 ist ein Beispiel für die gegebene Reduktion zu sehen. Analog zum $\mathcal{NP}$-Vollständigkeitsbeweis vom JOGGER-PROBLEM kann gezeigt werden, dass ein gegebener Pfad der Länge l genau $2n$ Kanten enthalten muss. Dementsprechend kommt es im Stellenwertsystem zur Basis b bei der Berechnung der Länge des Pfades zu keinerlei Überträgen und, um die Länge l zu erreichen, muss der Pfad genau n Kanten, die einen Knoten repräsentieren, enthalten. Dies zeigt die Äquivalenz einer Joggingtour im transformierten Graphen zu einem Hamiltonkreis im ursprünglichen Graphen. Somit ist auch KdJP $\mathcal{NP}$-schwer. Da aber zusätzlich auch alle Kantengewichte sowie die Ziellänge l offensichtlich polynomiell in n beschränkt sind und UHC selbst stark $\mathcal{NP}$-vollständig ist, ist auch KdJP stark $\mathcal{NP}$-vollständig. □

4. Effiziente Algorithmen

Wie im vorangegangenen Kapitel gezeigt wurde, ist das Jogger-Problem $\mathcal{NP}$-vollständig, weshalb wir uns nicht mit der exakten Lösung des Problems beschäftigen. In diesem Kapitel stellen wir einige Heuristiken vor, die für reale Probleminstanzen gute Lösungen in praktikabler Laufzeit berechnen. Wir werden dabei zwei verschiedene Ansätze betrachten. Darüber hinaus ist in der Praxis für eine gute Joggingtour nicht nur die Länge entscheidend. Wir betrachten deswegen weitere Eigenschaften, die die *Schönheit* einer Joggingroute charakterisieren können.

4.1 Problemvariationen

Für einen Jogger ist bei der Wahl der Trainingsstrecke nicht allein die exakte Einhaltung der gewünschten Länge ausschlaggebend. Wie schon zu Beginn erwähnt, soll eine Trainingsstrecke auch möglichst schön sein. Was eine schöne Strecke auszeichnet, ist aber nicht direkt ersichtlich. Einige Eigenschaften, die zur Schönheit einer Strecke beitragen könnten, sind recht naheliegend. So ist es im allgemeinen attraktiver durch einen Wald oder ein begrüntes Gebiet zu laufen als durch ein Industriegebiet. Eine andere Eigenschaft, die die Schönheit einer Strecke beeinflussen könnte, ist der Anteil des Weges, der doppelt gelaufen werden muss. Ein möglichst kanten- und knotendisjunkter Pfad wäre dementsprechend schöner als eine Strecke, bei der man, nachdem 50% der Strecke absolviert wurde, umdreht und denselben Weg zurück läuft.

Die Tatsache, dass eine Vielzahl von Kriterien existieren, welche die Schönheit beeinflussen, und dass diese Kriterien auch nach individuellen Vorlieben variieren, führt uns zu einer erweiterten Definition des Jogger-Problems. Wir erlauben dabei, dass die Länge der gesuchten Tour um einen Faktor ε von l abweicht, um Spielraum für schöne Touren zu schaffen.

Definition 4.1. JOGGER-PROBLEM mit weichen Optimierungskriterien.

Gegeben: *Ein Graph $G = (V, E)$, eine Gewichtsfunktion $c\colon E \to \mathbb{N}$, ein Startknoten $s \in V$, die Ziellänge $l \in \mathbb{N}$, ein Parameter $\varepsilon \in [0, 1]$ und eine Menge von weichen Optimierungskriterien.*

Gesucht: *Ein Pfad $P = (p_1, \ldots, p_k)$ in G mit $p_1 = p_k = s$ und $c(P) \in \mathcal{I}(l, \varepsilon) := [(1 - \varepsilon)l, (1 + \varepsilon)l]$, der bezüglich den weichen Optimierungskriterien optimal ist.*

Es ist klar, dass auch diese Variante des JP $\mathcal{NP}$-vollständig ist, denn für $\varepsilon = 0$ und eine leere Menge von Optimierungskriterien erhalten wir gerade die ursprüngliche Definition des JP. Im weiteren Verlauf der Arbeit werden wir uns auf die folgenden Optimierungskriterien konzentrieren:

Sharing. Der gesuchte Pfad soll möglichst kantendisjunkt sein. Wir definieren dazu das *Sharing* eines Pfades als den Anteil des Pfades, der mehrfach enthalten ist. Konkret gilt für einen Pfad $P = (p_1, \ldots, p_k)$:

$$\text{sharing}(P) := \frac{\sum\limits_{i \in \mathcal{D}} c(\{p_i, p_{i+1}\})}{c(P)}$$

Wobei $\mathcal{D}$ die Menge aller Knotenindices i ist, für die die Kante $\{p_i, p_{i+1}\}$ mehrmals in P enthalten ist. Es gilt, das Sharing des Ergebnispfades zu minimieren.

Badness. Der gesuchte Pfad soll möglichst angenehm zu laufen sein. Sei ein Graph $G = (V, E)$ mit Kantenlängen $c\colon E \to \mathbb{N}$ gegeben. Wir definieren zusätzlich die *Badness* b als eine Abbildung von der Kantenmenge E auf das Intervall $[0, 1]$, welche die Kanten bezüglich ihrer Schönheit bewertet. Dabei gilt für zwei Kanten $e_1, e_2 \in E$, dass e_1 genau dann angenehmer zu laufen ist als e_2, wenn $b(e_1) < b(e_2)$ gilt. Wir erweitern die Definition der Badness auf beliebige Pfade in G. Sei dazu $P = (p_1, \ldots, p_k)$ ein Pfad in G, dann ist seine Badness gegeben durch:

$$b(P) := \frac{\sum\limits_{i=1}^{k-1} b(\{p_i, p_{i+1}\}) \cdot c(\{p_i, p_{i+1}\})}{c(P)}$$

Die Badness eines Pfades ist also der Mittelwert der Badness der enthaltenen Kanten, gewichtet nach der Länge dieser Kanten. Durch die Normalisierung durch die Länge des Pfades $c(P)$, liegt auch die Badness eines Pfades im Intervall $[0, 1]$. Auf diese Weise können auch Pfade unterschiedlicher Länge miteinander verglichen werden. Ziel ist es, die Badness des resultierenden Pfades zu minimieren.

4.2 Facetten basierte Algorithmen

In unserem ersten Ansatz nutzen wir Zusatzinformationen, die durch die Tatsache gegeben sind, dass der Graph G ein Straßennetz repräsentiert. Dies gibt uns eine Einbettung des Graphen in die zweidimensionale Ebene. Eine gegebene Joggingtour entspricht dabei einem Polygon in der Ebene. Betrachten wir eine Karte und eine dort eingezeichnete, initiale Joggingtour, so ist es als Mensch leicht, die gegebene Tour durch das Hinzufügen oder Entfernen einzelner Häuserblöcke zu modifizieren oder gar zu optimieren. Abbildung 4.1 zeigt eine solche initiale Tour und wie sie ein Mensch intuitiv erweitern könnte, um eine schönere Tour zu erhalten.

Durch das Hinzufügen einzelner Straßenblöcke kann also iterativ eine Route aufgebaut und verbessert werden. Diese Vorgehensweise werden wir mit Greedy-Algorithmen, die eine Joggingtour kontinuierlich um weitere Gebiete erweitern, nachempfinden. Als initiale Joggingtour kann der Pfad gewählt werden, welcher nur aus dem Startknoten besteht. Anschließend muss der Algorithmus Gebiete identifizieren, welche an die bestehende Tour angrenzen und dann eines dieser Gebiete für die nächste Modifikation auswählen. Dies wird so lange wiederholt, bis die Tour der vorgegebenen Länge näherungsweise entspricht.

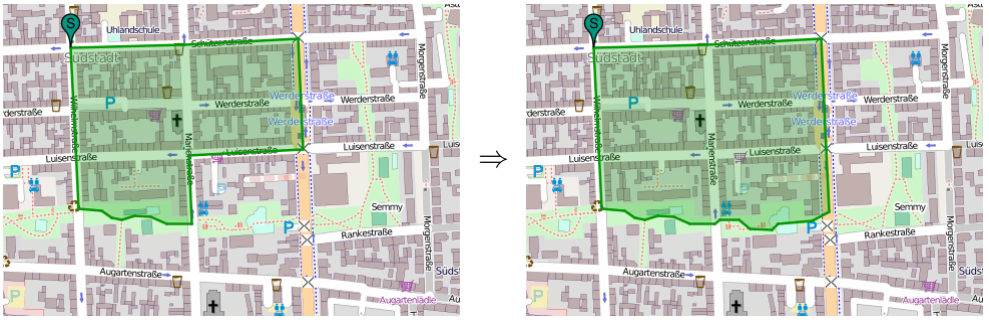


Abbildung 4.1: Intuitive Erweiterung einer Joggingtour.

Ein erstes Problem, das sich hierbei stellt, ist zu definieren, was genau diese möglichen Gebiete sind. Wir suchen also eine Entsprechung der Häuser- oder Straßenblocks in dem gegebenen Graphen G. Ist das Straßennetz, welches G repräsentiert, planar, so könnten die Gebiete einfach den Facetten von G entsprechen. Die an eine gegebene Joggingtour angrenzenden Facetten wären dann jene Facetten, die zu der gegebenen Tour inzident sind und deren Fläche sich nicht innerhalb des von der aktuellen Tour beschriebenen Polygons befindet.

Planarisierung. Ein Straßennetz enthält aber im allgemeinen Brücken und Unterführungen. Ein Graph der ein solches Straßennetz repräsentiert ist deshalb nicht planar und wir können keine Facetten bestimmen. Eine einfache Möglichkeit, einen Graphen mit gegebener Einbettung zu planarisieren, ist es, für jede Brücke oder Unterführung einen zusätzlichen Kreuzungsknoten im Graphen einzufügen. Wir würden hierdurch zwar einen planaren Graphen erhalten, allerdings würde ein Pfad in diesem Graphen nicht mehr unbedingt einer möglichen Joggingroute entsprechen, weshalb die Lösung nicht praktikabel ist.

Eine weitere Möglichkeit, den Graphen zu planarisieren, ist es, von jeweils zwei sich schneidenden Kanten eine aus dem Graphen zu entfernen. Auch hierdurch erhalten wir einen planaren Graphen, und Joggingrouten in diesem Graphen entsprechen auch in jedem Fall Routen im ursprünglichen Graphen. Ein Nachteil ist aber, dass Routen, die im ursprünglichen Graphen möglich waren, bei der Modifikation verloren gehen. Darüber hinaus stellt sich die Frage, welche der beiden sich schneidenden Kanten überhaupt gelöscht werden soll.

Wir lösen dieses Problem, indem wir zunächst für beide Möglichkeiten den entsprechenden Dualgraphen berechnen und anschließend die beiden Dualgraphen zu einem Graphen vereinen. Sei G also ein Graph mit Einbettung, für die sich die Kanten e_1 und e_2 schneiden. Dann berechnen wir zunächst die beiden Dualgraphen $G_1^* := (G \setminus \{e_1\})^*$ und $G_2^* := (G \setminus \{e_2\})^*$. Diese beiden Dualgraphen werden viele Knoten gemeinsam haben. Liegt nämlich auf dem Rand einer Facette weder e_1 noch e_2, so ist diese Facette sowohl in G_1^* als auch in G_2^* enthalten. Wir bilden nun den Dualgraphen $G^* = (V^*, E^*)$ als Vereinigung von G_1^* und G_2^*. Dabei gilt $V^* := V_1^* \cup V_2^*$ und zwei Facetten in V^* sind genau dann durch eine Kante verbunden, wenn es eine Kante in G gibt, welche auf dem Rand der beiden Facetten liegt. Wie ein normaler Dualgraph, lässt sich auch unsere Erweiterung allein auf der Grundlage einer kombinatorischen Einbettung bestimmen. Abbildung 4.2 zeigt ein Beispiel für die Konstruktion eines solchen Dualgraphen.

Im folgenden Abschnitt beschreiben wir ein Verfahren, mit dem wir die Facetten, wie wir sie hier beschrieben haben, auf der Grundlage eines gegebenen Graphen mit Einbettung, bestimmen. Dabei entfernen wir aber sich schneidende Kanten nicht von vornherein, sondern tun dies während der Identifizierung einzelner Facetten.

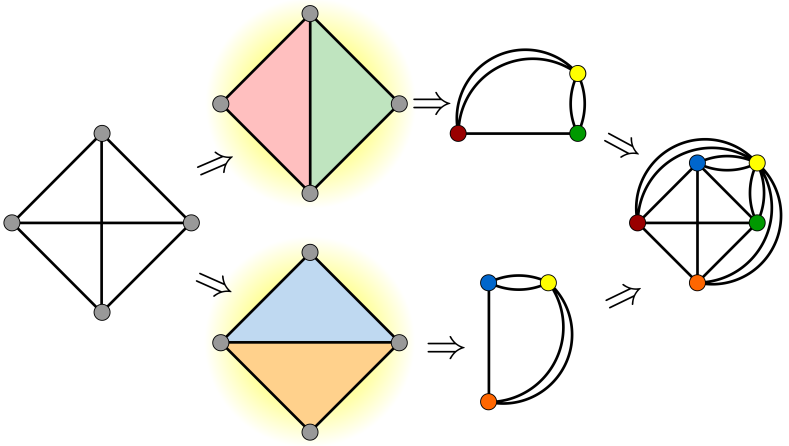


Abbildung 4.2: Ein Graph mit fester Einbettung und sein Dualgraph

4.2.1 Bestimmung der Facetten

In einem Vorverarbeitungsschritt berechnen wir zu einem gegebenen Graphen G und einer gegebenen Einbettung von G in die zweidimensionale Ebene einen *dualen* Graphen, dessen Knoten die Facetten von G repräsentieren. Zwei Facetten sind dabei auch in unserer Verallgemeinerung genau dann adjazent, wenn es mindestens eine Kante in G gibt, welche die beiden Facetten begrenzt. Um die Funktionsweise des Algorithmus zur Berechnung des Dualgraphen zu beschreiben, erklären wir zunächst, wie eine einzelne Facette erkannt wird, und gewinnen anschließend daraus den gesamten Algorithmus.

Zur Identifizierung einer einzelnen Facette im Graphen starten wir mit einer beliebigen Kante. Diese Kante liegt auf dem Rand der Facette, die identifiziert wird. Wir verfolgen von ihr ausgehend einen Pfad, der die Facette umschließt, bis wir wieder diese Kante erreichen. Dabei speichern wir jeweils die Knoten, die wir passieren, auf einem Stack, um so den Pfad, der die Facette umschließt, schrittweise aufzubauen. An jedem Knoten wählen wir jeweils jene Kante als nächste, welche mit der vorangegangenen Kante den kleinsten Winkel (im Uhrzeigersinn gerichtet) einschließt und fügen den über sie erreichbaren Knoten zum Stack hinzu. Anschließend überprüfen wir, ob sich die neu hinzugefügte Kante mit einer anderen bereits im Pfad enthaltenen Kante schneidet. Falls dem so ist, haben wir eine Kante passiert, die nicht zu dem Pfad, der die Facette umschließt, gehört. In diesem Fall wird der oberste Knoten wieder vom Stack entfernt und es wird die Kante mit dem nächst größeren Winkel gewählt.

Sackgassen. Da wir zum einen Joggingstrecken suchen, bei denen möglichst wenige Wege doppelt gelaufen werden müssen, und wir zum anderen die kleinstmöglichen Kreise suchen, ignorieren wir Sackgassen. Denn das Betreten einer Sackgasse führt zwangsläufig zu Teilstrecken, die doppelt gelaufen werden müssen. Sackgassen entsprechen Grad-1 Knoten im Graphen. Der Algorithmus zur Berechnung der umschließenden Pfade soll also Grad-1 Knoten ignorieren. Dies kann leicht erreicht werden, indem wir auch solche Kanten wieder vom Stack entfernen, die ein zweites Mal passiert werden. Beim Aufbau des Pfades wird so zunächst womöglich eine Kante gewählt, die zu einem Grad-1 Knoten führt. Im nächsten Schritt wird dieselbe Kante in die entgegengesetzte Richtung verfolgt. Da wir diese Kante entsprechend schon einmal passiert haben, wird der Grad-1 Knoten wieder vom Stack entfernt. Effektiv werden so ganze Bäume von Grad-1 Knoten ignoriert. Abbildung 4.3 zeigt, in welcher Reihenfolge die Kanten zur Identifizierung einer einzelnen Facette traversiert werden.

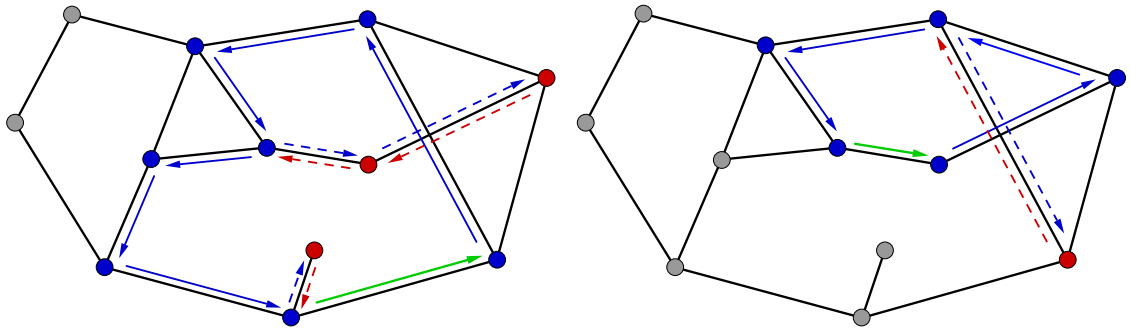


Abbildung 4.3: Betrachtete Kantenfolgen bei der Identifizierung einzelner Facetten, ausgehend von den Kanten in grün. Die blauen Knoten liegen am Ende des Algorithmus auf dem Stack, rote Knoten lagen zwischenzeitlich auf dem Stack, wurden aber wieder entfernt.

Wird beim Aufruf des späteren Algorithmus als Startpunkt s ein Knoten gewählt, der in einer Sackgasse liegt, so ist dieser Knoten zu keiner Facette inzident. Um den Algorithmus trotzdem anwenden zu können, wird der nächste Knoten im Graphen gesucht, der tatsächlich an eine Facette angrenzt. Für diesen Knoten s' kann dann normal eine Route berechnet werden. Zum Schluss wird die berechnete Route um den Pfad von s nach s' ergänzt.

Um alle Facetten eines Graphen zu identifizieren und seinen Dualgraphen aufbauen zu können, führen wir dieses Verfahren für jede Kante des gegebenen Graphen, und zwar in jede Richtung, durch. Hierbei kann es passieren, dass eine Facette mehrmals gefunden wird, Duplikate einer Facette können dementsprechend verworfen werden. Trotzdem ist es im allgemeinen nicht möglich, den Dualgraphen effizienter zu berechnen, da zum Beispiel ein vollständiger Graph $\Omega(m)$ Facetten besitzt, wobei m die Anzahl der Kanten ist.

Zusätzlich speichern wir beim Finden der Facetten für jede Kante aus G eine Liste mit den Facetten, in deren umschließendem Pfad diese Kante enthalten ist. Das Hinzufügen zu dieser Liste kann direkt beim Identifizieren der einzelnen Facetten passieren. Zuletzt müssen all diese Listen traversiert werden und die in ihnen enthaltenen Facetten paarweise durch eine Kante im Dualgraph verbunden werden.

Laufzeit. Der Algorithmus versucht für jede Kante des Graphen eine Facette zu identifizieren, demzufolge werden im schlimmsten Fall $\mathcal{O}(m)$ verschiedene Facetten berechnet. Der Pfad, der eine Facette umschließt, enthält maximal n Kanten bzw. Knoten, da sonst ein Knoten doppelt enthalten wäre. Trotzdem kann es passieren, dass beim Identifizieren dieses Pfades $\mathcal{O}(m)$ Kanten betrachtet werden, da es passieren kann, dass diese zunächst irrtümlich zum Pfad hinzugefügt werden. Für jede dieser Kanten, die zum Pfad hinzugefügt wird, muss überprüft werden, ob sie sich mit einer anderen Kante im Pfad schneidet. Da der Pfad zu keinem Zeitpunkt mehr als n Kanten enthält, hat dies einen Aufwand von $\mathcal{O}(n)$. Insgesamt ergibt sich somit eine Laufzeit von $\mathcal{O}(nm^2)$.

