Karlsruhe Institute of Technology

Approximative und heuristische Algorithmen für trajektorienbasierte Kartenbeschriftung

Bachelorarbeit
von

Stefan Orf

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. Benjamin Niedermann

Bearbeitungszeit: 3. Juli 2013 – 4. Oktober 2013

Versicherung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst, und weder ganz oder in Teilen als Prüfungsleistung vorgelegt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die benutzten Werken im Wortlaut oder dem Sinn nach entnommen sind, habe ich durch Quellenangaben kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen sowie für Quellen aus dem Internet.

Karlsruhe, 4. Oktober 2013

Zusammenfassung

Navigationsgeräte stehen heute einer breiten Masse zur Verfügung. Mittlerweile besitzt nahezu jedes Smartphone die nötige Hardware zur Positionsbestimmung. Ein Aspekt eines Navigationsgeräts ist es, eine Route zu visualisieren, die den Nutzer von einem Start- zu einem Zielort führt. Dem Nutzer wird dabei ein Ausschnitt der Karte mit der hervorgehobenen Route gezeigt. Zusätzlich werden meist nahe gelegene Orten, die von Interesse sein könnten (z.B. Einkaufsmöglichkeiten) durch Beschriftungen markiert.

In dieser Arbeit betrachten wir algorithmische Fragestellungen bezüglich des Beschriftens solcher Kartenausschnitte. Wir nehmen an, dass die Karte aus der Vogelperspektive dargestellt und die Route durch eine glatte Trajektorie modelliert ist. Die Position und Ausrichtung des für den Nutzer sichtbaren Ausschnitts folgt dabei der Trajektorie. Die Beschriftungen werden dem Nutzer bezüglich des angezeigten Kartenausschnitts horizontal angezeigt, was jedoch dazu führt, dass sich Beschriftungen im Laufe der dynamischen Visualisierung überschneiden können. Unter Berücksichtigung dieses Aspekts gilt es nun, zu bestimmten Zeitpunkten ausgewählte Beschriftungen anzuzeigen und andere auszublenden. Ziel ist es die Anzahl der angezeigten Beschriftungen über die Zeit zu maximieren. Um die Übersichtlichkeit für den Nutzer zu wahren, fordern wir zusätzlich eine Beschränkung der Anzahl der gleichzeitig sichtbaren Beschriftungen. Dieses Problem nennen wir k-RestrictedMaxTotal.

In dieser Arbeit werden wir verschiedene Algorithmen vorstellen, welche das Problem k-RestrictedMaxTotal approximativ und heuristisch lösen. Wir präsentieren anfangs einen Algorithmus, der k-RestrictedMaxTotal löst und $1/27$-approximativ ist. Das Problem k-RestrictedMaxTotal kann auch in zwei Teilprobleme zerlegt werden. Wir werden erst einen Algorithmus vorstellen, der die Überschneidungen approximativ löst. Für spezielle Instanzen präsentieren wir dann ein lineares Programm, welches dieses Teilproblem optimal löst. Neben einem approximativen und einem heuristischen Algorithmus zur Beschränkung der gleichzeitig sichtbaren Intervalle zeigen wir, wie dieses Teilproblem mithilfe eines Flussnetzwerkes in Linearzeit optimal gelöst werden kann. Wir geben dann einen Approximationsfaktor für einen Algorithmus, bei dem eine Ausführung der approximativen Teilalgorithmen nacheinander erfolgt, an. Abschließend werden wir die Güte und die Laufzeit der Algorithmen experimentell, anhand von Straßendaten aus Karlsruhe untersuchen. Die Ergebnisse zeigen, dass die von den Algorithmen berechneten Lösungen einen annähernd optimalen Wert besitzen.

Inhaltsverzeichnis

1. Einleitung **1**

2. Grundlagen **5**

3. Kombinierter Algorithmus **9**
3.1. Algorithmus . . . 9
3.2. Approximationsfaktor . . . 10
3.3. Laufzeit . . . 15
3.4. Varianten mit unterschiedlicher Laufzeit . . . 15

4. Algorithmen für die Teilprobleme **21**
4.1. Auflösung der Konflikte . . . 21
4.1.1. Approximativer Greedy-Algorithmus zur Konfliktauflösung . . . 21
4.1.2. Sonderfall: Zwei Konflikte pro Intervall . . . 22
4.2. Begrenzung der Anzahl gleichzeitig sichtbarer Beschriftungen . . . 25
4.2.1. Flussalgorithmus für die k-Eigenschaft . . . 25
4.2.2. Approximativer Greedy-Algorithmus für die k-Eigenschaft . . . 28
4.2.3. Heuristischer Greedy-Algorithmus für die k-Eigenschaft . . . 29
4.3. Approximationsfaktor zweier Teilalgorithmen . . . 31

5. Experimentelle Evaluation **33**

6. Fazit **39**

Literaturverzeichnis **41**

Anhang **43**
A. Ergänzende Diagramme zur experimentellen Evaluation . . . 43

1. Einleitung

Heutzutage erfordern viele Anwendungen eine Visualisierung geographischer Informationen. So werden in einem wichtigen Anwendungsfall, nämlich der zivilen Navigation, einem Nutzer Straßen und gegebenenfalls Gebäude auf einem Kartenausschnitt angezeigt, der sich entsprechend der Position des Anwenders verändert. Ein bedeutender Aspekt ist die Wegfindung. In der Navigation ist ein Weg von einem Punkt der Karte zu einem anderen durch eine, meist kürzeste, Route festgelegt. Diese Route dient dem Nutzer, um sich im Straßennetz von einem Startpunkt zu einem Ziel zu bewegen. Solche Routen verlaufen für gewöhnlich, bedingt durch Abzweigungen und Kurven, nicht geradlinig. Dabei ist es für den Nutzer nur wichtig, welche Abzweigungen er nehmen muss, die sich nicht innerhalb seines Aktionsraums befinden. So wird ihm nur ein Ausschnitt der Karte gezeigt, der sich an seiner aktuellen Position befindet. Die Position des Ausschnitts wird mit der Bewegung des Nutzers, entlang einer gegebenen Bahnkurve, einer sogenannten Trajektorie, verändert. Um eine leichte Navigation zu gewährleisten, wird der Ausschnitt nicht, wie sonst bei Karten üblich, in Richtung Norden, sondern bezüglich der Fahrt- bzw. Laufrichtung ausgerichtet. Dies wird dadurch erreicht, dass der Ausschnitt um den gleichen Winkel gedreht ist, wie eine Tangente, die an der aktuellen Position der Trajektorie angelegt ist. In Abbildung 1.1 ist dies anhand einer Karte und dem Ausschnitt, den der Nutzer sieht, veranschaulicht. Damit der Ausschnitt besonders an Abzweigungen einen ruhigen Übergang besitzt, nehmen wir an, dass die Trajektorie glatt verläuft. Ein weiterer Aspekt von Navigationslösungen ist die Kartenbeschriftung. Dem Nutzer sollen neben dem Straßennetz und dem zu folgenden Weg auch bestimmte Orte angezeigt werden. Dabei kann es sich beispielsweise um Einkaufsmöglichkeiten, Unternehmen oder öffentliche Einrichtungen handeln. Meistens wird ein Ort durch seinen Namen auf der Karte kenntlich gemacht. Die Beschriftung erfolgt dann durch den Namen des Ortes an dessen Position auf der Karte. Wir gehen hier von der vereinfachenden Annahme aus, dass die Beschriftungen durch achsen-parallele Rechtecke beschrieben werden, die den eigentlich darzustellenden Text umranden. Die linke untere Ecke eines Rechteckes ist sein Ankerpunkt und befindet sich an der entsprechenden Position auf der Karte. Die Beschriftungen sollen horizontal bezüglich des Kartenausschnitts angezeigt werden. Da der Ausschnitt jedoch bezüglich der Position auf der Trajektorie ausgerichtet ist, ändern die Beschriftungselemente ihre Ausrichtung anhand einer fest vorgegebenen Richtung. Aufgrund dieser dynamischen Ausrichtung, die von der aktuellen Position auf der Trajektorie und der eventuellen Fülle von Beschriftungen abhängt, können sich Überschneidungen zwischen Beschriftungen ergeben. Allerdings sind solche Überschneidungen nicht wünschenswert, da sie dazu führen, dass Beschriftungen ganz oder teilweise nicht lesbar sind. Um dies zu verhindern, werden Beschriftungen zu

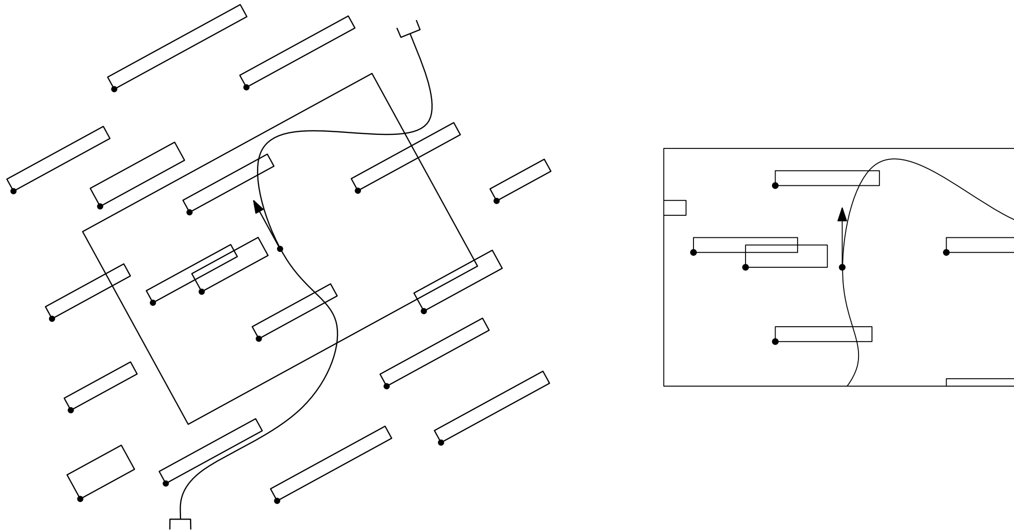


Abbildung 1.1: Eine beschriftete Karte mit einem Ausschnitt, dessen Position und Ausrichtung einer Trajektorie folgt. Ein Nutzer sieht nur den Ausschnitt der Karte, wie er rechts abgebildet ist

bestimmten Zeitpunkten ausgeblendet. Ein weiterer Aspekt ist die Anzahl der für den Nutzer sichtbaren Beschriftungen. Im Straßenverkehr birgt ein Navigationsgerät immer die Gefahr den Nutzer abzulenken. Besonders wenn zusätzlich Beschriftungen gelesen werden können, ist die Betrachtungszeit höher. Auch leidet die Sichtbarkeit des Straßennetzes und des Weges unter zu vielen Beschriftungen. Eine Begrenzung der maximalen Anzahl der gleichzeitg sichtbaren Beschriftungen ist also wünschenswert. In dieser Arbeit werden wir sowohl den Aspekt der Begrenzung der Zahl gleichzeitig sichtbarer Beschriftungen, als auch deren Überschneidungen behandeln. Dazu müssen die Zeitpunkte bestimmt werden, zu denen eine Beschriftung angezeigt werden darf. Bezüglich der Sichtbarkeit gibt es weitere Qualitätskriterien, die für den Nutzer wichtig sein können. Dazu zählt insbesondere das Beschriftungen nicht wiederholt ein- und ausgeblendet werden. Andernfalls kann es passieren, dass eine Beschriftung beispielsweise flackert (siehe [BDY06]). Wir betrachten daher nur den Fall, dass eine Beschriftung für den gesamten Zeitraum, in dem es sich innerhalb des Ausschnitts befindet, entweder sichtbar oder unsichtbar ist. Sind die Zeitpunkte der Sichtbarkeit für jede Beschriftung festgelegt, so bestimmt das Integral der Anzahl der Beschriftungen über die Zeit wie gut diese Lösung ist. Unter allen möglichen Lösungen finden sich dann eine oder mehrere optimale Lösungen. Es existiert ein Algorithmus, der so eine optimale Lösung berechnet (siehe [GNN13]). Beachtet man jedoch die begrenzte Hardware von Navigationsgeräten, ist die Laufzeit dieses Algorithmus für die Praxis eher ungeeignet. Ziel ist es also, einen Algorithmus zu entwickeln, der möglichst gute Lösungen in kurzer Zeit berechnet.

Forschungsstand

Betrachtet man statische Karten, so gibt es dort auch die Herausforderung Beschriftungen anzuzeigen. Für die Beschriftungen werden neben achsenparallelen Rechtecken auch andere Formen betrachtet. Im Artikel [SW01] wird hierzu beispielsweise ein approximativer Algorithmus vorgestellt, der kreisförmige Beschriftungen nutzt. Zudem muss die Größe der Beschriftungen nicht fest sein. Das Problem besteht dann darin, die Größe der Beschriftungen zu maximieren, so dass keine Überschneidungen auftreten (siehe dazu auch [SW01]). Beschränkt man sich bei den Beschriftungen auf achsenparallele Rechtecke fester Größe, sollen dann möglichst viele Beschriftungen überlappungsfrei angezeigt und andere

ausgeblendet werden. Die Position der Beschriftungen muss dabei nicht fest sein. Dazu wird in [vKSW98] ein „slider model“ vorgestellt. Bei diesem Modell muss eine Beschriftung den jeweiligen Ankerpunkt mit einer Kante berühren. Es stehen dann unendlich viele Positionen für das Rechteck zur Verfügung. Werden hingegen nur endlich viele Positionen zugelassen, spricht man vom „fixed model“. Hier wird zwischen den möglichen Positionen unterschieden. So gibt es ein Modell, bei dem die linke untere Ecke des Rechtecks auf dem Ankerpunkt liegen muss. Bei anderen Modellen sind zwei, vier oder acht Positionen möglich, so dass entsprechende Punkte des Rechtecks auf dem Ankerpunkt liegen. Wir betrachten hier jedoch nur das „fixed-position“ Modell, bei dem nur eine Position zugelassen ist. Das Maximierungsproblem für dieses Modell ist $\mathcal{NP}$-vollständig (siehe [FPT81]). Für das Problem werden in [AvKS98] approximative Algorithmen vorgestellt. Einer der dort präsentierten Algorithmen betrachtet Rechtecke beliebiger Größe. Der Algorithmus besitzt eine Laufzeit in $O(n \log n)$ und einen Approximationsfaktor, der in $O(\log n)$ liegt. In [CC09] wird hingegen ein Algorithmus beschrieben, dessen Approximationsfaktor in $O(\log \log n)$ liegt. Die beweisbaren Approximationsfaktoren für Beschriftungen beliebiger Größe liegen deutlich unter einer optimalen Lösung. Für Beschriftungen mit einheitlicher Höhe gibt es bessere Gütegarantien. Es existieren auch Veröffentlichungen, in denen heuristische Algorithmen vorgestellt werden, die in der Praxis annähernd optimale Ergebnisse erzielen (siehe [WWKS01]). In [WW97] wird zudem eine Heuristik beschrieben, die zusätzlich einen Approximationsfaktor aufweist und in der Praxis ebenfalls annähernd optimale Lösungen berechnet.

Eine Erweiterung des Problems auf dynamische Karten erscheint unter heutigen Gesichtspunkten sinnvoll. Für den Fall, dass die Karte vergrößerbar und der Ausschnitt verschiebbar ist, werden approximative Algorithmen in [BNPW10] beschrieben. Ein weiterer Aspekt ist das kontinuierliche Drehen einer Karte, bei der die Beschriftungen ihre waagrechte Ausrichtung beibehalten und sich nicht überschneiden dürfen. In [GNR11] wird ein Modell für dieses dynamische Drehen einer Karte entwickelt und ein Maximierungsproblem definiert. Dieses Maximierungsproblem ist $\mathcal{NP}$-vollständig. In der Arbeit werden auch approximative Algorithmen mit polynomieller Laufzeit für das Problem vorgestellt. Neben diesem Maximierungsproblem gibt es auch das Problem der Größenmaximierung der Beschriftung (siehe [YI13]). Wir werden in dieser Arbeit jedoch nicht näher darauf eingehen. Ein formales Modell für dynamische Karten und ihre Beschriftung wurde erstmals in [BDY06] eingeführt. Ein vollständiges Modell für die dynamische Kartenbeschriftung basierend auf Trajektorien, das dieser Arbeit zu Grunde liegt, wird in [GNN13] vorgestellt. Dort wird auch gezeigt, dass das zu Beginn beschriebene Maximierungsproblem ohne Beschränkung der Anzahl gleichzeitig sichtbarer Beschriftungen $\mathcal{NP}$-vollständig und $\mathcal{W}[1]$-schwer ist. Für den Fall, dass die Anzahl der sichtbaren Beschriftungen auf k begrenzt ist, wird ein optimaler Algorithmus vorgestellt, dessen Laufzeit in $O(n^{k^2+k-1})$ liegt. Dieses Problem werden wir in dieser Arbeit aufgreifen und dafür geeignete approximative Algorithmen mit besserer Laufzeit vorstellen.

Übersicht

Wir beginnen mit der Vorstellung der Grundlagen in Kapitel 2. Dort gehen wir auf das verwendete Modell für die dynamische Kartenbeschriftung und die Notation ein, welche in der gesamten Arbeit Anwendung findet. Ebenso nehmen wir dort eine formale Problemdefinition vor. In Kapitel 3 präsentieren wir einen Algorithmus, der sowohl die Überschneidungen auflöst, als auch die Anzahl der gleichzeitig sichtbaren Beschriftungen begrenzt. Wir zeigen, dass der Algorithmus einen Approximationsfaktor von $^1/_{27}$ besitzt. Zudem stellen wir verschiedene Varianten des Algorithmus vor, welche unterschiedliche Laufzeiten besitzen. Dabei besitzen die schnellsten Varianten eine Laufzeit in $O(n^2)$. Im Anschluss gehen wir auf Algorithmen für die Teilprobleme ein (siehe Kapitel 4). Dazu stellen wir zuerst einen approximativen Algorithmus vor, der nur die Überschneidungen auflöst. Die Laufzeit dieses Algorithmus liegt in $O(n \log n)$. Außerdem können die Überschnei-

dungen für spezielle Instanzen mit diesem Algorithmus optimal aufgelöst werden. Danach stellen wir ein lineares Programm vor, dass ebenfalls unter bestimmten Voraussetzungen eine optimale Lösung für das Teilproblem berechnet. Zur Begrenzung der Anzahl der gleichzeitig sichtbaren Beschriftungen zeigen wir die Transformation des Teilproblems auf Flussnetzwerke (siehe [CL95]). Durch effiziente Berechnung der kürzesten Wege können wir das Teilproblem in linearer Laufzeit optimal lösen. Es folgt, neben einem weiteren approximativen Algorithmus, ein heuristischer Algorithmus, der beliebig schlecht werden kann. Abschluss dieses Kapitels bildet eine Betrachtung der Ausführung der approximativen Algorithmen für die Teilprobleme hintereinander. Wir zeigen, dass der resultierende Algorithmus einen Approximationsfaktor von $1/72$ besitzt. Danach, in Kapitel 5, werden wir die vorgestellten Algorithmen anhand von realen Kartendaten experimentell evaluieren. Dazu haben wir die Ergebnisse der Algorithmen mit einer optimalen Lösung verglichen, die mithilfe eines ganzzahligen Programms berechnet wurde. Die Ergebnisse zeigen einen deutlichen Geschwindigkeitsvorteil der präsentierten Algorithmen. Trotz der teils niedrigen Approximationsfaktoren zeigte sich eine Güte, die nahe an der optimalen Lösung liegt. Der Ausblick (siehe Kapitel 6) rundet dann die Arbeit ab.

2. Grundlagen

In diesem Kapitel führen wir grundlegende Begriffe und Notationen ein, die in der gesamten Arbeit Anwendung finden. Die Problemstellung geht aus der dynamischen Kartenbeschriftung hervor. Bei dem hier betrachteten Anwendungsfall sollen in einem dynamischen Ausschnitt Beschriftungen einer statischen Karte dargestellt werden. Die Position und der Winkel der Beschriftungen ist dabei durch eine Bahnkurve festgelegt. Die Aufgabe besteht darin, Beschriftungen ein- bzw. auszublenden, um bestimmte Kriterien zu erfüllen. Solche Kriterien können zum Beispiel Lesbarkeit oder Übersichtlichkeit sein. Diese Bachelorarbeit betrachtet Fragestellungen, die auf dem Artikel [GNN13] aufbauen. Wir werden deshalb die dort eingeführte Notation teilweise übernehmen und gegebenenfalls verfeinern.

Wir betrachten zunächst eine statische Karte M. Bei solch einer Karte kann es sich beispielsweise um eine Straßenkarte handeln. Die Karte sei nach Norden ausgerichtet und besitzt einen festen Maßstab. Es befinden sich Punkte $P = \{p_1, \dots, p_N\}$, zusammen mit Beschriftungen $L = \{l_1, \dots, l_N\}$ auf der Karte. Jede Beschriftung l_i wird durch ein achsenparalleles Rechteck repräsentiert, dessen linke untere Ecke sich am Ankerpunkt p_i befindet. Dabei besitzt das Rechteck einer Beschriftungen individuelle Höhe und Breite. Eine Beschriftung besteht meist aus einem Text, bspw. einem Ortsnamen. In dieser Arbeit nehmen wir an, dass dieses Rechteck dem kleinsten achsenparallelen Rechteck entspricht, welches den Text enthält. Wir bezeichnen im Folgenden eine Beschriftung auch mit dem englischen Begriff *Label*.

Im betrachteten Anwendungsfall wird dem Nutzer ausschließlich ein Ausschnitt der Karte gezeigt. Dieser Ausschnitt richtet sich nach der Position des Nutzers auf der Karte und einem vordefinierten Weg. Der für den Anwender sichtbare Ausschnitt R wird auch *Viewport* genannt und wird ebenfalls durch ein Rechteck dargestellt. Er folgt einer *Trajektorie*, die durch eine kontinuierlich differenzierbare Funktion $T\colon [0,1] \to \mathbb{R}^2$ definiert ist. In [GNN13] wird R durch die Funktion $V\colon [0,1] \to \mathbb{R}^2 \times [0, 2\pi]$ beschrieben. Dabei bedeutet $V(t) = (c, \alpha)$, dass der Mittelpunkt von R sich an Position c befindet und um den Winkel α gedreht ist. Im Bezugssystem des Nutzers ist der Viewport in Ruhe und es sieht für ihn so aus, als ob sich die Karte dreht. Da sich R entlang T bewegt, wird $V(t) = (T(t), \alpha(t))$ definiert. Dabei ist $\alpha(t)$ die Richtung von T zum Zeitpunkt t. Dies bedeutet, dass der Viewport immer in Richtung der Trajektorie gedreht ist. Zusätzlich sollen die Labels zu jedem Zeitpunkt waagrecht zu R angezeigt werden, da so die Lesbarkeit gewährleistet ist. Somit werden die Labels zum Zeitpunkt t um den gleichen Winkel $\alpha(t)$ wie der Viewport gedreht. Eine Veranschaulichung findet sich in Abbildung Abbildung 1.1 im Kapitel 1. Die Labels werden dann durch $\ell(t)$ charakterisiert. Wir sehen $\ell(t)$ als Menge von Punkten $(x, y) \in \mathbb{R}^2$ in der Ebene an, welche durch das Rechteck des entsprechenden, gedrehten

Labels zum Zeitpunkt t definiert wird. Gleiches gilt für $V(t)$ bezüglich des Rechtecks des Viewports.

