Karlsruher Institut für Technologie

Bachelorarbeit

Schnelle Berechnung von Alternativgraphen

Marcel Radermacher

Abgabedatum: 13.11.2012

Betreuer: Prof. Dr. Peter Sanders
Dipl. Inform. Moritz Kobitzsch
Dipl. Phys., Dipl. Inform. Dennis Schieferdecker

Institut für Theoretische Informatik, Algorithmik
Fakultät für Informatik
Karlsruher Institut für Technologie

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den Datum

Zusammenfassung

Für das Kürzeste-Wege-Problem gibt es viele effiziente Algorithmen, die in dem zugrunde liegenden Modell die optimalen Lösungen liefern. Die Qualität ist allerdings rein subjektiv begründet. Alternativgraphen liefern eine Auswahl von Pfaden, die einen sinnvolle Alternative zum kürzesten Weg darstellen können. Wir geben die erste effiziente Implementierung eines von Bader et al. vorgestellten Verfahrens zur Berechnung von Alternativgraphen. Dabei schaffen wir es den klassischen Ansatz um eine Größenordnung zu beschleunigen.

Abstract

There are several efficient algorithms which solve the shortest-path-problem with respect to the given model. The quality of the this solution is purely subjectiv. Alternative graphs encode several paths which may be meaningful alternatives. We present the first efficient implementation of the method to calculate alternative graphs proposed by Bader et al. . We are able to accelerate the calculation about an order of magnitude with respect to the classic approach.

Danksagungen

Vielen Dank an Daniel Delling von Microsoft Research Silicon Valley, der uns eine Partitionierung für unseren Graphen zur Verfügung gestellt hat. Vielen Dank auch an meine Freunde und Familie, die mich über die ganze Zeit meiner Bachelorarbeit unterstützt haben.

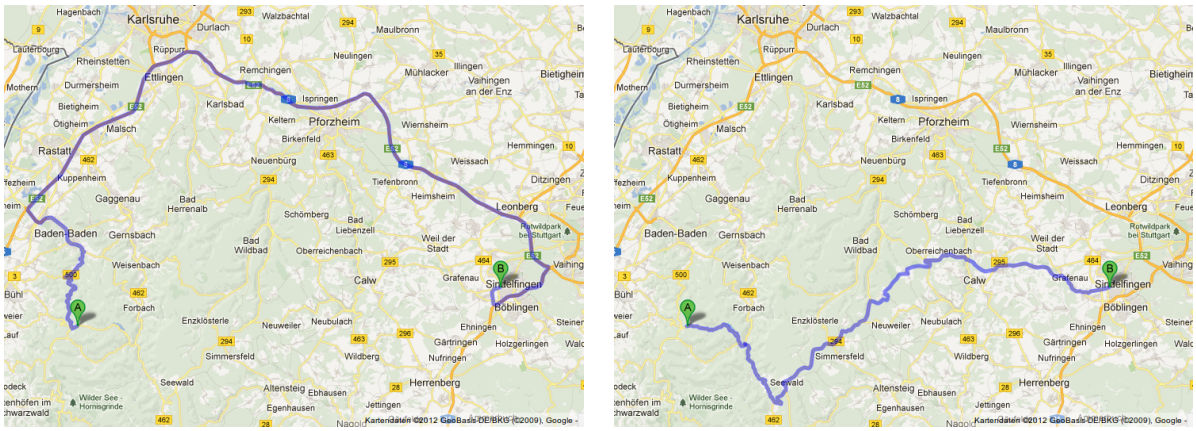
Inhaltsverzeichnis

1 Einführung

Die Berechnung von kürzesten Wegen wurde mit großen Fortschritten in den letzten Jahrzehnten voran gebracht. Neben kürzesten Wegen auf Straßennetzen gewinnen Alternativpfade immer mehr an Interesse. Alternativepfade sollen in der Routenplannung weitere Möglichkeiten darstellen mit einem geringen Umweg von dem gewünschten Startpunkt zum Ziel zu gelangen. Oft unterscheiden sich zwei Pfade in der Reisezeit nur um wenige Minuten, dafür kann die langsamere Variante kürzer sein. So zum Beispiel in Abbildung 1, die Route 1(a) über die Autobahn ist zwar schneller, dafür aber 40km länger. Alternativpfade liefern dem Anwender somit nicht nur die optimale Lösung in dem gegebenen Modell, sondern eine sinnvolle Auswahl an Pfaden, die ihn unterstützen können, seine Route nach persönlichen Präferenzen und aktueller Verkehrssituation auszuwählen.

Schon früh wurden erste Verfahren entwickelt, die nicht nur einen kürzesten Weg finden. Die Problemstellung wurde 1971 schon mit dem *k-shortest-path Problem* untersucht [15]. Die Berechnung der von Alternativgraphen ist eine eng verwandte Problemstellung. In den letzten Jahren wurde diese präzise formuliert [24, 14], Aussagen über die Schwere des Problems[24] getroffen und Qualitätsmerkmale festgelegt [24, 14].

Wir kombinieren ein bekanntes Verfahren aus der Routenplanung (MLD [5]) und der Alternativgraphberechnung (Penalty-Methode [24]). Mit der Kombination der beiden Verfahren sind wir in der Lage eine schnelle Berechnung von Alternativgraphen zu realisieren.



(a) Pfad mit 128km Länge und 1 Stunde und 32 Minuten Reisezeit.

(b) Pfad mit 81km Länge und 1 Stunde und 35 Minuten Reisezeit.

Abbildung 1: Zwei Pfade zum gleichen Ziel in Google Maps. Die Pfadlänge unterscheidet sich stark, die Reisezeit allerdings nur wenig.

1.1 Routing-Algorithmen

Edsger W. Dijkstra entwickelte 1956 [17] mit dem nach ihm benannten Verfahren die Grundlage für viele effiziente *Routing*-Algorithmen. Sein Verfahren bearbeitet die Knoten nach aufsteigender Distanz vom Startknoten aus. Die bidirektionale Variante seines Algorithmus kann für die Lösung des *single-pair-shortest-path* Problems genutzt werden. Diese Variante führt simultan zwei unabhängige Dijkstra Anfragen durch: eine *Vorwärtssuche*, die auf dem Ursprungsgraph vom Startknoten aus operiert, und eine *Rückwärtssuche*, die bei dem Zielknoten startet und nur den invertierten Graph betrachtet.

Unter der Annahme, dass ein statischer Graph vorliegt, d.h. dass die Topologie und die Kantengewichte sich nicht ändern, können weitere Beschleunigungen der Suchanfragen erreicht werden. In der Literatur finden sich *zielgerichtete* Verfahren (z.B. *A**, *ALT*, *Arc Flags*), *hierarchische Verfahren* (z.B. *Contraction Hierachies*, *MLD*) [8] und Kombinationen von beiden [21].

1.1.1 Zielgerichtete Verfahren

Zielgerichtete Verfahren versuchen die Suchanfrage mit einem Richtungssinn auszustatten. In vielen Fällen kann dadurch unnötige Arbeit gespart werden.

A* Der *A**[1] Algorithmus erweitert Dijkstras Algorithmus um eine Knotenpotentialfunktion, die für jede Suchanfrage unterschiedlich sein kann. Mit der Knotenpotentialfunktion werden die Kantengewichte verändert und dies führt, bei einer geeignete Wahl der Potentialfunktion, zu einem stark reduzierten Suchraum. Ein einfaches Beispiel für eine korrekte Potentialfunktion für Straßennetze, mit Distanzmetrik, ist die euklidische Distanz zum Ziel.

ALT *ALT* (A*, landmarks, triangle-inequality) [1] erweitert A* und automatisiert zu einem Graphen mit beliebiger nicht-negativer Metrik die Berechnung von einer gültigen Potentialfunktion. ALT nutzt aus, dass die Länge des kürzesten Weges von einem Knoten v zu einem fixierten Knoten l, einer Landmarke, und die Länge des kürzesten Weges von v zum Ziel mit der Dreiecksungleichung zu einer korrekten Potentialfunktion führt. Eine Menge von Knoten L (Landmarken) führt auf diese Weise zu einer Potentialfunktion, die den Suchraum reduzieren kann.

Arc Flags Ein Verfahren das ohne die Potentialfunktion auskommt ist *Arc Flags*. Die Technik basiert auf der Partitionierung des Graphen [25]. Bei n Partitionen wird jede Kante um n Bits erweitert. Dabei wird das k-te Bit mit `true` markiert, falls die Kante auf einem kürzesten Weg in die Partition k liegt, andernfalls mit `false`. Der Dijkstra-Algorithmus muss bei einer Anfrage zu einem Knoten in die Partition k nur um die Überprüfung ergänzt werden, ob das k-te Bit einer Kante gesetzt ist.

1.1.2 Hierarchien

Eine zweite Kategorie von Beschleunigungstechniken erstellt eine Hierarchie des Graphen. Mit deren Hilfe können *unwichtige* Teile des Graphen übersprungen werden.

Contraction Hierarchies Das *Contraction Hierachies*-Verfahren (CH) [23] versucht eine Ordnung der Knoten zu finden, die die Wichtigkeit eines Knoten repräsentiert. In der Vorberechnung werden alle Knoten v in der aufsteigenden Reihenfolge ihrer Wichtigkeit betrachtet. Ein KAnte (u, w), ein sogenannter Shortcut, wird in den Graphen eingefügt, wenn u und w wichtigere Nachbarn von v sind und der Pfad $\langle u, v, w\rangle$ der einzige kürzeste Pfad zwischen u und w ist. Anfragen werden mit einer bidirektionalen Dijkstra-Suche beantwortet, mit der Anpassung, dass nur Kanten zu Knoten betrachtet werden, die in der Hierarchie (Knoten mit einer höheren Wichtigkeit) aufsteigen.

Multilevel-Separatoren Der Multilevel-Separatoren-Ansatz [16, 28, 26, 12] basiert auf einer Multilevel-Partitionierung. Das Verfahren gliedert sich in drei Phasen. Die erste Phase partitioniert den Graphen und ist somit Metrik-unabhängig. Diese Phase ist somit für jede Metrik

identisch. Die zweite Phase berechnet kürzeste Wege innerhalb der Partitionen und ist somit Metrik-abhängig. Die letzte Phase stellt die Anfrage auf den vorberechneten Daten dar.

Aufbauend auf der Partitionierung erstellt der Separatoren-Ansatz eine Abstraktion des ursprünglichen Graphen. Die einfachste Möglichkeit ist es, nur ein Abstraktionslevel einzufügen. Für jede Partition werden die kürzesten Wege von einem Randknoten einer Partition zu allen anderen Randknoten der gleichen Partition berechnet(Metrik-abhängig). Die Suchanfrage benötigt für die Berechnung der kürzesten Wege nur noch die Partitionen, in denen der Start- oder Zielknoten liegt, und das Abstraktionslevel. Das Verfahren kann verallgemeinert werden, indem nicht nur eine Abstraktion sondern mehrere Abstraktionslevel eingeführt werden.

Delling et al. [5] haben den Separatoren-Ansatz erneut aufgegriffen und mit einer neuen Partitionierungstechnik ist es ihnen gelungen Suchanfragen ännlich schnell wie andere aktuelle Beschleunigungstechniken zu beantworten. Die *Multilevel Dijkstra* (MLD) genannte Technik wird unser Ausgangspunkt für die Berechnung von *Alternativgraphen* sein.

1.2 Alternativpfade und Alternativgraphen

Wir sind nicht direkt an der Berechnung von kürzesten Wegen interessiert, sondern an der Berechnung von Alternativpfaden. Für die Berechnung von Alternativen zeigt die Literatur zwei verschiedene Ansätze. Eine Grundlage bilden Alternativrouten oder ganze Alternativgraphen.