Alternativ wäre es auch möglich, zunächst alle Schnitte von Kanten im Graphen zu bestimmen. Dies ist in einer Laufzeit von $\mathcal{O}(m \log m + I)$ möglich, wobei I die Anzahl der Schnittpunkte ist [Bal95]. Anschließend wird für jeden gefundenen Schnitt einmal der Dualgraph berechnet, bei dem die eine am Schnitt beteiligte Kante entfernt wurde, und einmal der Dualgraph, bei dem die andere Kante fehlt. Für einen planaren Graphen kann der Dualgraph in $\mathcal{O}(n)$ berechnet werden, womit diese Vorgehensweise eine Gesamtlaufzeit von $\mathcal{O}(m \log m + nI)$ besitzt. Im schlimmsten Fall liegt dabei $I \in \mathcal{O}(m^2)$, wodurch auch dieses Verfahren in $\mathcal{O}(nm^2)$ liegt.

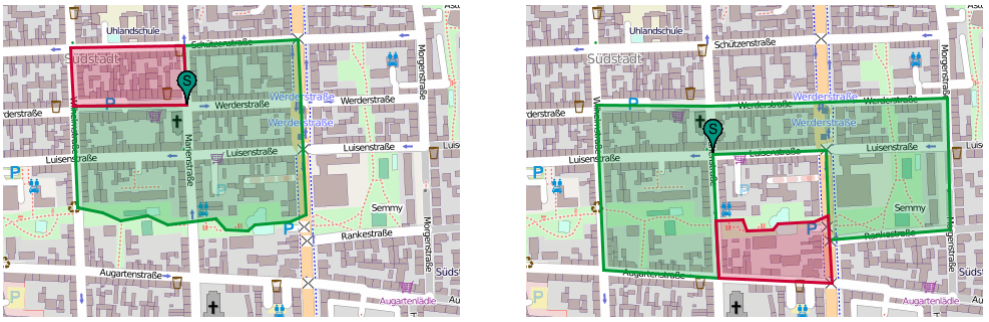


Abbildung 4.4: Beispiele für Facetten (Rot) die zu einer gegebenen Tour (Grün) nicht hinzugefügt werden dürfen.

Länge und Badness. Sind zusätzlich zum Originalgraphen Kantenlängen und eine Badnessfunktion gegeben, so kann auf den Kanten und den Knoten des Dualgraphen auch eine Längen- und Badnessfunktion definiert werden. Die Länge und Badness einer Kante im Dualgraphen sind dabei äquivalent zu der Länge und Badness der entsprechenden Kante im Ursprungsgraphen. Länge und Badness einer Facette ergeben sich aus den entsprechenden Werten für den die Facette umschließenden Pfad.

4.2.2 Hilfsdatenstruktur – FaceSet

Die Eingabe der Greedy-Algorithmen zur Berechnung der Joggingrouten enthält zusätzlich zum Graphen G, dem Startknoten s und der gesuchten Länge l auch den konstruierten Dualgraphen G^*. Im Laufe der Berechnung wird der aktuelle Pfad iterativ durch die Wahl weiterer Facetten, die zu ihm inzident sind, erweitert. Dabei treten zwei in Abbildung 4.4 gezeigte Probleme auf:

1. Es muss immer mindestens eine Facette geben, auf deren Rand s liegt, die noch nicht hinzugefügt wurde, da sonst s nicht mehr auf dem umschließenden Pfad liegt. Existiert nur noch eine solche Facette, so kann sie nicht hinzugefügt werden.

2. Die Fläche der bereits gewählten Facetten, darf keine *Löcher* enthalten, da sonst der umschließende Pfad in zwei nicht miteinander verbundene Teile zerfällt. Die Fläche enthält genau dann ein Loch, wenn der Graph ohne die ausgewählten Facetten nicht mehr zusammenhängend ist.

Um solche Probleme zu umgehen, verwenden wir eine Hilfsdatenstruktur, das *FaceSet*, welche es uns ermöglichen soll:

- eine Menge von bereits ausgewählten Facetten zu verwalten sowie einzelne Facetten hinzuzufügen oder zu entfernen, wobei das Hinzufügen oder Entfernen einer Facette zurückgewiesen wird, falls dies nicht möglich ist.

- eine Liste von Facetten anzufordern, welche potenziell zu der aktuellen Menge von Facetten hinzugefügt werden können.

- den Pfad abzufragen, der die aktuelle Menge von Facetten umschließt, sowie dessen Länge und Badness.

Die Datenstruktur kann die Menge $\mathcal{F}$ der enthaltenen Facetten, die Menge $\mathcal{A}$ der Facetten, die an Facetten aus $\mathcal{F}$ angrenzen, sowie die Länge l und Badness b des Pfades, der $\mathcal{F}$ umschließt, speichern. Diese Daten können beim Hinzufügen neuer Facetten leicht angepasst werden. Wird zum Beispiel eine Facette $f \in \mathcal{A}$ zu $\mathcal{F}$ hinzugefügt, so ergibt sich die neue Menge $\mathcal{F}'$ der enthaltenen Facetten zu $\mathcal{F}' = \mathcal{F} \cup \{f\}$. Die neue Menge der angrenzenden Facetten erhält man, indem man alle zu f adjazenten Facetten hinzunimmt und die bereits

in $\mathcal{F}'$ enthaltenen Facetten streicht. Es gilt $\mathcal{A}' = \mathcal{A} \cup \{f' \mid f' \text{ ist zu } f \text{ adjazent}\} \setminus \mathcal{F}'$. Die neue Länge des umschließenden Pfades ergibt sich aus der alten Länge plus der Länge von f minus der Längen aller Kanten, die zu f und einer Facette aus $\mathcal{F}$ inzident sind. Die Badness des neuen Pfades lässt sich äquivalent berechnen.

Die Funktionalität der Datenstruktur erfordert es zu entscheiden, ob eine Facette hinzugefügt werden darf oder ob dies wegen einem der zuvor genannten Randfälle nicht zulässig ist. Um dies zu berechnen, überprüfen wir zunächst, ob durch das Hinzufügen der Facette ein Loch entstehen würde. Sei dazu R der Rand der Facette f, die hinzugefügt werden soll, und P der Pfad, der $\mathcal{F}$ umschließt. Für einen Knoten $v \in R$ gilt nun entweder $v \in P$ oder $v \notin P$. Dasselbe gilt für die Kanten in R. Wir teilen R nun derart in Intervalle ein, dass für alle Knoten v und Kanten e eines Intervalls entweder $v, e \in P$ oder $v, e \notin P$ gilt. Lässt sich R auf diese Weise in genau zwei Intervalle zerlegen, so kann die Facette hinzugefügt werden, ohne dass ein Loch entsteht. Dies kann linear in der Länge des Pfades, der die Facette umschließt, berechnet werden. Hierbei ist es sehr wichtig, dass sowohl die Kanten als auch die Knoten des Randes der Facette betrachtet werden, da es Fälle gibt, in denen ein Intervall nur aus einem Knoten bzw. einer Kante besteht.

Zuletzt überprüfen wir noch, ob der Startknoten der Joggingtour durch das Hinzufügen der Facette aus dem Pfad herausfallen würde. Dies kann überprüft werden, während wir die zuvor genannten Intervalle bestimmen. Sei dazu u der Knoten, der in P vor s liegt, und v der Knoten, der in P nach s liegt. Der Startknoten fällt nun genau dann aus der Tour, wenn $\{u, s\} \in R \wedge s \in R \wedge \{s, v\} \in R$ gilt. Auf diese Weise lässt sich mit linearem Aufwand, in der Länge des Randes der Facette, entscheiden, ob die Facette hinzugefügt werden darf oder nicht. Beim Entfernen einer Facette sind die gleichen Randfälle zu beachten, zu deren Überprüfung kann die gleiche Vorgehensweise verwendet werden.

Wir werden im folgenden das hier beschriebene FaceSet in einigen Greedy-Algorithmen zur Lösung des Jogger-Problems benutzen.

4.2.3 Greedy-Algorithmen

Ein erster einfacher Greedy-Algorithmus, siehe Algorithmus 4.1, zur Lösung des Jogger-Problems initialisiert das FaceSet mit dem Dualgraphen und dem Startknoten. Anschließend wird jeweils jene Facette hinzugefügt, welche die beste Verbesserung des Badness Wertes verspricht. Dies wird wiederholt, bis die gesuchte Ziellänge l erreicht oder überschritten wurde. Auf Grund des Facettenansatzes ist der berechnete Pfad, sofern s nicht in einer Sackgasse liegt, auf jeden Fall knoten- und kantendisjunkt, hat also ein Sharing von 0. Die Länge der berechneten Route weicht, sofern überhaupt eine Route gefunden wurde, maximal um die Länge des längsten in G^* enthalten Facettenumfangs ab. Für die Güte der Badness der berechneten Tour kann allerdings keinerlei Garantie gegeben werden.

Laufzeit. Die Aufrufe von getSurroundingPathLength und getAdjacentFaces können nach unserer Modellierung des FaceSets in konstanter Zeit berechnet werden. Da zum FaceSet im schlimmsten Fall alle Facetten hinzugefügt werden, benötigt auch die Berechnung im schlimmsten Fall g Schritte, wobei g die Anzahl der Facetten ist. Auch die Berechnung in Zeile 6 betrachtet im schlimmsten Fall alle g Facetten. Die Berechnung von $\mathcal{F}.\text{Add}(f)$ ist linear in der Anzahl der Knoten des f umschließenden Pfades, der sicher kürzer als g ist. Der Algorithmus hat dementsprechend eine Laufzeit von $\mathcal{O}(g^2) = O(m^2)$.

Bei der Betrachtung einiger Ergebnisse des Algorithmus, wie in Abbildung 4.5a (Seite 21), fällt auf, dass die berechneten Joggingrouten eine sehr komplexe Struktur haben bzw. viele verschiedene Straßen benutzt werden. Auch dies entspricht intuitiv nicht der Vorstellung von einer guten Joggingtour, da es sehr schwer ist, sich die komplette Tour zu merken. Darüber hinaus ist in Abbildung 4.5a zu sehen, dass die gewählten Facetten alle in mehr

Algorithm 4.1: GreedyFaces

Input: Graph G, Dualgraph D von G, c, b, s, l
Output: $P = (s, p_1, \ldots, p_k, s)$

```
1 F ← FaceSet(D, s);
2 while F.getSurroundingPathLength() < l and F.getAdjacentFaces() ≠ ∅ do
3 |   A ← F.getAdjacentFaces();
4 |   foreach a ∈ A do
5 |   |   E_a ← {e ∈ E | e ist zu a und F inzident};
6 |   f ← arg min_{a∈A} (b(a) − 2 Σ_{e∈E_a} b(e));
7 |   F.Add(f);
8 return F.getSurroundingPath();
```

oder weniger unmittelbarer Nähe des Startknoten liegen. Auf Grund der Konzeption ist klar, dass der Algorithmus 4.1 keine Information, über die globale Struktur des Graphen G nutzt. Wünschenswert wäre es, wenn der Algorithmus Gebiete im Graphen mit niedriger Badness erkennen und sich zielstrebig in deren Richtung bewegen würde. Deshalb schlagen wir folgende Modifikation vor, bei der wir die Ausbreitungsrichtung des Algorithmus durch *Kräfte* steuern.

Der Algorithmus soll bei der Wahl der nächsten Facette, die hinzugefügt wird, von Facetten mit niedriger Badness angezogen werden und von Facetten mit hoher Badness abgestoßen werden. Kräfte modellieren wir dabei als zweidimensionale Vektoren, wobei der Betrag des Vektors dem Betrag der Kraft und die Ausrichtung des Vektors der Wirkungsrichtung der Kraft entspricht.

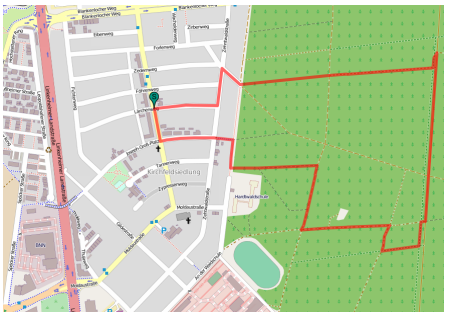
Bei der Modellierung der Kräfte orientieren wir uns im Groben an der Gravitations- oder Coulomb-Kraft. Wobei in dieser Analogie die Kräfte von den geometrischen Zentren der Facetten (center(f)) ausgehen sollen und die *Ladung* einer Facette von ihrer Badness und ihrem Umfang abhängt. Genau wie bei diesen Kräften, soll auch unsere Kraft quadratisch mit dem Abstand zu den Facetten abnehmen. Wir definieren die Kraft, welche die Facette f auf einen Punkt $p \in \mathbb{R}^2$ ausübt wie folgt:

$$\vec{\text{force}}(f, p) := \frac{(b(f) - 0.5) \cdot c(f)}{|\vec{d}|^2} \cdot \frac{\vec{d}}{|\vec{d}|} \text{ , mit } \vec{d} := p - \text{center}(f)$$

Um zu entscheiden, welche Facette als nächstes gewählt werden soll, muss für jede angrenzende Facette $f \in \mathcal{A}$ die Kraft, die von allen anderen Facetten im Dualgraph auf ihr Zentrum wirkt, berechnet werden. Damit diese Berechnung nicht in jedem Schleifendurchlauf aufs neue durchgeführt werden muss, bietet es sich an, die Kräfte zwischen allen Facetten initial zu berechnen. Dies führt allerdings zu einem Berechnungsaufwand mit Zeitkomplexität $\mathcal{O}(g^2)$, wobei g die Anzahl der im Graph enthaltenen Facetten ist. Wir präzisieren die Berechnung der Kräfte noch, indem wir sie zur Laufzeit, aber dafür nur für relevante Facetten ausführen. Ist eine Facette weiter als $l/2$ vom Startknoten entfernt, so ist klar, dass sie von einer Joggingtour, welche die Ziellänge l einhält, nicht erreicht werden kann. Die Berechnung der Kräfte kann also auf Facetten beschränkt werden, die nicht weiter als $l/2$ von s entfernt sind. Algorithmus 4.2 erweitert Algorithmus 4.1 um die hier beschriebenen Kräfte zur die Wahl der nächsten Facette.

Es bleibt noch zu beschreiben, wie genau die vorberechneten Kräfte bei der Wahl der nächsten Facette benutzt werden. Hierbei spielt nicht nur der Betrag der Kraft eine Rolle, sondern auch ihre Wirkungsrichtung. Diese Richtung muss mit der Richtung, in die die neue

(a) GreedyFaces
$l = 3303\text{m}, b = 75\%$



(b) GreedyFaces with Forces
$l = 3036\text{m}, b = 43\%$

Abbildung 4.5: Beispielergebnisse der Greedy-Algorithmen, gesuchte Länge: 3000m
Dabei ist l die Länge der berechneten Joggingroute und b ihre Badness.

Facette die bestehenden Facetten ergänzt, verglichen werden. Für die Wahl der nächsten Facette ist der Anteil der Kraft entscheidend, der in die Erweiterungsrichtung wirkt. Der Anteil der Kraft $\vec{force}$, der in Richtung des Vektors $\vec{d}$ wirkt ist gegeben durch $|\vec{force}| \cdot \cos(\measuredangle(\vec{force}, \vec{d}))$. Dies nutzen wir, um eine Variation des Greedy-Algorithmus zu erhalten.

Algorithm 4.2: GreedyFaces with forces

Input: Graph G, Dualgraph $D = (V = \{f_0, \ldots, f_{n-1}\}, E)$ von G, c, b, s, l, ε
Output: $P = (s, p_1, \ldots, p_k, s)$

1 Forces $\leftarrow$ Array von Force-Vektoren der Länge n;
2 M $\leftarrow \{i \mid l/2 \geq |s - \text{center}(f_i)|\}$;
3 **foreach** $i \in M$ **do**
4 $\quad$ Forces$[i] \leftarrow \sum_{j \in M} \vec{force}(f_i, \text{center}(f_j))$;
5 $\mathcal{F} \leftarrow$ FaceSet(D, s);
6 **while** $\mathcal{F}$.getSurroundingPathLength() $< l$ **do**
7 $\quad \mathcal{A} =\leftarrow \mathcal{F}$.getAdjacentFaces();
8 $\quad f \leftarrow \arg\max_{f_i \in \mathcal{A}} (|\text{Forces}[i]| \cdot \cos(\measuredangle(\text{Forces}[i], \text{center}(f_i) - \text{center}(\mathcal{F}))))$;
9 $\quad \mathcal{F}$.Add(f)
10 **return** $\mathcal{F}$.getSurroundingPath();

Auch für Algorithmus 4.2 können wir keinerlei Garantien über die Badness der gefundenen Lösung geben. Allerdings ist in Abbildung 4.5b zu erkennen, dass der Algorithmus wie erhofft, vom Startpunkt aus Facetten wählt, die in Richtung schöner Gebiete liegen. Die Berechnung der Kräfte liegt in $\mathcal{O}(g^2)$, wobei g die Anzahl der Facetten ist. Da im Vergleich zu Algorithmus 4.1 nur die Berechnung der Kräfte hinzugekommen ist, hat auch dieser Algorithmus eine Laufzeit von $\mathcal{O}(g^2)$.

Im nächsten Abschnitt werden wir einige Möglichkeiten zeigen, mit denen sich die Ergebnisse der Greedy-Algorithmen verbessern lassen.

4.2.4 Optimierungen

Um die Badness der gefundenen Lösung zu verbessern, können wir bei beiden Algorithmen den Parameter ε benutzen, welcher uns einen bestimmten Spielraum bei der Ziellänge einräumt. Um dies umzusetzen, ändern wir die Abbruchbedingung der Schleife in Zeile 6 von Algorithmus 4.2 zu $\mathcal{F}$.getSurroundingPathLength() $< (1+\varepsilon)l$ ab. Anschließend suchen wir unter allen Zwischenergebnissen mit einer Länge größer als $(1 - \varepsilon)l$ jenes mit der

niedrigsten Badness. Dazu prüfen wir am Ende jedes Schleifendurchlaufs, ob die Länge des Pfades größer als $(1-\varepsilon)l$ ist und ob die Badness geringer als die der bis dahin besten Lösung ist. Falls dem so ist, wird die beste Zwischenlösung aktualisiert.

Für einen Jogger wäre es, unabhängig von der Badness, wünschenswert, Joggingtouren zu erhalten, bei denen nicht allzu oft die Straße gewechselt wird, sondern solche, bei denen lange Teilstrecken geradeaus verlaufen. Dies soll es dem Jogger erleichtern, sich die Joggingtour einzuprägen. Es ist jedoch nicht direkt ersichtlich, welche Facette aus der Menge $\mathcal{A}$ hinzugenommen werden sollte, um möglichst lange gerade Teilstrecken im Ergebnis zu erhalten.

Da wir im Greedy-Algorithmus nur die Facetten betrachten, können wir keine direkten Rückschlüsse über die Geradlinigkeit des resultierenden Pfades treffen, ohne diesen jedes Mal explizit zu berechnen. Wir werden deshalb die Geradlinigkeit durch andere Eigenschaften approximieren. Eine erste Idee ist es hier, das Verhältnis der von der Joggingtour eingeschlossenen Fläche zu ihrer Länge heran zu ziehen. Umfasst eine Joggingtour eine sehr große Fläche, und ist die Strecke, die dabei zurückgelegt wird, trotzdem sehr klein ist, so muss die Route relativ *rund* sein. Der Gedanke ist nun, dass die Joggingroute deshalb möglichst wenig unnötigen Schlaufen, und damit auch Abbiegungen, enthält.