Die Algorithmen, die wir in dieser Arbeit vorstellen wollen, arbeiten auf *Zeitintervallen*, welche angeben, wann sich ein Label im Viewport befindet. Dabei ist ein Zeitintervall $[a, b]$ durch einen Startzeitpunkt a und einen Endzeitpunkt b charakterisiert. In dieser Arbeit werden wir die Zeitintervalle zudem auch *Zeitbereiche* nennen.

Von Interesse für das Ein- und Ausblenden der Labels sind nur die Labels, welche sich innerhalb des Viewports befinden. Wir bezeichnen ein Label ℓ in diesem Fall auch als *präsent* zum Zeitpunkt t, wenn $V(t) \cap \ell t \neq \emptyset$. Es besteht dabei ein Unterschied zur *Sichtbarkeit* eines Labels. Nicht alle Labels, die sich innerhalb des Viewports befinden, werden dem Nutzer später angezeigt. Es können also Labels existieren, die zwar präsent, aber nicht sichtbar sind. Bei $V(t)$ und $\ell(t)$ handelt es sich um geschlossene Mengen. Somit können wir geschlossene Zeitintervalle definieren, welche angeben, wann ein Label präsent ist. In [GNN13] ist die Menge, welche alle Intervalle enthält, in denen ein Label präsent ist, wie folgt definiert.

Definition 2.1. *Die Menge $\Psi_\ell = \{[a, b] \mid [a, b] \subseteq [0, 1]$ und $[a, b]$ ist maximal, so dass ℓ zu jedem Zeitpunkt $t \in [a, b]$ präsent ist$\}$ beinhaltet alle Zeitintervalle, in denen das Label ℓ präsent ist. Die Menge $\Psi = \{([a, b], \ell) \mid [a, b] \in \Psi_\ell$ und $\ell \in L\}$ fasst diese Intervalle aller Labels zusammen.*

Das Element $([a, b], \ell) \in \Psi$ nennen wir *Präsenzintervall des Labels ℓ* und kürzen es mit $[a, b]_\ell$ ab. Die Algorithmen werden dann auf diesen Präsenzintervallen arbeiten. Die Operationen, wie bei den üblichen Zeitintervallen, wie Schnitt, Vereinigung,..., finden auch bei Präsenzintervallen Anwendung. Insbesondere die Länge wird in dieser Arbeit auch auf sie angewandt. Wir werden zudem ein Präsenzintervall auch als *Intervall* bezeichnen, wenn klar ist was gemeint ist. Ein Label kann zudem mehrere Präsenzintervalle besitzen. Dies ist beispielsweise dann der Fall, wenn der Viewport mehrmals über den selben Kartenausschnitt wandert.

Die beiden Rechtecke zweier Labels ℓ und ℓ' überschneiden sich zum Zeitpunkt t, wenn $\ell(t) \cap \ell'(t) \neq \emptyset$ gilt. In diesem Fall befinden sich ℓ und ℓ' zum Zeitpunkt t in *Konflikt* miteinander. Aus Nutzersicht sind solche Konflikte nicht wünschenswert, wenn sie innerhalb des sichtbaren Bereichs liegen. Die Lesbarkeit der Labels ist dann meist eingeschränkt. Ein Konflikt zwischen zwei Labels ℓ und ℓ' zum Zeitpunkt t ist dann präsent, wenn $\ell(t) \cap \ell'(t) \cap V(t) \neq \emptyset$ ist. Ebenso wie die Präsenzintervalle eines Labels, können wir die Zeitintervalle, in denen zwei Labels in Konflikt miteinander stehen, in einer Menge zusammenfassen.

Definition 2.2. *Die Menge $C_{\ell,\ell'} = \{[a, b] \mid [a, b] \subseteq [0, 1]$ und $[a, b]$ ist maximal, so dass ℓ zu jedem Zeitpunkt t mit ℓ' in Konflikt steht$\}$ beinhaltet alle Zeitintervalle, während denen die zwei Labels ℓ und ℓ' in Konflikt miteinander stehen. Alle solche Konflikte sind in der Menge $C = \{([a, b], \ell, \ell') \mid [a, b] \in C_{\ell,\ell'}$ und $\ell, \ell' \in L\}$ zusammengefasst.*

Es bietet sich hier ebenfalls eine Abkürzung für $([a, b], \ell, \ell') \in C$ der Form $[a, b]_{\ell,\ell'}$ an. Das Intervall $[a, b]_{\ell,\ell'}$ heißt dann *Konfliktintervall der Labels ℓ und ℓ'*. Die Eingabe für die Algorithmen, besteht meist aus den Mengen Ψ und C. Hierzu definieren wir einen Konflikt auch zwischen zwei Präsenzintervallen.

Definition 2.3. *Gegeben eine Menge Ψ von Präsenzintervallen und die Menge C der Konflikte, dann stehen zwei Präsenzintervalle $[a, b]_\ell, [c, d]_{\ell'} \in \Psi$ miteinander in Konflikt, wenn ein Konflikt $[x, y]_{\ell,\ell'} \in C$ existiert, so dass gilt $[x, y] \cap [a, b] \cap [c, d] \neq \emptyset$.*

Weiterhin ist es für die Algorithmen wichtig, die Präsenzintervalle zu kennen, die mit einem bestimmten Präsenzintervall in Konflikt stehen. Dazu werden wir die folgende Menge definieren. Die Algorithmen, welche die Konflikte auflösen, nutzen dafür dann diese Menge.

Definition 2.4. *Die Menge* $C^{\ell}_{[a,b]} = \{[c,d]_{\ell'} \mid [c,d]_{\ell'} \in \Psi$ *und* $[c,d]_{\ell'}$ *steht in Konflikt mit* $[a,b]_{\ell}\}$ *ist die Menge aller Präsenzintervalle, welche mit* $[a,b]_{\ell}$ *in Konflikt stehen.*

Die Elemente dieser Menge bezeichnen wir als *Konfliktpartner* oder *Konfliktintervalle von* $[a,b]_{\ell}$.

Wie vorhin beschrieben, besteht ein Unterschied zwischen der Präsenz und der Sichtbarkeit eines Labels. Ein Präsenzintervall $[a,b]_{\ell}$ ist zum Zeitpunkt t präsent, wenn $\ell(t) \cap V(t) \neq \emptyset$ ist. Es werden später aber nicht unbedingt alle Labels dem Nutzer angezeigt. Der Algorithmus entscheidet welche präsente Labels dem Nutzer angezeigt werden und welche nicht. Sofern ein Label dem Nutzer angezeigt wird, gilt es als *aktiv.* Wann ein Label aktiv gesetzt werden darf, wird in [GNN13] durch Aktivitätsmodelle geregelt.

Aktivitätsmodelle

Die Karte soll nach bestimmten Regeln beschriftet werden. Ein Label darf also nur dann sichtbar sein, wenn bestimmte Einschränkungen erfüllt sind. Die Sichtbarkeit wird über die Aktivitätsmodelle geregelt. Dazu definieren [GNN13] drei Aktivitätsmodelle AM1, AM2 und AM3. Allen Modellen liegen die folgenden drei Konsistenzkriterien zu Grunde, die abgewandelt aus [BDY06] stammen.

1. Ein Label darf zum Zeitpunkt t nur dann aktiv sein, wenn es zu diesem Zeitpunkt präsent ist.

2. Es darf in einem Präsenzintervall nur maximal ein aktiver Zeitbereich existieren. Dies verhindert, dass Labels flackern.

3. Um sich überschneidende Labels zu verhindern, darf zu einem Zeitpunkt $t \in [a,b]_{\ell,\ell'}$ während eines Konfliktes nur maximal eines der beiden Labels ℓ und ℓ' aktiv sein.

Alle drei Aktivitätsmodelle unterscheiden sich durch verschiedene Vorschriften für die Start- und Endzeitpunkte der aktiven Intervalle, um die drei obigen Konsistenzkriterien zu beachten. Das Aktivitätsmodell **AM1** schreibt vor, dass ein Präsenzintervall $[a,b]_{\ell} \in \Psi$ entweder für den gesamten Zeitraum $[a,b]$ aktiv oder komplett inaktiv sein muss. Im zweiten Aktivitätsmodell **AM2** darf ein aktives Intervall eines Präsenzintervalls $[a,b]_{\ell}$ auch einen früheren Endzeitpunkt c als b besitzen. Ist dies der Fall, muss ein weiteres Label ℓ' existieren, so dass es einen Konflikt $[c,d]_{\ell,\ell'}$ gibt. Des Weiteren muss ℓ' zum Zeitpunkt c aktiv sein. Das Label ℓ' ist dann der *Zeuge* dafür, dass ℓ aktiv wird. In AM2 muss der Startzeitpunkt des aktiven Intervalls aber a entsprechen. Die Erweiterung auf das dritte Aktivitätsmodell **AM3** besteht darin, zusätzlich zum Endzeitpunkt, auch den Startzeitpunkt zu variieren. Der Startzeitpunkt c des aktiven Intervalls eines Präsenzintervalls $[a,b]_{\ell}$ darf von a abweichen, wenn ein Zeuge ℓ' und ein Konflikt $[a,c]_{\ell,\ell'}$ existiert. Das Label ℓ' muss wieder zum Zeitpunkt c aktiv sein. Für den Endzeitpunkt des aktiven Intervalls gilt die gleiche Bedingung wie in AM2. Das Erscheinen und Verschwinden von Labels muss also durch andere Labels begründet sein.

Charakterisierung einer Lösung

Eine Menge Ψ' von Präsenzintervallen gilt als *gültige Lösung*, um die Karte dynamisch zu beschriften, wenn die drei, oben genannten Konsistenzkriterien beachtet sind. Dies geschieht unter Beachtung eines der drei Aktivitätsmodelle. Um die Qualität einer Lösung zu definieren, übernehmen wir den, hier vereinfachten, gewichtsbasierten Ansatz aus [GNN13]. Demnach ist das *Gewicht* $w([a,b]_{\ell})$ eines Präsenzintervalls $[a,b]_{\ell}$ seine Länge $b-a$. Anhand des Werts einer Lösung können wir dann verschiedene Lösungen miteinander vergleichen. Mit dem Gewicht der einzelnen Präsenzintervalle können wir diesen Wert nun definieren.

Definition 2.5. *Der Wert einer Lösung* Ψ' *sei die Summe* $\sum_{[a,b]_{\ell} \in \Psi'} w([a,b]_{\ell})$ *der Gewichte der aktiven Präsenzintervalle.*

Wir können eine weitere Eigenschaft definieren, die zusätzlich zur Lesbarkeit erfüllt werden soll. Besonders in der Fahrzeugnavigation sind zu viele sichtbare Labels unvorteilhaft, da sie den Nutzer überfordern. Um Übersichtlichkeit zu gewährleisten, wollen wir also Lösungen betrachten, deren Anzahl an gleichzeitig aktiven Labels begrenzt ist. Dazu definieren wir die *k-Eigenschaft.*

Definition 2.6. *Die k-Eigenschaft einer gültigen Lösung ist erfüllt, wenn zu jedem Zeitpunkt die Anzahl der Intervalle höchstens k beträgt.*

Wir betrachten in dieser Arbeit hauptsächlich Algorithmen, welche neben den Regeln eines Aktivitätsmodells auch die k-Eigenschaft beachten. Da die Aktivitätsmodelle die Konflikte auflösen, werden wir dies in den Kapiteln auch so benennen.

Problemdefinitionen

In [GNN13] wird die Aufgabe, die Sichtbarkeit der Labels anhand obiger Kriterien zu bestimmen, in zwei Optimierungsproblemen definiert. Die Lösung eines der beiden Probleme nennen wir auch die *optimale Lösung.* Das Problem GeneralMaxTotal besteht darin, eine gültige Lösung für eine Instanz in einer der drei Aktivitätsmodelle zu finden, deren Wert unter allen gültigen Lösungen am größten ist. Das Problem ist, nach [GNN13], $\mathcal{NP}$-vollständig und $\mathcal{W}[1]$-schwer. Zur Lösung dieses Problems wird dort ein ganzzahliges lineares Programm vorgestellt. Fordern wir zusätzlich zur gültigen Lösung in einem Aktivitätsmodell, die k-Eigenschaft, sprechen wir vom Problem k-RestrictedMaxTotal. In [GNN13] wird auch hierfür eine Lösung, mittels eines dynamischen Programms, vorgestellt. Jedoch besitzen die Ansätze zur Lösung der beiden Probleme eine Laufzeit die in $O(n^{k^2+k-1})$. Deshalb wollen wir in dieser Arbeit nun verschiedene approximative und heuristische Verfahren zur Berechnung einer gültigen, nicht zwingend optimalen Lösung vorstellen.

3. Kombinierter Algorithmus

Wir werden in diesem Kapitel einen approximativen Algorithmus vorstellen, der eine gültige Lösung im Aktivitätsmodell AM1 berechnet, so dass die k-Eigenschaft beachtet wird. Wir wollen also das Problem k-RESTRICTEDMAXTOTAL approximativ lösen. Durch die Beschränkung auf AM1 wird ein Präsenzintervall, das an einem Konflikt beteiligt ist, entweder komplett aktiv oder nicht aktiv gesetzt. Das korrespondierende Label wird damit während des gesamten Präsenzintervalls angezeigt oder ist währenddessen nicht sichtbar. Im Folgenden werden wir zuerst das Verhalten des Algorithmus erläutern, anschließend folgt eine Betrachtung des Approximationsfaktors und der Laufzeit. Durch verschiedene Implementationen werden unterschiedliche Laufzeiten erreicht. Die Vorstellung dieser Varianten bildet dann den Abschluss dieses Kapitels.

3.1. Algorithmus

Der Algorithmus arbeitet auf der Menge der Präsenzintervalle. Diese Menge enthält die Intervalle, deren Labels im sichtbaren Bereich liegen, während sich der Viewport entlang der Trajektorie bewegt. Die Eingabemengen werden wir in diesem Kapitel mit Ψ für die Menge der Präsenzintervalle und C für die Menge der Konflikte bezeichnen. Als Lösung wird eine Teilmenge $\Psi' \subseteq \Psi$ erzeugt, die keine Konflikte aufweist und der k-Eigenschaft genügt.

Der Algorithmus folgt dem „Greedy-Prinzip" und betrachtet immer das größte Intervall, um eine gültige Lösung zu berechnen. Die Eingabe für den Algorithmus besteht aus der Eingabemenge Ψ und der Konfliktmenge C. Prinzipiell wird die Eingabemenge Ψ durch Betrachtung der einzelnen Elemente in die Lösungsmenge Ψ' überführt. Dazu wird ein Element $[a, b]_l \in \Psi$ der Intervallmenge entnommen und der Lösungsmenge Ψ', unter Beachtung der k-Eigenschaft, hinzugefügt. Zusätzlich werden die Konfliktintervalle von $[a, b]_l$ gelöscht. Dieser Schritt wird solange wiederholt bis alle Intervalle aus Ψ betrachtet bzw. gelöscht wurden.

Vorgehen

Zu Beginn werden die Intervalle der Eingabemenge Ψ absteigend nach ihrem Gewicht sortiert. Dies erlaubt es dem Algorithmus später, effizient auf das jeweils größte Intervall zugreifen zu können. Anschließend werden die Intervalle nacheinander in absteigender Reihenfolge in einer Schleife betrachtet. In jedem Schleifendurchlauf wird Ψ durch Entfernen des betrachteten Elements und gegebenenfalls seiner Konfliktintervalle verändert. Während der Schleifendurchläufe werden dann die Intervalle unter Beachtung der k-Eigenschaft der Lösungsmenge Ψ' hinzugefügt.

Wir betrachten nun einen einzelnen Schritt der Schleife. In diesem Schritt wird das erste und somit größte Element $[a,b]_l \in \Psi$ aus der Intervallmenge entnommen. Anschließend wird bestimmt, ob durch das Einfügen des Elements $[a,b]_l$ in Ψ' zu einem Zeitpunkt mehr als k Intervalle gleichzeitig aktiv sind. Im Algorithmus 3.1 geschieht diese Überprüfung mithilfe der Funktion PRÜFE-k-EIGENSCHAFT(). Als Parameter erhält die Funktion eine Menge Ψ' von Präsenzintervallen, das einzelne Intervall $[a,b]_l$ und eine natürliche Zahl $k \in \mathbb{N}$. Die Funktion prüft dann, ob $[a,b]_l$ eine Verletzung der k-Eigenschaft in Ψ' hervorruft. Ist dies der Fall, wird TRUE, andernfalls FALSE zurück gegeben.

Für den Fall, dass ein Einfügen von $[a,b]_l$ in Ψ' nicht zu einer Verletzung der k-Eigenschaft führt, wird $[a,b]_l$ in Ψ' eingefügt. Um eine konfliktfreie Menge zu erzeugen, dürfen nun aber alle Intervalle, welche mit $[a,b]_l$ in Konflikt stehen, nicht mehr in Ψ' eingefügt werden. Diese werden also aus Ψ entfernt.

Sowohl für den Fall, dass $[a,b]_l$ in Ψ' eingefügt wird, als auch für den gegensätzlichen Fall, muss $[a,b]_l$ aus Ψ entfernt werden. Denn in beiden Fällen darf $[a,b]_l$ nicht noch einmal betrachtet werden. Einerseits muss ein mehrmaliges Einfügen unterbunden werden und andererseits würde ein Widerspruch zur k-Eigenschaft weiterhin auftreten.

Die veränderte Menge Ψ wird dann wieder im nächsten Schritt benutzt. Die Iteration wird solange wiederholt bis $\Psi = \emptyset$ ist. Die Ausgabe des Algorithmus ist dann Ψ', wobei deren Intervalle alle als aktiv markiert werden.

Algorithmus 3.1 : GREEDY-k-RESTRICTEDMAXTOTAL(Ψ, C, K)

Eingabe : Menge Ψ von Intervallen, Menge C aller Konflikte, Parameter k
Ausgabe : Konfliktfreie Menge Ψ' von Intervallen, die AM1 genügt

```
1  begin
2      Ψ ← sort (Ψ) ; /* Sortiere Ψ in absteigender Reihenfolge */
3      Ψ' ← ∅;
4      while Ψ ≠ ∅ do
5          [a,b]_l ← größtes Element aus Ψ;
6          if !prüfeKEigenschaft(Ψ', [a,b]_l, k) then
7              Ψ' ← Ψ' ∪ {[a,b]_l};
8              Ψ ← Ψ \ C^l_[a,b];
9          Ψ ← Ψ \ {[a,b]_l};
10     return Ψ';
```

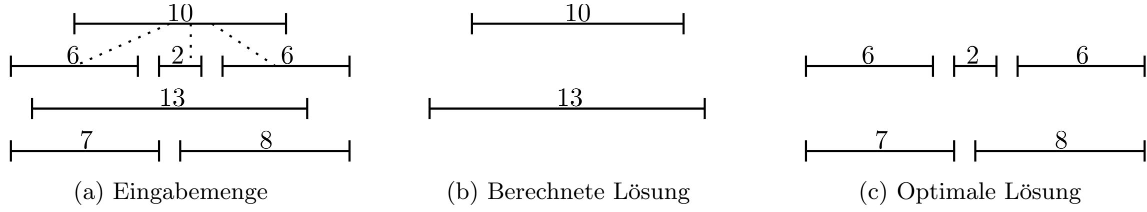
3.2. Approximationsfaktor

Der Algorithmus löst das Problem k-RESTRICTEDMAXTOTAL nicht optimal. Das in einem beliebigen Schritt betrachtete Intervall kann seinen Konfliktpartnern vorgezogen werden. Ebenso wird bei einem Widerspruch zur k-Eigenschaft (Funktion PRÜFE-k-EIGENSCHAFT()) das Intervall verworfen. Beides kann dazu führen, dass der Algorithmus Fehlentscheidungen trifft. In Abbildung 3.1 ist ein Beispiel zu sehen, für das der Algorithmus nicht die optimale Lösung findet. Für Einheitsquadrate als Labels zeigen wir am Ende dieses Abschnitts, dass der Algorithmus keine beliebig schlechte Lösung erzeugen kann, sondern eine beweisbare Approximationsgüte aufweist.

Vorbereitungen

Im Folgenden werden wir Lemmas und Definitionen beschreiben, die im Beweis des Approximationsfaktors Anwendung finden. Zur einfacheren Beschreibung bezeichnen wir die Menge Ψ im i-ten Schritt mit Ψ_i.

Wird ein Intervall vom Algorithmus für die Lösung ausgewählt, können eventuell andere Intervalle in einem späteren Schritt nicht mehr gewählt werden. Dies liegt in der Auflösung



(a) Eingabemenge (b) Berechnete Lösung (c) Optimale Lösung

Abbildung 3.1: Eine beispielhafte Eingabemenge von Intervallen und zwei verschiedene gültige Lösungen für $k = 2$. Die Beschriftungen der Intervalle entsprechen ihrer Länge und gestrichelte Linien repräsentieren Konflikte. Das Gesamtgewicht der Lösung (b) ist 23, wohingegen das der optimalen Lösung (c) 29 beträgt

der Konflikte und der Herstellung der k-Eigenschaft begründet. Wir können zeigen, dass die so beeinflussten Intervalle in einem besonderen Zeitbereich liegen. Zur Festlegung dieses Bereichs definieren wir nun allgemein den Einflussbereich eines Intervalls $[a, b]_l$, in dem die Intervalle einer Menge liegen, die sich mit $[a, b]_l$ zeitlich überschneiden.

Definition 3.1. *Gegeben sei ein Intervall $[a, b]_l$. Der **Einflussbereich von $[\mathbf{a}, \mathbf{b}]_\mathbf{l}$ auf Ψ** ist definiert als das kleinste Zeitintervall $[x, y]$, so dass für alle $[c, d]_{l'} \in \Psi$ mit $[c, d] \cap [a, b] \neq \emptyset$ gilt $[c, d]_{l'} \subseteq [x, y]$.*

Der Einflussbereich des Intervalls $[a, b]_l$ ist somit der zeitliche Bereich, in dem die Intervalle einer Menge liegen, die sich mit $[a, b]_l$ zeitlich überschneiden. Später werden wir jedoch noch genauer zwischen dem Einflussbereich von $[a, b]_l$ bezüglich seiner Konfliktepartner und bezüglich der k-Eigenschaft differenzieren. Das folgende Lemma schränkt die Größe des Einflussbereichs eines im i-ten Schritt gewählten Intervalls ein.