Ein Beispiel für die Berechnung von Alternativrouten sind *Pareto-Pfade* [11, 22]. Die grundlegende Idee ist es, Alternativpfade zu gewinnen, indem kürzeste Pfade auf unterschiedlichen Gewichtsfunktionen berechnet werden. Weiter kann auch eine Korridor-Suche [7] zu einer Alternativgraphberechnung erweitert werden. Die *Plateau-Methode* [3, 10] versucht aus der Schnittmenge der Vorwärts- und Rückwärtssuche Pfade mit möglichst großer Übereinstimmung zu extrahieren. Eine von der Idee her einfachere Methode, die aber schwierig effizient umzusetzen ist, ist die k-kürzesten-Wege (*k-shortest-paths* [9, 15]) zu berechnen. Als letztes Beispiel sei die *Penalty-Methode* [24, 27] genannt. Es werden iterativ kürzeste Wege berechnet und in den Alternativgraphen eingefügt. Die Kanten des aktuell kürzesten Pfad werden mit einer *Penalty* belegt, um so an neue kürzeste Wege zu gelangen.

2 Alternativgraphen

Die bisher vorgestellten Verfahren haben zum Ziel kürzeste-Wege-Anfragen effizient zu beantworten. Kürzeste Wege sind zwar theoretisch optimal, können in der Praxis aber Nachteile aufweisen. Subjektiv kann eine minimal längere Route die bessere Wahl sein. Aus diesem Grund sollen nicht nur ein kürzester Weg sondern zusätzlich *sinnvolle* Alternativen berechnet werden. Eine erste Idee, was eine gute Alternativroute darstellt, ist ein Weg im Graph, der möglichst kurz ist, sich aber in wesentlichen Teilen vom kürzesten Weg bzw. von anderen alternativen Routen unterscheidet.

2.1 Definitionen

Wir betrachten im Folgenden den gerichteten, gewichteten Graph $G = (V, E \subset V \times V, c : E \to \mathbb{R}_{\geq 0})$ mit der Knotenmenge V, der Kantemenge E und der Kantengewichtsfunktion c. Zu einem Pfad $\Pi = \langle v_1, v_2, \ldots, v_n \rangle, (v_i, v_{i+1}) \in E, i \in \{1, \ldots, (n-1)\}$ ist die Länge des Pfades als $c(\Pi) = \sum_{i=1}^{n-1} c((v_i, v_{i+1}))$ definiert. Für zwei Knoten $s, t \in V$ ist $\Pi_{s,t}$ der kürzeste Pfad von s nach t mit $d(s,t) = c(\Pi_{s,t})$. Mit $\Pi_G^{s,t}$ und $d_G(s,t)$ beschreiben wir entsprechend den kürzesten

Weg bzw. die Länge des kürzesten Weges von s nach t im Graphen G. Zu einem Knoten $v \in V$ bezeichnen wir mit $outdegree(v)$ die Anzahl an ausgehenden Kanten aus dem Knoten v und entsprechen mit $indegree(v)$ die Anzahl an eingehenden Kanten in v. Den Begrif *Vorwärtsgraph* benutzen wir als Synonym für G. Mit Rückwärtsgraph bezeichnen wir den Graphen, in dem alle Kanten invertiert sind.

Eine Partitionierung $\mathcal{P} = \{V_1, \ldots, V_p\}$ des Graphen G unterteilt die Knotenmenge V in p disjunkte Mengen. Eine Multilevel-Partitionierung mit l Leveln kann top-down wie folgt beschrieben werden. Aus der Partitionierung $\mathcal{P}_r = \{V_{r,1}, \ldots, V_{r,n_r}\}$ des Levels $r \in \{1, \ldots, l-1\}$ werden neue Partitionierungen $\mathcal{P}_{r-1,j}, j \in \{1, \ldots, n_r\}$ durch Partitionieren des von $V_{r,j}$, induzierten Graphen $G_{r,j} = (V_{r,j}, E \cap (V_{r,j} \times V_{r,j}))$ gewonnen. Die Partitionierung $\mathcal{P}_{r-1}$ ergibt sich dann durch Vereinigung der Mengen $\mathcal{P}_{r-1,j}$.

$$\mathcal{P}_{r-1} = \cup_{j=0}^{n_r} \mathcal{P}_{r-1,j}$$
$$\mathcal{P} = \cup_{j=0}^{l-1} \mathcal{P}_j$$

Mit $p(u) \in \mathcal{P}_0$ bezeichnen wir die Partition auf dem untersten Level zu Knoten $u \in V$, so dass u in $p(u)$ liegt. Die Zugehörigkeit zu Partitionen auf einem höheren Level lässt sich sukzessive mit der Funktion $parent(p) \in (\mathcal{P} \cup \emptyset), p \in \mathcal{P}$ oder *Vater von* p auflösen. Die Funktion liefert die Partition, aus der p hervorgegangen ist. Insbesondere gilt für die Partitionen p aus dem obersten Level $parent(p) = \emptyset$.

Eine Kante $e = (u, v) \in E$, die zu keiner Partition gehört, d.h. $p(u) \neq p(v)$, wird Randkante genannt. Insbesondere gilt: ist eine Kante e ein Randkante auf Level $l+1$, dann ist sie auch eine Randkante auf Level l. Existiert zu einem Knoten v eine Randkante $e = (u, v) \in E$ bzw. $e = (v, u)$, dann bezeichnen wir den Knoten als Randknoten.

2.1.1 Alternativgraphen

Bader et al. [24] schlagen folgende Definition für *Alternativgraphen* (AG) vor. Zu einem Graphen $G = (V, E)$ mit Startknoten s und Zielknoten t aus V, heißt $H = (V' \subset V, E' \subset V' \times V', c' : E' \to \mathbb{R}_{\geq 0})$ AG, wenn folgende Eigenschaften erfüllt sind.

- Knoten s, t sind in V' enthalten und es gibt keine isolierten Knoten,
- zu jeder Kante $e \in E'$ existiert ein einfacher s-t-Pfad in H, der e enthält (*einfache-Pfad-Eigenschaft*),
- jede Kante $e = (u, v) \in E'$ entspricht einem Pfad Π von u nach v in G mit $c'(e) = c(\Pi)$.

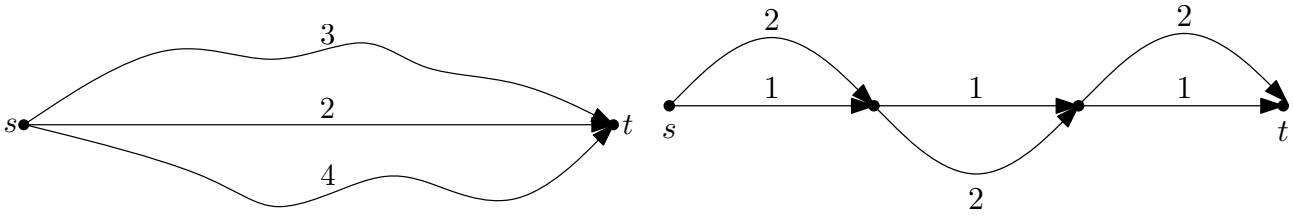
Wir nennen einen AG *reduziert*, wenn kein Knoten v in H existiert, für den gilt, dass $indegree(v) = 1$ und $outdegree(v) = 1$.

2.1.2 Bewertung

Die Definition des Alternativgraph sagt noch nichts über die Qualität der kodierten Alternativen aus. Nach der Definition darf sich auch ein Graph Alternativgraph nennen, wenn er alle möglichen Pfade von s nach t enthält. Daher schlagen Bader et al. [24] folgende Attribute zum Messen der Qualtität von Alternativgraphen vor.

Definition 2.1.

$$totalDistance := \sum_{e=(u,v) \in E'} \frac{c'(e)}{d_H(s,u) + c'(e) + d_H(v,t)}$$



(a) Drei disjunkte Pfade (b) Bis zu vier sich überlappende Pfade

Abbildung 2: Zwei kurze Beispiele, die zeigen, wie sich die Auswahl des Alternativgraphen auf die Qualitätsmerkmale auswirkt. Auf den Kanten sind die Kantengewichte aufgetragen.

Die Idee zu dem Wert *totalDistance* ist es, die Anzahl der Alternativen als ein Vielfaches des kürzesten Weges auszudrücken. Damit nimmt der Wert sein Maximum bei k disjunkten Pfaden an. Besteht der AG nicht nur aus disjunkten Pfaden, dann gibt der Wert darüber Auskunft, wie stark sich die einzelnen Routen überschneiden. In Abbildung 2(a) haben wir drei unabhängige Pfade und somit eine *totalDistance* von 3. Die in *totalDistance* betrachteten Pfade in Abbildung 2(b) haben bis auf den kürzesten Weg die Länge 4. Die betrachteten Pfade (ohne den kürzesten Pfad) haben allerdings jeweils einen gemeinsamen Anteil von 50% mit dem kürzesten Pfad. Insgesamt ergibt sich nur eine *totalDistance* von 5/2.

Definition 2.2.

$$averageDistance := \frac{\sum_{e \in E'} c'(e)}{d_G(s,t) \cdot totalDistance}$$

Da nicht alle Pfade aus einem Alternativgraphen eine sinnvolle Alternative darstellen, soll *averageDistance* die mittlere Distanz der sinnvollen Pfade widerspiegeln. In Abbildung 2(a) entspricht die *averageDistance* dem Mittelwert der Pfadlängen, wobei mit der Länge des kürzesten Weg normiert wird. Damit ergibt sich eine *averageDistance* von 3/2. Die *averageDistance* ist mit 6/5 in Abbildung 2(b) geringer, da die betrachteten Pfade weniger vom kürzesten Pfad abweichen.

Definition 2.3.

$$decisionEdges := \sum_{v \in V' \setminus \{t\}} (outdegree(v) - 1)$$

Die Anzahl der Entscheidungsmöglichkeiten gibt ein Maß für die Komplexität des Alternativgraphens an. Zu viele Entscheidungsmöglichkeiten können schnell überfordernd wirken. Die Anzahl an *decisionEdges* sollte somit gering gehalten werden. In Abbildung 2(a) haben wir zwar 3 unterschiedliche Pfade, allerdings nur 2 *decisionEdges*. Mit drei Möglichkeiten sich zu entscheiden, haben wir in der Abbildung 2(b) auch drei *decisionEdges*.

Definition 2.4.

$$objectiveFunction := totalDistance - \alpha(averageDistance - 1),\ \alpha \in (0, \infty) \tag{2.1}$$

Die Qualität des Graphen sollte möglichst durch einen Wert repräsentiert werden. Diesen gilt es dann zu optimieren. Bader et al. schlagen dazu die Funktion 2.1 vor.

$$averageDistance \leq 1.1,\ decisionEdges \leq 10 \tag{2.2}$$

Die Zielfunktion 2.1 soll unter Nebenbedingung 2.2 maximiert werden. Für den resultierenden Alternativgraph bedeutet dies, dass die mittlere Länge der Alternativen nicht mehr als 10% länger als der kürzeste Weg sind und dass die Anzahl der Entscheidungsmöglichkeiten auf 10 beschränkt ist.

2.2 Methoden

Das Problem ALTERNATIVE GRAPH lässt sich als Entscheidungsproblem formulieren.

Problem 1 (ALTERNATIVE GRAPH).

Gegeben:	*Graph* $G = (V, E, c : E \to \mathbb{R}_{\geq 0}), s, t \in V, C^*, K^* \in \mathbb{R}_{\geq 0}, N \in \mathbb{N}$
Frage:	*Gibt es einen Alternativgraph* H*, so dass gilt:*
	$totalDistance \geq K^*$
	$averageDistance \leq C^*$
	$decisionEdges \leq N$

Bader et al. [24] haben gezeigt, dass das Problem ALTERNATIVE GRAPH $\mathcal{NP}$-vollständig ist, indem sie es auf auf das Problem KNAPSACK reduziert haben. Eine exakte Lösung ist in der Praxis somit nicht zu erwarten. Daher werden im Folgenden verschiedene Heuristiken vorgestellt, mit denen (nicht optimale) Alternativgraphen berechnet werden können.

k-shortest-paths Eine naheliegende Möglichkeit Alternativrouten zu generieren, ist zu dem kürzesten Pfad die $k-1$ nächst kürzeren Pfade [9, 15] zu berechnen und in den Alternativgraphen einzufügen. Allerdings werden sich diese Routen häufig nicht sehr stark von dem kürzesten Weg unterscheiden. Im Falle eines Straßennetzes gibt es verschiedene Szenarien, die zu unerwünschten Ergebnissen führen. Denkbar ist, dass die k nächst kürzeren Pfade sich nur im mehrfachen Durchfahren eines Kreisverkehrs vom kürzesten Weg unterscheiden. Dieser Fall kann durch die Verschärfung der Problemstellung auf die Suche von k- kreisfreien kürzesten Wegen ausgeschlossen werden. Dies verhindert jedoch nicht weitere Szenarien wie das Abbiegen in eine Seitenstraße und das umgehende Zurückkehren auf den kürzesten Weg. Im Allgemeinen werden große k nötig sein, bis echte alternative Routen gefunden werden. Da kein Algorithmus bekannt ist, der das k-shortest-path Problem für praxisnahe Anwendungen schnell genug löst, ist dieser Ansatz momentan keine echte Option zum Berechnen von Alternativgraphen [24].