Wir ändern dafür die in Zeile 8 von Algorithmus 4.2 zu maximierende Formel zu einer Linearkombination aus dem Wert für die Kraft und dem Verhältnis von Fläche zu Umfang ab. Da diese beiden Werte möglicherweise verschiedene Wertebereiche abdecken, skalieren wir zusätzlich beide Werte auf das Intervall $[0,1]$. Um die Kräfte zu skalieren, bilden wir den Quotienten aus der Kraft für die einzelne Facette und der maximal vorkommenden Kraft. Beim Flächen-Umfang-Verhältnis nutzen wir aus, dass ein Kreis hier den maximalen Wert besitzt. Wir teilen deswegen den Umfang eines Flächengleichen Kreises durch den Umfang der Facette. Somit erhalten wir für Zeile 8 die Formel:

$$f \leftarrow \arg\max_{f_i \in \mathcal{A}} \left(\alpha \cdot \text{force}(f_i) + (1-\alpha) \cdot \text{shape}(f_i)\right), \text{ mit} \tag{4.1}$$

$$\text{force}(f_i) := \frac{|\text{Forces}[i]| \cdot \cos\left(\measuredangle(\text{Forces}[i], \text{center}(f_i) - \text{center}(\mathcal{F}))\right)}{\max(\text{Forces})},$$

$$\text{shape}(f_i) := \frac{2\sqrt{\text{area}(f_i)\pi}}{\text{perimeter}(f_i)}.$$

Lokale Optima. Da unser Algorithmus ein lokales Optimum verfolgt, kann eine Joggingtour *Einbuchtungen* enthalten, die nicht entfernt werden können. Ein Beispiel dafür ist in Abbildung 4.6 zu sehen. Besteht so eine Einbuchtung aus mehr als nur einer Facette, so kann es passieren, dass das Flächen/Längen-Verhältnis durch das Einfügen der ersten Facette zunächst verschlechtert wird, bevor es durch das Einfügen der zweiten Facette absolut verbessert wird. Leider reicht es auch nicht die nächsten 2 Facetten zeitgleich zu betrachten, da eine solche Einbuchtung aus beliebig vielen Facetten bestehen kann.

Ein weiterer Ansatz, die Geradlinigkeit des berechneten Pfades zu optimieren, ist es, gezielt derartige Einbuchtungen wie in Abbildung 4.6 zu reduzieren. Um solche Facetten zu identifizieren, kann der Abstand von einer Facette $f \in \mathcal{A}$ zum geometrischen Zentrum der bereits benutzten Facetten $\mathcal{F}$ benutzt werden. Ist dieser Abstand sehr klein, so ist es wahrscheinlich, dass f zu einer Einbuchtung gehört. Auch diese Überlegung können wir durch eine Linearkombination in Zeile 8 in den Algorithmus einfließen lassen. Die Ergebnisse des abgewandelten Algorithmus enthalten zwar keinerlei Einbuchtungen mehr und die berechneten Pfade haben auch eine sehr kompakte bzw. konvexe geometrische Form. Allerdings kann auch dieses Verfahren die Anzahl der Abbiegungen nicht wie gewünscht reduzieren.

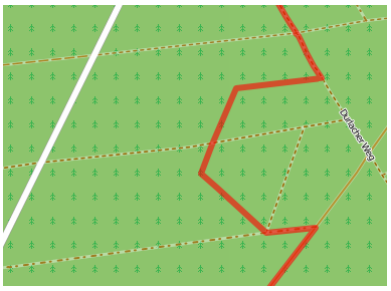


Abbildung 4.6: Beispiel einer Joggingtour mit Einbuchtung.

Da sich die Forderung nach langen geraden Teilstrecken nicht ohne weiteres mit der lokalen Optimierung des Greedy-Algorithmus vereinen lässt, werden wir in einem nächsten Schritt unseren Algorithmus nur noch nutzen, um der Joggingtour eine initiale Richtung und Ausdehnung zu geben und anschließend den erhaltenen Pfad durch längere gerade Straßenabschnitte ersetzen.

4.2.5 Glättungstrategien

Wir haben mit Hilfe von Algorithmus 4.2 einen Pfad $P = (s, p_1, \ldots, p_k)$ berechnet und wollen nachträglich die Eigenschaft, dass der Pfad möglichst gradlinig sein soll, optimieren. Dabei soll der veränderte Pfad nur so gering wie möglich vom ursprünglichen Pfad abweichen. Besonders der Knoten s muss auf jeden Fall weiterhin im Pfad enthalten sein.

Da in der Ebene eine gerade Linie zwischen zwei Punkten auch immer die kürzeste Verbindung zwischen diesen beiden Punkten darstellt, werden wir versuchen die Forderung nach gradlinigen Teilstrecken durch das Finden kürzester Wege zwischen einzelnen Knoten des Pfades P anzunähern. Zur Berechnung des kürzesten Weges zwischen zwei gegebenen Knoten werden wir hierbei auf Dijkstras Algorithmus zurückgreifen [Dij59]. Für einen Pfad $P = (p_1, \ldots, p_k)$ berechnen wir zuerst eine Folge $(\tilde{p}_0, \ldots, \tilde{p}_{h-1})$ von Knoten, die beibehalten werden sollen. Anschließend ergibt sich der *geglättete* Pfad P' aus der Konkatenation der kürzesten Wege für je zwei aufeinanderfolgende Knoten.

$$P' = \mathrm{SP}(\tilde{p}_1, \tilde{p}_2) \circ \cdots \circ \mathrm{SP}(\tilde{p}_{h-1}, \tilde{p}_h) \circ \mathrm{SP}(\tilde{p}_h, \tilde{p}_1)$$

Es müssen nun nur noch geeignete Kandidaten für beizubehaltende Knoten aus dem Pfad P gewählt werden. Dabei ist es sinnvoll, die gewählten Kandidaten relativ gleichmäßig über die Länge des Pfades P verteilt zu wählen, damit der Pfad seine Ausdehnung im Straßennetz beibehält.

Äquidistantes Glätten Eine erste simple Strategie zum Wählen der Kandidaten ist es, die Kandidaten äquidistant über die Länge des Pfades zu verteilen. Es sei hierfür ein Pfad $P = (p_1, \ldots, p_k)$, eine Längenfunktion c sowie die Anzahl h der zu wählenden Kandidaten gegeben. Dann ist die gesuchte Folge $(\tilde{p}_0, \ldots, \tilde{p}_{h-1})$ der Kandidatenknoten gegeben durch:

$$\tilde{p}_i := \arg\min_{p_j \in P} \left| \left(\frac{i \cdot c(P)}{h} \right) - c((p_1, \ldots, p_j)) \right| \text{ , für } i \in [0, h-1]$$

In Abbildung 4.7b auf Seite 26 ist die Glättung einer Joggingtour mit diesem Verfahren zu sehen. Auf Grund der Wahl der Kandidaten stehen diese in keinem Bezug zu Ein- und Ausbuchtungen des ursprünglichen Pfades. Deshalb können auch nach der Glättung noch Einbuchtungen im Pfad enthalten sein. In einigen Fällen ist es sogar möglich, dass Knoten mehrfach passiert werden und der Pfad P' somit ein Sharing > 0 aufweist. Auch dies ist in Abbildung 4.7b zu sehen. Darüber hinaus kann es passieren, dass, wenn zwei aufeinanderfolgende Knoten in verschiedenen Einbuchtungen liegen, der kürzeste Weg zwischen

ihnen sehr viel kürzer ist als der ursprüngliche Weg im Pfad P. Deshalb können wir keine Qualitätsgarantien bezüglich der Länge geben.

Laufzeit. Um einen Pfad mit k Knoten mit h beizubehaltenden Knoten zu glätten, müssen zunächst die beizubehaltenden Knoten gewählt werden, was in $\mathcal{O}(k)$ liegt. Anschließend wird h Mal Dijktras Algorithmus ausgeführt, was in $\mathcal{O}(h \cdot (n \log n + m))$ liegt. Da $k \in \mathcal{O}(n)$ gilt, hat diese Glättungsstrategie eine Gesamtlaufzeit von $\mathcal{O}(h \cdot (n \log n + m))$.

Beide Probleme der ersten Glättungsstrategie versuchen wir zu lösen, indem wir ausschließen, dass ein beizubehaltender Knoten innerhalb einer Einbuchtung liegt. Dementsprechend müssen also alle gewählten Knoten auf Ausbuchtungen liegen. Wir benötigen dafür eine formale Eigenschaft, die beschreibt, welche Knoten eines Pfades überhaupt auf einer Aus- bzw. Einbuchtung liegen. Nach der intuitiven Interpretation von *Ausbuchtungen* ist klar, dass zumindest die Knoten des Pfades P, die auch in seiner konvexen Hülle enthalten sind, als Knoten, die auf Ausbuchten liegen, verstanden werden können.

Konvexes Glätten. Diese Eigenschaft benutzen wir, um eine zweite Glättungsstrategie zu erhalten, bei der die Folge von beizubehaltenden Knoten in etwa der konvexen Hülle von P entspricht. Wir können nicht in jedem Fall die konvexe Hülle selber verwenden, da wir garantieren müssen, dass der Startknoten s in der Folge enthalten ist, aber s nicht notwendiger Weise auch in der konvexen Hülle enthalten ist. Wir benutzen deshalb zur Berechnung der Folge von beizubehaltenden Knoten eine geringfügig modifizierte Version des Graham Scan Algorithmus zur Berechnung von konvexen Hüllen [Gra72].

Der Graham Scan Algorithmus berechnet die konvexe Hülle einer Punktmenge P im Wesentlichen in drei Schritten. Zunächst wird ein Punkt p gesucht, der garantiert auf der konvexen Hülle liegt. Hierzu bietet es sich an, den Punkt mit der kleinsten x-Koordinate zu wählen. Haben mehrere Punkte die gleiche x-Koordinate wird unter ihnen derjenige Punkt mit der kleinsten y-Koordinate gewählt. Im zweiten Schritt werden die Punkte $q \in P$ nach dem Winkel der Geraden $\overline{pq}$ zur x-Achse sortiert. Im letzten Schritt werden zunächst p und der Punkt mit dem kleinsten Winkel auf einen Stack gelegt. Anschließend werden die restlichen Punkte, in der Reihenfolge ihrer Sortierung betrachtet. Liegt ein Punkt links der Geraden durch die letzten beiden Punkte auf dem Stack wird er zum Stack hinzugefügt, liegt er rechts, so wird der letzte Punkt vom Stack entfernt.

Nachdem alle Punkte von Graham Scan betrachtet wurden, bilden die Punkte die auf dem Stack liegen eine Folge von Knoten, die wir als Folge der beizubehaltenden Knoten benutzen wollen. Damit diese Folge aber unseren Anforderungen genügt, müssen wir noch sicherstellen, das der Knoten s in ihr enthalten ist. Sollte der Knoten s schon in der konvexen Hülle enthalten sein, sind wir fertig. Ist dies nicht der Fall, suchen wir den ersten Knoten des ursprünglichen Pfades, der auch in der konvexen Hülle enthalten ist. Wir fügen nun s vor diesem Knoten in die Folge ein, und erhalten so eine Folge die an allen Stellen bis auf s konvex ist.

Laufzeit. Die Laufzeit des Graham Scan Algorithmus wird vom Sortieren der Knoten dominiert, und liegt somit bei $\mathcal{O}(k \log k)$, wobei k die Anzahl der Knoten im zu glättendem Pfad ist. Zusammen mit den Aufrufen von Dijkstras Algorithmus zum verbinden der einzelnen beizubehaltenden Knoten ergibt sich eine Gesamtlaufzeit der Glättungsstrategie von $\mathcal{O}(k \cdot (n \log n + m))$.

Ein Beispiel für die Wirkung der Glättung mittels konvexer Hülle ist in Abbildung 4.7c auf Seite 26 zu sehen. Auch bei diesem Verfahren kann sich die Länge das Pfades theoretisch um einen beliebigen Faktor ändern. Deshalb können wir auch hier keine Aussage über die Qualität der berechneten Joggingroute treffen.

Die beiden vorgestellten Glättungsstrategien verfolgten bisher nur das Ziel, die Geradlinigkeit der Joggingtouren zu verbessern. Die Badness wurde bisher von keinem der Verfahren mit in Betracht gezogen. Wir werden als nächstes eine Glättungsstrategie vorstellen, welche sowohl die Geradlinigkeit als auch die Badness verbessern soll.

Wir verändern dafür zum einen die Dijkstra zugrunde liegende Metrik. Dijkstra soll nun nicht mehr, wie üblich, die kürzesten Wege bezüglich der Kantenlänge c berechnen, sondern eine Kombination aus Länge c und Badness b benutzen. Als neue Metrik ω für Dijkstra definieren wir das Produkt aus Länge und Badness. So ergibt sich für eine Kante $e \in E$ das für Dijkstra relevante Maß $\omega(e) := c(e) \cdot b(e)$. Mit diesem Maß drücken wir aus, dass eine *unschönere* Kante nur dann gewählt wird, wenn sich durch ihre Benutzung eine entsprechende Wegersparnis ergibt.

Glättung über beste Kreuzungen. Darüber hinaus wählen wir als beizubehaltende Knoten solche Knoten, welche ein hohes Potential für Pfade mit niedriger Badness versprechen. Um h solche Knoten zu bestimmen, unterteilen wir zunächst den Pfad in $h-1$ gleich große Intervalle. Anschließend wird aus jedem Intervall der Knoten gewählt, für den das Produkt der Badness Werte der zu ihm inzidenten Kanten minimal ist. Zusätzlich wird auch noch der Knoten s gewählt. Sei $P = (p_1, \ldots, p_k)$ ein zu glättender Pfad und $\tilde{p}_j$ der j-te beizubehaltende Knoten, dann gilt:

$$\tilde{p}_j := \begin{cases} p_1 & \text{falls } j = 0 \\ \arg\min\limits_{p \in I_j} \left(\prod\limits_{\{p,q\} \in E} b(\{p,q\}) \right) & \text{falls } j \in [1, h-1] \end{cases}$$

Wobei $I_j = \{p_i \in P \mid \frac{c(P)\cdot(j-1)}{n-1} < c(p_1, \ldots, p_i) \leq \frac{c(P)\cdot j}{n-1}\}$ gilt. Auf diese Weise werden Kreuzungsknoten gewählt, die zu möglichst vielen Kanten mit niedriger Badness inzident sind. Dies soll die Wahrscheinlichkeit erhöhen, dass Dijkstras Algorithmus einen von der ursprünglichen Strecke abweichenden Pfad mit niedrigerer Badness wählt.

Laufzeit. Bei der Berechnung der beizubehaltenden Knoten müssen für jeden Knoten alle zu ihm inzidenten Kanten betrachtet werden. Die Wahl der beizubehaltenden Knoten benötigt daher $\mathcal{O}(kn)$ Schritte. Die Gesamtlaufzeit des Verfahrens liegt deshalb in $\mathcal{O}(kn + h(n \log n + m))$. In Abbildung 4.7d auf Seite 26 ist das Ergebnis dieser Glättungsstrategie im Vergleich zu den beiden anderen Verfahren zu sehen.

4.3 Teilwege-Algorithmen

Da keine Glättungsstrategie zur Nachbearbeitung der Greedy-Algorithmen eine Aussage über die Länge der resultierenden Pfade zulässt, werden wir in diesem Abschnitt einen neuen Ansatz entwickeln, der uns garantiert, dass die Länge der gefundenen Joggingtour, sofern überhaupt eine gefunden wird, im Intervall $\mathcal{I}(l, \varepsilon) := [(1-\varepsilon)l, (1+\varepsilon)l]$ liegt. Während der Entwicklung des ersten Ansatzes hat sich gezeigt, dass nicht nur Sharing und Badness wichtige Maße für die *Schönheit* einer Joggingtour sind, sondern auch deren Geradlinigkeit. Darüber hinaus hat sich Dijkstras Algorithmus als geeignet erwiesen, diese Eigenschaft zwischen einzelnen Wegpunkten der Joggingroute sicherzustellen.

Als nächstes werden wir deshalb versuchen direkt geeignete Knoten des Graphen als Wegpunkte zu wählen, um sie anschließend über kürzeste Wege miteinander zu verbinden und so eine Joggingtour zu erhalten.

(a) (b) (c) (d)

Abbildung 4.7: Die verschiedenen Glättungsstrategien im Vergleich, bei gesuchte Länge 8000m. (a) ohne Glättung, $l = 7386\text{m}, b = 36\%$, (b) Äquidistante Glättung, $l = 5849\text{m}, b = 35\%$, (c) Glättung mittels konvexer Hülle, $l = 7515\text{m}, b = 27\%$, (d) Glättung mittels potentiell guter Knoten, $l = 7127\text{m}, b = 28\%$.

4.3.1 Joggingroute durch drei Wegpunkte

In einem ersten Ansatz werden wir die spätere Joggingtour durch ein Dreieck annähern. Wir müssen daher drei Punkte bestimmen, durch die die spätere Joggingroute verlaufen soll. Da eine Lösung des Jogger-Problems (JP) aber immer den Startknoten s enthalten soll, ist es naheliegend, dass einer der drei Wegpunkte s ist. Es müssen also nur noch zwei weitere Knoten gefunden werden. Um die Menge der Knoten, die als weitere Wegpunkte in Frage kommen, einzugrenzen, schränken wir unseren Ansatz ein und suchen nur nach gleichseitigen Dreiecken als Annäherung für die Joggingtour.

Sei mit $G = (V, E), c, b, l, \varepsilon$ eine JP-Instanz gegeben. Wir suchen dann zwei Wegpunkte $p_1, p_2 \in V$, so dass der bezüglich des Kantengewichts $\omega(e) := c(e) \cdot b(e)$ kürzeste Pfad von s nach p_i eine Länge von $l/3$ hat. Die Menge der Knoten $\mathcal{K}$, welche diese Eigenschaft erfüllen, kann leicht mit Dijkstras Algorithmus, ausgehend von s, bestimmt werden. Um auch hier schon den Spielraum, den uns der Parameter ε gewährt, mit zu berücksichtigen, bestimmen wir mittels Dijkstra alle Knoten, deren Entfernung von s im Intervall $\mathcal{I}(\frac{l}{3}, \varepsilon)$ liegt. Es ist zu beachten, dass wir zwar zunächst den kürzeste-Wege-Baum bezüglich ω von s aus aufbauen, jedoch anschließend als Kandidaten jene Knoten zu $\mathcal{K}$ hinzufügen, die bezüglich der tatsächlichen Länge c im passenden Abstand zu s liegen.

Nachdem die Kandidatenmenge $\mathcal{K}$ berechnet wurde, suchen wir unter allen Tupeln $(p_1, p_2) \in \mathcal{K}^2$, für die die Länge des Pfades $P_{p_1p_2} := \text{SP}(s, p_1) \circ \text{SP}(p_1, p_2) \circ \text{SP}(p_2, s)$ im Intervall $\mathcal{I}(l, \varepsilon)$ liegt, jenes Tupel mit minimaler Badness $b(P_{p_1p_2})$. Um dieses Tupel zu bestimmen, lassen wir ausgehend von jedem Punkt $p \in \mathcal{K}$ ein weiteres Mal Dijkstras Algorithmus laufen und bestimmen so seine Distanz zu jedem anderen Knoten $q \in \mathcal{K}$. Liegt die Länge des entsprechenden Pfades P_{pq} im gesuchten Bereich und ist dessen Badness geringer als die des bisher besten Tupels, so speichern wir uns (p, q) als neues bestes Tupel.

Laufzeit Da die Menge der Kandidaten $\mathcal{K}$ durch die Anzahl der Knoten n beschränkt ist, wird im schlimmsten Fall n Mal Dijkstras Algorithmus aufgerufen. Somit liegt die Laufzeit des Algorithmus in $\mathcal{O}(n^2 \log n + nm)$. Dies ist aber eine sehr konservative Abschätzung, da $\mathcal{K}$ auch durch l weiter eingeschränkt wird. So ergeben sich meist weitaus kürzere Laufzeiten.