Lemma 3.2. *Sei $[a, b]_l$ das im i-ten Schritt betrachtete Intervall und Ψ_i die Menge der in diesem Schritt verbleibenden Intervalle. Der Einflussbereich von $[a, b]_l$ auf Ψ_i ist dann $[2a - b, 2b - a]$.*

Beweis. Aus dem Algorithmus folgt, dass alle Intervalle in Ψ_i kürzer als oder gleich lang wie das betrachtete Intervall $[a, b]_l$ sind. Sei $[c, d]_{l'} \in \Psi_i$ das Intervall, für das gilt $[c, d] \cap [a, b] \neq \emptyset$. Dann gelten für den Startzeitpunkt c von $[c, d]_{l'}$ folgende Ungleichungen.

$$c \leq b$$
$$c \geq a - (b - a) = 2a - b$$

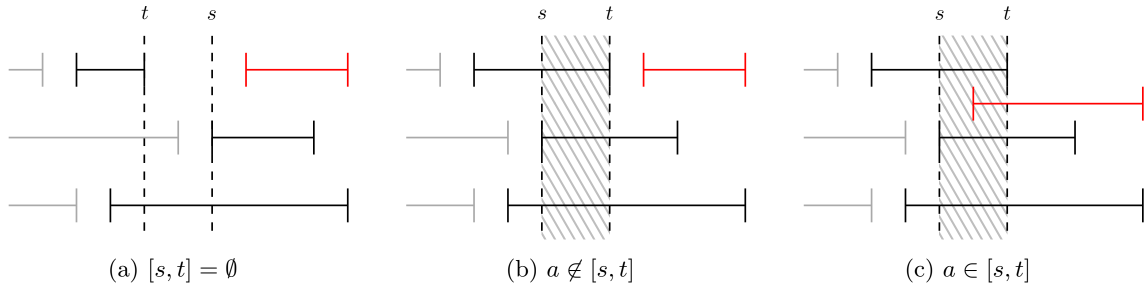
Falls $c < 2a - b$ gilt, ist mindestens eine der Voraussetzungen falsch. Das heißt, $[c, d] \cap [a, b] = \emptyset$ oder $[c, d]_{l'} \notin \Psi_i$. Ebenso, falls $c > b$ ist, muss $[c, d] \cap [a, b] = \emptyset$ gelten. Für den Endzeitpunkt d gelten, nach gleicher Begründung wie für c folgende Ungleichungen.

$$d \geq a$$
$$d \leq b + (b - a) = 2b - a$$

Damit ist $[2a - b, 2b - a]$ der Einflussbereich von $[a, b]_l$ auf Ψ_i. □

Später müssen wir zeigen können, dass die Konfliktpartner des betrachteten Intervalls in diesem Einflussbereich liegen. Ein Konflikt zwischen zwei Intervallen kann nur dann bestehen, wenn die beiden beteiligten Labels der Intervalle präsent sind. Außerdem existieren Intervalle nur für präsente Labels. Aus der Definition der Konflikte geht hervor, dass sich die Konftpartner eines Intervalls $[a, b]_l$ mit diesem zeitlich überschneiden müssen.

Es können nicht nur beim Auflösen der Konflikte, sondern auch beim Einfügen in die Lösung Fehler gemacht werden. Die Betrachtung einer Lösung ist daher für den Beweis des Approximationsfaktors ebenfalls hilfreich. Das nachfolgende Lemma enthält eine intuitive Möglichkeit für die Darstellung einer gültigen, insbesondere der optimalen Lösung.



(a) $[s,t] = \emptyset$ (b) $a \notin [s,t]$ (c) $a \in [s,t]$

Abbildung 3.2: Beispiel der möglichen Fälle, während dem, nach Startzeiten aufsteigenden, Einfügevorgang eines Intervalls (grau) in eine von k Mengen. In (c) ist ein Widerspruch zur k-Eigenschaft für k = 3 zu sehen

Lemma 3.3. *Die Intervalle einer gültigen Lösung können disjunkt in k Mengen eingefügt werden.*

Beweis. Wir zeigen das Lemma mithilfe eines konstruktiven Beweises. Dazu betrachten wir k, zu Beginn leere Mengen Ω_1 bis Ω_k und die Intervalle einer Lösung Ψ. Die Menge Ψ muss die k-Eigenschaft erfüllen. Wir sortieren die Intervalle aufsteigend nach ihrem Startzeitpunkt und fügen sie in dieser Reihenfolge in die k Mengen ein. Dass ein überlappungsfreies Einfügen in die k Mengen möglich ist, zeigen wir durch vollständige Induktion über die Anzahl der bereits eingefügten Intervalle.

Ein Intervall $[a,b]_l$ aus Ψ darf nur dann in eine Menge Ω_i eingefügt werden, wenn $[a,b] \cap [x,y] = \emptyset \; \forall [x,y]_{l'} \in \Omega_i$ ist. Da zu Beginn die k Mengen leer sind und der Induktionsschluss $\Omega_i \neq \emptyset$ erwartet, fügen wir das Intervall $[-2,-1]$ in jede der Mengen Ω_1 bis Ω_k ein. Diese Intervalle liegen außerhalb des Zeitbereichs $[0,1]$ und beeinflussen somit die einzufügenden Intervalle nicht.

Der **Induktionsanfang** besteht aus dem Einfügen des ersten Intervalls aus Ψ in eine der k Mengen. Das Einfügen dieses Intervalls ist immer möglich.

Als **Induktionsvoraussetzung** muss, nach dem Einfügen der ersten j Intervalle, für jedes Ω_i gelten, dass die darin enthaltenen Intervalle sich nicht überschneiden.

Für den **Induktionsschluss** beweisen wir, dass in Schritt j das Einfügen des entsprechenden Intervalls $[a,b]_l$ in eine der Mengen Ω_1 bis Ω_k möglich ist. Das Intervall $[s,t]$ sei der Schnitt der jeweils letzten Intervalle aus Ω_1 bis Ω_k. Aufgrund des sortierten Einfügens gilt immer $a > s$. Ist $[s,t] = \emptyset$, gilt zusätzlich $a > t$ (siehe Abbildung 3.2a). Betrachtet man $[a,b]_l$ bezüglich des Zeitbereichs $[s,t]$ kann man zwei mögliche Fälle unterscheiden. Demnach sind nur die beiden Fälle, $a \notin [s,t]$ (1. Fall) und $a \in [s,t]$ (2. Fall) zu betrachten.

1. Fall $\mathbf{a} \notin [\mathbf{s},\mathbf{t}]$: Es gibt in einer der Mengen Ω_1 bis Ω_k ein Intervall $[c,t]_{l'}$, welches den Endzeitpunkt t besitzt. Sei Ω_i die Menge, die $[c,t]_{l'}$ enthält. Offensichtlich endet $[c,t]_{l'}$ bevor $[a,b]_l$ beginnt. Es existieren keine späteren Intervalle. Damit ist in mindestens einer der k Mengen Platz für $[a,b]_l$. Wir können $[a,b]_l$ also in Ω_i einfügen. In den Abbildungen 3.2a und 3.2b sind diese Fälle an einem Beispiel verdeutlicht.

2. Fall $\mathbf{a} \in [\mathbf{s},\mathbf{t}]$: Würde a in $[s,t]$ liegen, befänden sich mit dem Intervall $[a,b]_l$ im Bereich $[s,t]$ mehr als k gleichzeitig präsente Intervalle (siehe dazu Abbildung 3.2c). Dies ist nach der Induktionsvoraussetzung nicht möglich.

Der Startzeitpunkt a von $[a,b]_l$ liegt also immer außerhalb von $[s,t]$. Damit haben wir bewiesen, dass ein Einfügen in eine der Mengen Ω_1 bis Ω_k immer möglich ist. Mithilfe der Induktion über die Einfügevorgänge ist dann gezeigt, dass alle Intervalle einer gültigen Lösung überlappungsfrei in k Mengen eingefügt werden können.

□

Gütegarantie

Für Einheitsquadrate als Labels liefert der Algorithmus keine beliebig schlechten Ergebnisse, sondern weist eine Approximationsgüte auf. Mit den vorangegangenen Vorbereitungen können wir diese nun zeigen.

Theorem 3.4. *Werden Einheitsquadrate als Labels und die Länge eines Intervalls als sein Gewicht verwendet, besitzt der Algorithmus* GREEDY-k-RESTRICTEDMAXTOTAL() *eine Approximationsgüte von* $^1/_{27}$.

Beweis. Um den Approximationsfaktor zu zeigen, betrachten wir einen beliebigen Schritt i des Algorithmus. Für jedes betrachtete Intervall werden wir Kosten berechnen, welche angeben, um welches Gewicht Schritt i schlechter ist als der entsprechende Schritt, der zur optimalen Lösung führt. Die Kosten erlauben es uns anschließend die Approximationsgüte zu bestimmen.

Sei $[a,b]_l$ das in Schritt i betrachtete Intervall. Nach Definition des Algorithmus ist dies das größte Intervall in Ψ_i. Falls ein Widerspruch zur k-Eigenschaft in der Lösungsmenge Ψ' zusammen mit $[a,b]_l$ vorliegt, wird $[a,b]_l$ verworfen (1. Fall). Andernfalls wird $[a,b]_l$ in die Lösungsmenge verschoben und die verbleibenden Intervalle aus $C^l_{[a,b]}$, die mit $[a,b]_l$ in Konflikt stehen, gelöscht (2. Fall).

1. Fall

Liegt ein Widerspruch zur k-Eigenschaft in $\Psi' \cup \{[a,b]_l\}$ vor, wird $[a,b]_l$ verworfen. Falls $[a,b]_l$ aber ein Teil der optimalen Lösung ist, geht damit das Gewicht $w([a,b]_l)$ dieses Intervalls für die Lösung verloren. Die Lösung weicht also in diesem Schritt um $w([a,b]_l)$ von der optimalen Lösung ab und es entstehen Kosten von $w([a,b]_l)$. Ist $[a,b]_l$ hingegen nicht in der optimalen Lösung vertreten, so kann dieses Vorgehen als optimal bezeichnet werden. Es entstehen also keine Kosten. Später werden wir sehen, dass die Kosten dieses Falls im nächsten Fall bereits erfasst sind.

2. Fall

Sofern kein Widerspruch zur k-Eigenschaft in $\Psi' \cup \{[a,b]_l\}$ vorliegt, wird $[a,b]_l$ der Lösungsmenge hinzugefügt. Zugleich werden die verbleibenden Intervalle aus $C^l_{[a,b]}$ aus Ψ entfernt. Liegt $[a,b]_l$ in der optimalen Lösung, wird in diesem Schritt kein Fehler gemacht und es entstehen ebenfalls keine Kosten. Wenn $[a,b]_l$ hingegen nicht in der optimalen Lösung vorhanden ist, werden Fehler gemacht. Diese Fehler entstehen einerseits durch das Löschen der verbleibenden Intervalle aus $C^l_{[a,b]}$ und andererseits durch das Einfügen von $[a,b]_l$ in die Lösung. Wird $[a,b]_l$ in die Lösungsmenge eingefügt, können in einem späteren Schritt bestimmte Intervalle, welche in der optimalen Lösung vorkommen, nicht mehr eingefügt werden, da sie sonst die k-Eigenschaft verletzen würden. In den nächsten zwei Paragraphen betrachten wir den Fehler der beiden Schritte dieses Falls genauer.

Konflikte Zurerst betrachten wir den Fehler, der durch die Konflikte entsteht. Nach Lemma 3.2 ist der Einflussbereich von $[a,b]_l$ auf Ψ_i auf $[2a-b, 2b-a]$ beschränkt. Da Konfliktpartner von $[a,b]_l$ gleiche Zeitpunkte mit ihm aufweisen müssen, liegen diese auch in dem Bereich $[2a-b, 2b-a]$. In Abbildung 3.3 ist beispielhaft dieser Einflussbereich zu sehen. Somit können alle Intervalle, welche sowohl mit $[a,b]_l$ in Konflikt stehen, als auch untereinander keine zeitliche Überschneidung aufweisen, ein maximales Gewicht von $3 \cdot (b-a) = 3 \cdot w([a,b]_l)$ besitzen.

Natürlich können sich die Konfliktintervalle auch untereinander zeitlich überschneiden. Um in der optimalen Lösung zu liegen, dürfen die Intervalle aber keine Konflikte untereinander aufweisen. Die Anzahl der gleichzeitig aktiven, nicht in einem Konflikt stehenden Intervalle ist aber nach dem PACKING-Lemma (siehe [GNN13, Lemma 2]) beschränkt. Das PACKING-Lemma besagt, dass maximal acht Labels, welche durch Einheitsquadrate dargestellt sind, gleichzeitig aktiv sein können, ohne miteinander in Konflikt zu stehen.

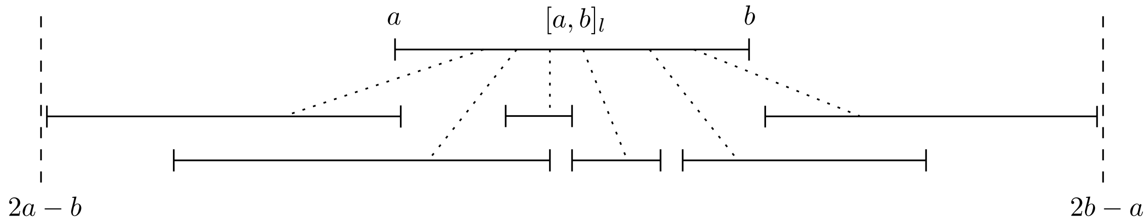


Abbildung 3.3: Der maximale Bereich, in dem die verbleibenden Konfliktpartner eines Intervalls $[a,b]_l$ in Schritt i des Algorithmus liegen können. Beispielhaft sind die verbleibenden Konfliktpartner von $[a,b]_l$ eingezeichnet

Im schlimmsten Fall liegen alle Konfliktpartner von $[a,b]_l$, die untereinander keine Konflikte besitzen und den gesamten Einflussbereich ausfüllen, in der optimalen Lösung. Diese Intervalle besitzen ein maximales Gesamtgewicht von $8 \cdot 3 \cdot w([a,b]_l)$. Da das Intervall $[a,b]_l$ in die Lösung eingefügt wird, ist die optimale Lösung für diesen Teilschritt um den Fehler $8 \cdot 3 = 24$ besser.

Einfügen in die Lösung Sobald ein Intervall, welches nicht in der optimalen Lösung vorhanden ist, in die Lösung eingefügt wird, entstehen ebenfalls Fehler. Der spätere Versuch ein Intervall einzufügen, welches in der optimalen Lösung liegt, kann dann aufgrund der k-Eigenschaft scheitern.

Das Intervall $[a,b]_l$ liegt in dieser Betrachtung des 2. Falls nicht in der optimalen Lösung. Zudem können wir nach Lemma 3.3 die Intervalle der optimalen Lösung überlappungsfrei in k Mengen einfügen. Durch Einfügen von $[a,b]_l$ in die Lösung werden Intervalle einer Menge der optimalen Lösung verdrängt. Dies bedeutet, dass Intervalle, die in der optimalen Lösung liegen, nicht in die Menge eingefügt werden können, da sie sich sonst mit $[a,b]_l$ überschneiden würden. Um nun die Kosten für $[a,b]_l$ berechnen zu können, müssen wir die Länge der verdrängten Intervalle betrachten.

Damit ein Intervall dieser Menge verdrängt wird, muss es sich mit $[a,b]_l$ zeitlich überschneiden. Somit können wir den Einflussbereich von $[a,b]_l$ auf der Menge der Intervalle einer optimalen Lösung verwenden (siehe Definition 3.1). Dabei sei Ψ^* die Menge der Intervalle einer optimalen Lösung. Über den Einflussbereich auf Ψ^* können wir jedoch keine Aussagen über eine beschränkte Größe machen, da die Intervalle in Ψ' auch länger als $[a,b]_l$ sein können.

Die Kosten für das einzufügende Intervall werden aber in jedem Schritt und somit auch in den vorherigen Schritten berechnet. Damit ist lediglich eine Betrachtung der Intervalle, die kürzer als $[a,b]_l$ sind, nötig. Für die größeren Intervalle wurden Kosten schon in einem vorherigen Schritt berechnet oder diese wurden vorher in die Lösung eingefügt. Die kürzeren Intervalle liegen in der Menge Ψ_i. Der Einflussbereich von $[a,b]_l$ ist also nur auf Ψ_i zu betrachten. Die Größe dieses Einflussbereichs ist $3 \cdot w([a,b]_l)$ (siehe Lemma 3.2). Im schlimmsten Fall füllen die verdrängten Intervalle den kompletten Einflussbereich. Diese Intervalle besitzen dann ein Gesamtgewicht von $3 \cdot w([a,b]_l)$. Da aber auch hier $[a,b]_l$ eingefügt wird, ist die optimale Lösung für diesen Teilschritt drei mal besser.

Gesamtkosten für den 2. Fall Somit nehmen wir insgesamt, für den Fall, dass $[a,b]_l$ in die Lösung aufgenommen wird, aber nicht in der optimalen Lösung liegt, Kosten von $24 \cdot w([a,b]_l) + 3 \cdot w([a,b]_l) = 27 \cdot w([a,b]_l)$ an. Diese Kosten werden durch das Löschen der Konflikte und das Einfügen von $[a,b]_l$ in die Lösung verursacht.

Betrachtung aller Schritte

Ein Intervall wird nur dann nicht in die Lösung aufgenommen und verworfen, wenn es zu einem Widerspruch der k-Eigenschaft in der Lösung kommen würde. Dies bedeutet, dass wir die Kosten für den ersten Fall schon vorher berechnet haben. Wir müssen also nur die Kosten der Schritte berechnen, für die das gewählte Intervall zur Lösung hinzugefügt wird.

Für die Betrachtung des Approximationsfaktors werden wir daher ausschließlich die Schritte annehmen, bei denen ein Intervall in die Lösung eingefügt wird, das nicht in der optimalen Lösung liegt. Für solch einen Schritt, mit betrachtetem Intervall $[a,b]_l$, belaufen sich die Kosten auf $27 \cdot w([a,b]_l)$. Es wird aber nur das Intervall $[a,b]_l$ mit dem Gewicht $w([a,b]_l)$ in die Lösung aufgenommen. Für den gesamten Approximationsfaktor des Algorithmus gilt somit im schlimmsten Fall folgende Gleichung.

$$\frac{w([a,b]_l)}{27 \cdot w([a,b]_l)} = \frac{1}{27}$$

□

3.3. Laufzeit

Der Algorithmus GREEDY-k-RESTRICTEDMAXTOTAL() besteht aus dem Sortieren der Eingabedaten und dem anschließenden Behandeln jedes Intervalls. Das Überprüfen geschieht mithilfe der Methode PRÜFE-k-EIGENSCHAFT() und kann auf verschiedene Weise umgesetzt werden (siehe Abschnitt 3.4). Die Laufzeit können wir also in Abhängigkeit der Funktion PRÜFE-k-EIGENSCHAFT() angeben.

Theorem 3.5. *Die worst-case Laufzeit des Algorithmus* GREEDY-k-RESTRICTEDMAX-TOTAL() *liegt in* $O(|\Psi| \cdot \log |\Psi| + |\Psi| \cdot T(\text{PRÜFE}-k-\text{EIGENSCHAFT}()) + I)$. *Hierbei sei* $T(\text{PRÜFE}-k-\text{EIGENSCHAFT}())$ *die Laufzeit der Funktion* PRÜFE-k-EIGENSCHAFT() *und* I *die Anzahl der Konflikte.*

Beweis. Das Sortieren der Eingabe erfolgt einmalig zu Beginn und besitzt bekanntlich eine Laufzeit in $\Omega(|\Psi| \cdot \log |\Psi|)$ (siehe [CLRS09]). Dies bildet den ersten Summanden der Laufzeit. Der zweite Summand entsteht durch die Überprüfung der Elemente der Menge Ψ, ob sie der Lösung hinzugefügt werden können. Im ungünstigen Fall müssen alle Intervalle der Eingabemenge betrachtet werden. Dieser Fall tritt ein, wenn keine Konflikte vorliegen. Die Überprüfung für ein einzelnes Intervall erfolgt schließlich mithilfe der Funktion PRÜFE-k-EIGENSCHAFT(), weshalb deren Laufzeit für jeden Schritt in die Gesamtlaufzeit mit einfließt. Eventuell müssen die Konfliktpartner des betrachteten Intervalls $[a,b]_l$ gelöscht werden. Dazu werden alle Konflikte von $[a,b]_l$ betrachtet. Insgesamt müssen alle Konflikte einmal betrachtet werden, wodurch deren Anzahl additiv in die Laufzeit mit einfließt. □

3.4. Varianten mit unterschiedlicher Laufzeit

Die Gesamtlaufzeit des Algorithmus hängt maßgeblich von der Laufzeit ab, die benötigt wird, um in einem Schritt die k-Eigenschaft zu überprüfen. Die Anzahl der Konflikte I fließt ebenfalls mit in die Laufzeit ein. Jedoch gilt $I \in |\Psi|^2$, weshalb I in den hier besprochenen Laufzeiten, nicht mehr explizit vorkommt. Der Algorithmus fügt ein Präsenzintervall der Lösungsmenge nur dann hinzu, wenn dieses Intervall dort keinen Widerspruch zur k-Eigenschaft auslöst. Diese Überprüfung geschieht mithilfe der Funktion PRÜFE-k-EIGENSCHAFT(). Die Funktion erwartet eine Menge Ψ' von Intervallen, ein einzelnes Intervall $[a,b]_l$ sowie den ganzzahligen Parameter k. Geprüft wird dann, ob das Präsenzintervall in der Menge der aktiven Intervalle einen Widerspruch zur k-Eigenschaft erzeugt. Die Ausgabe der Funktion sei TRUE, wenn $\Psi' \cup [a,b]_l$ zu einem Zeitpunkt mehr als k Intervalle besitzt. Falls die k-Eigenschaft verletzt ist, soll FALSE zurück gegeben werden. Im Folgenden werden wir verschiedene Varianten des Algorithmus vorstellen. Dabei unterscheiden sich diese Varianten hauptsächlich in der Funktion zur Überprüfung der k-Eigenschaft. Allen Varianten liegt ein *Sweepline*-Vorgehen zu Grunde. Dazu werden alle Zeitpunkte aufsteigend durchlaufen. Dieses Vorgehen ermöglicht die Bestimmung der Anzahl der aktiven Intervalle.