Pareto Oft steht nicht nur eine Gewichtsfunktion zur Verfügung, sondern mehrere wie Distanz, Reisezeit oder Spritverbrauch. Das Ziel des *Pareto-Ansatz* [11, 22] ist es nun pareto-optimale Pfade zu finden, also solche Pfade, die von keinem anderen Pfad dominiert werden. Ein Pfad dominiert einen anderen, wenn er in mindestens einer Zielfunktion besser und in allen anderen Argumenten mindestens genau so gut ist wie der Vergleichspfad. Nach Bader et al. [24] bedingt die hohe Anzahl der pareto-optimalen Pfade. Daher bieten sich zwei Lösungen für das Problem an: zum Einen eine Verschärfung des Dominanzkriteriums [6], zum Anderen müssen nicht alle pareto-optimalen Pfade in den AG eingefügt werden, sondern nur solche, die die Zielfunktion optimieren.

Plateau Der von CAMVIT [3] vorgeschlagene *Choice Routing* Algorithmus berechnet sogenannte Plateaus. In dem Verfahren wird der Suchbaum von Dijkstras Algorithmus im Vorwärts- und Rückwärtsgraph betrachtet (von s bzw. t aus). Maximal-lange Pfade, die in beiden Suchbäumen auftreten, werden Plateaus genannt. Der Pfad von s zum Plateau im Vorwärtssuchbaum mit dem Plateau und dem Pfad vom Plateau zu t im Rückwärtssuchbaum ergibt so eine

Alternativroute. Die Anzahl der so gefundenen Pfade kann sehr groß werden, daher schlagen Bader et al. [24] vor, die Routen nach einer Zielfunktionen zu ordnen und die Pfade, mit den besten Werten für die Zielfunktion, auszuwählen.

Penalty Die Penalty Methode berechnet iterativ einen kürzesten Pfad und fügt diesen in den Alternativgraphen ein. Um an neue kürzeste Wege zu gelangen, werden nach jeder Iteration die Kantengewichte des Pfades erhöht. Es werden so lange neue Pfade in den Alternativgraphen eingefügt bis Bedingung 2.2 verletzt ist. In jeder Iteration wird die Zielfunktion berechnet. Das Verfahren gibt den Alternativgraphen aus der Iteration mit der höchsten Zielfunktion zurück.

Wie stark die Kantengewichte auf dem kürzesten Pfad geändert werden, hat Einfluss auf die Qualität des Alternativgraphen. Einfaches Aufaddieren einer Konstante auf die Kantengewichte [27], führt zu einer Bestrafung von kurzen Pfaden mit einer hohen Anzahl an Kanten. Daher erhöhen Bader et al. [24] das Kantengewicht um einen Bruchteil des aktuellen Kantengewichts. Der zugrundeliegende Faktor wird $penalty\text{-}factor \in [0, 1]$ genannt.

$$\begin{aligned} &rejoin\text{-}penalty := rejoin\text{-}factor \cdot penalty\text{-}factor \cdot 0.5 \cdot d_G(s,t) \\ &rejoin\text{-}factor \in [0,1] \end{aligned} \tag{2.3}$$

In dieser Variante unterscheidet sich der neu berechnete Weg oft nur durch kleinere Umwege vom vorherigen Pfad. Umgangen wird dieser Effekt durch das Aufaddieren einer $rejoin\text{-}penalty$ auf die Kanten, die in den akutellen Pfad führen bzw. diesen verlassen [24]. Diese Kanten nennen wir *Rejoin-Kanten*. Die $rejoin\text{-}penalty$ sollte nach der Vorschrift 2.3 gebildet werden.

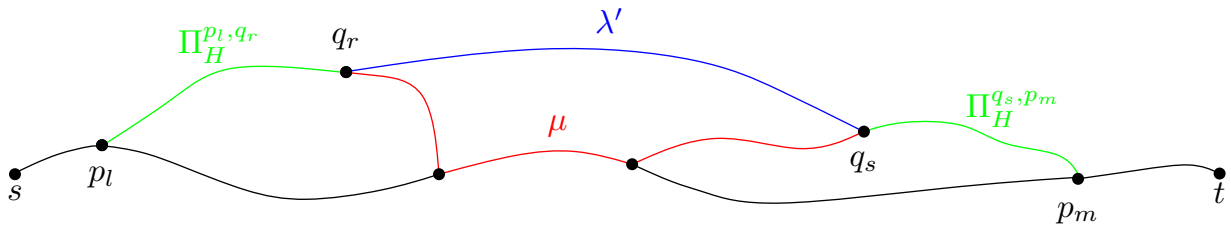
Um einen Alternativpfad berechnen zu können, muss es möglich sein, durch eine Kante den aktuellen Alternativgraphen zu verlassen und wieder zu betreten. D.h. die Summe dieser beiden Kanten darf nicht größer als das Gewicht des kürzesten Weges sein. Aus dieser Beobachtung lässt sich der Faktor $0.5 \cdot d_G(s,t)$ in der Gleichung erklären. Es ist leicht einsehbar, dass ein hoher Wert für $rejoin\text{-}factor$ das Verlassen und Eintreten in den Alternativgraphen erschwert. D.h. mit dem $rejoin\text{-}factor$ haben wir die Möglichkeit die Anzahl der $decisionEdges$ zu reduzieren.

Bader et al. [24] führen an, dass Teile des kürzesten Weges keine sinnvolle Alternative haben. Sie schlagen hier vor, den Pfad zu analysieren, um dann zu entscheiden, ob der Pfad in den Alternativgraphen eingefügt wird, führen aber nicht weiter aus unter welchen Kriterien ein Pfad in den Alternativgraphen eingefügt wird. Insgesamt führt dies zu Algorithmus 1.

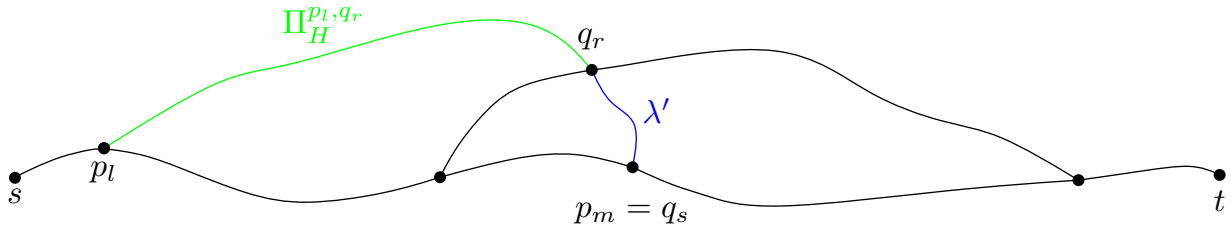
Algorithm 1 Penalty - Method

Input : Graph $G = (V, E), s, t \in V$
repeat
  $Path\ P = shortestPathQuery(s, t)$
  if *P sufficient* **then**
    $addPathToGraph(P, H)$
  if *H was improved and all constraints are met* **then**
    $H_{ret} = H$
  $updateEdges(G, H, P)$

until $\neg((averageDistance \leq 1.1)\ and\ (decisionEdges \leq 10))$;
return H_{ret}



(a) Der Teilpfad λ' stellt eine Alternative zu μ dar.



(b) Der Teilpfad λ' überbrückt zwei Pfade.

Abbildung 3: Visualisierung der Bedingung an Teilpfade.

3 Penalty Methode mit Pfadanalyse

Durch die Kriterien *limited sharing* und *uniformly bounded stretch* von [14] motiviert, haben wir uns für folgende Kriterien zur Analyse des Pfades entschieden: Der kürzeste Weg $\Pi_{s,t}$ aus G wird immer und vollständig in den Alternativgraphen eingefügt. Sei nun $\lambda = \Pi_G^{s,t} = \langle q_1, \ldots, q_n \rangle$ ein aktuell kürzester Pfad, der in den AG $H = (V', E')$ eingefügt werden soll. Wir betrachten die kürzesten Teilpfade $\lambda' = \Pi_G^{q_r,q_s} \subset \lambda$, für die gilt, $q_r, q_s \in V'$ und für alle $i \in \{r+1, \ldots, s-1\}$ gilt, dass q_i nicht in der Knotenmenge V' enthalten ist. Damit stellt λ' einen Teilpfad von λ dar, der bisher noch nicht im Alternativgraph vorkommt. Sei $\mu = \Pi_H^{q_r,q_s}$ und $\lambda_{p_l,p_m} = \Pi_H^{p_l,q_r} \cdot \lambda' \cdot \Pi_H^{q_s,p_m}$ der kürzest mögliche Pfad mit $p_l, p_m \in \Pi_{s,t}$. Der Pfad μ ist der Pfad der von λ' überbrückt wird. Dieser Pfad muss nicht existieren (siehe Abbildung 3(b)). Der Pfad λ_{p_l,p_m} überbrückt die Teilstrecke von p_l nach p_m auf dem kürzesten Pfad. Zur Veranschaulichung der Teilfpade siehe Abbildung 3(a). Die Teilpfade λ' werden in den Alternativgraphen eingefügt, wenn sie bestimmte Qualitätsmerkmale erfüllen.

$$c(\mu) \geq \textit{min-global-factor} \cdot c(\Pi_{s,t}) \qquad \textit{min-global-factor} \in [0,1] \tag{3.1}$$

$$c(\lambda_{p_l,p_m}) \leq \textit{max-global-factor} \cdot c(\Pi_H^{p_l,p_m}) \qquad \textit{max-global-factor} \in [1,\infty] \tag{3.2}$$

$$c(\lambda') \leq \textit{max-local-factor} \cdot c(\mu) \qquad \textit{max-local-factor} \in [1,\infty] \tag{3.3}$$

- Zum einen soll das Teilstück λ' eine echte Alternative darstellen, d.h. es soll einen möglichst langen Pfad überbrücken. Daher wird gefordert, dass das überbrückte Stück μ eine Mindestlänge einhält(siehe Bedingung 3.1).
- Eine alternative Route sollte keinen großen Umwegen darstellen. Dies kann man global fordern, indem die Teilstrecke λ_{p_l,p_m} nicht über eine bestimmte Länge hinausgehen darf (Bedingung 3.2).
- Lokal wird gefordert, dass λ' nur ein Bruchteil länger ist als μ (Bedingung 3.3).

Verbindet λ' zwei parallele Pfade in H, so existiert der Pfad μ nicht. In diesem Fall nehmen wir $c(\mu) = \infty$ an (siehe Abbildung 3(b)).

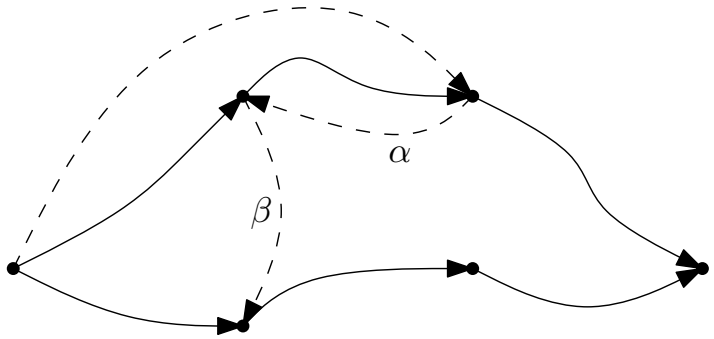


Abbildung 4: Wird der Pfad β nicht mit in den Alternativgraph aufgenommen, ist für die vorhergehende Kante α die einfache-Pfad-Eigenschaft für Alternativgraphen verletzt.