Algorithm 4.3: Teilwege-Algorithmus über drei Wegpunkte (TW3)

Input: Graph $G = (V, E)$, c, b, s, l, ε
Output: $P = (s, p_1, \ldots, p_k, s)$
`// dijkstra(`s, l`) berechnet hier den kürzeste-Wege-Baum mit Wurzel` s `und Radius` l `und gibt die entsprechenden Distanz- und parent-Zeiger-Arrays zurück`

```
1  (c_1, ω_1, parent_1) ← dijkstra(s, (1+ε)l/3);
2  K ← {v ∈ V | c_1[v] ∈ I(l/3, ε)};
3  ((p, q), badness) = (⊥, ∞);
4  foreach p' ∈ K do
5      (c_2, ω_2, parent_2) ← dijkstra(v_i, (1+ε)l/3);
6      foreach q' ≠ i ∈ K do
7          badness' ← (ω_1[p'] + ω_2[q'] + ω_1[q']) / (c_1[p'] + c_2[q'] + c_1[q']);
8          if c_1[p'] + c_2[q'] + c_1[q'] ∈ I(l, ε) and badness' < badness then
9              ((p, q), badness) ← ((p', q'), badness');
10 return SP(s, p) ∘ SP(p, q) ∘ SP(q, s);
```

Verschärftes Stoppkriterium Die praktische Laufzeit des Algorithmus kann reduziert werden, wenn die Kandidatenmenge $\mathcal{K}$ zunächst bezüglich der minimalen Badness der resultierenden Pfade sortiert wird. Anschließend betrachten wir die einzelnen Kandidaten in aufsteigender Reihenfolge. Für zwei Knoten $p, q \in \mathcal{K}$ definieren wir dementsprechend:

$$p < q :\Leftrightarrow \omega(\mathrm{SP}(s, p)) < \omega(\mathrm{SP}(s, q))$$

Tatsächlich erfordert diese Sortierung keinen zusätzlichen Rechenaufwand, da sie gerade der Reihenfolge entspricht, in welcher die Knoten von Dijkstras Algorithmus besucht werden. Sei nun $p \in \mathcal{K}$ der als nächstes zu betrachtende Kandidat. Dann müssen als zweiter Kandidat q nur solche Knoten betrachtet werden, für die $q > p$ gilt, da für Knoten mit niedrigerer Badness schon alle Kombinationen getestet wurden. Eine Joggingroute die zu diesem Zeitpunkt mit dem Knoten p gebildet wird, besteht also mindestens aus dem Streckenabschnitt dijkstra(s, p) und einem Weitere Abschnitt mit einem schlechterem Wert für ω. Auf diese Weise lässt sich die beste Route die im weiterem Verlauf noch berechnet werden kann wie folgt abschätzen.

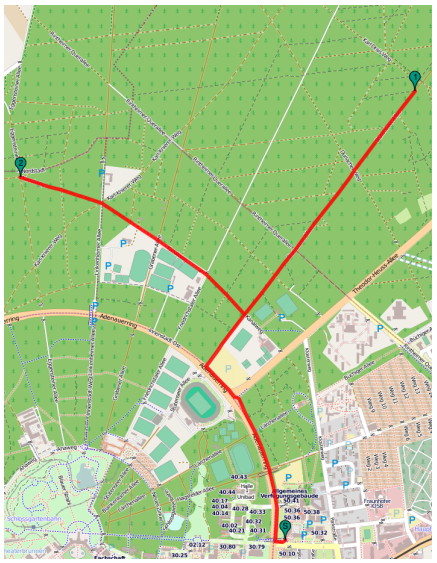
$$\min_{p', q' \geq p} b(P_{p'q'}) \geq \frac{2\omega(\mathrm{SP}(s, p))}{(1+\varepsilon)l}$$

Sei nun P zusätzlich der beste bisher gefundene Pfad. Gilt in diesem Fall die Gleichung

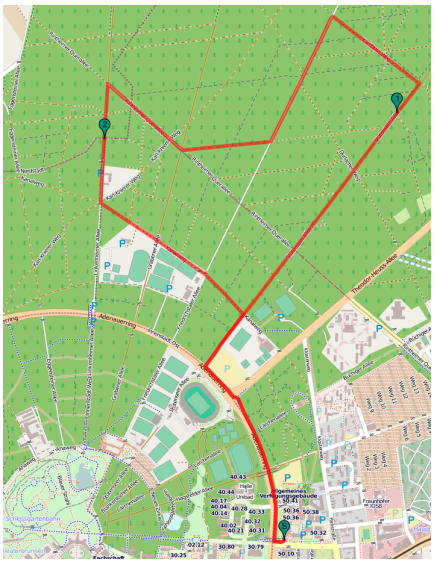
$$\min_{p', q' \geq p} b(P_{p'q'}) \geq \frac{2\omega(\mathrm{SP}(s, p))}{(1+\varepsilon)l} \geq b(P)$$

so kann keiner der noch nicht betrachteten Kandidaten mehr zu einer besseren Lösung als P führen. Dies gilt, da alle Kanten eine positive Badness und Länge besitzen. Das Erweitern des Pfades von s nach q kann also nur zu noch schlechteren Lösungen als P führen.

In Abbildung 4.8a ist das Ergebnis dieses Algorithmus für die gleiche Eingabe wie in Abbildung 4.11 und $\varepsilon = 0{,}1$ zu sehen. Dabei ist die erreichte Badness und Geradlinigkeit der Strecke schon zufriedenstellend, jedoch hat die berechnete Joggingtour in diesem Fall sogar ein Sharing von 1. Im nächsten Schritt werden wir den Algorithmus so modifizieren, dass er auch das Sharing reduziert. Wir wollen es aber auch erlauben, dass ein Teil des Weges in der Nähe von Startpunkt s doppelt gelaufen werden darf. Dies tun wir damit der Algorithmus auch dann Lösungen findet, falls der Startpunkt in einer Sackgasse liegt.



(a) Einfacher Algorithmus, l=8533m b=28%.



(b) Ohne Sharing, l = 8181m b=29%.

Abbildung 4.8: Ergebnisse des Teilwege-Algorithmus mit und ohne Sharing.

Reduzierung des Sharing. Um das Sharing, wie oben beschrieben, zu reduzieren, entfernen wir, bevor für einen Kandidaten $v_i \in \mathcal{K}$ in Zeile 5 von Algorithmus 4.3 Dijkstras Algorithmus ausgeführt wird, die Knoten auf dem Pfad von s nach p aus dem Graphen G. Dies führt dazu, dass der Pfad von s zum ersten Wegpunkt p und der Pfad von p zum zweiten Wegpunkt keine gemeinsamen Knoten, bis auf p selber, haben. Um gemeinsame Knoten auch zwischen dem Pfad vom ersten zum zweiten Wegpunkt und dem Pfad vom zweiten Wegpunkt zurück zum Startpunkt auszuschließen, überprüfen wir in Zeile 8 zusätzlich, ob der vorletzte Knoten auf dem Pfad von v_i nach v_j ungleich dem vorletzten Knoten auf dem Pfad von s nach v_j ist. Ist dies der Fall, so ist klar, dass die beiden Pfade kantendisjunkt sein müssen, denn sonst gäbe es von einem Knoten, der in beiden Pfaden enthalten ist, zwei verschiedene kürzeste Wege zu v_j. Abbildung 4.8b zeigt die Verbesserung, die diese Modifikationen bringen.

Kandidatenversatz. Am Anfang unserer Überlegungen stand die Idee die Joggingroute durch ein gleichseitiges Dreieck anzunähern. Deshalb bestimmen wir in Zeile 2 von Algorithmus 4.3 als Kandidaten solche Knoten, die einen Abstand von $l/3$ zu s haben. Dies muss natürlich nicht die Optimale Wahl für die Kandidaten sein. Wir definieren Deshalb den *Kandidatenversatz* als Abstand der Kandidatenknoten zum Startknoten in Relation zur gesuchten Gesamtlänge. Für den Kandidatenversatz können, als zusätzlicher Parameter für den Algorithmus, Werte zwischen 0 und 0,5 angegeben werden. Anschließend werden als Kandidaten solche Knoten gewählt, die in dem durch den Kandidatenversatz spezifiziertem Abstand zu s liegen.

Der Teilwege-Algorithmuss über drei Wegpunkte berechnet Joggingrouten mit niedrigem Sharing und garantiert zumindest zwischen den einzelnen Wegpunkten optimale Wege bezüglich ω. An den Wegpunkten selber kann aber keine Geradlinigkeit garantiert werden. Darüber hinaus können die Wegpunkte schlecht platziert sein, so dass sie zwar, dank Dijkstra, optimal erreicht werden, es jedoch womöglich noch günstiger wäre, sie gar nicht passieren zu müssen. Wir werden mit dem nächsten Algorithmus versuchen die Garantien, die Dijkstra uns zwischen den Wegpunkten gibt auch für die Wegpunkte selber zu erreichen.

4.3.2 Joggingroute durch vier Wegpunkte (TW4)

In diesem Abschnitt nähern wir die gesuchte Joggingroute durch ein Viereck an, im Gegensatz zu dem Dreieck des TW3 Algorithmus. Wie im Fall mit drei Wegpunkten berechnen wir zunächst mit Dijkstra eine Menge $\mathcal{K}$ von Kandidaten für den ersten Wegpunkt. In diesem Fall sind die Kandidaten all jene Knoten, deren Distanz zum Startknoten s im Intervall $\mathcal{I}(l/4, \varepsilon)$ liegt. Anschließend werden die Wegpunkte (s, p, m, q) gewählt, wobei $p, q \in \mathcal{K}$ gilt und m der noch zu findende vierte Wegpunkt, oder auch *Mittelpunkt*, ist. Auch bei diesem Algorithmus müssen die Kandidaten nicht zwingend im Abstand $l/4$ zum Startknoten liegen. Wieder ist theoretisch ein Kandidatenversatz im Bereich von 0 bis 0,5 möglich.

Darüber hinaus wollen wir erreichen, dass im resultierenden Pfad Knoten, die unmittelbar vor und nach einem Wegpunkt liegen, jeweils durch kürzeste Wege verbunden sind. Für einen Kandidaten $p \in \mathcal{K}$ muss der Pfad von s nach p also so fortgesetzt werden, dass er auch in der Umgebung von p einem kürzesten Pfad bezüglich ω entspricht. Um dies zu erreichen, benutzen wir als Startknoten für Dijkstra nicht den Knoten p selber, sondern gehen auf dem Pfad von s nach p zunächst ein Stück zurück und starten dann Dijkstra. Wir berechnen also den kürzeste-Wege-Baum ausgehend von einem Knoten, der vor p liegt. Anschließend betrachten wir nur den Teilbaum des kürzeste-Wege-Baums, der p enthält. Auf diese Weise entspricht der Pfad, ausgehend vom Startpunkt von Dijkstra, über p hinaus auch einem kürzesten Weg.

Der neue Algorithmus berechnet zunächst, wie auch Algorithmus 4.3, die Kandidaten $\mathcal{K}$. Anschließend wird für jeden Kandidaten $p \in \mathcal{K}$ der *Halbwegpunkt* h_p auf dem Pfad von s nach p berechnet. Ist $P = (s, p_1, \ldots, p_k, p)$ der Pfad von s nach p, so soll h_p in der Mitte dieses Pfades liegen, es gilt also:

$$h_p := \arg\min_{p_i \in P} \left| c\left((s, p_1, \ldots, p_i)\right) - \frac{c(P)}{2} \right|.$$

Anschließend wird Dijkstras Algorithmus nicht wie im ersten Algorithmus für alle Kandidaten ausgeführt, sondern für deren Halbwegpunkte. Als nächstes suchen wir nun mögliche Kandidaten für den vierten Wegpunkt m.

Um diesen zu finden, betrachten wir die Menge M_p der Knoten aus G, die von h_p aus gesehen *hinter* p liegen und deren Entfernung zu s in $\mathcal{I}(\frac{l}{2}, \varepsilon)$ liegt. Dementsprechend gilt:

$$M_p := \left\{ v \in V \mid p \in \mathrm{SP}(h_p, v) \wedge c(\mathrm{SP}(s, h_p) \circ \mathrm{SP}(h_p, v)) \in \mathcal{I}\left(\frac{l}{2}, \varepsilon\right) \right\}.$$

Für $m \in M_p$ liegt also p auf dem kürzesten Pfad von h_p nach m. Falls wir $p \in \mathcal{K}$ als ersten Wegpunkt wählen, so ist M_p die Menge der möglichen mittleren Wegpunkte. Sind $p, q \in \mathcal{K}$ zwei Kandidaten und $m \in M_p \cap M_q$ so ist (s, p, m, q) eine mögliche Folge von Wegpunkten. Unter allen Möglichkeiten, die sich auf diese Weise ergeben, suchen wir wieder diejenige, welche die Badness des resultierenden Pfades minimiert.

Laufzeit. Gehen wir wieder davon aus, dass $|\mathcal{K}|$ nur durch die Anzahl der Knoten n begrenzt ist, so müssen n^2 Schnittmengen bestimmt werden. Da auch M_p für jedes $p \in \mathcal{K}$ wieder bis zu n Elemente enthalten kann, besitzt der Algorithmus eine Laufzeit die mindestens in $\mathcal{O}(n^3 \log n)$ liegt. Wobei wir davon ausgehen, dass die Berechnung des Schnitts zweier Mengen mit n elementen in $\mathcal{O}(n \log n)$ liegt. Um den Rechenaufwand zu reduzieren, berechnen wir nicht für jeden Knoten p die Menge M_p, sondern speichern umgekehrt für jeden möglichen Mittelpunkt $m \in V$ die Menge P_m der möglichen Kandidaten, über die m erreicht werden kann. Die Menge P_m kann während der Berechnung von Dijkstra bestimmt

werden. Wird beim Aufruf von Dijkstra für den Startknoten h_p ein Knoten m erreicht, für den $p \in \mathrm{SP}(h_p, m)$ und $c(\mathrm{SP}(h_p, m)) \in \mathcal{I}(l/2, \varepsilon)$ gilt, so kann p zu P_m hinzugefügt werden. Diese Vorgehensweise wird in Algorithmus 4.5 und 4.4 (Seite 36) benutzt.

Des weiteren können wir dafür sorgen, dass die Menge P_m nie mehr als zwei Elemente enthält. Falls m als Mittelpunkt gewählt wird, werden nur die beiden Knoten aus P_m benötigt, deren Badness am niedrigsten ist. Deshalb kann jedes Mal, wenn ein weiterer Knoten zu P_m hinzukommt, der Knoten mit der niedrigsten Badness verworfen werden. Somit muss Dijkstras Algorithmus, beim Besuchen eines Knotens m, nur konstanten Rechenaufwand leisten, um die Menge P_m zu aktualisieren. Die Laufzeit von Dijkstras Algorithmus wird deshalb von dieser Modifikation nicht beeinträchtigt, und der resultierende Algorithmus hat eine Laufzeit von $\mathcal{O}(n^2 \log n + nm)$

Es kann natürlich passieren, dass zwei Knoten $p \neq q \in \mathcal{K}$ denselben Halbwegspunkt $h_p = h_q$ besitzen. In diesem Fall reicht es aus, nur einmal, mit Dijkstras Algorithmus, den kürzeste-Wege-Baum von h_p aus zu berechnen. Anschließend können der Teilbaum, der als Wurzel p enthält, und der Teilbaum mit Wurzel q separat betrachtet werden.

Um die Anzahl der Aufrufe, an Dijkstras Algorithmus, weiter zu reduzieren, können in manchen Fällen verschiedene Halbwegspunkte zusammengefasst werden. Seien hierzu h_p und h_q die Halbwegspunkte von p bzw. q. Falls h_q in diesem Fall in dem Teilbaum mit Wurzel h_p liegt (es gilt also $h_p \in \mathrm{SP}(s, h_q)$), so kann h_p auch als Halbwegspunkt für q benutzt werden.

Seien p, q zwei Knoten, die entweder auf dem ersten Teilstück der Joggingroute, also vor m, liegen oder beide auf dem zweiten Teilstück, also nach m, liegen. Darüber hinaus sei der Abstand zwischen p und q kleiner als $l/8$. Dann kann für Routen, die mit diesem Verfahren berechnet wurden, garantiert werden, dass der Weg, über den p und q innerhalb der Joggingroute miteinander verbunden sind, der kürzeste Weg zwischen den beiden Knoten bezüglich der Metrik ω ist. Es ist möglich, größere Werte als $l/8$ für die Länge dieser Strecke zu garantieren. Hierzu müssen wahlweise die Kandidaten in einem größeren Abstand als $l/4$ zum Startpunkt s gesucht werden oder die Halbwegspunkte näher an s als dem entsprechenden Kandidaten gewählt werden. Zeitgleich steigt dabei aber auch die Wahrscheinlichkeit, dass der Algorithmus keine gültigen Lösungen mehr findet.

Halbwegspunktversatz. Ähnlich wie beim Kandidatenversatz wollen wir es nun erlauben, dass auch die Halbwegspunkte an anderen Positionen als der Mitte zwischen Startpunkt und Kandidaten liegen. Hierzu definieren wir den *Halbwegspunktversatz* als das Verhältnis von $\mathrm{dist}(s, h_p)$ zu $\mathrm{dist}(s, p)$ für einen Kandidaten p. Für den Halbwegspunktversatz können dementsprechend Werte aus dem Intervall $[0, 1]$ gewählt werden.

In Abbildung 4.9a ist eine Joggingtour zu sehen, die mit diesem Verfahren berechnet wurde. Wie auch beim letzten Verfahren ergibt sich das Problem, dass die Joggingrouten an den Stellen, an denen die einzelnen Teilstrecken zusammengesetzt werden, Kanten doppelt enthalten können. Im letzten Verfahren galt dies für beide Wegpunkte, in diesem Verfahren tritt das Problem nur noch am Punkt m auf.

Eine Möglichkeit, dieses Problem zu lösen, ist es, Kombinationen von Kandidaten, die zu doppelt genutzten Kanten führen, strikt zu verbieten. Hierzu reicht es aus, die Knoten, die auf dem Weg, von einem Kandidaten zu m, direkt vor m liegen, zu betrachten. Sind diese Knoten für zwei Kandidaten $p, q \in \mathcal{K}$ verschieden, so sind die Pfade von p nach m und von q nach m disjunkt. Andernfalls wäre einer der beiden Pfade kein kürzester Pfad. Um dies umzusetzen, müssen bei der Berechnung der zu verwerfenden Kandidaten, in Zeile 19 des Algorithmus, zusätzlich die parent-Zeiger von v bezüglich der Kandidaten p betrachtet werden.

(a) Mit Sharing, l=8538m b=28%.

(b) Ohne Sharing, l = 8080m b=29%.

Abbildung 4.9: Ergebnisse des Teilwege-Algorithmus über vier Wegpunkte mit und ohne Sharing.

Ein Beispiel für eine Joggingroute, die mit dieser Modifikation des Algorithmus berechnet wurde, ist in Abbildung 4.9b zu sehen. Mit dieser Modifikation erhalten wir dieselben Garantien für das Sharing einer berechneten Lösung wie beim letzten Verfahren.

4.3.3 Joggingroute mit bidirektionaler Suche (TWB)

In diesem Abschnitt werden wir die Joggingroute wieder über ein Viereck annähern. Allerdings wollen wir die Einschränkung an die Position der Wegpunkte auflockern. So muss zum Beispiel der mittlere Wegpunkt m die Joggingroute nicht notwendigerweise in zwei gleichlange Teilstrecken zerteilen. Des weiteren versuchen wir den Umweg zu reduzieren, der auf dem Weg vom ersten Wegpunkt p zum dritten Wegpunkt q gelaufen werden muss, weil der Knoten m passiert werden muss.

Um dies zu erreichen, werden wir, aus einer Menge M_{pq} von möglichen Mittelpunkten für zwei feste Kandidaten $p, q \in \mathcal{K}$, denjenigen als mittleren Wegpunkt auswählen, der bezüglich ω zur kürzesten Gesamtstreckenlänge führt. Sollte die Länge der resultierenden Joggingroute nicht im gesuchten Intervall $\mathcal{I}(l, \varepsilon)$ liegen, so werden p und q als erster und dritter Wegpunkt verworfen. Anschließend wird unter allen Paaren von Kandidaten, für die ein gültiger mittlerer Wegpunkt gefunden wurde, jenes Paar gewählt, das zu der Joggingtour mit der niedrigsten Badness führt.