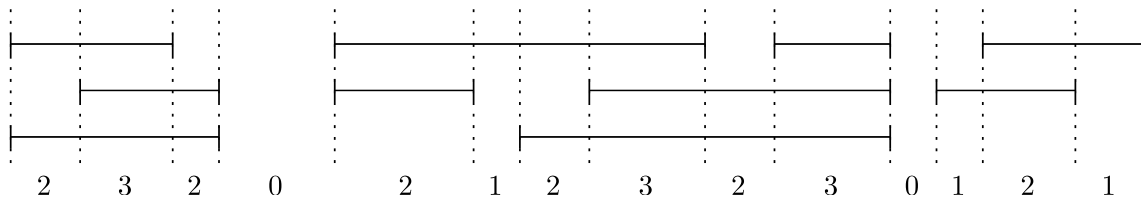


Abbildung 3.4: Anzahl der aktiven Intervalle für eine Intervallmenge

Sweepline-Vorgehen

Das Vorgehen mithilfe einer sogenannten Sweepline erlaubt es, die Anzahl der aktiven Intervalle für alle Zeitpunkte zu bestimmen. Unter allen Zeitpunkten können wir zusammenhängende Bereiche identifizieren, in denen sich die Anzahl der aktiven Intervalle nicht verändert (siehe Abbildung 3.4). Nach folgendem Lemma werden diese Bereiche von einer Start- bzw. Endzeit zweier Intervalle begrenzt.

Lemma 3.6. *Sei T die sortierte Menge aller Start- und Endzeitpunkte einer Intervallmenge, $x \in T$ ein beliebiger Zeitpunkte aus T und $y \in T$ der direkt darauffolgende Zeitpunkt. Dann ist die Anzahl der Intervalle für jeden Zeitpunkt $t \in [x, y]$ konstant.*

Beweis. Angenommen die Anzahl der Intervalle zu einem Zeitpunkt wäre im Bereich $[x, y]$ nicht konstant, dann muss mindestens ein Start- oder Endzeitpunkt zwischen x und y liegen. Dies ist aber ein Widerspruch zur Voraussetzung, dass y der direkt auf x folgende Zeitpunkt sei. □

Es müssen also nicht alle möglichen Zeiten betrachtet werden. Es reicht aus, nur die Start- und Endzeitpunkte aller Intervalle zu behandeln. Um dies umzusetzen, werden alle Zeitpunkte in zeitlich aufsteigender Reihenfolge betrachtet. Zusätzlich wird ein globaler Zähler geführt, welcher zu Beginn mit 0 initialisiert wird. Nun wird bei jeder Startzeit der Zähler um 1 erhöht und bei jeder Endzeit um 1 verringert. Damit ergibt sich, anhand des Zählerstandes, die Anzahl der aktiven Intervalle für den Bereich zwischen dem soeben betrachteten und den darauf folgenden Start- oder Endzeitpunkt.

Naive Methode

Dieses Vorgehen können wir nun ausnutzten, um die Methode Prüfe-k-Eigenschaft() zu entwerfen. Für das Sweepline-Vorgehen wird zu Beginn das einzelne Intervall in die Intervallmenge eingefügt. Von der so entstandenen Intervallmenge Ψ^* wird dann die Menge T aller Start- und Endzeitpunkte bestimmt. Anschließend müssen diese Zeitpunkte zeitlich aufsteigend sortiert werden. Nach diesen Vorbereitungen wird, wie oben beschrieben, mittels Sweepline und Zähler vorgegangen. Sobald der Zähler die vorgegebene Grenze k übersteigt, liegt ein Widerspruch zur k-Eigenschaft vor. Falls hingegen alle Zeiten betrachtet werden, ohne dass ein Widerspruch auftritt, liegen zu jedem Zeitpunkt maximal k Intervalle vor. Wir nennen die Funktion im Folgenden Prüfe-k-EigenschaftNaiv(). Algorithmus 3.2 skizziert das Verhalten dieser Funktion.

Nun werden wir die Laufzeit der Funktion Prüfe-k-EigenschaftNaiv() bestimmen. Die Laufzeit ist abhängig von der Anzahl der zu betrachtenden Zeitpunkte. Diese wiederum ergibt sich durch die Zahl der Intervalle. Jedes Intervall besitzt genau eine Start- und eine Endzeit. Damit wächst die Anzahl aller möglichen Zeitpunkte linear mit der Anzahl der Intervalle.

Beobachtung 3.7. *Die Anzahl aller Start- und Endzeiten für eine Menge Ψ' von Intervallen liegt in $O(|\Psi'|)$.*

Die Eingabemenge Ψ' der Funktion Prüfe-k-EigenschaftNaiv() besitzt im schlimmsten Fall so viele Intervalle wie die Eingabe für den gesamten Algorithmus. Damit können

Algorithmus 3.2 : PRÜFE-k-EIGENSCHAFTNAIV(Ψ', $[a,b]_l$, K)

Eingabe : Menge von Intervallen Ψ', Einzufügendes Interval $[a,b]_l$, Parameter k
Ausgabe : true falls $[a,b]_l$ in Ψ' einen Widerspruch zur k-Eigenschaft erzeugen würde, andernfalls false
Voraussetzung : Ψ' besitzt keinen Widerspruch zu k-Eigenschaft

```
1  begin
2      Ψ* ← Ψ' ∪ {[a,b]_l};
3      T ← {Start- und Endzeitpunkte der Intervalle aus Ψ* in aufsteigender
       Reihenfolge};
4      returnWert ← false;
5      for zeit ∈ T do
6          if IstStartzeit(zeit) then
7              counter ++;
8          else
9              counter −−;
10         if counter > k then
11             returnWert ← true;
12             break;
13     return returnWert;
```

wir die Laufzeit für die Funktion angeben. Zu Beginn werden die Zeiten sortiert und anschließend einzeln betrachtet. Im schlimmsten Fall müssen alle Zeiten betrachtet werden. So ergibt sich, zusammen mit Beobachtung 3.7, die Laufzeit der Funktion.

Korollar 3.8. *Die worst-case Laufzeit der Funktion* PRÜFE-k-EIGENSCHAFTNAIV() *liegt in* $O(|\Psi| \log |\Psi|)$.

Da die Laufzeit für Algorithmus GREEDY-k-RESTRICTEDMAXTOTAL() von der Laufzeit der Funktion zur Prüfung der k-Eigenschaft abhängt, können wir diese nun, mit Theorem 3.5 und Korollar 3.8, vollständig angeben.

Korollar 3.9. *Algorithmus* GREEDY-k-RESTRICTEDMAXTOTAL() *besitzt, unter Verwendung der Funktion* PRÜFE-k-EIGENSCHAFTNAIV(), *eine Laufzeit in* $O(|\Psi|^2 \log |\Psi|)$.

Verbesserung der Laufzeit

Das Einfügen eines Intervalls in die Lösungsmenge Ψ' ist endgültig. Zur Ψ' kommen also nur Intervalle hinzu. Genauso verhält es sich dementsprechend mit der Menge der Zeitpunkte T der Intervalle der Lösungsmenge. Jedoch wird T in der Funktion PRÜFE-k-EIGENSCHAFTNAIV() nach jedem Durchlauf verworfen. Daher muss T zu Beginn immer neu gebildet und sortiert werden. Ein besseres Vorgehen besteht darin, die Menge für jeden Aufruf der Funktion weiter zu verwenden. Dazu müssen wir jedoch Veränderungen an Algorithmus GREEDY-k-RESTRICTEDMAXTOTAL() vornehmen. Folglich muss die Funktion zur Überprüfung der k-Eigenschaft, die Menge der sortierten Zeitpunkte T der Elemente aus Ψ' erhalten. Die Menge T wird zu Beginn deklariert und mit der leeren Menge initialisiert. Auch die Arbeitsweise der Funktion muss angepasst werden. Einerseits fällt das Sortieren der Zeiten weg, andererseits müssen beim Sweepline-Vorgehen die Zeiten des einzufügenden Intervalls $[a,b]_l$ beachtet werden. Schließlich müssen die Zeiten von $[a,b]_l$ sortiert in T eingefügt werden, sofern kein Widerspruch zur k-Eigenschaft gefunden wurde. Es sei hier angenommen, dass die Änderungen an T beim Verlassen der Funktion erhalten bleiben.

Algorithmus 3.3 : PRÜFE-k-EIGENSCHAFTLINEAR(T, $[a,b]_l$, K)

Eingabe : Menge von sortierten Zeitpunkten T, Einzufügendes Intervall $[a,b]_l$, Parameter k

Ausgabe : true falls $[a,b]_l$ in Ψ', dessen Intervallzeitpunkte in T vorliegen, einen Widerspruch zur k-Eigenschaft erzeugen würde, andernfalls false

Voraussetzung : Ψ' besitzt keinen Widerspruch zu k-Eigenschaft

```
1  begin
2      t_s ← Startzeitpunkt von [a,b]_l;
3      t_e ← Endzeitpunkt von [a,b]_l;
4      Füge t_s und t_e, sortiert, in die Menge T ein;
5      for zeit ∈ T do
6          if IstStartzeit(zeit) then
7              counter ++;
8          else
9              counter −−;
10         if counter > k then
11             returnWert ← true;
12             break;
13     if returnWert = true then entferne t_s und t_e aus T;
14     return returnWert;
```

Wir bezeichnen diese Funktion als PRÜFE-k-EIGENSCHAFTLINEAR(). Algorithmus 3.3 zeigt das Verhalten der Funktion.

Das ständige Sortieren zu Beginn ist nun nicht mehr erforderlich. Dafür müssen die Zeitpunkte von x in die Menge sortiert eingefügt und eventuell wieder gelöscht werden. Diese Schritte lassen sich mithilfe eines Arrays in linearer Zeit und mit balancierten binären Bäumen in logarithmischer Zeit durchführen (siehe [CLRS09]). Insgesamt ergibt sich linearer Zeitaufwand für die Funktion PRÜFE-k-EIGENSCHAFTLINEAR().

Korollar 3.10. *Die Laufzeit der Funktion* PRÜFE-k-EIGENSCHAFTLINEAR() *liegt in* $O(|T|)$.

An der Laufzeit des angepassten Algorithmus GREEDY-k-RESTRICTEDMAXTOTAL() hat sich nichts geändert. Diese entspricht der Laufzeit des unangepassten Algorithmus (siehe Theorem 3.5). Die Menge T besitzt im schlimmsten Fall so viele Elemente wie die Eingabe für den gesamten Algorithmus. Es gilt also $|T| \in O(|\Psi|)$. Der Algorithmus GREEDY-k-RESTRICTEDMAXTOTAL() läuft somit, wenn die Funktion PRÜFE-k-EIGENSCHAFTLINEAR() verwendet wird, in quadratischer Zeit.

Korollar 3.11. *Algorithmus* GREEDY-k-RESTRICTEDMAXTOTAL() *besitzt, unter Verwendung der Funktion* PRÜFE-k-EIGENSCHAFTLINEAR()*, eine Laufzeit, die in* $O(|\Psi|^2)$ *liegt.*

Eine alternative Möglichkeit besteht darin, alle möglichen Zeitpunkte zu Beginn des Algorithmus zu sortieren und mit einem zusätzlichen Attribut zu versehen. Dieses Attribut gibt an, ob das entsprechende Intervall des Zeitpunktes in der Lösung enthalten ist oder nicht. Ist es enthalten, muss der Zeitpunkt in der Funktion PRÜFE-k-EIGENSCHAFTLINEAR() betrachtet werden. Das Attribut jedes Zeitpunktes wird mit FALSE initialisiert und bedeutet, dass diese Zeit nicht betrachtet werden muss. Die Funktion PRÜFE-k-EIGENSCHAFTLINEAR() setzt dann zu Beginn das Attribut der Start- und Endzeit von $[a,b]_l$ in

T auf TRUE. Dann wird das Sweepline-Verfahren auf alle Zeiten, deren Attribut TRUE ist, angewandt. Abschließend müssen die Attribute wieder zurück gesetzt werden, wenn die k-Eigenschaft verletzt wurde. Dieser Ansatz besitzt offensichtlich ebenfalls quadratische Laufzeit.

Kleinen Bereich prüfen

Die k-Eigenschaft wird durch die Funktion PRÜFE-k-EIGENSCHAFTLINEAR() bei jedem Einfügen eines Intervalls überprüft. Für einen erneuten Durchlauf der Funktion werden wieder alle möglichen Zeitpunkte in T betrachtet. Somit werden die Anzahlen der aktiven Intervalle komplett neu berechnet. Wie in Lemma 3.6 beschrieben, existiert eine Menge B von disjunkten Zeitbereichen, innerhalb derer die Anzahl der Intervalle konstant ist. Das Hinzufügen eines Intervalls $[a,b]_l$ ändert die Anzahlen der aktiven Intervalle. Jedoch geschieht dies ausschließlich in den Bereichen $B' = \{[c,d] \mid [c,d] \in B, [c,d] \subseteq [a,b]\} \subseteq B$. Da die Intervalle der Größe nach absteigend in die Lösung aufgenommen werden, ist davon auszugehen, dass die Anzahl der Bereiche in B' deutlich geringer ist als die der Menge B. Damit liegt es nahe, die Anzahl der aktiven Intervalle für jeden Bereich zu speichern und nur für die Bereiche in B' zu aktualisieren.

Die Bereiche in B sind zeitlich konsekutiv. Da nur ein Intervall eingefügt wird und dieses Intervall zusammenhängend ist, können wir für B' Folgendes beobachten.

Beobachtung 3.12. *Die Bereiche in B' sind zeitlich konsekutiv.*

Aus Beobachtung 3.12 folgt, dass nur der Startzeitpunkt $x \in T$ des ersten und der Endzeitpunkt $y \in T$ des letzten Bereiches in B' identifiziert werden muss. Das Sweepline-Vorgehen von oben muss also nur auf dem Zeitbereich $[x,y]$ durchgeführt werden. Eine Verkettung der Zeitpunkte in T durch Referenz auf den Nachfolger bietet sich für das Sweepline-Vorgehen an. Sobald x bekannt ist, können die folgenden Zeitpunkte aus T nacheinander, bis zum Erreichen von y abgearbeitet werden. Die Suche nach einem Element in einer verketteten Liste benötigt linearen Zeitaufwand in der Eingabegröße. Deshalb ist es sinnvoll, die Elemente in T zusätzlich in einem balancierten binären Baum zu organisieren. Nun lässt sich die Suche mit logarithmischem Zeitaufwand durchführen (siehe [CLRS09]). Im Folgenden sei T eine Datenstruktur, deren Elemente sowohl in einem balancierten binären Baum gespeichert, als auch untereinander in zeitlicher Reihenfolge verkettet sind. Zusätzlich wird die Anzahl der aktiven Intervalle für den zeitlich folgenden Bereich in jedem Element in T als zusätzliches Attribut gespeichert.

Im Folgenden skizzieren wir das Vorgehen der Funktion PRÜFE-k-EIGENSCHAFTLOKAL(). Die Funktion erhält die Menge T der Zeitpunkte der Lösungsintervalle, das zu prüfende Intervall $[a,b]_l$ und den Parameter k. Wir setzen voraus, dass die Intervalle der Zeitpunkte in T die k-Eigenschaft nicht verletzen. Zu Beginn wird der Startzeitpunkt x ermittelt. Dieser Zeitpunkt liegt immer direkt vor a. Das zusätzliche Attribut von x, welches die Anzahl der aktiven Intervalle, für den Bereich ab diesem Zeitpunkt angibt, wird als Wert für den Zähler übernommen. Nun wird das nachfolgenden Element von x, in der verketteten Liste, betrachtet. Dabei muss ständig überprüft werden, ob a oder b schon überschritten wurde und ob das Ende y erreicht ist. Das Sweepline-Vorgehen wird analog angewandt. Wird kein Widerspruch zur k-Eigenschaft gefunden, wird a und b eingefügt und die zusätzlichen Attribute aktualisiert.

Die worst-case Laufzeit verbessert sich nicht, da $B = B'$ auftreten kann. Dieser Fall tritt beispielsweise für ein Intervall ein, welches die gesamte Zeitachse füllt.

Korollar 3.13. *Die worst-case Laufzeit der Funktion* PRÜFE-k-EIGENSCHAFTLOKAL() *liegt in $O(|\Psi|)$.*

Die worst-case Laufzeit des Algorithmus GREEDY-k-RESTRICTEDMAXTOTAL() mit der Funktion PRÜFE-k-EIGENSCHAFTLOKAL() unterscheidet sich damit nicht von Laufzeit des

Algorithmus mit der Funktion Prüfe-k-EigenschaftLinear() (siehe Korollar 3.11). Der worst-case Fall kommt jedoch in der Praxis selten vor, womit zu erwarten ist, dass die tatsächliche Laufzeit des Algorithmus mit der Funktion Prüfe-k-EigenschaftLokal() besser ist als die anderen Varianten. Eine experimentelle Analyse, welche in Kapitel 5 präsentiert wird, bestätigt diese Annahme.

Wir haben in diesem Kapitel einen Algorithmus vorgestellt, der eine beweisbare Gütegarantie aufweist. Verschiedene Varianten dieses Algorithmus besitzen unterschiedliche Laufzeiten, die deutlich besser sind, als die Laufzeit des Algorithmus aus [GNN13]. Wie gut sich der Algorithmus in der Praxis verhält wird in Kapitel 5 untersucht.

4. Algorithmen für die Teilprobleme

Der Algorithmus aus Kapitel 3 löst gleichzeitig die Konflikte unter Beachtung von AM1 auf und stellt die k-Eigenschaft her. Hier wollen wir Algorithmen vorstellen, welche die beiden Teilprobleme separat behandeln. Die Konflikte werden hier ebenfalls unter Beachtung des Aktivitätsmodells AM1 aufgelöst. Zu Beginn werden wir Algorithmen zur Konfliktauflösung und anschließend, in Abschnitt 4.2, zur Herstellung der k-Eigenschaft behandeln. Abschließend betrachten wir die Approximationsgüte für die Ausführung der Algorithmen nacheinander.

4.1. Auflösung der Konflikte

In einer gültigen Lösung dürfen, neben der k-Eigenschaft, keine Konflikte mehr vorhanden sein. Im Aktivitätsmodell AM1 bedeutet dies, dass eines, der an einem Konflikt beteiligten Intervalle nicht angezeigt werden darf. Werden nur die Konflikte, unter Beachtung von AM1 aufgelöst, entspricht dies dem Problem GeneralMaxTotal. Wir werden nun Algorithmen vorstellen, welche ausschließlich die Konflikte auflösen. Dabei arbeitet der erste Algorithmus nach gleichem Prinzip wie der Algorithmus aus Kapitel 3. Mit der Einschränkung auf einen Konflikte pro Intervall arbeitet dieser Algorithmus optimal. Anschließend präsentieren wir ein lineares Programm, welches die Konfliktauflösung optimal vornimmt, wenn maximal zwei Konflikte pro Intervall vorliegen.

4.1.1. Approximativer Greedy-Algorithmus zur Konfliktauflösung

Zur approximativen Lösung von GeneralMaxTotal wird in [GNN13] ein Algorithmus vorgestellt. Dieser verfährt ähnlich wie der Algorithmus aus Kapitel 3, erzeugt aber eine ausschließlich konfliktfreie Intervallmenge als Lösung. Eine Überprüfung der k-Eigenschaft ist für die reine Konfliktauflösung nicht nötig. Daher kann auf die Funktion Prüfe-k-Eigenschaft() verzichtet werden. Dementsprechend besitzt dieser Algorithmus einen anderen Approximationsfaktor und eine andere Laufzeit. Wir werden das Vorgehen des Algorithmus kurz wiederholen und anschließend die Approximationsgüte und die Laufzeit angeben.

Vorgehen

Der Algorithmus betrachtet die Intervalle der Eingabe Ψ der Größe nach absteigend. In einem Schritt des Algorithmus wird das betrachtete Intervall $[a, b]_l$ in die Lösungsmenge Ψ' verschoben. Zugleich werden die Konfliktpartner $C^l_{[a,b]}$ entfernt, damit diese in einem späteren Schritt nicht mehr betrachtet werden können. Nach Betrachtung aller möglichen Intervalle gibt der Algorithmus die Menge Ψ' als Lösung zurück. Algorithmus 4.1 entspricht diesem Verhalten.

Algorithmus 4.1 : GreedyGeneralMaxTotal(Ψ, C, κ)

Eingabe : Menge von Intervallen Ψ, Menge aller Konflikte C
Ausgabe : Konfliktfreie Intervallmenge Ψ'

```
1 begin
2     Ψ ← sort (Ψ);
3     Ψ' ← ∅;
4     while Ψ ≠ ∅ do
5         [a, b]_l ← größtes Element aus Ψ;
6         Ψ' ← Ψ' ∪ {[a, b]_l};
7         Ψ ← Ψ \ (C^l_[a,b] ∪ {[a, b]_l});
8     return Ψ';
```

Approximationsfaktor und Laufzeit

Der Beweis des Approximationsfaktors des Algorithmus GreedyGeneralMaxTotal() wird ebenfalls in [GNN13] präsentiert. Wir verzichten deshalb darauf den Beweis zu wiederholen und geben nur die Laufzeit und den Approximationsfaktor aus [GNN13, Theorem 4] an.

Korollar 4.1. *Werden Einheitsquadrate als Labels benutzt und für das Gewicht eines Intervalls seine Länge verwendet, besitzt der Algorithmus* GreedyGeneralMaxTotal() *einen Approximationsfaktor von* $^1/_{24}$ *und seine Laufzeit liegt in* $O(|\Psi|\ log|\Psi|)$.

Beschränkung auf einen Konflikt pro Intervall

Wir betrachten nun die Einschränkung der Eingabemenge auf maximal einen Konflikt pro Intervall für den Algorithmus GreedyGeneralMaxTotal(). In diesem Fall besitzt jedes Intervall maximal einen Konfliktpartner. Eine solche Instanz kann mithilfe des Algorithmus GreedyGeneralMaxTotal() optimal gelöst werden. Ein optimaler Algorithmus entscheidet sich für jeden Konflikt für das Intervall mit dem größeren Gewicht. Da der Algorithmus GreedyGeneralMaxTotal() die Intervalle der Größe nach absteigend in die Lösung aufnimmt, ist gewährleistet, dass immer das größere, an einem Konflikt beteiligte Intervall gewählt wird.

4.1.2. Sonderfall: Zwei Konflikte pro Intervall

Im Folgenden beschreiben wir ein dynamisches Programm, welches eine optimalen Lösung liefert, wenn die Eingabemenge maximal zwei Konflikte pro Intervall aufweist. Für die spätere Betrachtung ist es nötig, den *Konfliktgraph einer Intervallmenge* zu definieren.