3.1 Zyklen

Angenommen, der aktuelle Alternativpfad λ erzeugt in dem Alternativgraphen einen Zyklus und mindestens ein Teilpfad wird nicht mit in den Alternativgraphen aufgenommen. Dann kann es wie in Abbildung 4 dazu kommen, dass die einfache-Pfad-Eigenschaft für Alternativgraphen verletzt ist. Eine einfache Lösung, die dieses Problem vermeidet, ist, Teilpfade, die einen Zyklus schließen, nicht mit in den Alternativgraphen aufzunehmen.

3.2 Abbruchbedingung

Es gibt Szenarien, in denen keine Alternativen zum kürzesten Weg existieren. Wählt man zum Beispiel den Start und Endpunkt einer Brücke zu einer Insel zu der diese Brücke die einzige Zufahrt ist, dann ändern sich die Werte für *averageDistance* und *decisionEdges* nicht. Bedingung 2.2 wird nie verletzt. In Algorithmus 1 führt dies zu unendlich vielen Iterationen. Daher verschärfen wir die Abbruchbedingung mit einer Obergrenze *limitN* für die Anzahl an Iterationen, in der sich keine Änderung im Alternativgraphen ergeben haben.

Mit diesen Ergänzungen stellt sich das Verfahren wie in Algorithmus 2 dar.

Algorithm 2 Penalty - Method with Path-Analysis

Input : Graph $G = (V, E), s, t \in V$
repeat
  $Path\ P = shortestPathQuery(s, t)$
  devide P into new subpaths q'
  forall the *subpaths* q' **do**
    if *constraints 3.1 - 3.3 are met and* q' *does not close a cycle* **then**
      $addPathToGraph(q', H)$
  if *H was improved and all constraints are met* **then**
    $H_{ret} = H$
  $updateEdges(G, H, P)$
until $\neg$*((averageDistance* ≤ 1.1*) and (decisionEdges* ≤ 10*) and (H changed during the last limitN iterations))*;
$resetEdgeWeights(G)$
return H_{ret}

3.3 Laufzeit

Eine einfache Implementierung von Algorithmus 2 führt eine einfache (bidirektionale) Dijkstra-Suchanfrage für *shortestPathQuery* durch. Nach dem Ändern der Kantengewichte kann direkt die nächste Suchanfrage gestartet werden. Die Laufzeit dieser Variante wird maßgeblich von *shortestPathQuery* bestimmt. Soll der Suchraum weiter eingeschränkt werden, kann für die Suchanfrage einer der in Abschnitt 1.1.1 und 1.1.2 vorgestellten Verfahren eingesetzt werden. Die Änderung der Kantengewichte kann aber bei diesen Methoden aufwendige Neuberechnungen zur Folge haben. Es muss also ein Kompromiss zwischen der Zeit zur Berechnung der kürzesten Wege und der Zeit zum Aktualisieren des Graphen gefunden werden. Da die Änderungen sehr lokal sind, bietet es sich an die Penalty-Methode mit MLD als Grundlage zu implementieren.

4 Schnelle Implementierung

Zur Berechnung von Alternativgraphen kombinieren wir zwei verschiedene bekannte Verfahren. MLD [5] soll als Beschleunigungstechnologie für die Penalty-Methode[24] zur Berechnung von Alternativgraphen dienen.

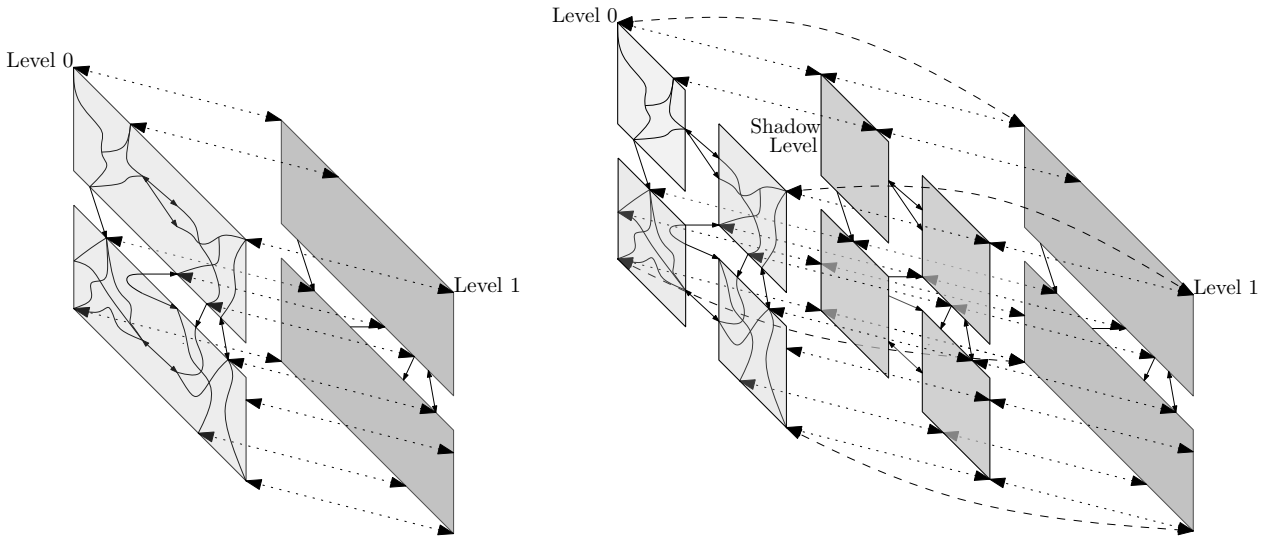
4.1 MLD

Das MLD Verfahren besteht aus drei verschiedenen Phasen: die metrik-unabhängige Partitionierung des Graphen, der metrik-abhängigen Vorberechnung aller kürzesten Pfade innerhalb einer Partition und den eigentlichen Kürzesten-Wege-Anfragen. Im Folgenden gehen wir auf die von uns genutzte Datenstruktur und die Umsetzung Vorberechnung ein. Für die Partitionierung sei auf [4] verwiesen.

4.1.1 Graph-Datenstruktur

Der Graph setzt sich aus dem ursprünglichen Graphen und den Cliquen zusammen, die die kürzesten Wege zwischen den Randknoten einer Partition repräsentieren. Der ursprüngliche Graph wird als Adjazenzarray dargestellt. Delling et al. [5] schlagen vor die Cliquen als Matrizen abzuspeichern in denen nur die Gewichte der kürzesten Wege abgespeichert werden. Existiert kein kürzester Weg, wird dies mit einem *unendlich*-Wert markiert. Zu jeder Matrix wird ein zusätzliches Adjazenzarray benötigt, das auf benachbarte Randknoten verweist, die durch die Kanten im Ursprungsgraphen induziert werden. In dem Array wird ebenfalls eine Kante, mit Gewicht Null, zu den Repräsentanten in das vorherige und nächste Level hinterlegt (siehe Abbbildung 5(a)). Der Ursprungsgraph kann weiterhin als Adjazenzarray abgespeichert werden, der nun für die Randknoten auch Kanten in das nächste Level enthält. Die Knoten werden mit einer natürlichen Zahl aus dem Intervall $[O, N)$ identifiziert, mit N als Anzahl an Knoten im MLD-Graph. Den Knoten auf dem untersten Level werden eine Zahl kleiner als n zu geordnet, wobei n die Anzahl an Knoten in dem ursprünglichen Graphen sind. Um Knoten und den Matrizen ihrem Level zu zu ordnen, werden weitere Abbildung eingesetzt.

Für die bidirektionale Suche wird der Vorwärts- und Rückwärtsgraph benötigt. Eine Möglichkeit, die beiden Graphen zu repräsentieren ist die Vorwärts- und Rückwärtskanten im gleichen Graphen zuspeichern und mit zwei Werten *forward, backward* zu unterscheiden, ob die Kante im Vorwärtsgraph bzw. im Rückwärtsgraph vorkommt. Hierbei sollte davon Gebrauch gemacht werden, dass der Großteil der Kanten im Vorwärts- und Ruckwärtsgraphen auftauchen, eine



(a) Gestrichelte Kanten stellen die Abbildung in das nächste bzw. vorherige Level dar. Durchgehende Linien sind Randkanten zwischen den grau dargestellten Partitionen.

(b) Das mittlere Level stellt das Shadow Level dar. Die gepunkteten Linien sind vor der Suchanfragen versteckt. Die gestrichelten Linien sind traversierbar. (Hinweis: Um die Darstellung nicht weiter zu überlasten, wurden Kanten vom untersten ins mittlere Level weggelassen.)

Abbildung 5: Schematische Darstellung der Partitionierung und ihre Verknüpfung untereinander. Das Level 0 stellt den Ursprünglichengraph dar. Die Partitionierung im folgenden Level ist schon angedeutet.

Kante also mit *forward* sowie *backward* markiert werden kann. Die Rückwärtskanten der Matrizen werden implizit durch das Transponieren der Matrizen geliefert. Ein erneutes Abspeichern dieser Matrizen ist somit nicht nötig.

Delling et al. schlagen zur Beschleunigung der Vorberechnung vor, an unterster Stelle ein weiteres Level in den Graphen einzufügen. Dieses Level soll ausschließlich für die Vorberechnung genutzt werden. Dieses *Shadow Level* kann in unserer Darstellung des Graphen einfach implementiert werden. Für Suchanfragen wird die gleiche *parent*-Funktion genutzt wie sie bei der Partitionierung ohne Shadow Level entstehen würde. Für die Vorberechnung mit Shadow Level wird die *parent*-Funktion genutzt, wie sie durch die Multilevel-Partitionierung berechnet wird. Der Graph selbst wird fast identisch, wie zuvor beschrieben, aufgebaut. Bei Kanten, die in das Shadow Level und aus dem Level hinaus zeigen, werden die Werte *forward, backward* entfernt, somit bleiben diese vor den Suchanfragen verdeckt. Bei der Vorberechnung werden die Kanten als Abbildung genutzt und zum Initialisieren der Suchanfrage genutzt. Ob die Werte *forward, backward* gesetzt sind, beeinflusst die Vorberechnung nicht. Damit die Suchanfragen aber weiterhin korrekt bleiben, müssen weitere Kanten eingefügt werden, die das Shadow Level überspringen (siehe Abbildung 5(b)). Ist ein Knoten ein Randknoten auf dem untersten Level und auch ein Randknoten auf dem ersten Level, dann werden diese mit einer Kante mit Gewicht Null verbunden.

Den Alternativgraphen kodieren wir mit einem Zeitstempel einen Kanten direkt in dem MLD-Graphen. Wird ein zweiter Alternativgraph berechnet, können vorherige Alternativegraphen überschrieben werden. Dies kann einen zusätzlichen Extraktionnsschritt nötig machen.

4.1.2 Vorberechnung

Der Schlüssel für eine schnelle Vorberechnung liegt in der Erkenntnis, dass die kürzeste Wege-Suche für die Randknoten auf die Partition eingeschränkt werden kann ohne die Korrektheit zu verlieren. Für die Vorberechnung des l-ten Level muss nur das $(l-1)$-te Level betrachtet werden. Jedes Level enthält die korrekten Kürzesten-Wege-Distanzen von einem Randknoten zu allen anderen Randknoten. Die Vorberechnung eines Levels kann daher ausschließlich auf dem darunter liegenden Level durchgeführt werden. Möchten wir also die kürzesten-Wege für die Partition P berechnen, müssen nur die Partitionen p betrachtet werden, für die gilt $parent(p) = P$.

Da die Vorberechnung einzelner Partitionen innerhalb eines Levels unabhängig voneinander sind, profitiert diese stark von einer parallelen Vorberechnung. Dazu mehr in Abschnitt 4.2.1.