Als möglichen mittleren Wegpunkt in der Menge M_{pq} erlauben wir in diesem Verfahren auch Knoten, die nicht in der Mitte der Joggingroute liegen. Es können also Knoten $m \in M_{pq}$ existieren, für die die Länge der Teilstrecke von s nach m nicht im Intervall $\mathcal{I}(l/2, \varepsilon)$ liegt. Wir wollen aber weiterhin garantieren können, dass zwei Knoten, deren Abstand unterhalb eines gewissen Grenzwertes liegt, innerhalb der Joggingtour über den kürzesten Weg miteinander verbunden sind. Deshalb enthält M_{pq} nur solche Knoten m, für die p auf dem kürzesten Weg von h_p nach m und q auf dem kürzesten Weg h_q nach m liegt. Dementsprechend definieren wir M_{pq} wie folgt:

$$M_{pq} := \{m \in V \mid p \in \mathrm{SP}(h_p, m) \wedge q \in \mathrm{SP}(h_q, m)\}$$

Der Algorithmus muss dann für jedes Paar $p, q \in \mathcal{K}$ den Knoten $m \in M_{pq}$ bestimmen, der die Länge der Joggingtour minimiert. Es gilt also:

$$m = \arg\min_{m \in M_{pq}} c(\mathrm{SP}(h_p, m) \circ \mathrm{SP}(m, h_q))$$

Um den Knoten m effizient zu berechnen, gehen wir ähnlich vor, wie beim letzten Algorithmus. Für jeden Kandidaten $p \in \mathcal{K}$ wird wieder der kürzeste-Wege-Baum von h_p aus berechnet. Hierzu nutzen wir wieder Dijkstras Algorithmus. Anschließend betrachten wir wieder nur den Teilbaum, der p als Wurzel hat. Anstatt aber im Nachhinein den Schnitt zweier berechneter Teilbäume zu berechnen, um M_{pq} zu erhalten, berechnen wir m direkt beim Aufbau des kürzeste-Wege-Baums.

Hierzu berechnen wir den kürzeste-Wege-Baum nicht für jeden Knoten einzeln, sondern betrachten immer Paare $p, q \in \mathcal{K}$ von Kandidaten zusammen. Wir starten nun eine bidirektionale Suche von h_p und h_q aus. Der gesuchte Knoten m ist dann einer der Knoten, an dem sich der von h_p ausgehende Suchraum und der von h_q ausgehende Suchraum treffen. Im Gegensatz zu einer normalen bidirektionalen Suche ist der gesuchte Pfad aber noch nicht beim ersten Treffen der Suchräume gefunden. Dies würde dem kürzesten Weg von h_p nach h_q entsprechen. Wir beenden die Suche erst, wenn beide Suchräume einen Knoten erreicht haben, für den die kürzesten Pfade über p bzw. q verlaufen.

Der beschriebene Algorithmus muss im schlimmsten Fall, wenn $\mathcal{K} = V$ gilt, für n^2 verschiedene Paare von Kandidaten eine bidirektionale Suche ausführen. Somit liegt seine Laufzeit in $\mathcal{O}(n^3 \log n + n^2 m)$. In Abbildung 4.10a auf Seite 33 ist eine Joggingtour zu sehen, die mit diesem Verfahren berechnet wurde.

4.3.4 Verbesserungen

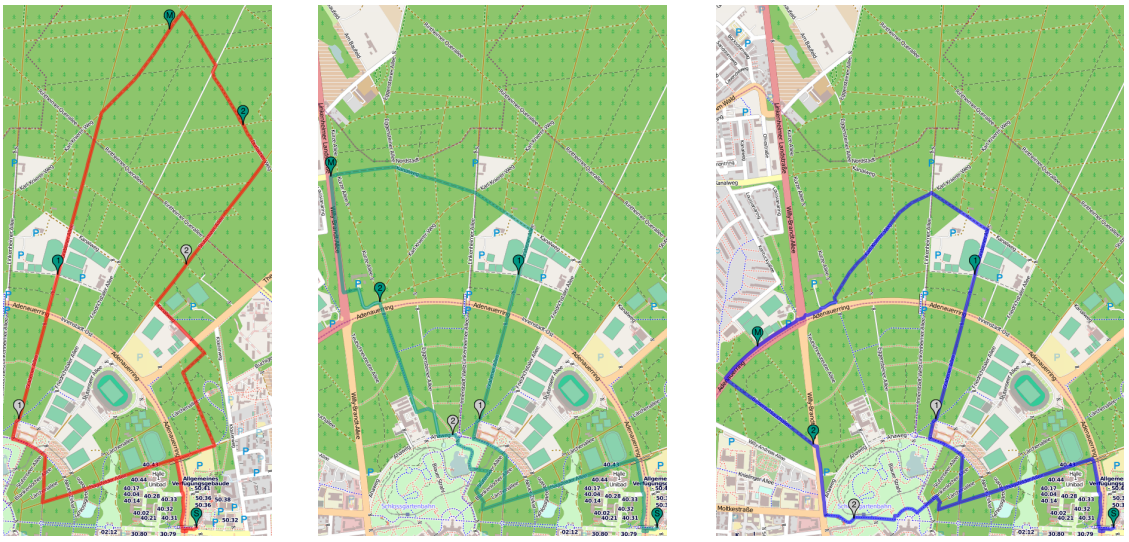
Nachdem wir unsere grundlegenden Algorithmen zur Annäherung von Joggingrouten beschrieben haben, werden wir im Folgenden noch einige Verbesserungsmöglichkeiten, die sich auf alle Verfahren anwenden lassen, beschreiben.

Parallelisierung. Um die benötigte Rechenzeit zum Finden der Joggingrouten zu reduzieren, bietet es sich an, die vorgestellten Algorithmen zu parallelisieren. Wir werden sehen, dass dies bei den Verfahren, die auf Teilwegen zwischen einzelnen Wegpunkten basieren, leicht möglich ist.

Alle vorgestellten Verfahren berechnen zunächst eine Menge $\mathcal{K}$ von Kandidaten. Anschließend werden für alle diese Kandidaten die aus ihnen resultierenden Routen betrachtet, um die Route mit minimaler Badness zu bestimmen. Die Kandidaten werden deshalb von allen Verfahren größtenteils unabhängig voneinander betrachtet. Darüber hinaus dominiert das Prüfen der Kandidaten die Laufzeit des gesamten Algorithmus.

Es bietet sich dementsprechend an, die Überprüfung der Kandidaten zu parallelisieren. Dazu wird die Menge $\mathcal{K}$ in so viele Teilmengen zerteilt, wie Prozesse zur Verfügung stehen. Anschließend wird jedem Prozess eine dieser Teilmengen zugewiesen. Jeder Prozess berechnet dann jene Joggingtour, deren erster Wegpunkt in seiner Kandidatenteilmenge liegt. Aus der Menge dieser Joggingtouren wird anschließend jene gesucht, welche die niedrigste Badness besitzt. Zum Schluss wird unter allen Ergebnissen der einzelnen Prozesse die Route mit der niedrigsten Badness gewählt.

Alternative Routenvorschläge. Da die tatsächliche *Schönheit* einer Joggingroute zuletzt immer im Ermessen des Nutzers liegt, wäre es schön, wenn in dem Fall, dass die berechnete Joggingtour nicht den Vorstellungen des Nutzers entspricht, alternative Strecken ausgegeben werden könnten.



(a) l=7600m, b=20% (b) l=7654m, b=24% (c) l=7697m, b=27%

Abbildung 4.10: Ergebnis des Teilwege-Algorithmus mit bidirektionaler Suche (a) sowie die zwei ersten von zwölf Alternativen (b), (c).

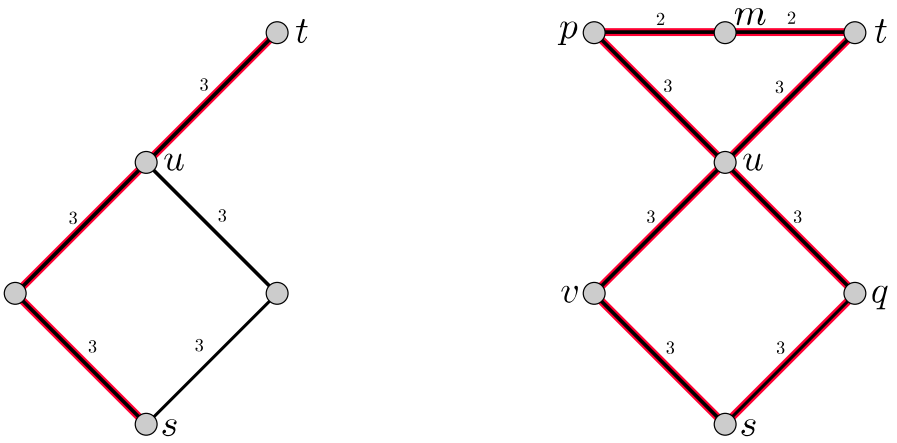
Da die Algorithmen aus diesem Abschnitt zunächst eine Vielzahl von möglichen Kombinationen für die Wegpunkte berechnen und anschließend aus den entsprechenden Joggingtouren jene mit minimaler Badness auswählen, ist es ohne zusätzlichen Zeitaufwand möglich die x-Besten Routen auszugeben. Auf diese Weise kann dem Benutzer die Möglichkeit gegeben werden auf einer alternative Route auszuweichen. Falls ihm das beste Resultat nicht zusagt. Allerdings werden die so entstehenden Alternativen oft nur um wenige Kanten voneinander abweichen.

Wir wollen nun nur solche Alternativen zulassen, die sich stärker voneinander unterscheiden und sogar komplett verschiedene Gebiete des Graphen abdecken. Um dies zu erreichen, bietet es sich an, nicht die Route mit der nächstgrößeren Badness als Alternative zu wählen. Statt dessen suchen wir unter den berechneten und nach Badness sortierten Routen, die erste Route welche über einen Halbwegspunkt führt, der noch von keiner anderen Route genutzt wird. Abbildung 4.10 zeigt das Ergebnis des TWB Algorithmus sowie die ersten beiden Alternativen die gefunden wurden.

Abbiegekosten Zum Joggen möchte ein Jogger vermutlich keine Karte mitnehmen. Deshalb könnte es wichtig sein, dass sich der Jogger eine neue Joggingroute gut und schnell einprägen kann. Dies kann erreicht werden, wenn der Jogger nur an wenigen Punkten abbiegen muss und ansonsten geradeaus laufen kann.

In den einzelnen Verfahren haben wir schon versucht möglichst geradlinige Resultate zu berechnen. Wir werden nun eine Möglichkeit beschreiben, wie die Anzahl der Knoten, an denen der Jogger abbiegen muss, noch weiter reduziert werden kann. Dazu werden wir die Algorithmen so modifizieren, dass wir für Knoten, an denen abgebogen wird, eine Strafe auf den Badness-Wert rechnen.

Diese Art, Pfade, die viele Abbiegungen enthalten, durch schlechte Badness-Werte auszuschließen, ist eine einfache Möglichkeit, geradlinige Strecken zu erzwingen. Dies hat aber auch einen großen Nachteil. Dijkstras Algorithmus, den wir in allen Verfahren benutzen, beruht auf der sogenannten Subpfadoptimalität. Ist $P = (p_1, \ldots, p_k)$ der kürzeste Pfad von p_1 nach p_k, so gilt $P'(p_1, \ldots, p_i)$ ist der kürzeste Pfad von p_1 nach p_i, für alle $i \in [1, k]$. Diese Annahme gilt nicht mehr, wenn wir das Abbiegen bestrafen, wie am Beispiel in Abbildung 4.11a zu erkennen ist.



(a) Der kürzeste Weg von s nach u ist nicht Teil des kürzesten Wegs von s nach t.

(b) Kreis der den Knoten u doppelt enthält.

Abbildung 4.11: Beispiel für die Auswirkung von Abbiegekosten.

In diesem Beispiel wird auf dem kürzesten Weg von Knoten s zu Knoten t ein Umweg gelaufen, um die Anzahl der Abbiegungen zu reduzieren. Deshalb ist der im kürzesten Pfad von s nach t enthaltene Pfad von s nach u nicht der kürzeste Pfad von s nach u. Um bezüglich der neuen Längenfunktion mit Abbiegekosten ein optimales Ergebnis zu berechnen, ist deswegen Dijktras Algorithmus im Allgemeinen nicht mehr geeignet. Es gibt Möglichkeiten, auch bezüglich Länge und Kosten fürs Abbiegen optimale Pfade zu berechnen ([DGPW11], [GV11]). Für ein solches Verfahren wäre im Graph 4.11b der Kürzeste Weg von s nach t gegeben durch (s, v, u, t) und der Kürzeste Weg von s nach p wäre gegeben durch (s, q, u, p). Werden nun vom Teilwege-Algorithmus die Knoten p und t als Kandidaten gewählt, so erhalten wir den Pfad $(s, v, u, t, m, p, u, q, s)$. Dieser Pfad enthält den Knoten u aber doppelt, was nicht erwünscht ist.

Da wir Dijkstras Algorithmus nur in einem heuristischen Algorithmus zur Verbesserung der Ergebnisse benutzen, werden wir für unsere Zwecke auf die Korrektheit der kürzesten Wege verzichten. Wird von Dijkstras Algorithmus eine Kante exploriert, die mit dem Parentzeiger des aktuellen Knoten eine Abbiegung bildet, so werden die Kosten für das Abbiegen auf die Länge des Pfades addiert. Zwei Kanten in unserem Graphen sollen genau dann eine Abbiegung darstellen, wenn ihr gemeinsamer Knoten zu mindestens 3 Kanten inzident ist und der Winkel, den die beiden Kanten bezüglich der gegebenen Einbettung einschließen, einen Schwellenwert überschreitet.

4.3.5 Implementierungsdetails

Die Funktion FindMidwayPoints (4.4) berechnet, ausgehend von einem Halbwegspunkt h, alle möglichen Mittelpunkte der späteren Route, die über h erreichbar sind. Der Knoten h selber kann dabei Halbwegspunkt für beliebig viele Kandidaten sein. Die zu ihm gehörenden Kandidaten werden im Array candidate übergeben. Dieses Array hat dabei die feste Länge n und der Knoten $v \in V$ wird als Kandidat aufgefasst, wenn candidate$[v] = v$ gilt. Das candidate-Array wird im Verlauf des Algorithmus außerdem benutzt, um festzustellen, welche Knoten von h aus über welchen Kandidaten erreicht wurden. Die so gefundenen Mittelpunkte werden aber nicht zurückgegeben. Stattdessen wird über einen Seiteneffekt direkt der beste Mittelpunkt aktualisiert, falls ein besserer gefunden wird.

Der eigentliche Teilwege-Algorithmus über vier Wegpunkte (4.5) berechnet nun zunächst, wie schon die anderen Teilwege-Algorithmen zuvor, die Menge $\mathcal{K}$ der Kandidaten. Anschließend wird in Zeile 3 bis 8 für jeden Kandidaten p der Pfad zum Startknoten s solange

zurückverfolgt, bis die Hälfte des Weges erreicht wurde, um so den Halbwegspunkt h_p zu ermitteln. Wird h_p gefunden, so wird zugleich p im candidate-Array von h_p eingetragen. Anschließend wird für jeden Halbwegspunkt in der Funktion FindMidwayPoints nach möglichen Mittelpunkten gesucht. Zum Ende des Algorithmus ist der dabei beste gefundene Mittelpunkt in der Variablen bestMidwayPoint gespeichert.

Algorithm 4.4: FindMidwayPoints

Input: s, ω_s, c_s, candidate
Data: P, badness, bestMidwayPoint, bestBadness

foreach $v \in V$ **do**
  $(\omega'[v], c'[v], \text{parent}'[v]) \leftarrow (\infty, \infty, \perp)$;
$(\omega'[s], c'[s]) \leftarrow (\omega_s, c_s)$;
$\mathrm{Q} \leftarrow \{s\}$;
while $\mathrm{Q} \neq \emptyset$ **do**
  $v \leftarrow \arg\min_{q \in \mathrm{Q}}(\omega'[q])$;
  $\mathrm{Q} \leftarrow \mathrm{Q} \setminus \{v\}$;
  foreach $e = \{v, u\} \in E$ **do**
    if $\omega'[v] + (c(e) \cdot b(e)) < \omega'[u]$ **then**
      $(\omega'[u], c'[u], \text{parent}'[u]) \leftarrow (\omega'[v] + (c(e) \cdot b(e)), c'[v] + c(e), v)$;
      if $c'[u] < (1+\varepsilon)l/2$ **then**
        $\mathrm{Q} \leftarrow \mathrm{Q} \cup \{u\}$;
  if candidate$[v] = \perp$ **then**
    candidate$[v] \leftarrow$ candidate[parent$[v]$];
  if $c'[v] \in \mathcal{I}(l/2, \varepsilon)$ **and** *candidate*$[v] \neq \perp$ **then**
    $P_v \leftarrow P_v \cup \{\text{candidate}[v]\}$;
    $\text{badness}_v[\text{candidate}[v]] \leftarrow \omega'[v]$;
    if $|P_v| > 2$ **then**
      $P_v \leftarrow P_v \setminus \{\arg\max_{p \in P_v}(\text{badness}_v[p])\}$;
    if $P_v = \{p_1, p_2\}$ **and** $\text{badness}_v[p_1] + \text{badness}_v[p_2] <$ bestBadness **then**
      (bestMidwayPoint, bestBadness) $\leftarrow (v, \text{badness}_v[p_1] + \text{badness}_v[p_2])$;

Algorithm 4.5: Teilwege-Algorithmus über vier Wegpunkte

Input: Graph $G = (V, E)$, c, b, s, l, ε
Output: $P = (s, p_1, \ldots, p_k, s)$
Data: P, badness, bestMidwayPoint, bestBadness

$(\omega', c'\text{parent}') \leftarrow \text{dijkstra}(s, (1+\varepsilon)l/4)$;
$\mathcal{K} \leftarrow \{v \in V \mid c'[v] \in \mathcal{I}(l/4, \varepsilon)\}$;
foreach $p \in \mathcal{K}$ **do**
  $h_p \leftarrow p$;
  while $|c'[\text{parent}'[h_p]] - c'[p]/2| < |c'[h_p] - c'[p]/2|$ **do**
    $h_p \leftarrow \text{parent}'[h_p]$;
  $\mathcal{H} \leftarrow \mathcal{H} \cup \{h_p\}$;
  candidate$[h_p]$[p] $\leftarrow p$;
(bestMidwayPoint, bestBadness) $\leftarrow (\perp, \infty)$;
$\forall v \in V : P_v = \emptyset$;
foreach $h \in \mathcal{H}$ **do**
  FindMidwayPoints(h, $\omega'[h]$, $c'[h]$, candidate$[h]$);
$m \leftarrow$ bestMidwayPoint;
$\{p_1, p_2\} \leftarrow P_m$;
return $\text{SP}(s, p_1) \circ \text{SP}(p_1, m) \circ \text{SP}(m, p_2) \circ \text{SP}(p_2, s)$;

5. Evaluierung der Algorithmen

In diesem Kapitel werden wir die Leistung der vorgestellten Algorithmen näher untersuchen. Dazu beschreiben wir zunächst die Daten auf denen wir die Algorithmen laufen lassen. Anschließend analysieren wir die Algorithmen, dabei betrachten wir erst ihre spezifischen Parameter und vergleichen danach die Algorithmen miteinander. Schließlich zeigen wir noch einige Fallbeispiele von Joggingrouten die mit unseren Algorithmen berechnet wurden.

Die vorgestellten Algorithmen haben wir dabei alle in C++ implementiert und mit `g++` Version 4.6.2 (64 bit) kompiliert. Für die Experimente verwendeten wir einen Rechner mit zwei Intel Xeon E5-2670 Prozessoren, mit jeweils acht Kernen und einer Taktung von 2,6GHz und 64GiB DDR3-1600 RAM.