Definition 4.2. *Sei Ψ eine Intervallmenge mit Konflikten. Dann erhält man den* ***Konfliktgraph von*** Ψ*, indem man für jedes Intervall einen Knoten erstellt und eine Kante zwischen zwei Knoten, deren korrespondierende Intervalle in Konflikt miteinander stehen, einfügt. Zusätzlich erhält jeder Knoten ein Gewicht, das gleich dem Gewicht des entsprechenden Intervalls ist.*

Der Konfliktgraph einer Intervallmenge besteht aus einzelnen Zusammenhangskomponenten. Kommen nur maximal zwei Konflikte pro Intervall in der Intervallmenge vor, dann besteht eine Komponente entweder aus einem Pfad oder einem Zyklus. Abbildung 4.1 veranschaulicht dies anhand eines einfachen Konfliktgraphen.

Da keine Konflikte zwischen zwei Zusammenhangskomponenten bestehen, können diese separat gelöst werden. Die Lösung der gesamten Intervallmenge setzt sich dann aus den Lösungen der Zusammenhangskomponenten zusammen.

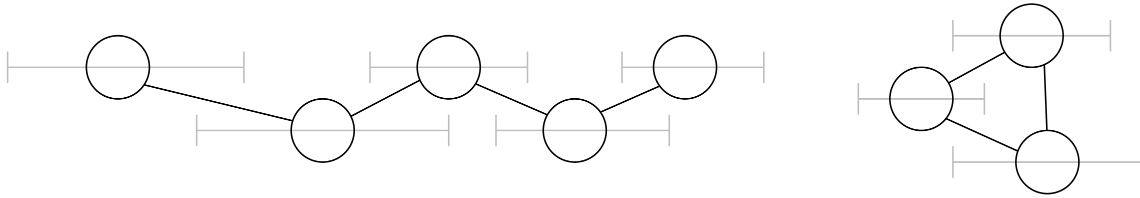


Abbildung 4.1: Beispiel eines Konfliktgraphen einer Intervallmenge, bei der jedes Intervall maximal zwei Konflikte besitzt. Zu sehen sind die beiden möglichen Zusammenhangskomponenten, Pfad und Zyklus

Lösung eines Pfades

Wir beschreiben nun das dynamische Programm LösePfad() zur Lösung eines Pfades des Konfliktgraphen. Sei der Pfad als Sequenz $[v_0, \ldots, v_{t-1}]$ der Knoten gegeben. Der Pfad darf dabei keinen Zyklus bilden. Das dynamische Programm betrachtet jeden Knoten des Pfades der Reihe nach von v_0 beginnend und bestimmt für jeden Knoten v_i den Distanz-Wert $\textsc{Dist}[i]$. Der Distanz-Wert von v_i gibt den Wert der optimalen Lösung aller bis v_i betrachteten Knoten an. Bei der Betrachtung eines Knotens muss der Algorithmus entscheiden, ob dieser zur optimalen Lösung der, bis zu diesem Zeitpunkt betrachteten Knoten hinzu zählt oder nicht. Diese Entscheidung stützt sich dabei auf das Gewicht des Knotens und die Werte $\textsc{Dist}[i-1]$ und $\textsc{Dist}[i-2]$ der beiden letzten Knoten. Wir betrachten zuerst die Berechnung des Wertes $\textsc{Dist}[i]$ eines beliebigen Knotens v_i. Das korrespondierende Intervall des Knoten v_i bezeichnen wir dann mit $[a, b]_l$.

Der Algorithmus muss entscheiden, ob der Knoten v_i hinzu genommen wird oder nicht. Diese Entscheidung basiert darauf, ob der Wert der bisherigen Lösung mit dem Knoten besser ist als ohne ihn. Dazu werden die beiden möglichen Werte berechnet und miteinander verglichen. Wird Knoten v_i hinzugenommen, darf der direkt vorhergehende Knoten nicht in der Lösung vorkommen. Der Wert $\textsc{Dist}[i]$ würde sich also aus der Summe des Wertes $\textsc{Dist}[i-2]$ und des Gewichts des Intervalls $[a, b]_l$ berechnen. Falls hingegen Knoten v_i nicht aufgenommen wird, würde der $\textsc{Dist}[i]$-Wert nur aus dem $\textsc{Dist}[i-1]$-Wert bestehen. Der Algorithmus berechnet also $\textsc{Dist}[i] = \max\{\textsc{Dist}[i-2] + w([a, b]_l), \textsc{Dist}[i-1]\}$. Es wird ebenfalls eine Markierung gespeichert, welche angibt, ob Knoten v_i für diesen Wert $\textsc{Dist}[i]$ in die Lösung aufgenommen wird oder nicht. Für die ersten beiden Knoten stehen die nötigen Werte $\textsc{Dist}[-1]$ und $\textsc{Dist}[-2]$ noch nicht zur Verfügung. Wir setzen deshalb $\textsc{Dist}[-1] = \textsc{Dist}[-2] = 0$.

Nachdem für jeden Knoten der entsprechende Distanz-Wert berechnet wurde, gibt der Wert $\textsc{Dist}[t-1]$ des letzten Knotens im Pfad den Wert der optimalen Lösung für den gesamten Pfad an. Durch eine weitere Betrachtung aller Knoten kann dann die Intervallmenge bestimmt werden. Hierzu wird beim letzten Knoten begonnen. Der Index bezeichnet wieder die Position des Knotens im Pfad. Wir betrachten einen beliebigen Knoten v_i. Die vorher erstellte Markierung gibt nun an, ob das korrespondierende Intervall $[a, b]_l$ des Knotens in der Lösung enthalten ist. Das Intervall wird dementsprechend in die Lösung aufgenommen oder verworfen. Der nächste zu betrachtende Knoten ergibt sich anhand der Markierung des Knotens. Gibt die Markierung an, dass der Knoten v_i in der Lösung enthalten ist, so ist der als nächstes zu betrachtende Knoten v_{i-1}. Andernfalls wird Knoten v_{i-2} als nächstes betrachtet. Der Algorithmus nimmt so sukzessive die entsprechenden Intervalle, die zur optimalen Lösung führen, in die Lösung auf. Beendet wird der Algorithmus sobald der Knoten v_{-1} oder v_{-2} betrachtet werden soll. In Algorithmus 4.2 ist das Verhalten der Funktion LösePfad() veranschaulicht.

Im ersten Teil wird der $\textsc{Dist}[i]$-Wert für alle Knoten v_i berechnet, dazu werden alle Knoten des Pfades der Reihe nach betrachtet. Anschließend werden die Intervalle der

Algorithmus 4.2 : LÖSEPFAD($[v_0, \ldots, v_{t-1}]$)

```
Eingabe : Pfad [v_0, ..., v_{t-1}]
Ausgabe : Entsprechende Intervallmenge Ψ' mit aufgelösten Konflikten
1  begin
2      dist[−2] ← 0;
3      dist[−1] ← 0;
4      Ψ' ← ∅;
5      for i ← 0; i < t; i++ do
6          [a, b]_l ← entsprechendes Intervall des Knotens v_i;
7          dist[i] = max{dist[i − 2] + w([a, b]_l), dist[i − 1]};
8          if dist[i − 2] + w([a, b]_l) > dist[i − 1] then
9              Markiere Knoten v_i als gewählt;

10     i ← t − 1;
11     while i > 0 do
12         if Knoten v_i ist als „gewählt“ markiert then
13             [a, b]_l ← entsprechendes Intervall des Knotens v_i;
14             Ψ' ← Ψ' ∪ {[a, b]_l};
15             i ← i − 2;

16         else
17             i −−;

18     return Ψ';
```

gewählten Knoten in die Lösung aufgenommen. Damit wird jeder Knoten höchstens zwei mal betrachtet. Es ergibt sich demnach folgende Laufzeit für die Funktion LÖSEPFAD().

Beobachtung 4.3. *Die Laufzeit der Funktion* LÖSEPFAD() *liegt in $O(t)$. Hierbei ist t die Anzahl der entsprechenden Intervalle des Konfliktgraphs.*

Lösung eines Zyklus

Die andere Zusammenhangskomponente des hier besprochenen Konfliktgraphen ist ein Zyklus. Dabei bilden die Knoten dieses Teilgraphs einen Pfad, bei dem Start- und Endknoten gleich sind. Wir können einen Zyklus mithilfe der Funktion LÖSEPFAD() lösen. Aufgrund der Konflikte kann ein Knoten niemals gleichzeitig mit seinen Nachbarn in der Lösung auftauchen. Deshalb entfernt der Algorithmus einen beliebigen Knoten x aus dem Teilgraph, wodurch der Zyklus aufgetrennt wird. Es ist nun ein Pfad entstanden, der mit der Funktion LÖSEPFAD() gelöst werden kann. Die zurückgelieferte Lösung bezeichnen wir mit Ψ'_1. Da aber x nicht mehr im Konfliktgraph vorhanden ist, kommt dessen Präsenzintervall in Ψ'_1 nicht vor. Deshalb wird der ursprüngliche Zyklus ein zweites Mal aufgetrennt, indem nun einer der beiden Nachbarn von x gelöscht wird. Die Funktion LÖSEPFAD() liefert wieder eine optimale Lösung Ψ'_2 für den entstandenen Pfad. Nun vergleicht der Algorithmus den Wert von Ψ'_1 mit dem von Ψ'_2. Die Lösung mit dem größeren Wert wird dann gewählt.

Zum Lösen eines Zyklus muss also die Funktion LÖSEPFAD() zweimal aufgerufen werden. Die Laufzeit für die Funktion LÖSEPFAD() ist in Beobachtung 4.3 angegeben. Unter der Annahme, dass das Entfernen eines Knotens und der Vergleich der Lösungen jeweils konstante Laufzeiten besitzen, lässt sich dann folgendes beobachten.

Beobachtung 4.4. *Die Laufzeit zum Lösen eines Zyklus mithilfe der Funktion* LÖSEPFAD() *liegt in $O(t)$. Hierbei ist t die Anzahl der Knoten des Zyklus.*

Identifizierung der Zusammenhangskomponenten

Der Konfliktgraph kann aus mehreren Zusammenhangskomponenten bestehen. Um den richtigen Lösungsweg auf einen Teilgraphen anzuwenden, muss der Algorithmus die Art des Teilgraphen bestimmen. Ein einfaches Vorgehen ist eine Tiefensuche, bei der jeder, schon besuchte Knoten markiert wird.

Zu Beginn wird ein beliebiger nicht markierter Knoten aus dem Konfliktgraph betrachtet. Nun werden mittels Tiefensuche, alle erreichbare Knoten identifiziert und ebenfalls markiert. Trifft der Algorithmus auf einen schon markierten Knoten, kann es sich nur um einen Zyklus handeln. Dieser wird dann mit obigem Vorgehen gelöst. Werden hingegen nur Knoten gefunden, die noch nicht markiert sind, handelt es sich um einen Pfad. Es wird dann die Funktion LösePfad() direkt darauf angewendet. Anschließend wird wieder von vorne begonnen und ein Knoten betrachtet, der noch nicht markiert ist. Nach und nach werden so alle Knoten des Konfliktgraphen markiert.

Damit ein Knoten als Ausgangsknoten für die Tiefensuche gewählt werden kann, darf dieser nicht markiert sein. Um zu verhindern, dass ein markierter Knoten beliebig oft betrachtet wird, werden alle Knoten der Reihe nach als Ausgangsknoten für die Tiefensuche betrachtet. Ist ein Knoten schon markiert, wird der nächste betrachtet.

Lemma 4.5. *Die Identifizierung der Zusammenhangskomponenten eines Graphen, dessen Knoten über maximal zwei Kanten verfügen, besitzt lineare Laufzeit.*

Beweis. Mit allen Tiefensuche-Vorgängen wird jeder Knoten des Graphen genau einmal betrachtet und markiert. Zum Auswählen der Knoten für die Tiefensuche müssen alle Knoten der Reihe nach betrachtet und ihre Markierung überprüft werden. Damit besitzt die Identifizierung lineare Laufzeit. □

Gesamtlaufzeit

Einerseits hängt die Gesamtlaufzeit von der Identifizierung der Teilgraphen und andererseits von der Berechnung der Lösung ab. Zum Lösen der Teilgraphen wird die Funktion LösePfad() verwendet. Mit der Laufzeit für die Identifizierung der Teilgraphen (Lemma 4.5) und der Laufzeit zur Lösung eines Pfades und eines Zyklus (Beobachtung 4.3 und Beobachtung 4.4) können wir nun die Laufzeit des gesamten linearen Programms angeben.

Korollar 4.6. *Die Laufzeit des hier beschriebenen linearen Programms liegt in $O(|\Psi|)$. Dabei ist $|\Psi|$ die Anzahl der zu betrachtenden Präsenzintervalle.*

4.2. Begrenzung der Anzahl gleichzeitig sichtbarer Beschriftungen

Damit eine Menge Ψ die k-Eigenschaft erfüllt, dürfen sich in Ψ zu jedem Zeitpunkt nicht mehr als k Intervalle schneiden. In diesem Abschnitt stellen wir zuerst das Verfahren aus [CL95] vor, welches die k-Eigenschaft mithilfe eines Flussnetzwerks herstellt. Um mehrere Verfahren für die experimentelle Evaluation (siehe Kapitel 5) zur Auswahl zu haben, zeigen wir anschließend einen Algorithmus, der dem Vorgehen aus Kapitel 3 folgt. Danach werden wir einen weiteren approximativen Algorithmus vorstellen, der mithilfe eines Sweepline-Vorgehens arbeitet.

4.2.1. Flussalgorithmus für die k-Eigenschaft

Flussnetzwerke werden in der Informatik häufig zur Lösung von Problemen benutzt. Dabei betrachtet man einen Graph, über dessen Kanten ein Fluss von einer Quelle zu einer Senke fließen soll. In [CLRS09] wird ein Flussnetzwerk durch einen gerichteten Graph $G = (V, E)$ mit zwei ausgezeichneten Knoten $s, t \in V$ und einer Kostenfunktion $c \colon E \to \mathbb{R}_0^+$ beschrieben. Der Knoten s wird als Quelle, der Knoten t als Senke bezeichnet. Ein Fluss auf G ist definiert als die Funktion $f \colon V \times V \to \mathbb{R}$, welche die beiden folgenden Bedingungen erfüllt.

Kapazitätsbedingung: $0 \leq f(u,v) \leq c(u,v),\ \forall u,v \in V$

Flusserhaltung: $\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v),\ \forall u \in V \setminus \{s,t\}$

Das bedeutet anschaulich, dass der Fluss über eine Kante nie größer sein darf als dessen Kapazität und, dass der in einen Knoten fließende Fluss genauso groß sein muss wie der von ihm weg fließende Fluss. Der Wert eines Flusses f ist $|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$. Häufig ist man an einem maximalen Fluss interessiert, das bedeutet, dass ein Fluss f gefunden werden soll, dessen Wert unter allen möglichen Flüssen maximal ist. Des Weiteren kann man eine Kostenfunktion $a\colon E \to \mathbb{R}$ einführen, die jeder Kante Kosten zuweist. Die Gesamtkosten für einen Fluss f betragen dann $\sum_{(u,v) \in E} a(u,v) f(u,v)$. Für gewöhnlich soll ein maximaler Fluss gefunden werden, dessen implizierte Gesamtkosten minimal sind.

In [CL95] wird ein Verfahren beschrieben, das eine Teilmenge einer Menge von Präsenzintervallen berechnet, deren Gewicht unter allen Teilmengen am größten ist und welche die k-Eigenschaft erfüllt. Diese Teilmenge nennen wir hier die *optimale Lösung, welche die k-Eigenschaft erfüllt*. Dieses Verfahren transformiert die Menge der Intervalle in ein Flussnetzwerk und findet anschließend einen maximalen Fluss mit minimalen Kosten. Die optimale Lösung ergibt sich dann anhand des Flusses. Im Folgenden wollen wir die Transformation und die Bestimmung der Lösung beschreiben, da wir in Kapitel 5 den Algorithmus experimentell auswerten.

Transformation

Um einen Flussalgorithmus anwenden zu können, muss die Intervalmenge Ψ zuerst in ein Flussnetzwerk transformiert werden. Wir werden nun die Transformation aus [CL95] beschreiben. Dazu werden, für jedes Intervall $[a,b]_l$, zwei Knoten a und b erstellt. Diese Knoten repräsentieren den Start- und Endzeitpunkt von $[a,b]_l \in \Psi$. Zusätzlich werden Knoten für die Quelle s und die Senke t hinzugefügt. Insgesamt besitzt damit das Flussnetzwerk $|\Psi| \cdot 2 + 2$ Knoten. Anschließend werden alle Knoten in zeitlicher Reihenfolge mit gerichteten Kanten verbunden. Dabei bildet die Quelle den Anfang und die Senke das Ende. Existieren für einen Zeitpunkt mehrere Knoten, werden diese so sortiert, dass sich die Endknoten vor den Startknoten befinden. Andernfalls könnten Startpunkte übersprungen und damit Intervalle ausgeschlossen werden. Jede dieser Kanten erhält Kapazität k und erzeugt keine Kosten. Der Benennung aus [CL95] folgend, bezeichnen wir diese Kanten als Clique-Kanten. Nun erhält jeder Knoten, der den Startzeitpunkt eines Intervalls repräsentiert, eine Kante zu dem entsprechenden Endknoten des Intervalls. Wir bezeichnen diese Kanten als Intervall-Kanten. Die Kanten besitzen jeweils eine Kapazität von 1 und erzeugen Kosten, die dem negativen Gewicht $-w([a,b]_l)$ des korrespondierenden Intervalls entsprechen. In Abbildung 4.2 ist diese Transformation veranschaulicht.

Lösung

Nun wird ein maximaler Fluss mit minimalen Kosten auf diesem Flussnetzwerk berechnet. Die Clique-Kanten bilden einen Pfad von Knoten s nach Knoten t mit Kapazität k. Knoten s hat zudem nur diese eine ausgehende Clique-Kante. Das führt zur folgenden Beobachtung.

Beobachtung 4.7. *Der maximale Fluss des so konstruierten Flussnetzwerks hat den Wert k.*

Die Kosten des Flusses der Größe k sind, laut [CL95, Lemma 4], gleich des negativen Wertes der optimalen Lösung. Es muss also der kostenminimale Fluss der Größe k, welcher nach Beobachtung 4.7 der maximale Fluss ist, berechnet werden. Dann lässt sich anhand des Flusses auf die Intervalle der Lösung schließen. Nach [CL95, Theorem 3] bilden die korrespondierenden Intervalle, der saturierten Intervallkanten eine optimale Lösung, welche die k-Eigenschaft erfüllt.

Die Berechnung des maximalen Flusses mit minimalen Kosten kann natürlich mittels eines Flussalgorithmus berechnet werden. Da aber das verwendete Flussnetzwerk ausschließlich

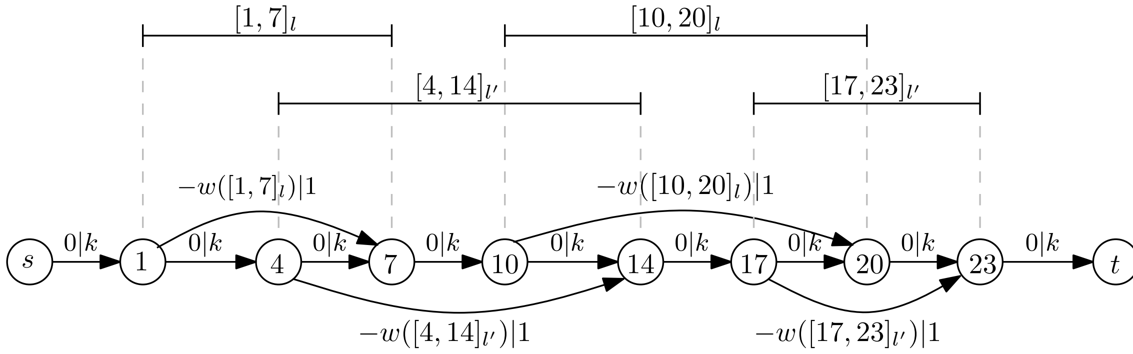


Abbildung 4.2: Veranschaulichung der Transformation einer Intervallmenge in das entsprechende Flussnetzwerk zur Bestimmung einer optimalen Teilmenge, auf der die k-Eigenschaft erfüllt ist. Die Kantenbeschriftungen folgen dem Schema Kosten | Kapazität

Vorwärtskanten besitzt, ist eine einfache Betrachtung der kürzesten Wege möglich. Die Laufzeit, um einen kostenminimalen Fluss der Größe k zu finden, beschränkt sich damit auf $O(k \cdot S(n))$ ([CL95, Theorem 3]). Hierbei ist $S(n)$ die Laufzeit eines Algorithmus zur Berechnung eines kürzesten Pfades.

Kürzester Pfad

Mit dem Algorithmus von Bellman und Ford kann man einen kürzesten Pfad in $O(n \cdot m)$ berechnen (siehe [CLRS09]). Der Algorithmus von Dijkstra besitzt eine Laufzeit in $O(n \cdot \log n + m)$ (siehe [CLRS09]), wobei die Kantengewichte nicht negativ sein dürfen. Wir stellen nun ein Verfahren vor, mit dem der kürzeste Pfad, auf einem solchen Flussnetzwerk in linearer Laufzeit berechnet werden kann. Wir benennen einen Knoten mit dem entsprechenden Index in der aufsteigend sortierten Liste aller Start- und Endzeitpunkte, beginnend bei 1. Dabei sei n die Anzahl aller Zeitpunkte. Der Knoten s bekommt den Index 0 und der Knoten t entsprechend den Index $n + 1$. Jeder Knoten i bekommt zusätzlich einen Distanz-Wert $\textsc{Dist}[i]$, der die Länge bzw. das Gewicht des kürzesten Pfades vom Startknoten 0 bis zum diesem Knoten i enthält. Der Wert $\textsc{Dist}[0]$ wird mit 0 initialisiert.

Nun werden alle Knoten aufsteigend betrachtet und die jeweilige Distanz berechnet. Wir betrachten dies nun für einen beliebigen Knoten $i > 0$. Der Knoten i besitzt immer eine eingehende Clique-Kante mit Kosten 0. Repräsentiert der Knoten einen Endzeitpunkt eines Intervalls $[a, b]_l$, besitzt er zusätzlich noch eine eingehende Intervall-Kante mit Kosten $-w([a, b]_l)$. Falls eine solche Intervall-Kante existiert, benennen wir den entsprechenden Startknoten des Intervalls mit h. Es gilt immer $h < i$. Anschließend wird $\textsc{Dist}[i] = \min\{\textsc{Dist}[i-1], \textsc{Dist}[h] - w([a, b]_l)\}$ gesetzt. So wird für jeden Knoten nacheinander vorgegangen. Der Wert $\textsc{Dist}[n{+}1]$ enthält die Kosten des kürzesten Pfades im Flussnetzwerk von der Quelle zur Senke. Werden nun die Knoten von $n + 1$ entgegen der Kantenrichtung durchlaufen, können die Intervall-Kanten bestimmt werden, welche den kürzesten Pfad bilden. Dabei muss für jeden Knoten die Kante gewählt werden, die zu dem entsprechenden Distanz-Wert beigetragen hat.