4.1.3 Suchanfragen

Bei einer s-t-Suchanfrage kann der Suchraum signifikant verkleinert werden. Sei $\mathcal{P}_{s,t}$ die Menge der Partitionen, in denen s oder t enthalten ist, dann ergibt sich der Suchgraph G_S aus allen Partitionen p, für die gilt $parent(p) \in (\mathcal{P}_{s,t} \cup \emptyset)$. Auf diesem Graph kann ohne weitere Anpassung der bekannte Dijkstra-Algorithmus bzw. seine bidirektionale Variante angewendet werden. In der Praxis können die Partitionen $\mathcal{P}_{s,t}$ und der Wächter $\emptyset$ als *aktiv* markiert werden. Ist der Vater einer Partition aktiv, wird die Partition in der Suche betrachtet. Eine einfache Implementierung des Dijkstra-Algorithmus für MLD kann dann wie in Algorithmus 3 dargestellt, aussehen.

Die bidirektionale Suchanfrage wird parallel ausgeführt, in dem die Vorwärts- und Rückwärtssuche in einem eigenständigen Thread durchgeführt wird.

Pfadentpackung Benötigt man den kürzesten Pfad im ursprünglichen Graphen, dann muss der, von der Suchanfrage gefundene, Pfad Π entpackt werden. Dazu muss man im wesentlichen die benachbarten Knoten (v, w) auf den Pfad Π betrachten, die zur gleichen Partition gehöhren. Für diese wird erneut die kürzeste Pfad von v nach w in dem darunter liegenden Level berechnet. Der daraus entstehende Pfad wird nach diesem Schema rekursiv bis zum Ursprungsgraph aufgelöst.

Algorithm 3 Undirectional MLD Query

Input : MLD-Graph $G = (V, E)$, Startnode s, Targetnode t
Mark $\mathcal{P}_{s,t}$ as active
priority queue q
$q.push(s, 0)$
while $q \neq \emptyset$ **do**
  $current = q.getMin$
  $q.deleteMin$
  if $current == target$ **then**
    return $q.getKey(target)$
  forall the $e = (current, target) \in E$ **do**
    if $parent(target)$ *is active* **then**
      if $e.target \notin q$ **then**
        $q.push(e.target, c(e) + q.getKey(current))$
      else if $c(e) + q.getKey(current) < q.getKey(target)$ **then**
        $q.decreaseKey(target, c(e) + q.getWeight(current))$

4.1.4 Partitionsplitting

Wie wir in Abschnitt 5.4.2 sehen werden, steigt mit der Anzahl an Randknoten in unserer Partitionierung auch die Zeit zum Aktualisieren der Partitionen auf dem obersten Level. Daher brechen wir Partitionen auf dem obersten Level mit mehr als l Randknoten auf. Dazu wird der, durch die Partitionierung P aus dem obersten Level induzierte Graph G_P mit Metis [13] in k Partitionen unterteilt. Wobei sich k aus $\min(\lceil \#Randknoten(P)/(2 \cdot l) \rceil, \lceil \#\{p = parent(P)\}/4 \rceil)$ ergibt. Die Anzahl der Partitionen soll als Verhältnis der Randknoten zu l gewählt werden. Dies allein kann dazu führen, dass k größer ist als die Anzahl an Partitionen aus der sich P zusammensetzt. Als Variante wird k in Abhängigkeit von der Anzahl an Partitionen mit P als Vater-Partition gewählt. Die Faktoren 2 bzw. 4 haben bei unseren Tests zu guten neuen Partitionierungen geführt.

Definition 4.1. *Der ungerichtete Graph $G_P = (V_P, E_P, c_P)$ wird durch den Graphen $G = (V, E, c)$ und die Partition P induziert und bildet sich wie folgt.*

$$
\begin{aligned}
V_P &= \{p \in \mathcal{P} \mid parent(p) = P\} \\
E_P &= \{(p_1, p_2) \mid p_1, p_2 \in V_P \wedge \exists (u, v) \in E : u \in p_1 \wedge v \in p_2\} \\
c_P((p_1, p_2)) &= \sum_{e \in p_1 \times p_2 \cap E} c(e) + \sum_{e \in p_2 \times p_1 \cap E} c(e)
\end{aligned}
$$

Der von P induzierte Graph G_P spiegelt das Verhältnis zwischen den Partitionen wider. Partitionen werden als Knoten dargestellt. Existiert eine Randkante im ursprünglichen Graph, die zwischen den Partitionen p_1 und p_2 verläuft, dann existiert auch eine Kante zwischen diesen beiden Partitionen im induzierten Graphen. Da im Allgemeinem mehrere Randkanten zwischen von p_1 zu p_2 und umgekehrt verlaufen, wird als Kantengewicht die Summe der Kantengewichte von diesen Kanten gewählt.

4.2 MLD zur Berechnung von Alternativgraphen

Multilevel-Dijkstra ist auf Grund der lokalen Vorberechnung besonders geeignet, um als Beschleunigungstechnik für die Penalty-Methode zu dienen. Während bei vielen Beschleunigungstechniken Änderungen an den Kantengewichten eine komplett neue Vorberechnung nötig machen, müssen bei MLD nur die Partitionen aktualisiert werden in denen Kanten liegen, die verändert wurden.

Der konzeptionelle Algorithmus 2 muss nur an wenigen Stellen an MLD angepasst werden. Anstatt des Ursprungsgraphen wird der *MLD Graph* mit den Partitionen übergeben. Dijkstras Algorithmus wird durch die bidirektionale Version für den MLD-Graphen ersetzt. Die aufwendigste Änderung steckt in der Funktion *updateEdges*, hier müssen zunächst die Kantengewichte auf dem ursprünglichen Graph angepasst werden (Abschnitt *update edges on path* und *update rejoin edges* in Algorithmus 4). Da der Pfad im Allgemeinen über mehrere Partitionen hinweg verlaufen wird, müssen auch alle Gewichte der Randkanten in allen Leveln aktualisiert werden. Schließlich müssen *bottom-up*, wie bei der Vorberechnung des Graphen, alle kürzesten Wege der Partitionen auf allen Leveln aktualisiert werden, in denen eine geänderte Kante liegt (Abschnitt *MLD-update* in Algorithmus 4).

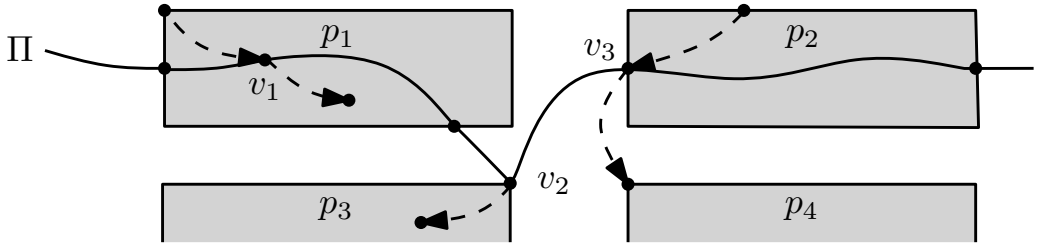


Abbildung 6: Visualisierung der traversierten Partitionen und der Rejoin-Kanten. v_1 liegt innerhalb einer Partition und kann somit keine Kante in einer anderen Partition manipulieren. v_2 ist ein Randknoten und hat eine Kante die in p_3 zeigt, allerdings liegt v_2 selber schon in p_3. v_3 liegt in p_2 und hat eine Kante die zu p_4 zeigt. Es wird allerdings keine Kante in p_4 geändert. p_4 muss somit nicht aktualisiert werden.

Satz 4.2. *Es müssen nur die Partitionen $\mathcal{P}_{update}$ aktualisiert werden, in denen ein Knoten des Pfades Π liegt.*

$$\mathcal{P}_{update}(\Pi) = \{p \in \mathcal{P} \mid \exists v \in \Pi : v \in p\}$$

Beweis. Wird die *rejoin-penalty* außer acht gelassen, ist leicht einsehbar, dass die Behauptung korrekt ist. Es werden nur Kanten $e = (u, v)$ auf dem Pfad geändert, folglich müssen die Partition $p(v)$ und $p(u)$ aktualisiert werden und damit alle Partitionen von denen ein Knoten auf dem Pfad liegt. Hier können sogar zu viele Partitionen aktualisiert werden. Liegt zum Beispiel der Startknoten am Rand einer Partition, allerdings keine Kante des Pfades, dann wird diese Partition aktualisiert, obwohl keine Änderung in der Partition vorgenommen wurde.

Zu zeigen bleibt nun, dass die *rejoin-penalty* die Menge $\mathcal{P}_{update}$ nicht ändert. Dafür unterscheiden wir folgende Fälle. Sei $v \in \Pi$ ein Knoten des Pfades Π

v ist kein Randknoten Wir betrachten alle Kanten $e = (u, v)$ bzw. $e = (v, u)$. Angenommen es gilt $p(u) \neq p(v)$, dann ist e eine Randkante und v somit ein Randknoten. Somit kann in diesem Fall kein Kantengewicht außerhalb von p geändert werden. (siehe v_1 in Abbildung 6)

v ist ein Randknoten Es existiert mindestens eine Kante $e = (u, v)$ oder $e = (v, u)$ mit $p(v) \neq p(u)$. Ohne Beschränkung der Allgemeinheit lassen wir hier die Unterscheidung fallen, ob e eine *Rejoin-Kante* ist (oder Pfad auf dem Pfad liegt). Die beschriebene Kante ist eine Randkante und liegt somit in keiner Partition. Damit ändert das Aktualisieren der Kante keinen kürzesten Weg innerhalb der Partition $p(u)$. Die Menge $\mathcal{P}_{update}$ ändert sich also nicht. Für weitere Kanten mit $p(u) = p(v)$ ändert sich zwar ein Kantengewicht innerhalb einer Partition, allerdings ist $p(u)$ schon in $\mathcal{P}_{update}$ enthalten.(siehe v_2 und v_3 in der Abbildung 6)

Damit ist gezeigt, dass die *rejoin-penalty* keinen Einfluss auf die Menge $\mathcal{P}_{update}$ hat.

□

Algorithm 4 updateEdges

Input : Graph $G = (V, E), H = (V', E')$, Path Π
list of partitions $P_{update} = \emptyset$
`// update edges on path`
forall the *edges* $e \in \Pi$ **do**
  update edge weights
  $P_{update} = P_{update} \cup parent(u) \cup parent(v)$
  if *e is boundary edge* **then**
    update edges in upper layers
`// update rejoin edges`
forall the $v \in \Pi$ **do**
  $RejoinEdges = \{(v, w) \in E\} \cup \{(w, v) \in E\}$
  forall the $e \in RejoinEdges$ **do**
    update edge weights
    if *e is boundary edge* **then**
      update edges in upper layers
`// MLD-update`
while $P_{update} \neq \emptyset$ **do**
  list of partitions $P'_{update} = \emptyset$
  forall the $p \in P_{update}$ **do**
    update shortest paths in p
    $P'_{update} = P'_{update} \cup parent(p)$ `// parent (p) = ∅ on top layer`
  $P_{update} = P'_{update}$

4.2.1 Parallelisierung

Die Aktualisierung von Partitionen P_{update} aus einem Level l lässt sich *extern* und *intern* parallelisieren. Die externe Methode (Algorithmus 5) nutzt aus, dass die Partitionen $\mathcal{P}_l$ innerhalb eines Levels l unabhängig voneinander berechnet werden können. Die Partitionen P_{update} werden in n disjunkte Teilmengen $P_{update,i}$ mit $\cup_{i=1}^{n} P_{update,i} = P_{update}$ unterteilt. Jede dieser Teilmenge kann unabhängig voneinander betrachtet werden und bietet somit den Ausgangspunkt für eine parallele Berechnung. Die genaue Anzahl der Partitionen n kann größer als die Anzahl der Threads sein, so dass einem Thread eine neue Teilmenge zugewiesen wird, sobald dieser eine Teilmenge komplett abgearbeitet hat.