5.1 Datenaufbereitung

Für die Experimente in diesem Kapitel, sowie die Beispielbilder in den vorangegangenen Kapiteln, benutzen wir Kartenmaterial von OpenStreetMap[1]. Aus diesen Daten erstellen wir einen Graph mit fester Einbettung sowie eine Längenfunktion für die Kanten. Um die Badness der Kanten zu modellieren, benutzen wir Informationen über die Art der Straßen bzw. Wege und über die Nutzung einzelner Gebiete, die in den Daten von OpenStreetMap hinterlegt sind.

Für Straßen ist in den Daten, unter der Kennzeichnung *highway*, der Typ der Straße hinterlegt. Es können zum Beispiel Schnellstraßen, Fahrradwege, Wanderwege und weitere Typen unterschieden werden. In Tabelle 5.1b sind alle Straßentypen, die wir benutzen, aufgeführt. Autobahnen und andere für Jogger ungeeignete Straßenarten haben wir dabei ignoriert und auch nicht mit in den Graphen aufgenommen. Für den highway-Typ *Track*, welcher für Feld- und Waldwege steht, ist zusätzlich ein Maß (*grade*) angegeben, welches die Befestigung des Weges widerspiegelt. Auch diese Information nutzen wir zur Modellierung der Badness.

In den Daten sind einzelne Gebiete als Polygone dargestellt. Für diese ist unter der Kennzeichnung *landuse* die Nutzung des Gebietes angegeben. Es können so zum Beispiel Wohngebiete, Industriegebiete oder Parks unterschieden werden. In Tabelle 5.1a sind die verschiedenen Gebietsarten aufgelistet. Wir ordnen die Knoten unseres Graphen genau dann

[1] http://www.openstreetmap.org/

landuse-Typ	Badness
allotments	0,5
brownfield	1
cemetery	1
commercial	1
construction	1
farm	0,2
farmland	0,2
farmyard	0,3
forest	0,1
garages	1
grass	0,15
greenfield	0,1
greenhouse_horticulture	0,6
industrial	1
landfill	1
leisure	0,15
meadow	0,1
military	1
orchard	0,5
plant_nursery	0,6
quarry	1
railway	0,5
recreation_ground	0,2
reservoir	0,3
residential	0,8
retail	1
unclassified	1
village_green	0,2
vineyard	0,4

(a) Badnesswerte nach Gebietskategorie

highway-Typ	Badness
bridleway	0,6
crossing	0,6
cycleway	0,2
footway	0,5
ford	1
living_street	0,7
path	0,5
pedestrian	0,8
residential	0,9
road	0,8
secondary	1
secondary_link	1
service	0,9
steps	0,5
tertiary	1
tertiary_link	1
track	0,15
track, grade1	0,1
track, grade2	0,15
track, grade3	0,25
track, grade4	0,35
track, grade5	0,45
unclassified	0,9

(b) Badnesswerte nach Straßenkategorie

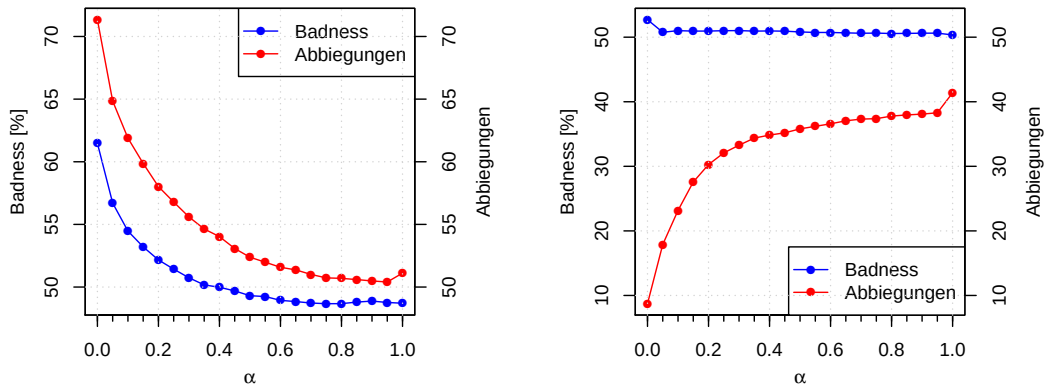
Tabelle 5.1: Zuordnungen für die Berechnung der Badness

einem Gebiet zu, wenn der entsprechende Punkt des Knotens innerhalb des Polygons dieses Gebiets liegt.

Wir haben somit für alle Knoten und Kanten des Graphen eine Zuordnung zu Gebiets- bzw. Straßentypen. Für diese Typen legen wir Badness-Werte fest. In den Tabellen 5.1 sind die Badnesswerte, die wir für die Experimente genutzt haben, zu sehen. Anschließend berechnen wir die Badness b einer einzelnen Kante $\{v, u\}$ des Graphen wie folgt.

$$b(\{v,u\}) := \begin{cases} b(\text{highway}(\{v,u\})) & \text{falls highway}(\{v,u\}) = \text{track} \\ \frac{1}{2}(b(\text{highway}(\{v,u\})) + \max(b(v), b(u)) & \text{sonst} \end{cases}$$
$$b(v) := b(\text{landuse}(v))$$

In den folgenden Abschnitten werden wir, als Grundlage für die Algorithmen, Kartenmaterial von OpenStreetMap vom 5.8.2012 benutzen. Der gewählte Ausschnitt umfasst die Region Karlsruhe und erstreckt sich von den Koordinaten 8°18'19" N, 48°56'21" O bis 8°30'0" N, 49°5'52" O. Auf anderen Kartenausschnitten haben wir ähnliche Resultate erhalten.



(a) Ergebnisse für den Straßengraph von Karlsruhe.

(b) Ergebnisse für einen Gittergraph mit zufälliger Badness.

Abbildung 5.1: Auswirkung der Wahl von α auf die Badness und die Anzahl der Abbiegungen.

5.2 Auswertung der Parameter

Die vorgestellten Algorithmen besitzen eine Vielzahl von Parametern zur Steuerung ihres Verhaltens. In diesem Abschnitt ermitteln wir zunächst, wie diese Parameter belegt werden sollten, um gute Ergebnisse zu erzielen. Anschließend vergleichen wir die einzelnen Verfahren.

5.2.1 Parameter des FacesGreedy-Algorithmus

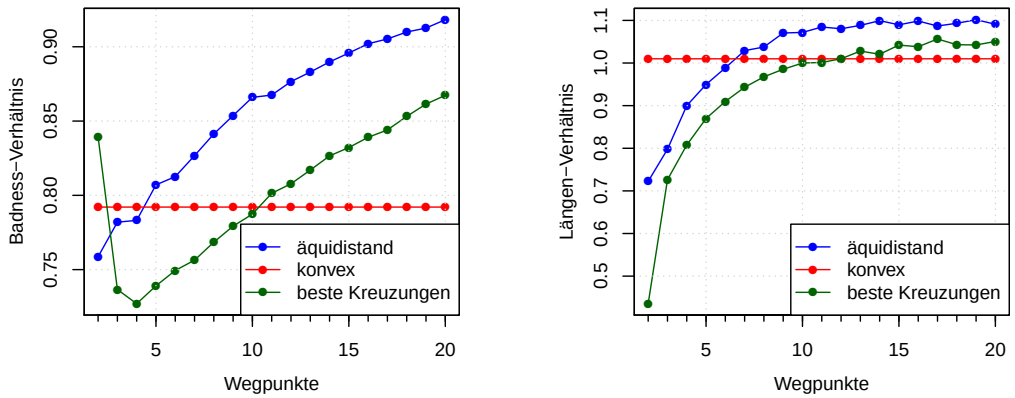
Zunächst analysieren wir die Parameter des Facetten-basierten Ansatz. Dieser besitzt zum einem Ein Parameter der festlegt wie stark die Kräfte die Wahl der nächsten Facette beeinflussen, und zum anderen ein Parameter der Angibt wie viele Wegpunkte für das Glätten verwendet werden.

Zur Auswahl der nächsten Facette maximiert der FacesGreedy-Algorithmus den folgenden Ausdruck 4.1 von Seite 22, welcher eine Linearkombination von Kräften und dem Flächen-Umfang-Verhältnis bildet.

$$\arg\max_{f_i \in \mathcal{A}} \left(\alpha \cdot \text{force}(f_i) + (1-\alpha) \cdot \text{shape}(f_i)\right)$$

Für den Wert α kann dabei ein beliebiger Wert aus dem Intervall $[0, 1]$ gewählt werden. Ist $\alpha = 1$, so werden nur die Kräfte berücksichtigt. In diesem Fall erwarten wir eine niedrige Badness der Resultate. Ist $\alpha = 0$, so wird nur das Verhältnis von Fläche zu Umfang berücksichtigt. Wir erhoffen uns hiervon eine niedrige Zahl von Abbiegungen.

Abbildung 5.1a zeigt die Auswirkung der Wahl von α auf den FacesGreedy-Algorithmus. Jeder Messpunkt entspricht dabei dem Mittelwert von 1000 zufälligen Anfragen, wobei Routen der Länge 10km bei einer erlaubten Abweichung von $\varepsilon = 0{,}1$ gesucht wurden. Wie erwartet, sinkt die Badness mit steigender Gewichtung der Kräfte. Allerdings sinkt parallel zur Badness auch die Anzahl der Abbiegungen. Der Verlauf der Kurve zeigt, dass in diesem Versuch die Anzahl der Abbiegungen nicht durch ein besseres Verhältnis von Fläche zu Umfang reduziert wird. Interessant ist dabei auch der starke Zusammenhang zwischen Badness und Abbiegungen. Es scheint, dass eine niedrige Badness direkt mit einer niedrigen Anzahl von Abbiegungen zusammenhängt.



(a) Einfluss der Wegpunkte auf die Badness. (b) Einfluss der Wegpunkte auf die Länge.

Abbildung 5.2: Resultate der Glättungsstrategien im Verhältnis zur Anzahl der gewählten Wegpunkte.

Vergleich mit zufälligem Graphen. Um dies näher zu untersuchen, haben wir den FacesGreedy Algorithmus ein weiteres Mal auf einem regelmäßigem Gittergraphen mit zufälliger, gleichverteilter Badness getestet. Das Gitter hat eine Kantenlänge von 100 Metern und umfasste 100×100 Knoten.

Abbildung 5.1b zeigt die Auswirkung der Wahl von α auf einem solchen Gittergraphen. Es ist zu erkennen, dass die Badness etwa den Erwartungswert der Badness, im zufällig generiertem Graphen, erreicht und nur gering durch die Kräfte beeinflusst wird. Die Anzahl der Abbiegungen nimmt allerdings, mit steigender Gewichtung des Flächen-Umfang-Verhältnis wie erwartet ab. Wir schließen daraus, dass in einem natürlichen Straßennetzwerk die Badness und die Anzahl der Abbiegungen nicht vollständig unabhängig voneinander sind. Es ist vorstellbar, dass es einige schöne Straßen mit niedriger Badness gibt, die über mehrere Kreuzungen hinweg verlaufen und keine Abbiegungen beinhalten. Ist eine solche Straße auf Grund der niedrigen Badness komplett im Ergebnis enthalten führt dies auch zu einer niedrigen Anzahl von Abbiegungen.

Auf Grund dieser Ergebnisse wählen wir im weiteren Verlauf der Arbeit einen Wert für α der hier zu niedriger Badness führte. Wir wollen aber dennoch das Flächen zu Umfang Verhältnis nicht komplett vernachlässigen. Deshalb und weil sich die Badness für noch größere Werte von α im Versuch nicht weiter verbessert hat, setzen wir $\alpha = 0{,}8$.

Anzahl der Wegpunkte für die Glättungsstrategien

Zwei der drei in Kapitel 4.2.5 vorgestellten Glättungsstrategien besitzen einen Parameter, der die Anzahl der beizubehaltenden Wegpunkte bestimmt. Wir untersuchen, wie sich die Anzahl der Wegpunkte auf die Badness der Pfade auswirkt. Wir vergleichen dazu die Ergebnisse der Algorithmen, angefangen mit nur 2 Wegpunkten bis zu 20 Wegpunkten. Zum Vergleich ist auch das Ergebnisse der konvexen Glättung angegeben, auf welche die Anzahl der Wegpunkte aber keine Auswirkung hat. Auch in diesem Versuch wurden wieder pro Messpunkt 1000 zufällige Anfragen betrachtet.

Abbildung 5.2a zeigt die Badness der Geglätteten Pfade im Verhältnis zur Badness der ungeglätteten Pfade. Es ist zu sehen, dass bei allen drei Glättungsstrategien die Badness der berechneten Lösung deutlich unter der Badness des ursprünglichen Pfades liegt. Es ist außerdem zu erkennen, dass eine geringere Anzahl von Wegpunkten im Allgemeinen zu einer besseren Badness führt.

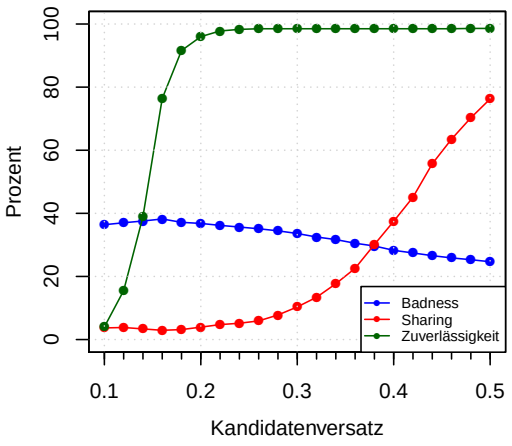


Abbildung 5.3: Resultate des Teilwege-Algorithmus in Bezug zum Abstand der Kandidaten zum Startknoten.

Abbildung 5.2b zeigt die Veränderung der Länge der geglätteten Pfade. Hier ist zu erkennen, dass eine geringe Anzahl von Wegpunkten in einer deutlichen Abnahme der Länge resultiert. Hierdurch lässt sich auch der Ausreißer in der Badness bei der Glättung über beste Kreuzungen mit nur zwei Wegpunkten erklären. Die Strecke wird hier derart verkürzt, dass sie einfach nicht mehr lang genug ist, um Gebiete mit niedriger Badness zu erreichen.

Die Tatsache, dass die geglätteten Pfade zum Teil sogar länger sind als die ursprünglichen Pfade, liegt daran, dass die Wegpunkte über den bezüglich ω kürzesten Weg miteinander verbunden werden. Dieser Weg muss nicht mit Weg übereinstimmen, der der Länge nach am kürzesten ist.

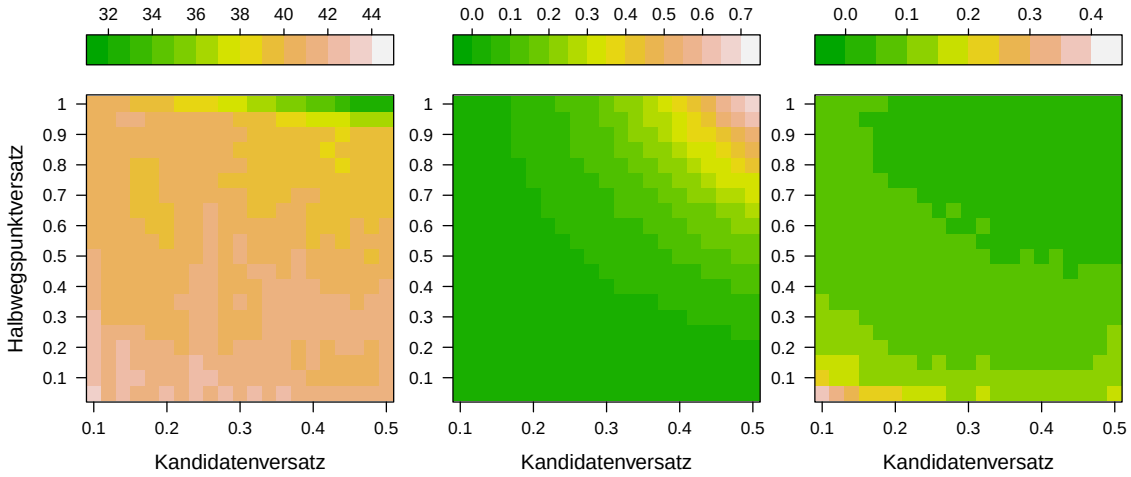
Da die Glättungsstrategien die Länge eines Pfades möglichst erhalten sollen, können wir nicht beliebig wenig Wegpunkte wählen. Deshalb wählen wir im weiteren Verlauf stets so viele Wegpunkte wie in diesem Versuch benötigt wurden um die Länge des Pfades in etwa zu erhalten, Dies waren sechs Wegpunkte bei äquidistantem Glätten und neun Wegpunkte bei der Glättung über die besten Kreuzungen.

5.2.2 Parameter der Teilwege-Algorithmen

Als nächstes werden wir die Parameter der Teilwege-Algorithmen analysieren. Bei diesen Algorithmen lässt sich die Position der Kandidaten, sowie die Position der Halbwegspunkte, sofern vorhanden, über Parameter steuern.

Der TW3 Algorithmus kann über den Kandidatenversatz parametrisiert werden. Hierbei wurde zunächst ein Kandidatenversatz von 1/3 vorgeschlagen. Wir wollen nun feststellen, für welchen Wert der Algorithmus die besten Ergebnisse erzielt. Dazu testen wir, wie sich die berechneten Touren verändern, wenn wir den Kandidatenversatz zwischen 0,1 und 0,5 variieren. Abbildung 5.3 zeigt die Ergebnisse, dabei entspricht ein Messpunkt jeweils dem Mittelwert von 1000 zufälligen Anfragen.

Für Werte kleiner 0,2 findet der Algorithmus nur noch wenig Ergebnisse. Dies liegt daran, dass alle drei Seiten des Dreiecks mit Dijkstras Algorithmus berechnet werden, also kürzeste Wege sind. Sind nun die beiden Kandidaten nur $0{,}2l$ (wobei l die gesuchte Gesamtlänge ist) von s entfernt, so ist ihr gegenseitiger Abstand maximal $0{,}4l$. Die fehlende Dreiecksseite muss aber $0{,}6l$ lang sein, was keinem kürzesten Weg entspricht. Der Grund,



(a) Badness der Resultate in Prozent.

(b) Sharing der Resultate.

(c) Rechenzeit pro Alternative in Sekunden.

Abbildung 5.4: Auswirkung der Position von Kandidaten und Halbwegspunkten auf den Teilwege-Algorithmus mit vier Wegpunkten.

warum der Algorithmus für Werte kleiner 0,25 überhaupt noch Ergebnisse findet, ist der Spielraum bei der Länge, die der Parameter ε einräumt.

Dass das Sharing der berechneten Routen mit steigendem Abstand der Kandidaten zum Startknoten zunimmt, entspricht der Erwartung. Der Algorithmus ist so aufgebaut, dass ein Sharing der Strecke von s aus gesehen nur vor den Kandidaten möglich ist. Somit ist klar, dass bei einem größeren Abstand der Kandidaten zum Startknoten s auch ein höheres Sharing möglich ist.