Insgesamt werden mit diesem Verfahren alle Knoten maximal zweimal betrachtet. Wie oben erwähnt gilt $n \in O(|\Psi|)$. Die Laufzeit um einen kürzesten Pfad auf dem oben vorgestellten Flussnetzwerk zu finden, liegt somit in $O(|\Psi|)$. Insgesamt ergibt sich, nach [CL95, Lemma 4], für die Berechnung einer Teilmenge die folgende Laufzeit.

Korollar 4.8. *Eine Teilmenge einer Menge von Präsenzintervallen, welche die k-Eigenschaft erfüllt, kann in $O(k \cdot n)$ berechnet werden.*

4.2.2. Approximativer Greedy-Algorithmus für die k-Eigenschaft

Der Algorithmus aus Kapitel 3 kann so abgewandelt werden, dass er eine Teilmenge der Eingabe findet, die nur die k-Eigenschaft erfüllt. Die Konflikte werden dabei nicht berücksichtigt und kommen, sofern vorher vorhanden, auch in der Lösungsmenge vor.

Vorgehen

Ähnlich zum Algorithmus aus Kapitel 3, werden die Intervalle der Eingabemenge Ψ der Größe nach absteigend betrachtet. Sei nun $[a,b]_l$ das in Schritt i betrachtete Intervall des Algorithmus. Dann wird geprüft, ob das Intervall $[a,b]_l$ der Lösungsmenge Ψ' hinzugefügt werden kann. Die Überprüfung geschieht wieder mit der Funktion Prüfe-k-Eigenschaft() (siehe Abschnitt 3.4). Die Funktion prüft, ob das Einfügen von $[a,b]_l$ in die Lösungsmenge Ψ' die k-Eigenschaft verletzt. Ist dies nicht der Fall, kann $[a,b]_l$ eingefügt werden. Andernfalls wird $[a,b]_l$ verworfen. Nachdem alle Intervalle bearbeitet wurden, wird die Menge Ψ' als Lösung zurückgegeben. Dieses Verhalten ist in Algorithmus 4.3 zu sehen. Dabei findet die Funktion Prüfe-k-EigenschaftNaiv() (siehe Algorithmus 3.2) Anwendung. Eine Verwendung der Varianten (siehe Abschnitt 3.4) ist nach entsprechenden Anpassungen ebenfalls möglich.

Algorithmus 4.3 : Greedy-k-Eigenschaft(Ψ, k)

Eingabe : Menge von Intervallen Ψ, Parameter k
Ausgabe : Intervallmenge Ψ', die keinen Widerspruch zur k-Eigenschaft besitzt

```
1 begin
2     Ψ ← sort (Ψ);
3     Ψ' ← ∅;
4     while Ψ ≠ ∅ do
5         [a,b]_l ← größtes Element aus Ψ;
6         if !prüfeKEigenschaft(Ψ', [a,b]_l, k) then
7             Ψ' ← Ψ' ∪ {[a,b]_l};
8         Ψ ← Ψ \ {[a,b]_l}
9     return Ψ';
```

Approximation

Der Beweis für die Approximationsgüte dieses Algorithmus verläuft ähnlich, wie der des Algorithmus Greedy-k-RestrictedMaxTotal() aus Kapitel 3. Da die Intervalle der Größe nach absteigend betrachtet werden, findet auch hier der Einflussbereich eines Intervalls (siehe Definition 3.1) Anwendung. Damit können wir den Approximationsfaktor beweisen.

Theorem 4.9. *Der Algorithmus* Greedy-k-Eigenschaft() *besitzt eine Approximationsgüte von* $^1/_3$.

Beweis. Im Folgenden werden wir die Menge Ψ im i-ten Schritt wieder mit Ψ_i bezeichnen. Sei $[a,b]_l$ das im i-ten Schritt betrachtete Intervall. Jedes Intervall aus Ψ_i ist kleiner oder gleich groß wie $[a,b]_l$. Wenn $[a,b]_l$ in die Lösung eingefügt wird, können andere Intervalle nicht mehr hinzugefügt werden. Liegt $[a,b]_l$ nicht in der optimalen Lösung und die Intervalle, die aufgrund von $[a,b]_l$ nicht mehr eingefügt werden können, hingegen schon, weicht der Wert der Lösung von dem einer optimalen Lösung ab.

Nach Lemma 3.3 können die Intervalle der optimale Lösung in k Mengen angeordnet werden, ohne dass sich zwei Intervalle einer Menge überschneiden. Wird $[a,b]_l$ in die Lösung eingefügt, können bestimmte Intervalle nicht mehr hinzugefügt werden. Diese Intervalle können aber in einer optimalen Lösung liegen. Eine Verdrängung geschieht nur dann, wenn

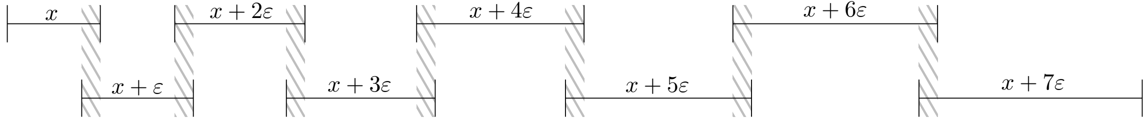


Abbildung 4.3: Beispiel für eine Instanz, für welche der Algorithmus SWEEPLINE-k-EIGENSCHAFT() eine sehr schlechte Lösung für $k = 1$ berechnen würde

sich ein Intervall mit $[a, b]_l$ überschneidet. Dabei müssen wir nur die Intervalle beachten, welche kürzer als $[a, b]_l$ sind. Die Kosten für die längeren Intervalle wurden entweder schon vorher berechnet oder diese Intervalle wurden in die Lösung eingefügt. Wir können also wieder den Einflussbereich von $[a, b]_l$ auf Ψ_i verwenden (siehe Lemma 3.2). Dessen Länge beträgt $3 \cdot w([a, b]_l)$.

Im schlimmsten Fall können Intervalle nicht mehr eingefügt werden, die den kompletten Einflussbereich füllen. Diese Intervalle besitzen dann ein Gesamtgewicht von $3 \cdot w([a, b]_l)$, welches der Länge des Einflussbereichs entspricht. Vom Algorithmus wird aber das Intervall $[a, b]_l$ hinzugefügt. Die optimale Lösung besitzt im schlimmsten Fall jedoch die Intervalle, die aufgrund von $[a, b]_l$ nicht mehr eingefügt werden können, da deren Gesamtgewicht größer ist als $w([a, b]_l)$. Wird für jeden Schritt der schlimmste Fall angenommen, ist der Approximationsfaktor für diesen Algorithmus $1/3$. □

Laufzeit

Im Vergleich zu Algorithmus GREEDY-k-RESTRICTEDMAXTOTAL() (siehe Kapitel 3) werden in einem Schritt des Algorithmus GREEDY-k-EIGENSCHAFT() die Konfliktpartner nicht gelöscht. Damit besitzt dieser Algorithmus eine andere Laufzeit. Die Laufzeit unterscheidet sich aber lediglich um den Teil der Konfliktauflösung.

Korollar 4.10. *Die worst-case Laufzeit des Algorithmus* GREEDY-k-EIGENSCHAFT() *liegt in* $O(|\Psi| \cdot \log |\Psi| + |\Psi| \cdot T(\text{PRÜFE}-k-\text{EIGENSCHAFT}()))$. *Hierbei sei* $T(\text{PRÜFE}-k-\text{EIGENSCHAFT}())$ *die Laufzeit der Funktion* PRÜFE-k-EIGENSCHAFT().

Unter Verwendung der verschiedenen Varianten der Funktion PRÜFE-k-EIGENSCHAFT() (siehe Abschnitt 3.4) können dann unterschiedliche Laufzeiten erzielt werden.

4.2.3. Heuristischer Greedy-Algorithmus für die k-Eigenschaft

Nun wollen wir einen weiteren Algorithmus zur Herstellung der k-Eigenschaft vorstellen. Der Algorithmus arbeitet nach dem Sweepline-Prinzip und erkennt so die Bereiche, in denen die k-Eigenschaft verletzt ist. Gleichzeitig werden entsprechende Intervalle entfernt, um die k-Eigenschaft wiederherzustellen. Der Algorithmus kann beliebig schlecht werden.

Vorgehen

Der Algorithmus SWEEPLINE-k-EIGENSCHAFT() betrachtet die Start- und Endzeitpunkte jedes Intervalls der Eingabemenge Ψ in aufsteigender Reihenfolge. Hier sei wieder auf das Lemma 3.6 verwiesen, welches besagt, dass sich die Anzahl der Intervalle zwischen aufeinanderfolgenden Zeitpunkten nicht ändert. Zusätzlich führt der Algorithmus eine Datenstruktur der Intervalle, welche von der Sweepline geschnitten werden. Trifft der Algorithmus auf einen Bereich, der die k-Eigenschaft verletzt, wird das kleinste Intervall, das diesen Bereich schneidet, aus Ψ entfernt. In Algorithmus 4.4 ist der Algorithmus SWEEPLINE-k-EIGENSCHAFT() skizziert.

Approximation

Wir zeigen, dass der Algorithmus SWEEPLINE-k-EIGENSCHAFT() eine beliebig schlechte Lösung produzieren kann. Zuerst konstruieren wir ein Gegenbeispiel für $k = 1$. Wir konstruieren eine Instanz, bei der sich alle Intervalle bis auf das linkeste und das rechteste mit je zwei anderen Intervallen schneiden. Wir setzen die Startzeitpunkte der Intervalle

Algorithmus 4.4 : SWEEPLINE-k-EIGENSCHAFT(Ψ, k)

Eingabe : Menge von Intervallen Ψ, Parameter k
Ausgabe : Intervallmenge Ψ', die keinen Widerspruch zur k-Eigenschaft besitzt

```
1  begin
2      T ← Start- und Endzeitpunkte der Intervalle aus Ψ;
3      T ← sort(T);
4      Φ ← ∅;
5      for t ∈ T do
6          Π ← {zu t entsprechende Intervalle} ∩ Ψ;
7          if t ist Startzeitpunkt then
8              Φ ← Φ ∪ Π;
9              if |Φ| > k then
10                 [a, b]_l ← kleinstes Intervall aus Φ;
11                 Ψ ← Ψ \ {[a, b]_l};
12                 Φ ← Φ \ {[a, b]_l};
13                 Entferne die zu [a, b]_l entsprechenden Zeitpunkte aus T;
14         else
15             Φ ← Φ \ Π;
16             Ψ' ← Ψ' ∪ Π;
17     return Ψ';
```

so, dass sich zwei aufeinanderfolgende Intervalle zeitlich überschneiden. Das Gewicht jedes Intervalls wählen wir dann so, dass der Wert um ε größer ist als das, des davor beginnenden Intervalls. In Abbildung 4.3 ist dies veranschaulicht. Eine Lösung für so eine Instanz, die mit dem Algorithmus SWEEPLINE-k-EIGENSCHAFT() berechnet wurde, enthält *nur* das letzte Intervall. Der Algorithmus beginnt beim frühesten Startzeitpunkt und betrachtet die folgenden Start- bzw Endzeitpunkte aufsteigend. Betrachtet der Algorithmus den Startzeitpunkt des i-ten Intervalls, mit $i > 1$, ist sofort die k-Eigenschaft verletzt, da mehr als ein Intervall an diesem Zeitpunkt existiert. Zu diesem Zeitpunkt ist das Intervall i und $i-1$ aktiv. Das Intervall i besitzt ein größeres Gewicht, weshalb das Intervall $i-1$ gelöscht wird. Dies geschieht nacheinander für alle Intervalle. Wird der Startzeitpunkt jedes Intervalls zusätzlich so spät wie möglich, innerhalb des vorhergehenden Intervalls gewählt, enthält eine optimalen Lösung jedes zweite Intervall. Dieses Gegenbeispiel können wir für einen allgemeinen Parameter k erweitern, indem wir das oben beschriebene Konstrukt k-mal untereinander platzieren. In der Praxis können Instanzen mit ähnlichen Eigenschaften vorkommen, indem zum Beispiel durch Drehung des Viewports die Labels eine längere Präsenzzeit besitzen als die direkt vorhergehenden.

Laufzeit

Der Algorithmus betrachtet alle Start- und Endzeiten der Intervalle aus Ψ. Die Anzahl der Zeitpunkte ist linear zur Anzahl der Intervalle (siehe Beobachtung 3.7). Es muss jedes Intervall in die Datenstruktur der gerade präsenten Intervalle eingefügt und später wieder gelöscht werden. Damit hängt die Laufzeit von der verwendeten Datenstruktur ab. Wird ein balancierter binärer Baum (Laufzeiten siehe [CLRS09]) zum Verwalten der gerade präsenten Intervalle benutzt, so können wir die Laufzeit wie folgt angeben.

Beobachtung 4.11. *Werden für die Datenstruktur der aktuell präsenten Intervalle balancierte binäre Bäume verwendet, liegt die Laufzeit des Algorithmus* SWEEPLINE-k-EIGENSCHAFT() *in* $O(|\Psi| \cdot \log |\Psi|)$.

4.3. Approximationsfaktor zweier Teilalgorithmen

Um das Problem k-RestrictedMaxTotal zu lösen, müssen beide Teilalgorithmen nacheinander angewandt werden. Die Reihenfolge spielt dabei für die Gültigkeit einer Lösung keine Rolle. Wie wir jedoch in Kapitel 5 sehen werden hängt die Güte deutlich von der Reihenfolge ab. Zudem werden wir hier nur die beiden nicht-optimalen Algorithmen GreedyGeneralMaxTotal() (siehe Abschnitt 4.1.1) und Greedy-k-Eigenschaft() (siehe Abschnitt 4.2.2) besprechen. Der Approximationsfaktor für einen zusammengesetzten Algorithmus, bei dem einer oder beide Teilalgorithmen eine optimale Lösung erzeugen, wird in dieser Arbeit nicht besprochen. Im Allgemeinen gibt die Ausführung optimaler Teilalgorithmen keine Garantie für einen optimalen Gesamtalgorithmus. Auch hier betrachten wir wieder Einheitsquadrate als Labels und setzen $w([a, b]_l) = b - a$. Zuerst untersuchen wir den Approximationsfaktor für die Reihenfolge, bei der zu Beginn der Teilalgorithmus GreedyGeneralMaxTotal() und anschließend Greedy-k-Eigenschaft() verwendet werden

Zur Berechnung des Approximationsfaktors werden wir, genauso wie oben, Kosten für jedes Präsenzintervall berechnen. Die Kosten für ein Intervall $[a, b]_l$ setzen sich aus den Gewichten der Intervalle zusammen, die gelöscht werden mussten, damit $[a, b]_l$ in die Lösung aufgenommen werden konnte. Wir betrachten die Kosten für den zweiten Teilalgorithmus Greedy-k-Eigenschaft(). Diese Kosten geben dann den worst-case Fall für den Approximationsfaktor des Gesamtalgorithmus an. Wie wir im Beweis von Theorem 4.9 beschrieben haben, entstehen im schlimmsten Fall für das Intervall $[a, b]_l$ Kosten von $3 \cdot w([a, b]_l)$. Diese Kosten entstehen dadurch, dass Intervalle mit dem Gesamtgewicht $3 \cdot w([a, b]_l)$ nicht mehr in die Lösung eingefügt werden können, da $[a, b]_l$ gewählt wurde. Jedes dieser verdrängten Intervalle wurde jedoch vorher im Algorithmus GreedyGeneralMaxTotal() ausgewählt. Dabei wurden andere Intervalle entfernt, welche nun nicht mehr betrachtet werden können. Deshalb müssen diese Kosten für jedes, im zweiten Schritt, verdrängte Intervall mit einkalkuliert werden. Die Kosten für ein Intervall $[c, d]_{l'}$, welches im ersten Teilalgorithmus GreedyGeneralMaxTotal() gewählt wurde, betragen $24 \cdot w([c, d]_{l'})$ (siehe Korollar 4.1). Insgesamt berechnen sich die Kosten eines, im zweiten Schritt gewählten Intervalls $[a, b]_l$ zu $3 \cdot (24 \cdot w([a, b]_l))$. Nehmen wir nun für jeden Schritt den schlimmsten Fall an, ergibt sich der Approximationsfaktor.

Korollar 4.12. *Wird das Problem k-RestrictedMaxTotal mithilfe der separaten Algorithmen GreedyGeneralMaxTotal() und anschließend Greedy-k-Eigenschaft() gelöst, ergibt sich ein Approximationsfaktor von $1/72$, wenn die Länge eines Intervalls als sein Gewicht und Einheitsquadrate als Labels verwendet werden.*

Die Argumentation von oben lässt sich auf die umgekehrte Reihenfolge der beiden Teilalgorithmen anwenden. Es ergibt sich der gleiche Approximationsfaktor wenn zuerst Greedy-k-Eigenschaft() und anschließend GreedyGeneralMaxTotal() benutzt werden.

5. Experimentelle Evaluation

In diesem Kapitel werden wir einige Algorithmen aus Kapitel 3 und 4 experimentell evaluieren. Wir vergleichen dabei die Ergebnisse der hier vorgestellten Algorithmen mit denen einer optimalen Lösung. Insgesamt liegt der Fokus dieses Kapitels auf der Evaluation der Güte und der Laufzeit der Algorithmen. Für die Experimente haben wir eine beschriftete Straßenkarte der Karlsruher Innenstadt aus OpenStreetMap-Daten erzeugt. Wir haben für diese Karte eine Vergrößerung von 1 : 2000 gewählt. Der Ausschnitt, den der Viewport zeigt, besitzt eine Ausdehnung in der Realität von $339m \times 254m$. Insgesamt wurden 1000 zufällige Trajektorien und die dazugehörigen Präsenzintervalle bearbeitet. Eine Trajektorie wurde aus dem kürzesten Weg zwischen zwei zufälligen Punkten auf der Karte erstellt. Dabei wurden Start- und Zielpunkt gleichverteilt zufällig gewählt. Verschiedene Messwerte, insbesondere die Laufzeit und der Wert der erzeugten Lösung, geben darüber Aufschluss, wie sich die Algorithmen unter realen Bedingungen verhalten.

Das verwendete Testsystem besitzt 16 Prozessoren vom Typ Intel Xeon E5-2670 mit 2,60 GHz. Es standen 64 GB Arbeitsspeicher zur Verfügung. Die Algorithmen wurden in Java implementiert und mit OpenJDK in der Version 1.7.0_21 ausgeführt. Es wurden keine Parallelisierungstechniken benutzt, weshalb die Algorithmen auf einem einzelnen Prozessor liefen.

Im Zuge der Arbeit [GNN13] ist eine Implementation entstanden, welche das dort vorgestellte ganzzahlige lineare Programm umsetzt und eine optimale Lösung des Problems k-RestrictedMaxTotal berechnet. Um die Ergebnisse miteinander vergleichen zu können, geschah die Berechnung auf dem selben Testsystem. Die Implementation der Algorithmen aus Kapitel 3 und 4 nutzte eine textuelle Beschreibung der Präsenzintervalle einer Trajektorie und bearbeitete diese dann entsprechend. Aus Kapitel 3 wurde der Algorithmus Greedy-k-RestrictedMaxTotal() in verschiedenen Varianten getestet. Insgesamt wurden fünf Varianten dieses Algorithmus betrachtet. Die Varianten unterscheiden sich hauptsächlich in der verwendeten Funktion Prüfe-k-Eigenschaft(). Wir bezeichnen die Variante, welche die Funktion Prüfe-k-EigenschaftNaiv() nutzt, als **A1**. Wird die Funktion Prüfe-k-EigenschaftLinear() verwendet, bezeichnen wir diesen Algorithmus mit **A2** und **A3**, je nachdem ob die entsprechenden Zeitpunkte bei jedem Aufruf der Funktion eingefügt werden oder zu Beginn feststehen und nur durch ein Attribut als „Eingefügt“ markiert werden. Die Funktion Prüfe-k-EigenschaftLokal() nutzt entweder eine SkipList oder balancierte binäre Bäume zur Verwaltung der Zeitpunkte. Werden SkipLists benutzt, sprechen wir von Algorithmus **A4**, bei Verwendung von balancierten binären Bäumen von **A5**. SkipLists besitzen eine erwartete Laufzeit für das Suchen eines Elements in $O(\log n)$, wohingegen balancierte binäre Bäume diese Laufzeit garantieren

(siehe [CLRS09]) Die Algorithmen A1 bis A5 unterscheiden sich nur in der Laufzeit und nicht in der Güte. Deshalb werden wir für die Betrachtung der Güte insgesamt von Algorithmus **A** sprechen. Die separaten Algorithmen aus Kapitel 4 lösen die Konflikte und berechnen eine Teilmenge, welche die k-Eigenschaft erfüllt, separat. Insgesamt wurden sieben Varianten getestet. Bei allen Varianten wurde zum Lösen der Konflikte der Algorithmus GreedyGeneralMaxTotal() benutzt. Damit unterscheiden sich die Algorithmen nur im Algorithmus, welcher die k-Eigenschaft betrachtet. Da beide Teilalgorithmen unabhängig voneinander arbeiten, können wir auch die Reihenfolge verändern. Zuerst betrachten wir die getesteten Algorithmen, welche erst die Konflikte lösen und anschließend die k-Eigenschaft betrachten. Der Algorithmus Greedy-k-Eigenschaft() (siehe Abschnitt 4.2.2) wurde wieder in fünf Varianten getestet, welche die jeweils gleiche Funktion wie A1 bis A5 für Prüfe-k-Eigenschaft() verwenden. Wir benennen den gesamten Algorithmus dann entsprechend **B1** bis **B5**. Auch hier unterscheiden sich die Varianten B1 bis B5 nur in der Laufzeit. Wenn wir hingegen von der Güte sprechen, bezeichnen wir den Algorithmus mit **B**. Wurde zum Betrachten der k-Eigenschaft der Sweepline-Algorithmus (siehe Abschnitt 4.2.3) verwendet, bezeichnen wir den resultierenden Algorithmus mit **C**. Wird der Flussalgorithmus verwendet, sprechen wir von Algorithmus **D**. Die separaten Algorithmen haben wir auch in umkehrter Reihenfolge getestet. Das heißt, es wurde zuerst die k-Eigenschaft betrachtet und anschließend wurden die Konflikte aufgelöst. In diesem Fall bezeichnen wir die Algorithmen mit **B1r** bis **B5r**, **Cr** und **Dr**.