Algorithm 5 parallel execution of MLD-update section in updateEdges(extern)

Input : Graph $G = (V, E)$
while $P_{update} \neq \emptyset$ **do**
  list of partitions $P'_{update} = \emptyset$
  parallel forall the $p \in P_{update}$ **do**
    update shortest paths in p
    $P'_{update} = P'_{update} \cup parent(p)$ `// parent(p) = ∅ on top layer`
  $P_{update} = P'_{update}$

Bei der internen Methode (Algorithmus 6) werden nicht die Partitionen auf einem Level betrachtet, sondern eine einzelne Partition P mit ihren Randknoten B. Die Randknoten B werden in n disjunkte Teilmengen B_i unterteilt mit $\cup_{i=1}^{n} B_i = B$. Alle kürzeste-Wege-Anfragen von den Randknoten aus beeinflussen sich nicht. Zum Berechnen der Matrizen auf dem l-ten Level werden die Werte dieser Matrizen nicht benötigt. Die j-te Zeile einer Matrix aus dem l-ten Level,

ist dem Randknoten b_j zugeordnet. Diese wird nur bei der Berechnung der kürzesten Wege von b_j gefüllt.

Algorithm 6 update shortest paths(parallel, intern)

Input : Graph $G = (V, E)$, Partition P
B = boundary nodes of P in on level below
parallel forall the $b \in B$ **do**
    run shortest-path-query from b on nodes v with $parent(p(v)) = P$
    forall the $b' \in B$ **do**
        update Matrix-Edge (b, b') to new weight, according to the shortest-path-query

5 Experimente

Die einzelnen Komponenten des Algorithmus werden auf zwei gleichwertigen Rechnern mit einem Intel(R) Core(TM) i7 920 Prozessor mit vier echten Kernen (acht mit Hyperthreading), (2.67GHz), 12GB DDR3-1333 RAM untersucht. Das Programm wurde mit g++ (SUSE Linux) 4.5.0 und den Parametern `-std=c++0x -fopenmp -O3 -DNDEBUG -mtune=native` kompiliert. Zur Parallelisierung wird OpenMP eingesetzt. Das Programm wurde mit acht Threads ausgeführt. Es wurde der Linux-Kernel in Version 2.6.34.10-0.6-default genutzt.

5.1 Graphen und Partitionierung

Zum Testen des Algorithmus kommt ein Straßennetzwerk von Europa zum Einsatz, das von der PTV-AG [20] zur *9th DIMACS Implementation Challenge* [2] veröffentlicht wurde. Der Graph enthält 18 Millionen Knoten und 44 Millionen Kanten. Als Gewichtsfunktion wurde die Reisezeit betrachtet. Die Partitionierung wurde mit Punch [4], wie von Delling et al. in [5] angegeben, mit dem Parameter u (maximale Anzahl der Knoten in einer Partition) $[2^8, 2^{12}, 2^{16}, 2^{20}]$ und für das Shadow Level mit 2^5, berechnet.

Wir haben insgesamt drei Varianten an Partitionierungen (*AG-MLD-4, AG-MLD-4-S, AG-MLD-4-S-Splitted*), die zur Untersuchung herangezogen werden. Die ersten beiden Varianten werden wir nutzen um unsere Implementierung von MLD mit der von Delling et al. zu vergleichen. Die Varianten AG-MLD-4-S und AG-MLD-4-S-Splitted nutzen wir um einen Laufzeitvorteil für die Alternativgraphberechnung zu zeigen. Die Evaluation und weitere Optimierungen werden mit AG-MLD-4-S-Splitted durchgeführt.

Allen drei Varianten liegt die gleiche Partitionierung zugrunde. Die Partitionierung AG-MLD-4-S(-Splitted) verwendet im Gegensatz zu AG-MLD-4 ein Shadow Level. Alle anderen Level in AG-MLD-4-S und AG-MLD-4 sind identisch. Die Partitionierung AG-MLD-4-S-Splitted unterscheidet sich von AG-MLD-4-S nur im obersten Level. Bei dieser Partitionierung wurden nach Abschnitt 4.1.4 alle Partitionen aufgesplittet, die mehr als $l = 100$ Randknoten besitzen (siehe Abschnitt 5.4.2). Das sind insgesamt 45% aller Partitionen auf dem obersten Level.

Tabelle 1 gibt eine Übersicht über die Eigenschaften der Partitionierungen in jedem Level. Die Anzahl an Randknoten bezieht sich einmal auf die Anzahl der Randknoten zu einer Partition P und einmal auf die Summe der Anzahl an Randknoten auf dem darunter liegenden Level mit der gleichen Vater-Partition. Für das erste Level sind die Anzahl an Knoten in einer Partition eingetragen. Analog verhält es sich mit der Anzahl an Partitionen. Für die Partitionierung AG-MLD-4 existieren also 20 Partitionen auf dem obersten Level. Im Schnitt besteht jeder dieser Partitionen aus 16 Partitionen in dem Level darunter. Die Partitionierung AG-MLD-4

Attribut	Partitionierung					
Maximale # Knoten		2^5	2^8	2^{12}	2^{16}	2^{20}
#Partitionen	AG-MLD-4	-	82278	5046	319	20
	AG-MLD-4-S	664300	82278	5046	319	20
	AG-MLD-4-S-Splitted	664300	82278	5046	319	46
Maximale #Randknoten pro Partition	AG-MLD-4	-	55	96	208	384
	AG-MLD-4-S	21	55	96	208	384
	AG-MLD-4-S-Splitted	21	55	96	208	415
Mittlere #Randknoten pro Partition	AG-MLD-4	-	10	26	68	137
	AG-MLD-4-S	5	10	26	68	137
	AG-MLD-4-S-Splitted	5	10	26	68	182
Maximale #Partitionen mit gleicher Vater-Partition	AG-MLD-4	-	-	22	20	19
	AG-MLD-4-S	-	12	22	20	19
	AG-MLD-4-S-Splitted	-	12	22	20	19
Mittlere #Partitionen mit gleicher Vater-Partition	AG-MLD-4	-	-	16	16	16
	AG-MLD-4-S	-	8	16	16	16
	AG-MLD-4-S-Splitted	-	8	16	16	7
Maximale #(Rand)knoten in gleicher Vater-Partition	AG-MLD-4	-	256	474	966	2038
	AG-MLD-4-S	32	131	474	966	2038
	AG-MLD-4-S-Splitted	32	131	474	966	1140
Mittlere #(Rand)knoten in gleicher Vater-Partition	AG-MLD-4	-	219	163	418	1083
	AG-MLD-4-S	27	42	163	418	1083
	AG-MLD-4-S-Splitted	27	42	163	418	471

Tabelle 1: Eigenschaften der verschiedenen Partitionierungen. Spalte 2^5 stellt das Shadow Level für AG-MLD-4-S(-Splitted) dar, daher sind hier für AG-MLD-4 keine Werte eingetragen. Für die Anzahl der (Rand-)Knoten mit gleicher Vater-Partition wurden in dem ersten Level (2^8 für AG-MLD-4, sonst 2^5) die Anzahl der Knoten in einer Partition akkumuliert.

hat im ersten Level maximal 256 Knoten und im Mittel 219 Knoten. Diese Partition teilt sich im AG-MLD-4-S in 664300 Partitionen auf mit im Mittel 27 Knoten. Die Anzahl an Partitionen auf dem obersten Level erhöht sich von 20 auf 46 für die Partitonierung AG-MLD-4-S-Splitted, im Gegensatz zu AG-MLD-4-S. Durch das Partitionieren des Graphen G_P in Abschnitt 4.1.4 wird das Gewicht der Schnittkanten optimiert, dies muss aber keine Verbesserung der Anzahl an Randknoten ergeben. Dies tritt bei der Partitionierung AG-MLD-4-Splittet auch ein. Es erhöhen sich die Anzahl an Randknoten im Mittel, sowie im Maximum, dafür kann etwa eine Halbierung der Anzahl an Randknoten in der gleichen übergeordneten Partition erreicht werden.

5.2 Test-Mengen und Test-Konfiguration

Es wurden zwei Test-Mengen generiert. Die *s-t-Menge* besteht aus insgesamt 10 000 zufällig ausgewählten Knotenpaaren aus dem Europa-Graphen. Die *Dijkstra Rang* Analysen basieren auf 1 000 zufälligen Dijkstra-Suchanfragen. Wird ein Knoten v als 2^i-tes, mit $i \in \{10, \ldots, 24\}$, in einer Suchanfrage von s aus betrachtet, dann stellt (s, v) die Eingabe für den zu analysierenden Algorithmus zu *Dijkstra Rang* i dar. Um Nebeneffekte zu vermeiden wurden diese Knotenpaare

mit `std::random_shuffle` zufällig permutiert .

$$\begin{array}{llll} \textit{rejoin-penalty} & (RP) & = & 0.4 \\ \textit{rejoin-factor} & (RF) & = & 0.01 \\ \textit{min-global-factor} & (minGF) & = & 0.1 \\ \textit{max-global-factor} & (maxGF) & = & 1.3 \\ \textit{max-local-factor} & (minLF) & = & 1.3 \\ limitN & & = & 15 \end{array} \tag{5.1}$$

Wenn nicht anders aufgeführt, wird die Standardkonfiguration 5.1 für die Berechnung von Alternativgraphen genutzt. Für Tests können einzelne Parameter variiert werden. Wird keine weitere Aussage über die verbleibenden Parameter gemacht, bleiben diese wie in der Standardkonfiguration gewählt.

5.2.1 Auswertung

$$objectiveFunction := totalDistance - (averageDistance - 1) \tag{5.2}$$

Die Qualität der Alternativgraphen wird mit $\alpha = 1$ für die Zielfunktion ausgewertet. Für die Zielfunktion ergibt sich also Formel 5.2 [24].

5.3 MLD

Im Folgenden sollen die Implementierung von Delling et al. unserer Implementierung von MLD gegenübergestellt werden. Die Daten zu dem Graphen MLD-4 stammen aus [5]. Dieser Graph ist eine MLD Variante eines Europa Straßennetzwerkes aus der *9th DIMACS Implementation Challenge*[2] mit 18 Millionen Knoten und 42 Millionen Kanten. In unserem Graphen werden parallele Kanten zugelassen, da dies für in Straßennetzwerken auch eine zulässige Eingabe darstellt. Dies kann der Grund für unseren etwas größeren Graphen sein. Die Messungen für MLD-4 wurden auf einem vergleichbaren Rechner durchgeführt. Die Tests für die Anfragen basieren 10 000 zufälligen Knotenpaaren. Die Vorberechnung nutzt ein Shadow Level, dieses wird aber nicht in dem benötigten Speicher für die Matrizen mit aufgeführt, da das Shadow Level nur für die Vorberechnung aber nicht für die kürzesten-Wege-Anfragen relevant ist. Für die Berechnung der Alternativgraphen wird das Shadow Level hingegen zur Aktualisierung in jeder Iteration benötigt. Der Speicherverbrauch für die Matrizen des Shadow Levels sind bei AG-MLD-4-S(-Splitted) also mit enthalten.

Um einen direkten Vergleich der Implementierung von Delling et al. (MLD-4) mit unserer Implementierung zu ermöglichen, haben wir in Tabelle 2 eine neue Variante AG-MLD-Hybrid eingefügt. Die Werte für AG-MLD-Hybrid setzen sich aus den Varianten AG-MLD-4 und AG-MLD-4-S aus Tabelle 3 zusammen. Die Zeit zur Vorberechnung konnte um 0.3 Sekunden von 4.7 Sekunden auf 4.4 Sekunden reduziert werden. Durch das Aufsplitten von Partitionen kann eine weitere Verbesserung der Vorberechnungzeit um 0.5 Sekunden erzielt werden (Tabelle 3).