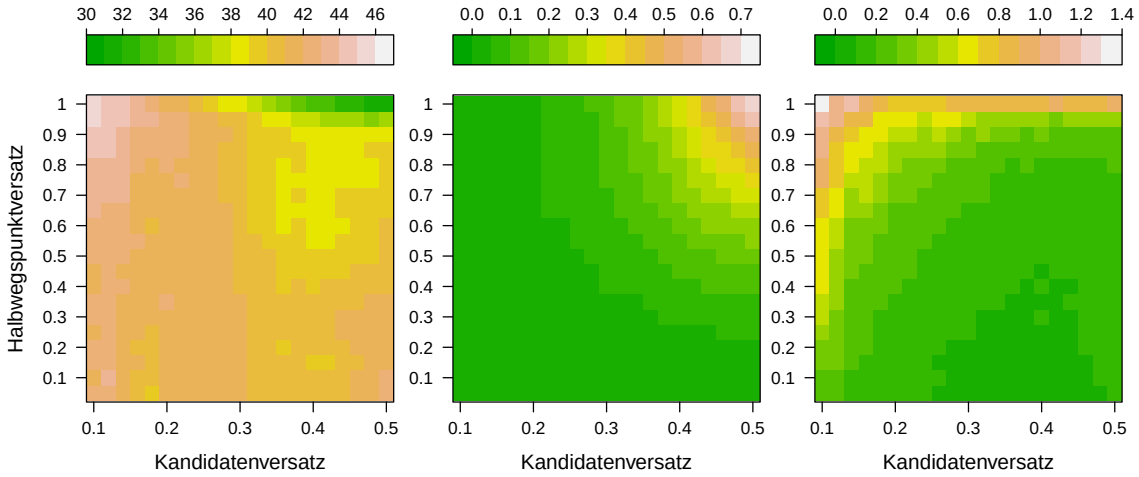
Das erhöhte Sharing ist auch eine Begründung dafür, warum die Badness mit steigendem Abstand der Kandidaten zum Startknoten abnimmt. Gibt es einen bezüglich Badness optimalen Pfad, der vom Startknoten wegführt, so kann dieser bei großem Kandidatenversatz doppelt im Resultat vorhanden sein. Dies führt zu hohem Sharing und niedriger Badness. Ist der Kandidatenversatz gering, so kann dieser optimale Pfad nur einmal enthalten sein. Da die Route aber ein Kreis sein muss, muss ein zweiter schlechterer Pfad zum Startknoten s aufgenommen werden, dadurch sinkt das Sharing und die Badness steigt.

Wir müssen nun für die folgenden Experimente einen Kompromiss zwischen Badness und Sharing finden. In den weiteren Versuchen entscheiden wir uns für einen Kandidatenversatz von 0,4 für den Teilwege-Algorithmus über drei Wegpunkte.

Kandidatenwahl des Teilwege-Algorithmus über vier Wegpunkte

Auch beim Teilwege-Algorithmus über vier Wegpunkte (TW4) sowie dem Teilwege-Algorithmus mit bidirektionaler Suche (TWB) kann der Kandidatenversatz variiert werden. Zusätzlich bieten diese Algorithmen die Möglichkeit, die Lage der Halbwegspunkte über den Halbwegspunktversatz zu steuern. Im folgenden untersuchen wir, wie sich Kandidatenversatz und Halbwegspunktversatz auf die Badness, das Sharing und die Laufzeit der beiden Algorithmen auswirken. In den Heatmaps 5.4 und 5.5 entspricht dabei ein Datenpunkt je dem Mittelwert des Ergebnisses von 100 zufälligen Anfragen.

Abbildung 5.4a zeigt die Badness der Lösungen des Teilwege-Algorithmus über vier Wegpunkte. Diese sinkt vor allem bei einem großen Halbwegspunktversatz. Die besten Werte bezüglich Badness werden dabei bei einem Kandidatenversatz von 0,5 und einem Halbwegspunktversatz von 1 erreicht. Aus demselben Grund wie beim Teilwege-Algorithmus



(a) Badness der Resultate in Prozent.

(b) Sharing der Resultate.

(c) Rechenzeit pro Alternative in Sekunden.

Abbildung 5.5: Auswirkung der Position von Kandidaten und Halbwegspunkten auf den Teilwege-Algorithmus mit bidirektionaler Suche.

über drei Wegpunkte ergibt sich auch bei diesem Algorithmus ein großes Sharing, wenn die Halbwegspunkte bzw. Kandidaten weit vom Startpunkt entfernt liegen. Dies ist in Abbildung 5.4b zu sehen.

In Abbildung 5.4c ist die Rechenzeit in Millisekunden pro gefundener Alternative dargestellt. Der Algorithmus ist vor allem bei niedrigem Halbwegspunktversatz langsam. Es ist aber auch zu erkennen, dass bei einem Halbwegspunktversatz von 1 die benötigte Rechnenzeit wieder ansteigt.

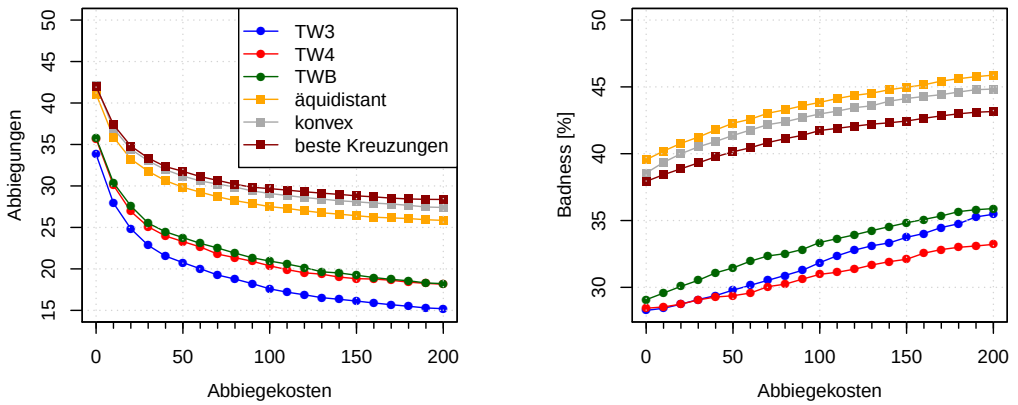
Um Routen mit niedriger Badness und niedrigem Sharing zu erhalten, wählen wir in den weiteren Experimenten einen Kandidatenversatz von 0,4 und einen Halbwegspunktversatz von 0,6.

Für den Teilwege-Algorithmus mit bidirektionaler Suche ergeben sich die gleichen Möglichkeiten zur Variation des Kandidaten Versatzes bzw. des Halbwegspunkt Versatzes. Abbildung 5.5 zeigt dementsprechend die gleichen drei Heatmaps für diesen Algorithmus.

Interessant ist, dass bei diesem Algorithmus die Badness, zu sehen in Abbildung 5.5a, nicht mehr nur vom Halbwegspunktversatz dominiert wird. Es ist zu erkennen, dass auch der Kandidatenversatz die Badness deutlich beeinflusst. Vor allem für einen Halbwegspunktversatz kleiner als 0,9 führt ein Kandidatenversatz von etwa 0,4 zu den besten Ergebnissen bezüglich Badness.

Beim Sharing, in Abbildung 5.5b, zeigt sich hingegen ein ähnliches Bild wie für den Teilwege-Algorithmus über vier Wegpunkte. Ein deutlicher Unterschied ist hingegen bei der Laufzeit zu erkennen. Wie in Abbildung 5.5c zu sehen, nimmt diese deutlich mit steigendem Halbwegspunktversatz zu. Dies liegt vor allem daran, dass mit steigendem Halbwegspunktversatz auch mehr Halbwegspunkte gefunden werden. Da die Laufzeit des Algorithmus quadratisch in der Anzahl der Halbwegspunkte ist, aber die Anzahl der tatsächlich gefunden Alternativen nur linear mit der Anzahl der Halbwegspunkte steigt, führt dies zu schlechteren Ergebnissen bei der Laufzeit.

Wir nutzen den Teilwege-Algorithmus mit bidirektionaler Suche im weiteren Verlauf mit einem Kandidatenversatz von 0,4 und einem Halbwegspunktversatz von 0,6.



(a) Einfluss auf die Anzahl der Abbiegungen.

(b) Einfluss auf die Badness.

Abbildung 5.6: Auswirkung von Abbiegekosten auf die vorgestellten Verfahren.

Abbiegekosten

Für alle Algorithmen, die Dijkstras Algorithmus verwenden, haben wir zusätzliche Abbiegekosten eingeführt. Die Abbiegekosten werden dabei in Metern Umweg auf Strecken mit Badness 1 angegeben. Im Folgenden untersuchen wir, wie sich diese Abbiegekosten tatsächlich auf die Anzahl der in den Joggingrouten enthaltenen Abbiegungen auswirken. Darüber hinaus wollen wir feststellen, inwiefern sich die Abbiegekosten auf die Badness der Routen auswirken.

Abbildung 5.6a zeigt den Mittelwert der in einer Joggingtour enthaltenen Abbiegungen. Wieder wurden dafür pro Messpunkt 1000 zufällige Anfragen ausgewertet. Es ist zu erkennen, dass sich bei allen Algorithmen schon für geringe Abbiegekosten eine deutliche Reduktion der tatsächlich enthaltenen Abbiegungen ergibt. Es ist aber auch für immer weiter steigende Abbiegekosten zu erkennen, dass kaum noch eine Reduktion der tatsächlichen Abbiegungen erreicht werden kann.

In Abbildung 5.6b ist die Veränderung der Badness der gleichen Touren zu sehen. Es ist zu erkennen, dass eine Erhöhung der Abbiegekosten zu Touren mit schlechterer Badness führt. Im Gegensatz zu den Abbiegungen selber, scheint die Steigung der Badness auch für hohe Abbiegekosten nicht nachzulassen.

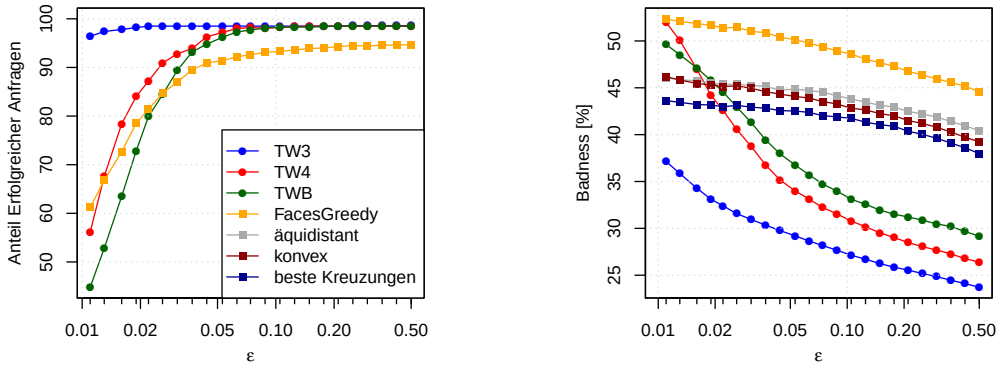
Auch bei den Abbiegekosten müssen wir deshalb einen Kompromiss zwischen Badness und Abbiegungen eingehen. Für die weiteren Versuche wählen wir Abbiegekosten von 100 Metern Umweg.

5.3 Vergleich der vorgestellten Algorithmen

In diesem Abschnitt wollen wir zunächst betrachten, wie sich die in der Problemstellung enthaltenen Ziellänge l und die erlaubte Abweichung ε auf die einzelnen Algorithmen auswirken. Schließlich werden wir die vorgestellten Algorithmen vergleichen.

Auswirkung von ε

Zunächst wollen wir betrachten, welche Auswirkung die Wahl von ε hat. Es ist zu erwarten, dass für Werte nahe oder gleich 0 kaum noch Lösungen bzw. gar keine Lösungen gefunden werden. Abbildung 5.7a zeigt in wie viel Prozent der Fälle die einzelnen Verfahren mindestens eine Lösung finden konnten. Die einzelnen Glättungsstrategien des FacesGreedy Algorithmus wurden hierbei nicht separat aufgeführt, da sie die Zuverlässigkeit des zugrunde liegenden FacesGreedy Algorithmus nicht verändern.



(a) Anteil der Aufrufe, bei denen ein Ergebnis gefunden wurde.

(b) Veränderung der Badness in Abhängigkeit von ε.

Abbildung 5.7: Einfluss von ε auf die Algorithmen.

Wie erwartet, haben die meisten Algorithmen für kleine Werte von ε ein Problem, gültige Lösungen zu finden. Wird keine Lösung gefunden, kann aber keine Aussage darüber getroffen werden ob dies am Algorithmus liegt, oder ob tatsächlich keine gültige Route existiert. Einzig der TW3 Algorithmus findet auch für $\varepsilon = 0{,}02$ noch in etwa 98% der Fälle eine Lösung. Ab einem Wert von etwa $\varepsilon = 0{,}1$ sind die Algorithmen relativ stabil und finden in den meisten Fällen Lösungen. Darüber hinaus ist zu erkennen, dass der FacesGreedy Algorithmus auch für große Werte von ε deutlich hinter den Teilwege-Algorithmen liegt.

Wie in Abbildung 5.7b zu erkennen ist, wirkt sich ein größeres ε auch positiv auf die Badness der berechneten Joggingrouten aus. Besonders die Teilwege-Algorithmen über vier Wegpunkte berechnen für kleines ε sehr schlechte Touren, verbessern sich dann aber mit steigendem ε schnell. Der FacesGreedy Algorithmus profitiert hingegen nur gering von einem großen ε. Unabhängig von ε liefert der TW3 Algorithmus die Ergebnisse mit der niedrigsten Badness und höchsten Zuverlässigkeit.

Auswirkung der Länge

Bei der theoretischen Betrachtung der Laufzeit für die einzelnen Algorithmen haben wir diese nur von der Größe des Graphen abhängig gemacht. Rein praktisch wirkt sich aber auch die Länge der gesuchten Joggingroute auf die Laufzeit der Algorithmen aus. Wir untersuchen nun, wie stark die Laufzeit mit wachsender Streckenlänge steigt.

Abbildung 5.8 zeigt die Mittelwerte der erzielten Laufzeiten für je 1000 zufällige Anfragen. Wie erwartet, steigt bei allen Algorithmen die Laufzeit mit der Länge der gesuchten Route. Besonders beim FacesGreedy Algorithmus steigt die benötigte Rechenzeit stark mit

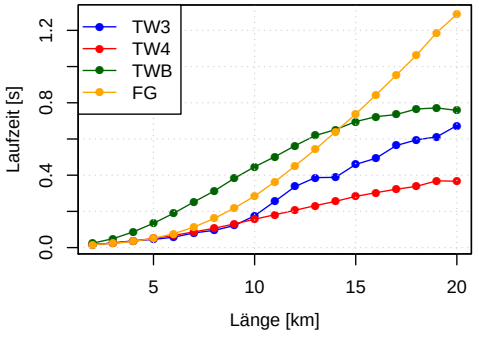


Abbildung 5.8: Laufzeit der Algorithmen in Abhängigkeit zur gesuchten Länge l.

Algorithmus	Zeit	Länge	Std-Abw.	Bad.	Shring	Abb.	Zuv.
TW3	179ms	10,01km	0,57	27,2%	52,5%	16,1	98,5%
TW3-komplett	3579ms	10,01km	0,57	27,2%	52,2%	16,1	98,5%
TW4	155ms	10,13km	0,41	30,9%	23,5%	20,3	98,1%
TWB	446ms	10,05km	0,52	33,3%	13,8%	20,9	98,3%
FacesGreedy	285ms	9,89km	0,58	48,6%	0,2%	50,6	93,3%
mit äqu. S.	288ms	9,61km	2,06	43,8%	6,4%	27,5	93,3%
mit kon. S.	296ms	9,72km	2,23	42,9%	6,8%	29,0	93,3%
mit best. S.	293ms	9,47km	1,98	41,7%	5,9%	29,6	93,3%

Tabelle 5.2: Die vorgestellten Algorithmen im Vergleich, getestet mit jeweils 1000 zufälligen Anfragen.

zunehmender Streckenlänge. Ab einer Länge von 15km ist der FacesGreedy Algorithmus der langsamste Algorithmus. Vor allem für lange Strecken ist der TW4 Algorithmus der schnellste hier vorgestellte Algorithmus.

Vergleich der Algorithmen

Wir vergleichen nun die Resultate der einzelnen Algorithmen. Dabei wählen wir für die Algorithmen jeweils jene Konfiguration, die im letzten Abschnitt gute Ergebnisse erzielte.

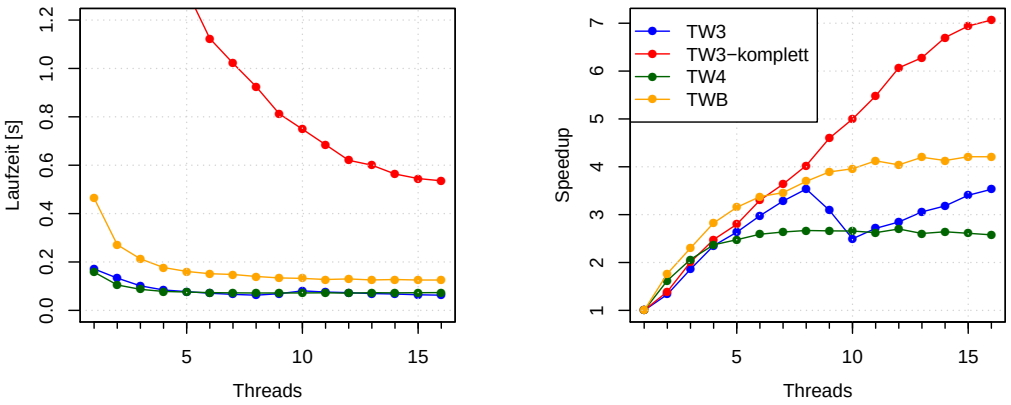
Tabelle 5.2 zeigt die erzielten Ergebnisse der einzelnen Algorithmen. Mit TW3-komplett wird dabei der Teilwege-Algorithmus über drei Wegpunkte, bei dem das verschärfte Stoppkriterium, wie auf Seite 27 beschrieben, nicht verwendet wird. Wir interessieren uns vor allem für die Qualität der berechneten Lösungen. Diese ist Gegeben durch die Mittelwerte der Badness, des Sharings und der Anzahl der Abbiegungen. Zusätzlich sind in der Tabelle die Rechenzeit, die erzielten Länger, die Standartabweichung der Länge und die Zuverlässigkeit angegeben.

Betrachten wir nur die Badness, so liefert TW3 die besten Ergebnisse, wobei alle Teilwege-Algorithmen deutlich vor den FacesGreedy Algorithmen liegen. Betrachten wir das Sharing, so erzielt der normale FacesGreedy Algorithmus die besten Werte, da dieser Lösungen mit minimal notwendigem Sharing berechnet. Zwar kann das Sharing durch die einzelnen Glättungsstrategien nachträglich wieder erhöht werden. Trotzdem besitzen auch diese geglätteten Lösungen noch ein deutlich geringeres Sharing als die Lösungen der Teilwege Algorithmen.

Betrachtet man die Anzahl der Abbiegungen, so erzielt vor allem der normale FacesGreedy Algorithmus sehr schlechte Ergebnisse. Die Anzahl der Abbiegungen kann aber durch das Anwenden einer Glättungsstrategie beinahe halbiert werden. Die geringste Anzahl an Abbiegungen erreicht der TW3 Algorithmus.

Auffällig ist auch, dass im Mittel die Teilwege-Algorithmen Routen berechnen, die geringfügig länger sind als die gesuchte Länge, wohingegen FacesGreedy eher kürzere Routen berechnet. Darüber hinaus wird die mittlere Länge der Routen durch eine Glättungsstrategie nur geringfügig gesenkt. Die Standardabweichung (Std-Abw.) der Länge hingegen steigt durch die Glättungsstrategien deutlich. Bei einer gesuchten Länge von 10km und $\varepsilon = 0{,}1$ sind Abweichungen von bis zu einem Kilometer zulässig. Die Standardabweichung aller Glättungsstrategien überschreitet diesen Wert.

Des weiteren kann der Tabelle entnommen werden, dass wie erwartet die stärkere Abbruchbedingung für TW3 die Qualität der Ergebnisse nicht beeinträchtigt, wohl aber die Rechenzeit reduziert. In diesem Beispiel erreichen wir hierdurch eine Beschleunigung um den Faktor 20.



(a) Absolute Laufzeit

(b) Speedup

Abbildung 5.9: Verbesserung der Laufzeit die durch die Parallelisierung erreicht wurde.

Parallelisierung

In Abschnitt 4.3.4 auf Seite 32 haben wir eine einfache Möglichkeit zur Parallelisierung der Teilwege-Algorithmen vorgestellt. Wir werden nun untersuchen, wie sich die Anzahl der benutzten Threads auf die Laufzeit der Algorithmen auswirkt. Die Ergebnisse dieser Versuche sind in Abbildung 5.9 zu sehen. Wie zuvor, entspricht dabei ein Messpunkt dem Mittel von 1000 zufälligen Anfragen.