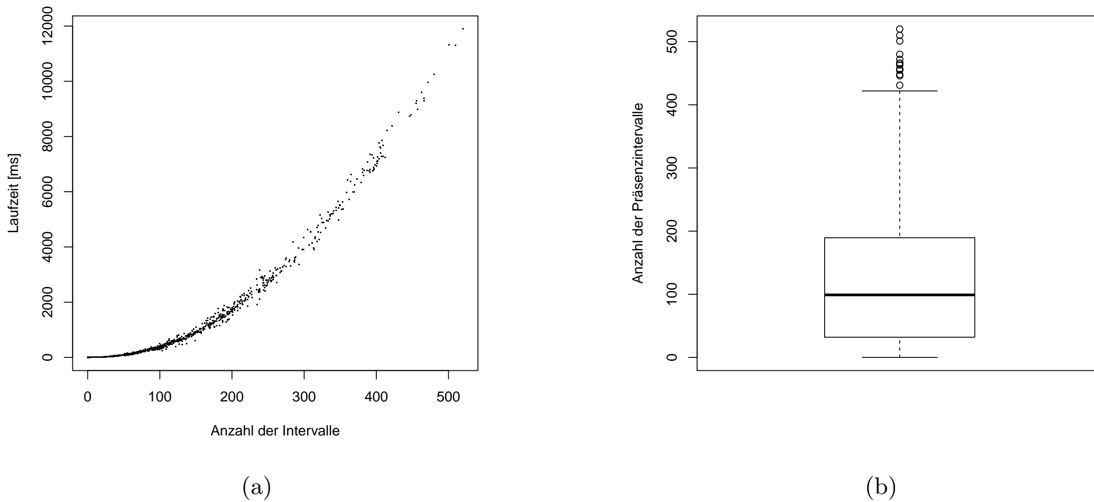
Wir haben verschiedene Werte für den Parameter k betrachtet. Dabei haben wir realistische Werte für den Anwendungsfall der Fahrzeug- oder Fußgängernavigation gewählt. Alle Algorithmen wurden jeweils mit dem Parameter $k = 3$, $k = 4$, $k = 5$ und $k = 10$ getestet. Im Anhang, in Abbildung A.1 sieht man die durchschnittliche Laufzeit und Güte für jeden Algorithmus im Vergleich.

Laufzeit

Es hat sich herausgestellt, dass der Parameter k keinen signifikanten Einfluss auf die Laufzeit der Algorithmen besitzt. Im Anhang in den Abbildungen A.2, A.3, A.4 und A.5 sind die Laufzeiten jedes Algorithmus für alle betrachteten Werte für k zu sehen. Wir betrachten für die Laufzeit deshalb stellvertretend nur den Fall $k = 4$.

Wir beginnen mit der Betrachtung der Laufzeiten der Implementation des ganzzahligen linearen Programms aus [GNN13]. In Abbildung 5.1 ist die Laufzeit des Programms in Abhängigkeit von der Anzahl der zu bearbeitenden Intervalle für den Parameter $k = 4$ gezeigt. Dabei ist die Anzahl der zu bearbeitenden Intervalle die Eingabegröße. Insgesamt ist zu erkennen, dass eine Erhöhung der Eingabegröße eine Vergrößerung der Laufzeit zur Folge hat. Die Zunahme der Laufzeit ist hier deutlich über einem linearen Wachstum. Für die Hälfte der Instanzen liegt die Eingabegröße unterhalb von 99 Präsenzintervallen (siehe Abbildung 5.1). Das hat zur Folge, dass der Median der Laufzeit bei 354 Millisekunden liegt (siehe Abbildung 5.2). Insgesamt lassen sich die optimalen Lösungen von 658 Instanzen in weniger als $1,0$ Sekunden berechnen. Jedoch liegt die Laufzeit für Instanzen mit sehr großen Eingabegrößen bei einigen Sekunden.

Die Abbildung 5.2 zeigt die Laufzeiten der verschiedenen Varianten des Algorithmus A. In den Abbildungungen A.7 im Anhang ist die durchschnittliche Laufzeit der Varianten in Abhängigkeit von der Anzahl der Intervalle abgebildet. Für die getesteten Instanzen besitzen alle Varianten Laufzeiten von unter 100 Millisekunden und sind damit deutlich schneller als das ganzzahlige lineare Programm. Vergleicht man die Algorithmen untereinander, fällt die höhere Laufzeit der Variante A1 gegenüber den anderen Varianten auf. Dies spiegelt die größere asymptotische Laufzeit, von $O(n^2 \log n)$ gegenüber $O(n^2)$ der anderen Varianten, wieder. Es zeigt sich auch ein kleiner Geschwindigkeitsvorteil der Variante A2 gegenüber A3. Das sortierte Einfügen der Zeitpunkte ist also insgesamt etwas schneller als die Betrachtung aller möglichen Zeitpunkte. Wie angenommen, bringt die Funktion



(a) (b)

Abbildung 5.1: figure
Laufzeit des optimalen Algorithmus in Abhängigkeit der Eingabegröße, für $k = 4$ (a) und die Verteilung der Trajektorien hinsichtlich der Anzahl der Präsenzintervalle (b)

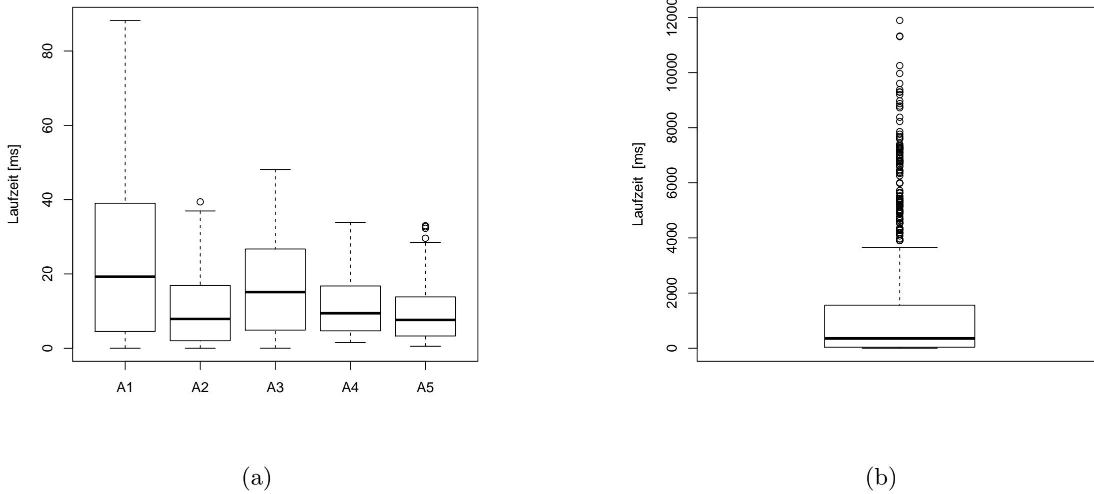
Prüfe-k-EigenschaftLokal() (siehe Variante A4 und A5) trotz gleicher asymptotischer Laufzeit einen Geschwindigkeitsvorteil.

Die Laufzeiten der separaten Algorithmen (siehe Kapitel 4) sind in Abbildung 5.3 dargestellt. Insgesamt liegen auch hier die Laufzeiten deutlich unter denen des ganzzahligen linearen Programms. Bezüglich der Laufzeit zeigen die Varianten B1 bis B5 ein ähnliches Verhalten wie die Algorithmen aus Kapitel 3. Dies ist dadurch zu begründen, dass die Varianten B1 bis B5 die jeweils selbe Funktion Prüfe-k-Eigenschaft() wie die Varianten A1 bis A5 nutzen. Betrachtet man den Median, besitzen die Varianten C und D die besten Laufzeiten. Die asymptotischen Laufzeiten ließen dies erwarten. Eine Lösung, welche die k-Eigenschaft erfüllt, kann mit dem Algorithmus Sweepline-k-Eigenschaft() (Variante C) in $O(n \log n)$ und mit dem Flussalgorithmus (Variante D) in $O(k \cdot n)$ berechnet werden. Zum Vergleich, die Laufzeit der Variante B5 liegt in $O(n^2)$. Zusätzlich werden jedoch die Konflikte mit dem Algorithmus GreedyGeneralMaxTotal() aufgelöst. Dieser besitzt eine asymptotische Laufzeit in $O(n \log n)$. Deshalb fällt der Vorteil in der Laufzeit gering aus. Die Laufzeiten für die separaten Algorithmen sind nur für den Fall angegeben, dass die Konflikte zuerst aufgelöst werden. Die Reihenfolge hat aber nur einen geringen Einfluss auf die Laufzeiten. Dies ist im Anhang in den Abbildungen A.3, A.4 und A.5 zu sehen.

Güte

Nun werden wir die Güte der Algorithmen betrachten. Wir geben dabei den Wert einer Lösung relativ zum Wert der optimalen Lösung an. Da es sich bei k-RestrictedMaxTotal um ein Maximierungsproblem handelt, ist die Güte immer kleiner gleich 1.

Abbildung 5.4 zeigt die Güte aller Algorithmen. Auffällig ist die schlechte Güte der separaten Algorithmen in gegensätzlicher Reihenfolge (Br, Cr und Dr). Die entsprechenden Algorithmen, welche zuerst die Konflikte lösen, erreichen im Vergleich deutlich bessere Werte. Dies kann daran liegen, dass nachdem die k-Eigenschaft hergestellt wurde, noch viele Konflikte bestehen. Werden diese anschließend aufgelöst, bleiben nur wenige Präsenzintervalle übrig. Umgekehrt können nach der Auflösung der Konflikte, zu vielen Zeitpunkten noch mehr als k Präsenzintervalle zur Verfügung stehen. Des Weiteren weist der Algorithmus B eine ähnliche Güte auf, wie Algorithmus A. Zudem arbeitet der Teilalgorithmus GreedyGeneralMaxTotal() optimal, falls nur ein Konflikt pro Intervall besteht. Die Algorithmen arbeiten nach ähnlichem Prinzip, was eine Erklärung für die Ähnlichkeit der



(a) (b)

Abbildung 5.2: Durchschnittliche Laufzeiten der Varianten des Algorithmus A (a) und des optimalen Algorithmus (b), für $k = 4$

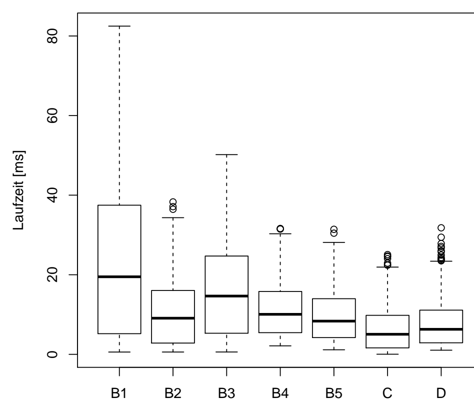


Abbildung 5.3: Durchschnittliche Laufzeiten der getesteten separaten Algorithmen (siehe Kapitel 4) in Abhängigkeit der Eingabegröße, für $k = 4$

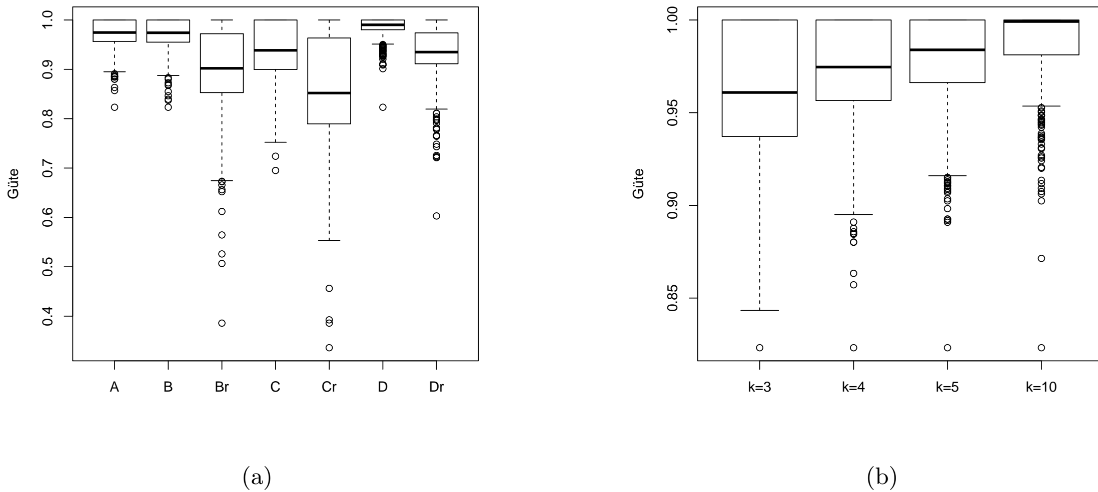


Abbildung 5.4: Die durchschnittliche Güte aller Algorithmen für $k = 4$ (a) und des Algorithmus A für verschiedene Parameter k (b)

Güte ist. Die insgesamt beste durchschnittliche Güte mit $0,9902$ erreicht der Algorithmus D. Bei diesem Algorithmus wird die k-Eigenschaft mithilfe des Flussnetzwerks optimal gelöst. Die Algorithmen C und Cr besitzen im Vergleich die schlechteste Güte, was sich darauf zurückführen lässt, dass der verwendete Algorithmus Sweepline-k-Eigenschaft() beliebig schlecht werden kann.

Weitere Auswertungen haben gezeigt, dass, anders als für die Laufzeit, die Güte der Algorithmen vom gewählten Parameter k abhängt. Mit steigendem Parameter k erhöht sich bei fast allen Algorithmen die Güte. Ausnahme bilden hier die separaten Algorithmen D und Dr. In Abbildung 5.4 ist das Verhalten für den Algorithmus Greedy-k-Restricted-MaxTotal() (A) für verschiedene Parameter k gezeigt. Bis auf die Algorithmen D und Dr weisen alle diesen Trend auf. Die fehlenden Abbildungen sind im Anhang in Abbildung A.6 zu finden. Die Anzahl der Bereiche, in denen mehr als k Intervalle präsent sind, wird mit der Erhöhung von k geringer. Damit hat einer der Algorithmen für die k-Eigenschaft (A, B und C) mit steigendem k weniger Einfluss auf den Wert der Lösung. Fehler wirken sich dann nicht mehr so stark aus. Da Algorithmus D und Dr eine optimale Teilmenge berechnen, welche die k-Eigenschaft betrachtet, ist es denkbar, dass die Verschlechterung der Güte für aufsteigendes k bis $k = 5$ durch die Auflösung der Konflikte hervorgerufen wird.

6. Fazit

Wir haben in dieser Arbeit approximative und heuristische Algorithmen für trajektorienbasierte Kartenbeschriftung betrachtet. In [GNN13] wurde ein Algorithmus vorgestellt, der Überschneidungen in polynomieller Laufzeit löst, wenn die Anzahl der gleichzeitig sichtbaren Beschriftungen beschränkt ist. Jedoch besitzt dieser Algorithmus eine schlechte Laufzeit. Hinsichtlich der begrenzten Hardware von Navigationsgeräten sind Ansätze wünschenswert, die eine bessere Laufzeit aufweisen. Eine Möglichkeit, die in dieser Arbeit verfolgt wurde, um bessere Laufzeiten zu erreichen, besteht durch approximative und heuristische Algorithmen. Nach den grundlegenden Definitionen für die trajektorienbasierte Kartenbeschriftung haben wir erst einen approximativen Algorithmus besprochen, der das Problem vollständig löst. Wir haben den Approximationsfaktor gezeigt und verschiedene Varianten des Algorithmus mit unterschiedlicher Laufzeit formuliert. Für die Auflösung der Überschneidungen haben wir einen approximative Algorithmus vorgestellt, der das Teilproblem auf speziellen Instanzen optimal löst. Zusätzlich haben wir ein lineares Programm angegeben, welches auch auf eingeschränkten Instanzen eine optimale Lösung berechnet. Zur Begrenzung der Anzahl der gleichzeitig sichtbaren Beschriftungen haben wir zuerst die Transformation des Problems in ein Flussproblem (siehe [CL95]) vorgestellt. Damit kann eine optimale Lösung für das Teilproblem in Linearzeit berechnet werden. Anschließend haben wir einen approximativen und einen heuristischen Algorithmus für das Teilproblem präsentiert. Werden zwei Teilalgorithmen wird das Problem komplett gelöst. Für den resultierenden Algorithmus zeigen wir dann den Approximationsfaktor, wenn approximative Teilalgorithmen verwendet werden. Die Ergebnisse der experimentellen Evaluation haben dann gezeigt, dass die Algorithmen mit realen Eingabedaten sehr gute Lösungen berechnen und trotzdem eine kurze Laufzeit aufweisen. So erreicht der beste Algorithmus eine durchschnittliche Güte von über 99% und besitzt dabei eine durchschnittliche Laufzeit von weniger als 10 Millisekunden. Somit haben wir gezeigt, dass das Problem der trajektorienbasierten Kartenbeschriftung gut angegangen werden kann.

Trotzdem gibt es noch offene Fragestellungen. Alle Algorithmen berechnen eine Lösung unter der Beachtung des Aktivitätsmodells AM1 (siehe Kapitel 2). Das bedeutet, eine Beschriftung wird für die gesamte Dauer, die es sich im sichtbaren Bereich befindet, entweder angezeigt oder ausgeblendet. In den Aktivitätsmodellen AM2 und AM3 besteht die Möglichkeit, dass eine Beschriftung früher aus- bzw. später eingeblendet werden kann. Hierfür existieren noch keine Algorithmen mit effizienter Laufzeit. Desweiteren wurden die Experimente auf einem Rechner mit deutlich höherer Leistung ausgeführt, als ein Navigationsgeräts überlicherweise aufweist. Es wäre als von Interesse die Algorithmen unter realen Bedingungen, z.B. auf einem Navigationsgerät oder Smartphone zu testen.

Literaturverzeichnis

[AvKS98] Pankaj K. Agarwal, Marc van Kreveld, and Subhash Suri: *Label placement by maximum independent set in rectangles.* Computational Geometry, 11(3–4):209 – 218, 1998. http://www.sciencedirect.com/science/article/pii/S0925772198000285.

[BDY06] Ken Been, Eli Daiches, and Chee Yap: *Dynamic map labeling.* IEEE Transactions on Visualization and Computer Graphics, 12(5):773–780, September 2006. http://dx.doi.org/10.1109/TVCG.2006.136.

[BNPW10] Ken Been, Martin Nöllenburg, Sheung Hung Poon, and Alexander Wolff: *Optimizing active ranges for consistent dynamic map labeling.* Computational Geometry, 43(3):312 – 328, 2010. http://www.sciencedirect.com/science/article/pii/S0925772109000649, Special Issue on 24th Annual Symposium on Computational Geometry (SoCG'08).

[CC09] Parinya Chalermsook and Julia Chuzhoy: *Maximum independent set of rectangles.* In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 892–901. Society for Industrial and Applied Mathematics, 2009. http://dl.acm.org/citation.cfm?id=1496770.1496867.

[CL95] Martin C. Carlisle and Errol L. Lloyd: *On the k-coloring of intervals.* Discrete Applied Mathematics, 59(3):225 – 235, 1995. http://www.sciencedirect.com/science/article/pii/0166218X9580003M.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[FPT81] Robert J. Fowler, Michael S. Paterson, and Steven L. Tanimoto: *Optimal packing and covering in the plane are np-complete.* Information Processing Letters, 12(3):133 – 137, 1981. http://www.sciencedirect.com/science/article/pii/0020019081901113.

[GNN13] Andreas Gemsa, Benjamin Niedermann, and Martin Nöllenburg: *Trajectory-based dynamic map labeling.* Proceedings of the 24th International Symposium on Algorithms and Computation (ISAAC'13), Lecture Notes in Computer Science, 2013.

[GNR11] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter: *Consistent labeling of rotating maps.* In Frank Dehne, John Iacono, and Jörg Rüdiger Sack (editors): *Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, pages 451–462. Springer, 2011.

[SW01] Tycho Strijk and Alexander Wolff: *Labeling points with circles.* International Journal of Computational Geometry Applications, 11(02):181–195, 2001. http://www.worldscientific.com/doi/abs/10.1142/S0218195901000444.

[vKSW98] Marc van Kreveld, Tycho Strijk, and Alexander Wolff: *Point set labeling with sliding labels.* In *Proceedings of the fourteenth annual symposium on Computational geometry*, SCG '98, pages 337–346. ACM, 1998. `http://doi.acm.org/10.1145/276884.276922`.

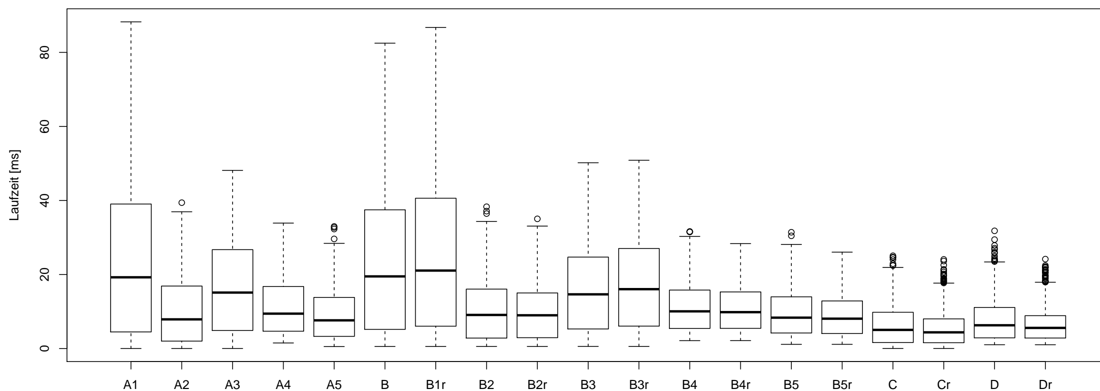
[WW97] Frank Wagner and Alexander Wolff: *A practical map labeling algorithm.* Computational Geometry, 7(5–6):387 – 404, 1997. `http://www.sciencedirect.com/science/article/pii/S0925772196000077`, 11th ACM Symposium on Computational Geometry.

[WWKS01] Frank Wagner, Alexander Wolff, Vikas Kapoor, and Tycho Strijk: *Three rules suffice for good label placement.* Algorithmica, 30(2):334–349, 2001. `http://dblp.uni-trier.de/db/journals/algorithmica/algorithmica30.html#WagnerWKS01`.