Die Anfragezeiten werden um einen Faktor 3.6 langsamer. Die Partitionierung und der leicht andere Graph können sich auf die Anzahl der gescannten Knoten auswirken, die mit 4277 Knoten etwa um den Wert 400 höher liegt gegenüber MLD-4. Wie sich im Vergleich mit AG-MLD-4-Splitted zeigt (Tabelle 3), werden die Anfragezeiten noch einmal um den Faktor 1.58 langsamer. Die Anzahl an gescannten Knoten wird allerdings nur um einen Faktor 1.17 größer. Dies legt die Vermutung nahe, dass das Scannen der Partitionen in unserer Implementierung

Algorithmus	Vorberechnung [s]	Matrizen [MB]	Anfragezeit [ms]	gescannte Knoten
MLD-4	4.7	59.9	0.72	3828
AG-MLD-Hybrid	4.4	66.5	2.59	4277

Tabelle 2: Vergleich der MLD-Implementierung von Delling et al. [5] mit unserer Implementierung.

Algorithmus	Vorberechnung [s]	Matrizen [MB]	Speicher [MB]	Anfragezeit [ms]	gescannte Knoten
AG-MLD-4	5.4	66.5	676.0	2.59	4277
AG-MLD-4-S	4.4	149.3	1102.5	2.57	4276
AG-MLD-4-S-Splitted	3.9	154.7	1108.4	4.07	5497

Tabelle 3: Vergleich zwischen unseren einzelnen Varianten der Partitionierung.

einen Flaschenhals darstellt. Die Suchanfragen wurden allerdings nicht weiter optimiert, da die Alternativgraphberechnung stark von der Zeit zum Aktualisieren der Partitionen dominiert wird (siehe Abbildung 11).

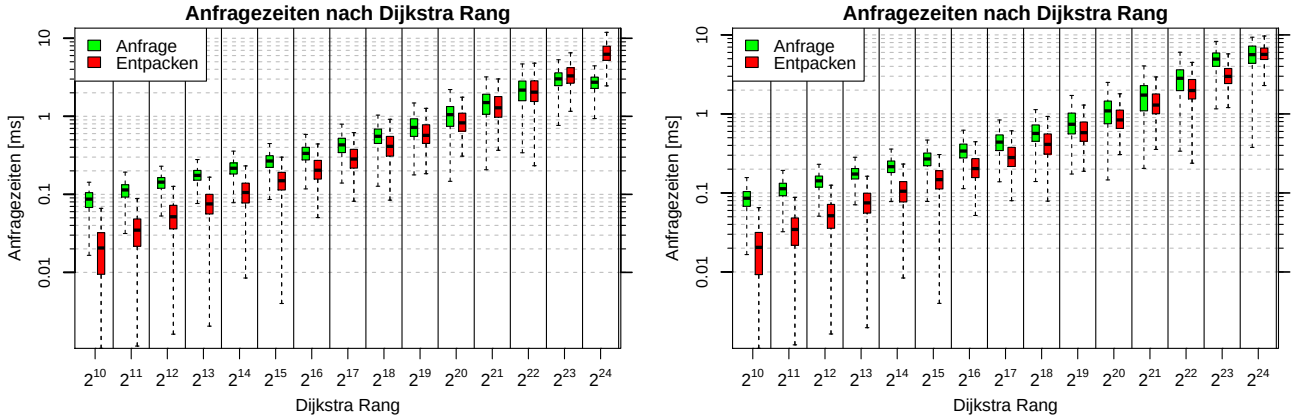
Der zusätzliche Speicher für die Matrizen für AG-MLD-Hybrid liegt bei uns etwa 7MB über dem Wert von MLD-4 (Tabelle 3). Dies kann an einem leicht anderen Graphen und einer unterschiedlichen Partitionierung liegen. Für die Alternativgraphberechnung werden insgesamt 676MB bzw. 1.1GB zusätzlicher Speicher benötigt. Dieser zusätzliche Speicher wird für die Matrizen und verschiedenen Abbildungen benötigt, die in dem Ursprünglichengraphen nicht nötig waren.

Zur Parallelisierung wurde der externe Ansatz gewählt, wie in [5] vorgeschlagen.

5.3.1 Suchanfragen nach Dijkstra Rang

Abbildung 7 zeigt die Zeiten der Suchanfragen und Pfadentpackung nach Dijkstra Rang. Die hervorgehebone Linie in der Box stellt den *Median* dar. Die obere bzw. untere Grenze der Box entspricht dem oberen bzw. unteren Quartil. Der größte Wert der weniger als das 1.5 fache des Quartilsabstandes von dem oberen Quartil entfernt ist, stellt die obere Grenze der gestrichelten Linie dar. Analog verhält es sich mit der unteren Linie und dem unteren Quartil. Kreise deuten Ausreißer an. Also solche Werte, die nicht mehr innerhalb der zuvor definierten Grenzen liegen. Für die Analyse der Ränge wurden die Ausreißer nicht mit geplottet, da sie uns keine weiteren wichtigen Informationen zum Verhalten des Verfahrens liefern.

In beiden Abbildungen wird bei 2^{18} die Millisekunden-Grenze das erste Mal überschritten. Die Partitionierung AG-MLD-4-S erreicht das Maximum der Suchanfragen bei 2^{23}. Die zweite Variante erreicht das Maximum erst bei 2^{24}. Die maximale Zeit für Suchanfragen verdoppelt sich nahezu von etwa fünf Millisekunde auf etwa neun Millisekunde. Erste Änderungen der Laufzeiten lassen sich ab etwa 2^{16} beobachten. Der Median sowie das Maximum sind für AG-MLD-4-S-Splitted leicht nach oben versetzt. Das bedeutet, dass ab diesem Rang erste Anfragen das oberste Level zur Berechnung der kürzesten Wege mit nutzen müssen. Eine deutliche Veränderung kann ab 2^{21} festgestellt werden, die Maximalzeit verschiebt sich um eine Millisekunde nach oben, der Median rückt näher an die eine Millisekunde. Das Minimum bleibt unverändert. Der auffälligste Unterschied ist bei 2^{24}. Die Laufzeit von AG-MLD-4-S wird sogar leicht bes-



(a) Anfragezeiten für den Graphen *AG-MLD-4-S*.

(b) Anfragezeiten für den Graphen *AG-MLD-4-S-Splitted*. Das Aufspallten macht sich besonders bei langen Pfaden bemerkbar.

Abbildung 7: Anfragezeiten der bidirektionalen Dijkstra-Suchanfragen.

ser, im Gegensatz zum vorherigen Rang. AG-MLD-4-Splitted benötigt hingegen noch einmal mehr Laufzeit und kommt fast an die zehn Millisekundengrenze. Im Median bleibt AG-MLD-4-Splitted zwischen fünf und sechs Millisekunden. Dies ist allerdings etwa doppelt so viel wie AG-MLD-4-S im Median benötigt. Diese Änderungen lassen sich durch die höhere Anzahl an Partitionen erklären. Wurde eine Partition auf dem obersten Level gesplittet, dann konnten die Distanzen mit AG-MLD-4-S direkt aus der Matrix gelesen werden. Die Partitionierung AG-MLD-4-S-Splitted muss für die gleiche Anfrage mehrere Partitionen scannen. Dies bekräftigt auch noch einmal die Vermutung, dass sich die steigende Anzahl an Partitionen negativ auf unsere Implementierung des bidirektionalen Dijkstra auswirkt. Erst mit langen Pfaden muss das oberste Level gescannt werden, auf dem sich nun die Anzahl an Partitionen erhöht hat. Ab diesem Zeitpunkt werden die Anfragen zum Vergleich zu AG-MLD-4-S langsamer.

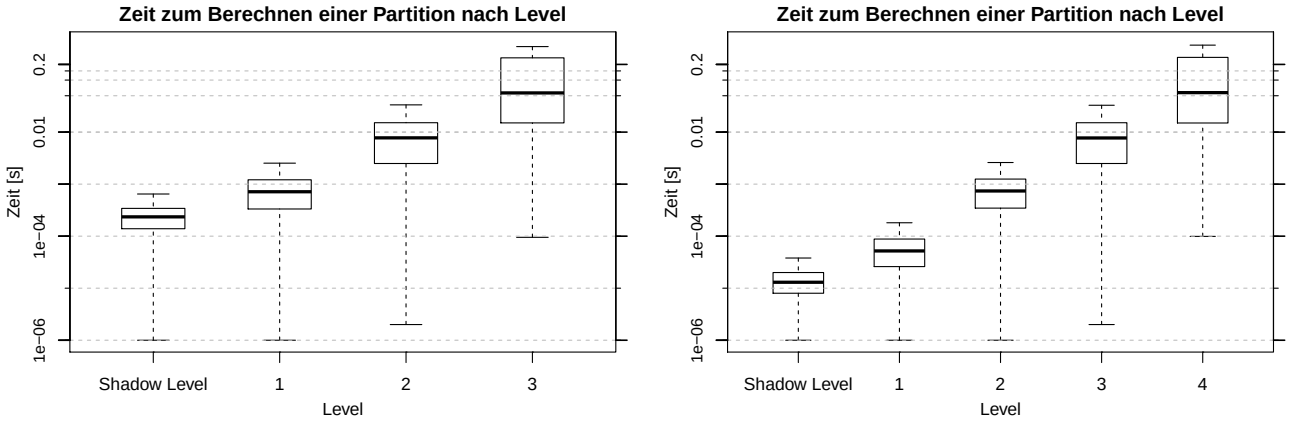
Delling et al. geben an, dass die Pfadentpackung die Anfragezeiten weniger als verdoppelt [5]. Mit Abbildung 7(a) kann die Aussage nachvollzogen werden. Die Zeit zum Entpacken selbst bleibt immer unterhalb der eigentlichen Anfragezeit, mit Ausnahme des Werte 2^{24}. Die Entpackung der Pfade bei der Partitionierung AG-MLD-4-S benötigt mehr Zeit als die Anfragezeit. Der Median benötigt mehr Zeit als das Maximum der Anfragezeit. Diese Auswirkung ist bei der Partitionierung AG-MLD-4-Splitted nicht zu erkennen. Der Median und das Maximum benötigt für die Anfrage und Pfadentpackung die gleiche Zeit.

5.4 Optimierung

Bei der Alternativgraphberechnung werden nur die Partitionen aktualisiert, durch die der Pfad läuft. Es muss also nur ein Bruchteil der Partitionen auf einem Level aktualisiert werden. Im Folgenden soll untersucht werden, ob die Techniken, die die Vorberechnung beschleunigt haben, auch einen Vorteil für die Alternativgraphberechnung darstellen und welche Techniken eine weitere Beschleunigung erwirken. Wir betrachten daher vorerst nur die Zeiten zum Berechnen einer Partition nach Level getrennt.

5.4.1 Shadow Level

Das Shadow Level konnte schon erfolgreich eingesetzt werden, um die Vorberechnung zu beschleunigen (Tabelle 3). Abbildung 8 zeigt, wie sich das Shadow Level auf die Zeit zum Aktualisieren der Partitionen aus dem ersten Level auswirkt. Das Shadow Level verschiebt die Vorberechnungszeit des oberen Quartils vom ersten Level auf unter 10^{-4} Sekunden. Die Anzahl der Partitionen bleibt bei diesen Level identisch. Allerdings muss für die Vorberechnung des ersten Levels von AG-MLD-4-S noch zusätztlich das Shadow Level vorberechnet werden. Im Shadow Level handelt es sich um kleine lokale Graphen, auf denen die Suchanfragen sehr schnell beantwortet werden können. Tabelle 3 zeigt, dass die Arbeit für das Shadow Level und für das erste Level von AG-MLD-4-S zusammen geringer ist als die Arbeit für das erste Level von AG-MLD-4 .



(a) Aktualisierungszeiten pro Partition für die Partitionierung *AG-MLD-4*.

(b) Aktualisierungszeiten pro Partition für die Partitionierung *AG-MLD-4-S*.

Abbildung 8: Zeit zum Vorberechnen einzelner Partitionen mit und ohne Shadow Level.