Wir betrachten nun den Speedup, der durch die parallele Ausführung der Algorithmen erreicht wird (Abbildung 5.9b). Hier kann man erkennen das der TW4 Algorithmus sowie der TWB Algorithmus ab einer gewissen Anzahl benutzter Threads keine Verbesserung mehr zeigen. Dies ist aber recht leicht zu erklären. Die Anzahl der gefundenen Halbwegspunkte schwankt bei Anfragen dieses Testlaufs in etwa zwischen fünf und 20. Da unsere Parallelisierung nur die gefundenen Halbwegspunkte an die einzelnen Threads verteilt, ist es klar, dass dies ineffizient ist, falls weniger Halbwegspunkte existieren als Threads zur Verfügung stehen.

Darüber hinaus fällt auf, dass der TW3 Algorithmus bei neun und zehn Threads deutlich schlechteren Speedup aufweist als noch bei nur acht Threads. Um dies zu erklären, vergleichen wir die Kurve des TW3 Algorithmus mit der des TW3-komplett. In der Speedup-Kurve des TW3-komplett ist keinerlei Einbruch zu erkennen woraus wir folgern, dass die verschärfte Abbruchbedingung, die der TW3 nutzt, mit Ursache für den Einbruch des Speedups beim TW3 Algorithmus ist.

Trotzdem bleibt die Frage was genau diesen Einbruch verursacht. Wegen der Abbruchbedingung muss der TW3 Algorithmus nicht alle Kandidaten behandeln, eine Vermutung war deshalb, dass sich durch die Parallelisierung die Anzahl der tatsächlich überprüften Kandidaten erhöht. Messungen haben aber gezeigt, dass diese Anzahl gerade mal von im Schnitt 65 überprüften Kandidaten bei einem Thread auf 68 Kandidaten bei 16 Threads steigt. Auch bei neun bzw. zehn Threads weist die Zahl der tatsächlich überprüften Kandidaten keine Ausreißer auf.

Eine weitere Überlegung die den Einbruch des Speedups erklären könnte, ist, dass bei der von uns verwendeten Maschine ab neun Threads ein zweiter Prozessor neu dazugeschaltet wird. Es könnte sein, dass dieser Einbruch durch die Speicherbus-Architektur im Fall des TW3 Algorithmus mit verschärfter Abbruchbedingung zu erklären ist.

5.4 Fallbeispiele

Wir werden die Evaluierung der Algorithmen damit abschließen, dass wir einige Beispiele für Routen zeigen, die mit unseren Algorithmen generiert wurden. Hierzu betrachten wir drei Anfragen im Gebiet von Karlsruhe, mit den Längen 8km, 12km und 5km. Anhand diesen Beispielen werden wir noch ein mal die wichtigsten Unterschiede, zwischen den Algorithmen, herausstellen.

Abbildung 5.10a zeigt das Ergebnis des Teilwege-Algorithmus mit bidirektionaler Suche (TWB). Dieser Algorithmus optimiert aller Kriterien für schöne Joggingrouten relativ gut. Im Beispiel liegt er nur 38 Meter über der gesuchten Länge. Mit nur elf Abbiegungen ist es zudem leicht sich die Route zu merken.

Deutlich wird der Unterschied zum TW3 Algorithmus (Abbildung 5.10c). Dessen Ergebnisse haben im Allgemeinem ein höheres Sharing. Dies erlaubt es zu nächst das Stadtgebiet auf dem kürzestem Weg zu verlassen, dann im Wald eine lange und schöne Strecke zu laufen, und anschließend zum Startpunkt zurückzukehren.

Die FacesGreedy-Algorithmen schneiden deutlich schlechter ab. Die Badness sowie die Anzahl der Abbiegungen liegt deutlich höher als bei den Teilwege-Algorithmen. Die Glättungsstrategien können dies nur zum Teil ausgleichen, führen aber zu einer deutlichen Abnahme der Länge.

Betrachten wir nun die Zweite Anfrage der Länge 12km (Abbildung 5.11). Auch hier liegen die FacesGreedy-Algorithmen hinter der Teilwege-Algorithmen zurück. Vor allem für die ungeglättete Variante (Abbildung 5.11f) gilt, das sie viele Abbiegungen produziert, weshalb es schwer wird sich die Route zu merken.

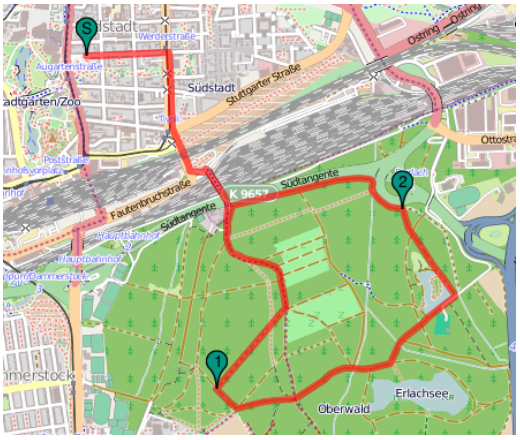
In Abbildungen 5.11a und 5.11b kann man einen wesentlichen Unterschied zwischen dem TWB und dem TW4 Algorithmus sehen. Der TW4 Algorithmus sucht eine Verbindung der Kandidaten, die die gesuchte Gesamtlänge gut annähert. Daher gibt es in Abbildung 5.11b einen Schlenker am Mittelpunkt, der sich am rechten Bildrand befindet. Dieser Schlenker führt zu zusätzlichen Abbiegungen. Der TWB Algorithmus sucht unter gewissen Randbedingungen den kürzesten Weg zwischen den Kandidaten, und produziert diesen Schlenker daher nicht.

Wir betrachten nun die letzte Anfrage, der Länge 5km (Abbildung 5.12). Auch hier zeigt sich ein ähnliches Bild wie zuvor. Es ist zu erkennen, dass die Algorithmen auch im Stadtgebiet kleinere Grünstreifen finden und diese erfolgreich nutzen um schöne Joggingtouren zu erstellen.

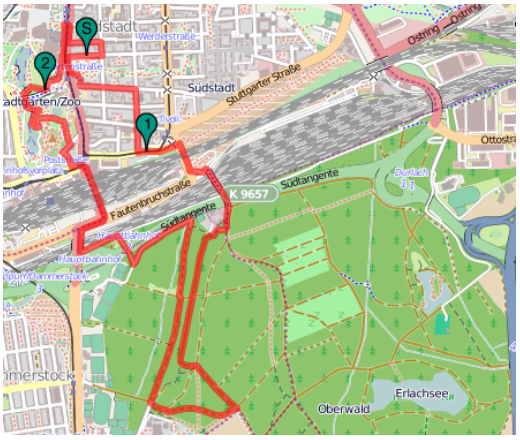
Hier ist zu erkennen das die Versionen des FacesGreedy-Algorithmus, die eine Glättung verwenden (Abbildung 5.12d und 5.12e) bezüglich Badness und der Anzahl Abbiegungen durchaus mit den Teilwege-Algorithmen mithalten können. Durch die deutliche Reduzierung der Länge die damit einhergeht liegen sie aber trotzdem hinter den Teilwege-Algorithmen zurück.

(a) TWB l=8038m b=35% Abbiegungen=11

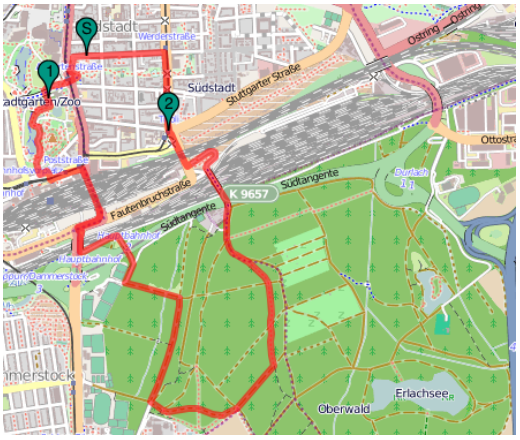
(b) TW4 l=7996m b=35% Abbiegungen=14



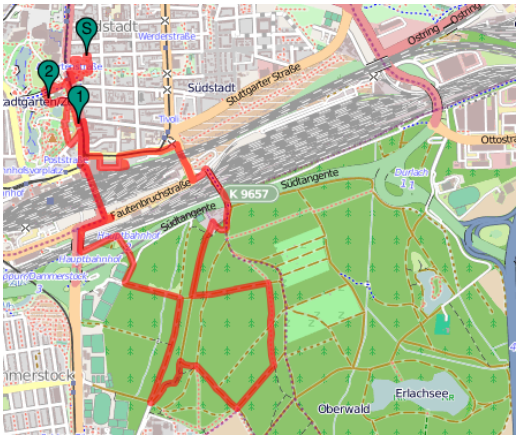
(c) TW3 l=7923m b=34% Abbiegungen=15



(d) FacesGreedy, beste Kreuzungen l=7767m b=43% Abbiegungen=43

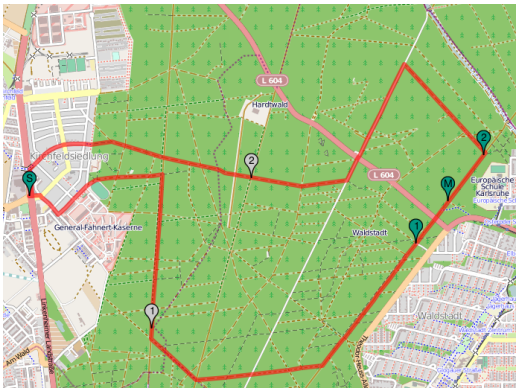


(e) FacesGreedy, konvex l=6637m b=44% Abbiegungen=28

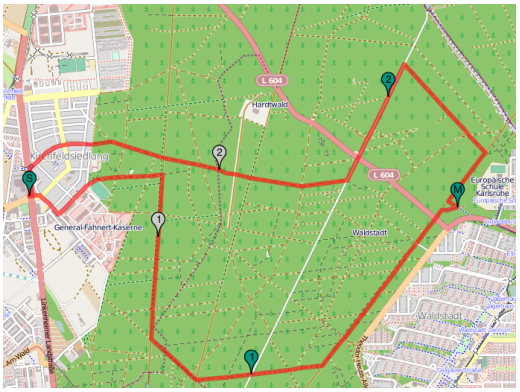


(f) FacesGreedy l=7883m b=44% Abbiegungen=39

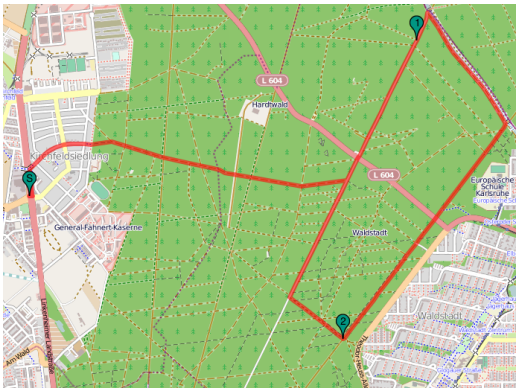
Abbildung 5.10: Ergebnisse der Algorithmen, bei einer gesuchten Länge von 8km.



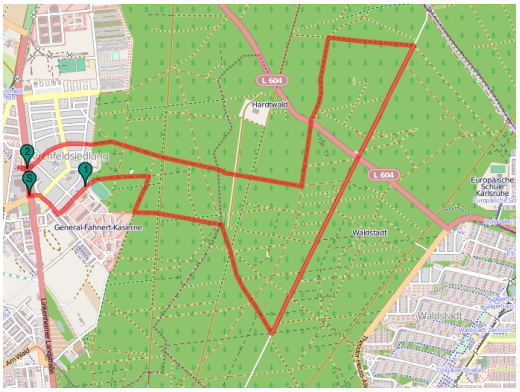
(a) TWB l=11687m b=23% Abbiegungen=12



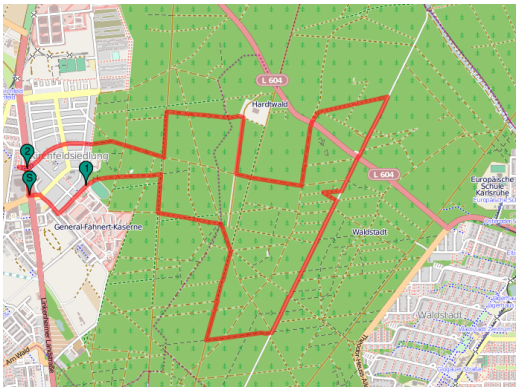
(b) TW4 l=11822m b=23% Abbiegungen=15



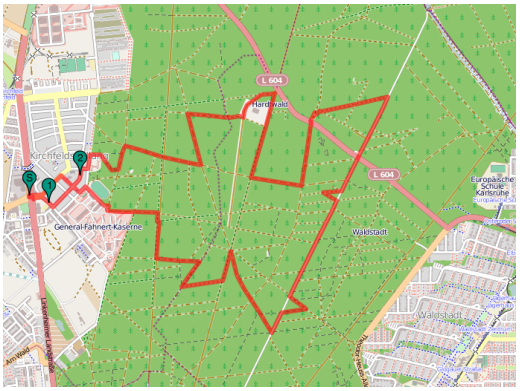
(c) TW3 l=12753m b=22% Abbiegungen=11



(d) FacesGreedy, beste Kreuzungen l=10811m b=27% Abbiegungen=14



(e) FacesGreedy, äquidistant l=11129m b=30% Abbiegungen=23



(f) FacesGreedy l=12254m b=29% Abbiegungen=40

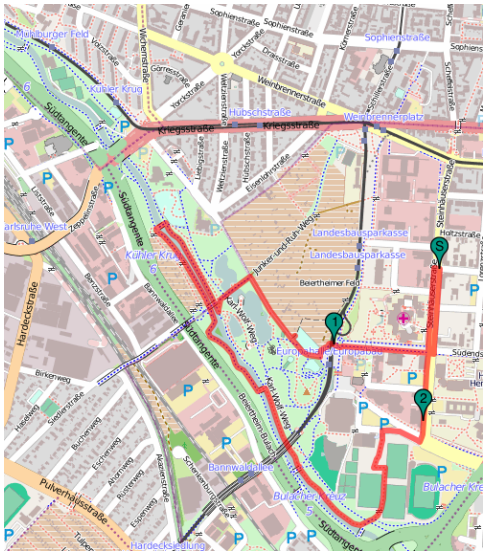
Abbildung 5.11: Ergebnisse der Algorithmen, bei einer gesuchten Länge von 12km.

(a) TWB l=4755m b=30% Abbiegungen=16

(b) TW4 l=4769m b=48% Abbiegungen=22

(c) TW3 l=4838m b=30% Abbiegungen=18

(d) FacesGreedy, beste Kreuzungen l=3987m b=37% Abbiegungen=21



(e) FacesGreedy, äquidistant l=3982m b=40% Abbiegungen=19

(f) FacesGreedy l=4620m b=41% Abbiegungen=44

Abbildung 5.12: Ergebnisse der Algorithmen, bei einer gesuchten Länge von 5km.

6. Zusammenfassung

Ziel dieser Arbeit war es, formal zu definieren was eine schöne Joggingroute ist, und anschließend Algorithmen zu entwerfen, die solche Joggingrouten berechnen.

Im Laufe dieser Arbeit haben sich Badness, Sharing und die Anzahl der Abbiegungen als gute Maße für die Qualität von Joggingrouten herausgestellt. Darüber hinaus haben wir verschiedene Algorithmen gefunden, die bezüglich diesen Maßen gute Joggingrouten finden können. Die tatsächliche Schönheit einer Route hängt aber immer auch von den Vorlieben des Betrachters ab.

Die einzelnen Algorithmen liefern zum Teil für jeweils verschiedene Kriterien besonders gute Ergebnisse. Legt man besonderen Wert auf ein niedriges Sharing, so sollte man den FacesGreedy-Algorithmus verwenden. Sind hingegen eher niedrige Badness und eine niedrige Anzahl von Abbiegungen von Interesse, so liefern die Teilwege-Algorithmen bessere Ergebnisse. Darüber hinaus können die Ergebnisse der einzelnen Algorithmen auch noch über die verschiedenen vorgestellten Parameter, zu Gunsten einzelner Kriterien, verbessert werden.

Die vorgestellten Algorithmen können mit einer großen Zuverlässigkeit schöne Routen finden. Dennoch lassen unsere Verfahren keine Aussage über Qualitätsgarantien zu. Es wäre hier interessant weitere Nachforschungen bezüglich der Approximierbarkeit des Kantendisjunkten Jogger-Problems zu betreiben. Würde ein Approximationsalgorithmus gefunden, so könnten aus ihm Algorithmen erwachsen, die tatsächlich Garantien für die Güte der berechneten Lösung geben.

Auch die hier vorgestellten Algorithmen können durchaus noch verbessert werden. Als grundlegend gute Vorgehensweise hat es sich erwiesen, zunächst Kandidaten oder Wegpunkte zu bestimmen und diese anschließend über kürzeste Wege miteinander zu verbinden. Insofern wäre es nun denkbar, weitere Verfahren zu suchen, die in kurzer Zeit gute Kandidaten finden.

Darüber hinaus wären auch erweiterte Eingabemöglichkeiten vorstellbar. Es könnte sinnvoll sein, mehr Knoten als nur den Startknoten anzugeben, durch die die Route verlaufen soll. Eine andere Idee wäre das der Benutzer zusätzlich ein Gebiet, vielleicht in Form eines Polygons, angibt, durch das er bevorzugt laufen möchte. Oder aber, dass er eine Richtung angibt, in die er vom Startpunkt aus laufen möchte.

Literaturverzeichnis

[Bal95] Ivan J. Balaban: *An Optimal Algorithm for Finding Segments Intersections.* In: *Proceedings of the eleventh annual symposium on Computational geometry*, SCG '95, Seiten 211–219. ACM, 1995, ISBN 0-89791-724-3.

[BKK+07] Kevin Buchin, Christian Knauer, Klaus Kriegel, André Schulz und Raimund Seidel: *On the Number of Cycles in Planar Graphs.* In: *Proc. 13th International Computing and Combinatorics Conference (COCOON)*, Seiten 97–107, 2007.

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor und Renato Fonseca F. Werneck: *Customizable Route Planning.* In: *Proceedings of the 10th international conference on Experimental algorithms*, Band 6630 der Reihe *Lecture Notes in Computer Science*, Seiten 376–387. Springer, 2011, ISBN 978-3-642-20661-0.

[Dij59] Edsger W. Dijkstra: *A Note on Two Problems in Connexion with Graphs.* Numerische Mathematik, 1:269–271, 1959.

[For09] Lance Fortnow: *The status of the P versus NP problem.* Communications of the ACM, 52(9):78–86, 2009, ISSN 0001-0782.

[GJ78] M. R. Garey und D. S. Johnson: *"Strong" NP-Completeness Results: Motivation, Examples, and Implications.* Journal of the ACM, 25(3):499–508, 1978, ISSN 0004-5411.

[GJ90] Michael R. Garey und David S. Johnson: *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990, ISBN 0716710455.

[Gra72] Ronald L. Graham: *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set.* Information Processing Letters, 1(4):132–133, 1972.

[GV11] Robert Geisberger und Christian Vetter: *Efficient routing in road networks with turn costs.* In: *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, Seiten 100–111. Springer-Verlag, 2011, ISBN 978-3-642-20661-0.

[Kar72] Richard M. Karp: *Reducibility among Combinatorial Problems.* In: Raymond E. Miller und James W. Thatcher (Herausgeber): *Complexity of Computer Computations*, Seiten 85–103. Plenum Press, 1972.

[LW06] Hongbo Liu und Jiaxin Wang: *A new way to enumerate cycles in graph.* In: *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*, AICT-ICIW '06, Seiten 57–60. IEEE Computer Society, 2006, ISBN 0-7695-2522-9.

[YZ95] Raphael Yuster und Uri Zwick: *Color-coding.* Journal of the ACM, 42(4):844–856, 1995, ISSN 0004-5411.