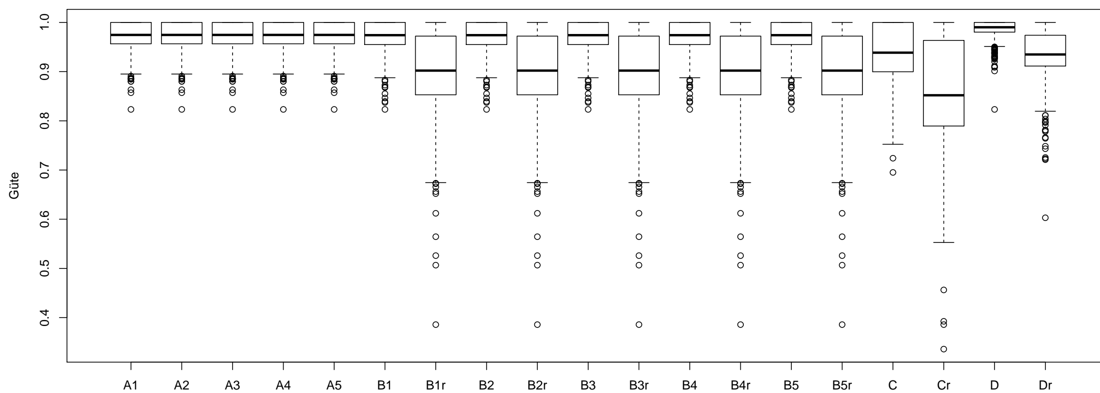
[YI13] Yusuke Yokosuka and Keiko Imai: *Polynomial time algorithms for label size maximization on rotating maps.* In *Proceedings of the 25th Canadian Conference on Computational Geometry*, pages 187–192, 2013. `http://www.cccg.ca/proceedings/2013/papers/paper_38.pdf`.

Anhang

A. Ergänzende Diagramme zur experimentellen Evaluation

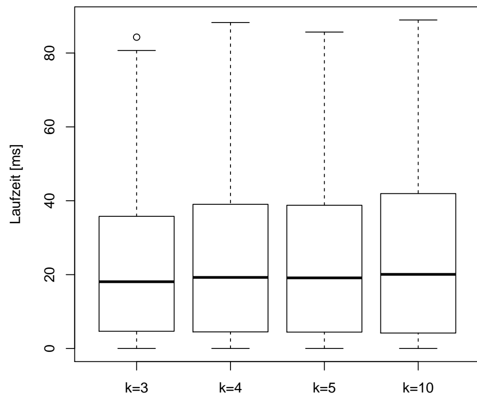


(a) Durchschnittliche Laufzeiten aller getesteten Algorithmen, für $k = 4$

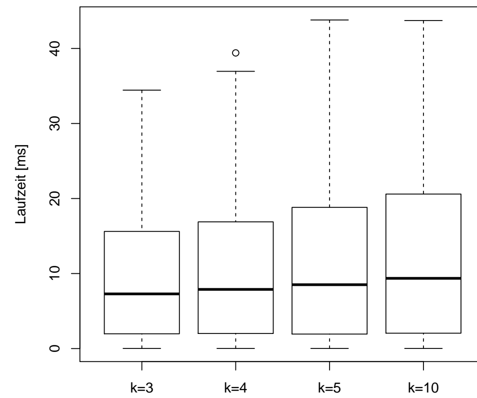


(b) Durchschnittliche Güte der Lösung aller getesteten Algorithmen relativ zum Wert der optimalen Lösung, für $k = 4$

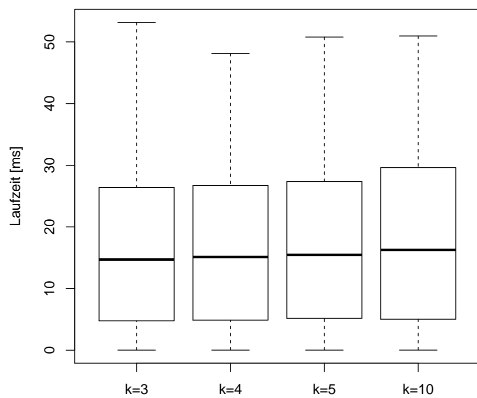
Abbildung A.1: Die durchschnittlichen Laufzeiten und Güten aller Algorithmen im direkten Vergleich, für $k = 4$



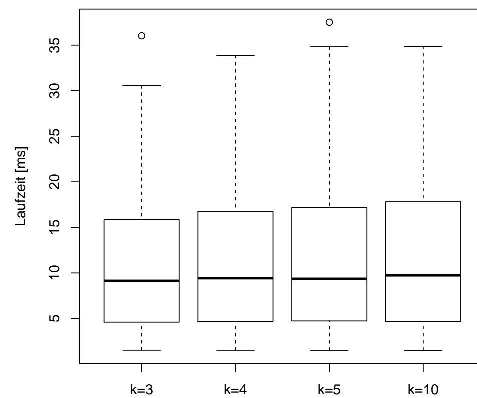
(a) Algorithmus A1



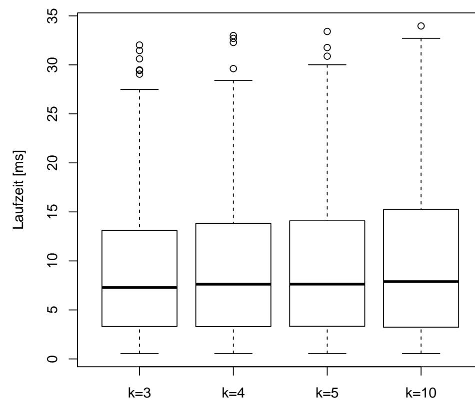
(b) Algorithmus A2



(c) Algorithmus A3

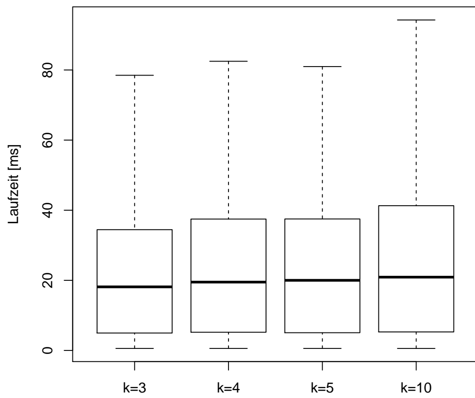


(d) Algorithmus A4

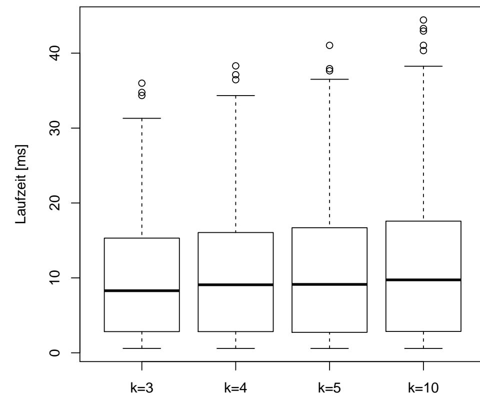


(e) Algorithmus A5

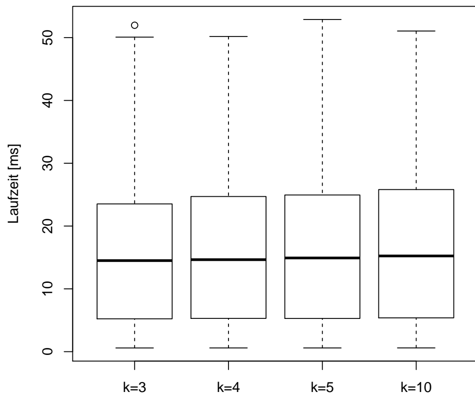
Abbildung A.2: Durchschnittliche Laufzeiten der jeweils angegebenen Algorithmen für verschiedene Parameter k



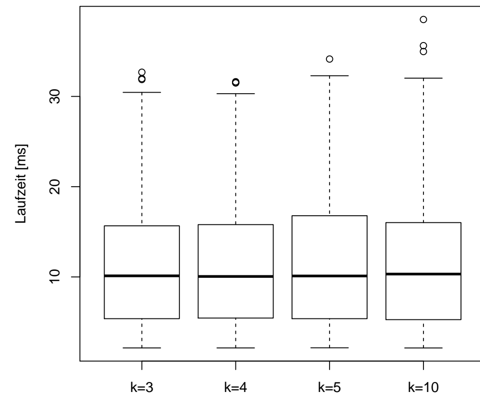
(a) Algorithmus B1



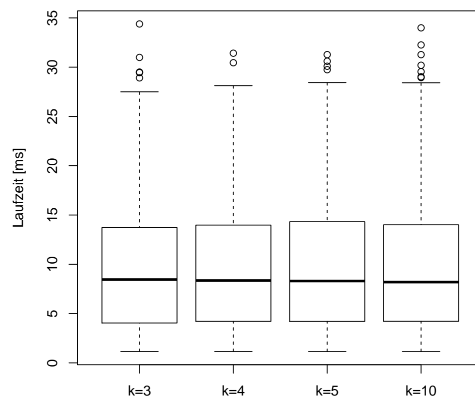
(b) Algorithmus B2



(c) Algorithmus B3

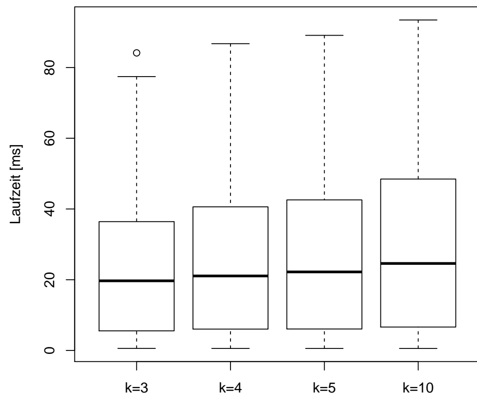


(d) Algorithmus B4

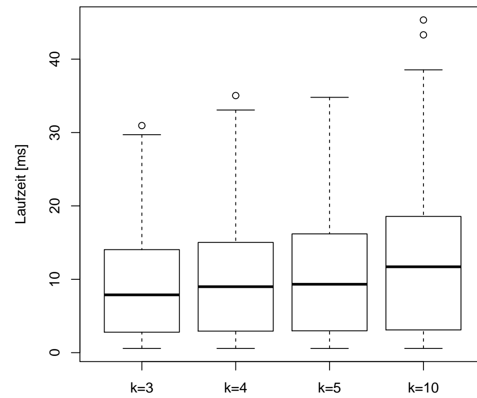


(e) Algorithmus B5

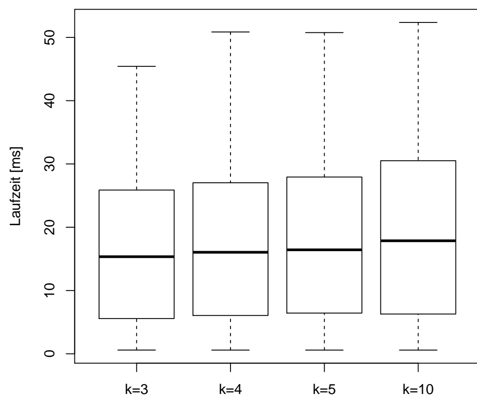
Abbildung A.3: Durchschnittliche Laufzeiten der jeweils angegebenen Algorithmen für verschiedene Parameter k



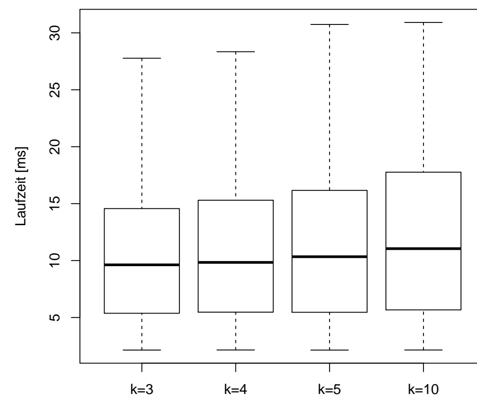
(a) Algorithmus B1r



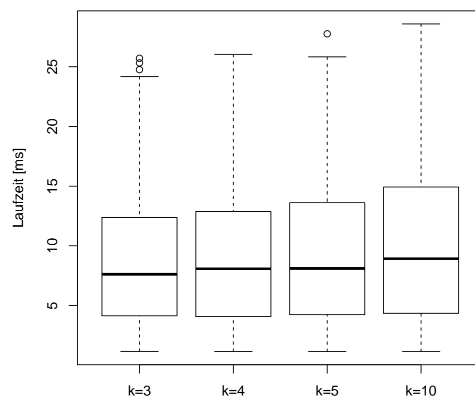
(b) Algorithmus B2r



(c) Algorithmus B3r

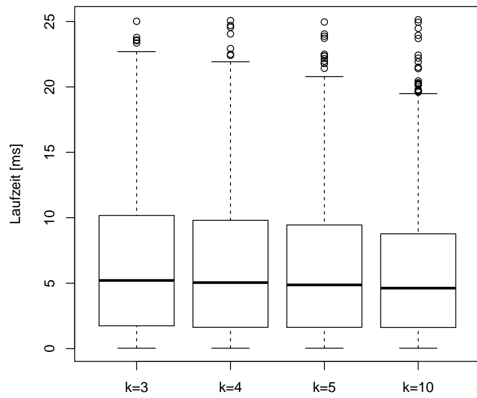


(d) Algorithmus B4r

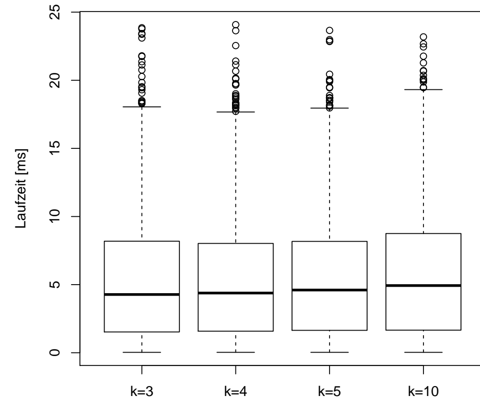


(e) Algorithmus B5r

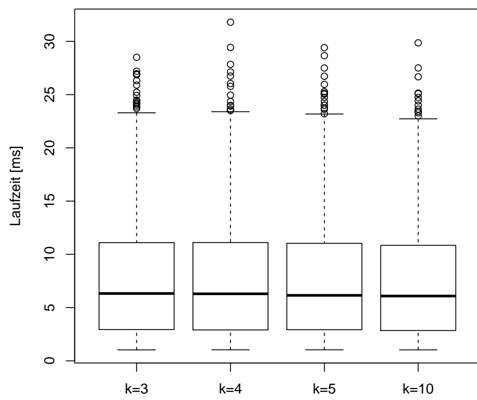
Abbildung A.4: Durchschnittliche Laufzeiten der jeweils angegebenen Algorithmen für verschiedene Parameter k



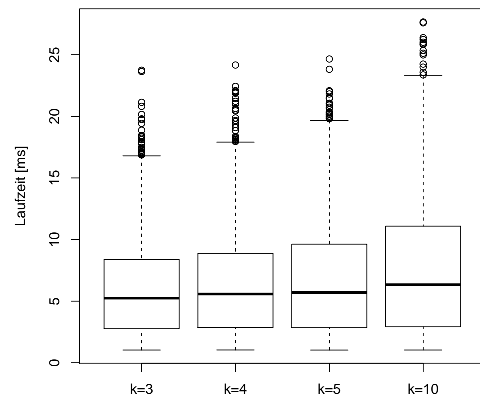
(a) Algorithmus C



(b) Algorithmus Cr

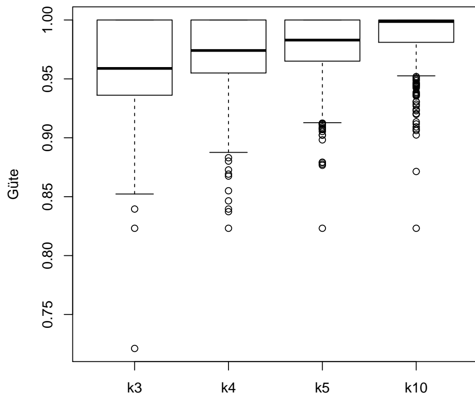


(c) Algorithmus D

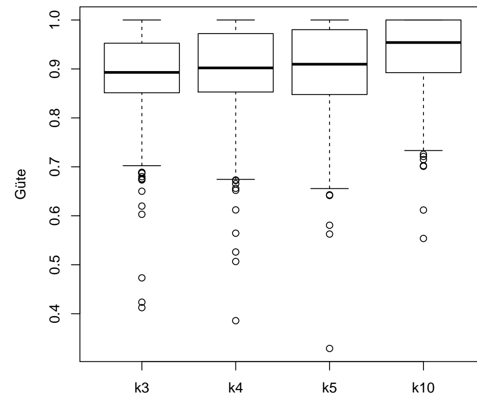


(d) Algorithmus Dr

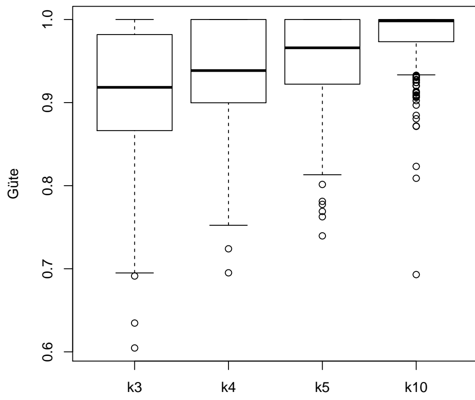
Abbildung A.5: Durchschnittliche Laufzeiten der jeweils angegebenen Algorithmen für verschiedene Parameter k



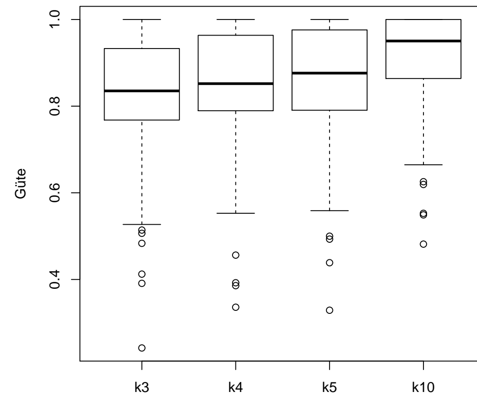
(a) Algorithmus B



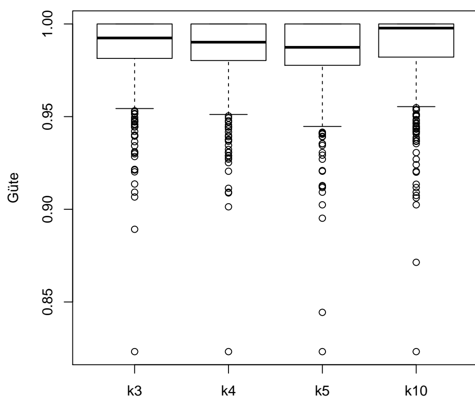
(b) Algorithmus Br



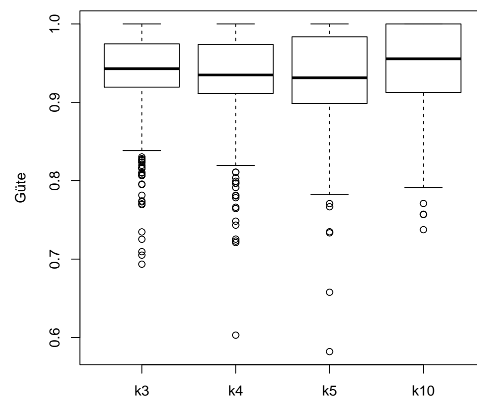
(c) Algorithmus C



(d) Algorithmus Cr

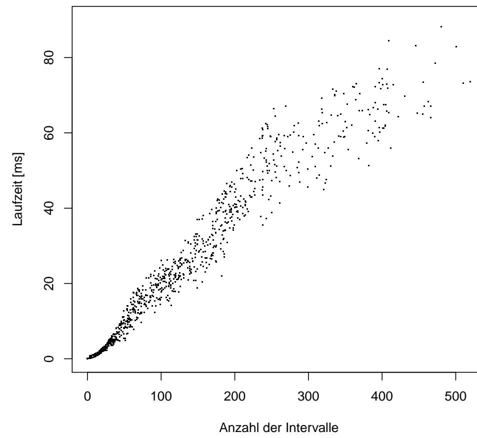


(e) Algorithmus D

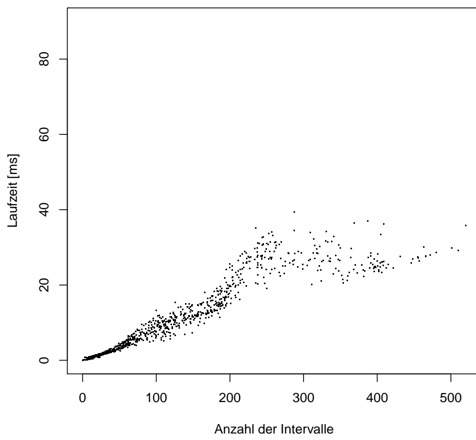


(f) Algorithmus Dr

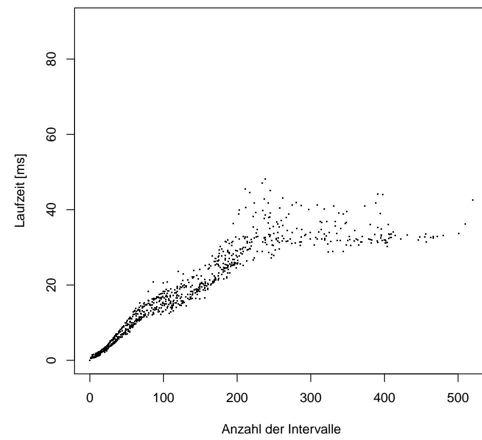
Abbildung A.6: Durchschnittliche Güte der jeweils angegebenen Algorithmen für verschiedene Parameter k



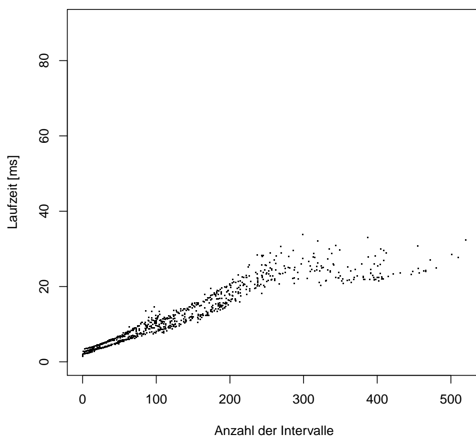
(a) A1



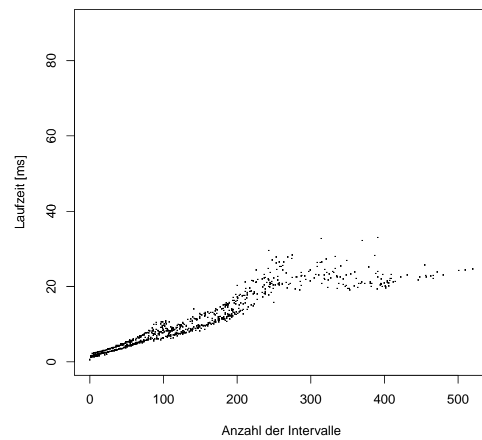
(b) A2



(c) A3



(d) A4



(e) A5

Abbildung A.7: Durchschnittliche Laufzeiten der jeweils angegebenen Varianten des Algorithmus Greedy-k-RestrictedMaxTotal() (siehe Kapitel 3) in Abhängigkeit der Eingabegröße, für $k = 4$