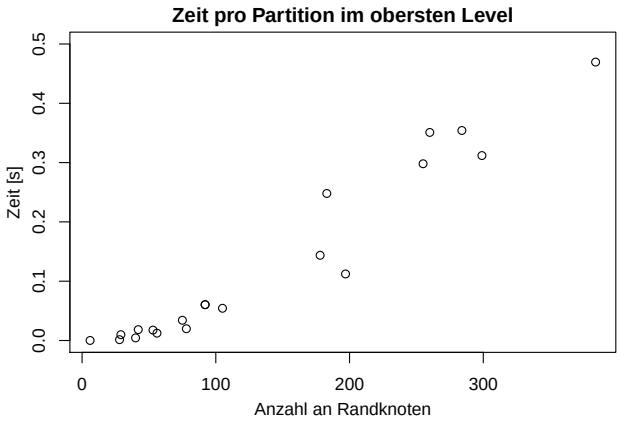
5.4.2 Partitionsplitting

Wie das Shadow Level, erhöht auch das Partitionsplitting die Anzahl der Partitionen. Die Partitionierung AG-MLD-4-Splitted hat in etwa doppelt so viele Partitionen wie AG-MLD-4 auf dem höchsten Level (siehe Tabelle 1). Abbildung 9 verdeutlicht, dass mit steigender Anzahl an Randknoten auf dem obersten Level von AG-MLD-4-S auch die Zeit zum Aktualisieren einer Partition sowie die mittlere Anzahl gescannter Knoten steigt.

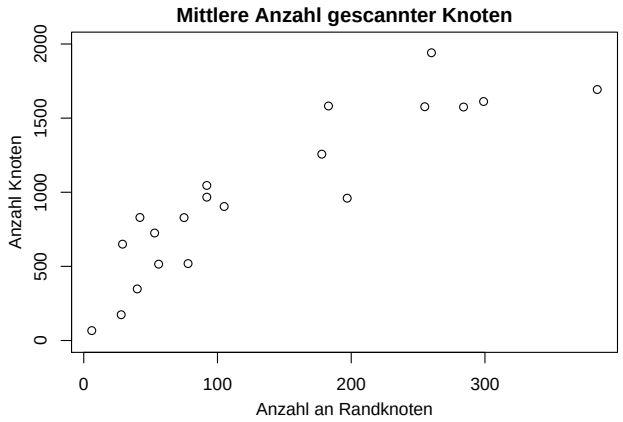
Abbildung 9(a) verdeutlicht, dass Partitionen mit weniger als 100 Randknoten weniger als 100ms Berechnungszeit benötigen. Für mehr Randknoten steigen die Berechnungszeiten auf bis zu 460ms. Ein ähnliches Verhalten zeigt sich für die Anzahl der gescannten Knoten (Abbildung 11(b)). Daher wurde für das Vorgehen aus Abschnitt 4.1.4 die Grenze $l = 100$ zur Berechnung von AG-MLD-4-S-Splitted gewählt.

Das Aufsplitten der Partitionen reduziert die maximale Berechnungszeit von 460ms auf 140ms (Abbildung 10(a)). Die mittlere Anzahl gescannter Knoten pro Dijkstra-Suche von einem Randknoten aus bleibt für alle Partitionen mit mehr als 100 Randknoten unter 800(Abbildung 10(b)). Dieser Wert lag zuvor zwischen etwa 1000 und 2000, diese Knoten werden nun auf die 26 zusätzlichen Partitionen aufgeteilt.

Die Abbildung 11 stellt die Berechnungszeiten für die Partitionierung AG-MLD-4 und AG-MLD-4-Splitted noch einmal gegenüber. Das Aufsplitten hat keine Auswirkung auf die Parti-

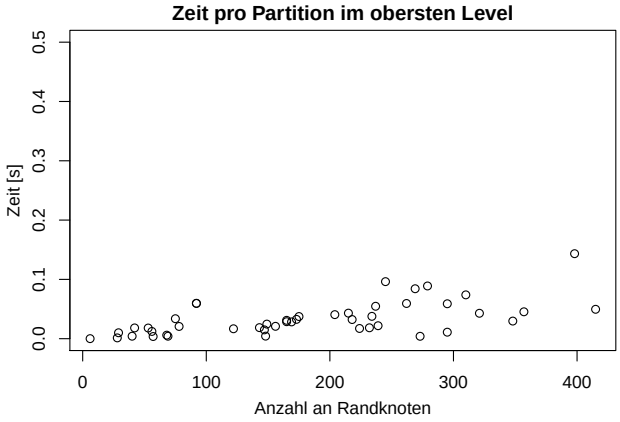


(a) Zeit zum Aktualisieren einzelner Partitionen.

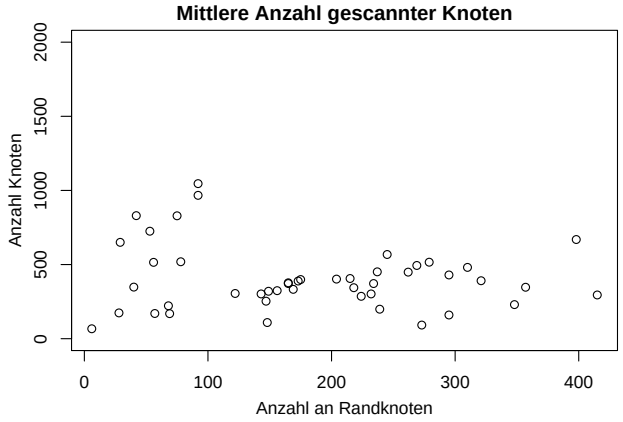


(b) Mittlere Anzahl gescannter Knoten pro Dijkstra-Suche von einem Randknoten.

Abbildung 9: Zeiten und Anzahl der gescannten Knoten für die Partitionierung AG-MLD-4-S.



(a) Zeit zum Aktualisieren einzelner Partitionen.



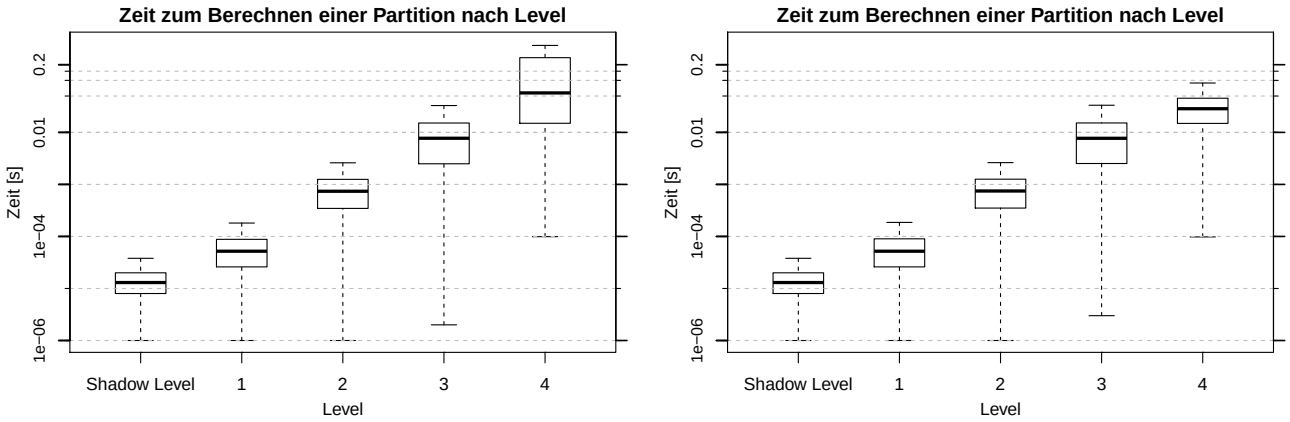
(b) Mittlere Anzahl gescannter Knoten pro Dijkstra-Suche von einem Randknoten.

Abbildung 10: Zeiten und Anzahl der gescannten Knoten für die Partitionierung AG-MLD-4-S-Splitted.

tionen in den ersten vier Level. Auf dem oberen Level ist der Suchraum durch das Partition-Splitting um mindestens ein Faktor 2 kleiner wie zuvor, die einzelnen Suchanfragen können also schneller Beantwortet werden. Dies führt insgesamt sogar zu einer Reduzierung der maximalen Berechnungszeit um einen Faktor von etwa 3.2. Die Anzahl der Partitionen auf dem obersten Level erhöht sicht allerdings nur um den Faktor 2.3. Daher ergibt sich ein echter Laufzeitvorteil durch das Partitionsplitting. Dass das Aufsplitten der Partition keine reine Umverteilung der Arbeit auf mehrere Partitionen darstellt, lässt sich anhand der Vorberechnung feststellen (Tabelle 3). Der Vorteil für die Berechnung von Alternativgraphen zeigt sich in Abschnitt 5.4.4.

5.4.3 Parallelisierung

Die Auswirkung der *internen* und *externen* Parallelisierung auf die Alternativgraphberechnung wurde mit der kompletten s-t-Menge getestet. Der genutzte bidirektionale Dijkstra wurde unabhängig von der Anzahl der Threads parallel ausgeführt. Bei gleichbleibender Qualität ist die interne Methode etwa 20% schneller als im externen Fall (siehe Tabelle 4). Für alle folgenden



(a) Aktualisierungszeiten pro Partition für die Partitionierung *AG-MLD-4-S*.

(b) Aktualisierungszeiten pro Partition für die Partitionierung *AG-MLD-4-S-Splitted*.

Abbildung 11: Zeit zum Aktualisieren einzelner Partitionen.

Methode	Zeit[s]	relativer Speedup
Ohne	3.379	1
Intern	0.829	4.076
Extern	1.035	3.265

Tabelle 4: Zeit zum Berechnen von Alternativgraphen mit *interner* und *externer* Parallelisierung.

Tests zur Berechnung von Alternativgraphen wird die interne Methode genutzt.

Die bessere Zeit der internen Methode kann durch eine bessere Auslastung der Prozessoren erklärt werden. Die Arbeit zum Aktualisieren einzelner Partitionen kann stark variieren. Bei der externen Methode kann es vorkommen, dass einem Thread mehr Arbeit zugeordnet wird als den anderen Threads. Die Bearbeitung des nächsten Level muss solange warten, bis alle Threads ihre Berechnungen beendet haben. Bei der internen Methode ist der Suchraum für alle Threads identisch, daher ist zu erwarten, dass alle Threads kurz nacheinander ihre Berechnung beenden.

Dass die einzelnen Partitionen getrennt voneinander berechnet werden können, wirkt sich direkt auf die Parallelisierbarkeit aus (Tabelle 5). Eine Verdopplung der Anzahl an genutzter Kerne hat auch fast eine Halbierung der benötigten Rechenzeit zur Folge. Da der genutzte Prozessor nur vier echte Kerne hat, lässt sich mit acht Threads leider keine weitere Halbierung der Rechenzeit erwirken. Die Zeit kann trotzdem um noch einmal 12.5ms verbessert werden.

# Threads	Zeit [s]	relativer Speedup
1	3.379	1
2	1.729	1.955
4	0.954	3.543
8	0.829	4.076

Tabelle 5: Benötigte Zeit bei Variation an Anzahl der Threads bei interner Parallelisierung. Bei insgesamt acht Threads wird das Hyperthreading des Prozessors ausgenutzt.

Partitionierung	$totalDist.$	$avgDist.$	$dec.Edges$	$objFct.$	Zeit [s]
AG-MLD-4	2.941	1.048	7	2.892	1.462
AG-MLD-4-S	2.940	1.048	7	2.892	1.418
AG-MLD-4-S-Splitted	2.940	1.048	7	2.892	0.829

Tabelle 6: Auswirkung der Partitionierung auf die Berechnungszeit von Alternativgraphen. Hinweis: Die Schwankung in der *totalDistance* von AG-MLD-4 zu AG-MLD-4-S kann durch eine leicht andere Pfadauswahl erklärt werden.

Konfiguration	$penalty\text{-}factor$	$rejoin\text{-}penalty$
AG-04	0.4	0.0020
AG-03	0.3	0.0015

Tabelle 7: Ausgangskonfigurationen

5.4.4 Zusammenfassung der Optimierungen

Bisher wurde nachgewiesen, dass das Shadow Level und das Partitionsplitting eine Beschleunigung der Vorberechnung erreichen (Tabelle 3). Bei der Vorberechnung konnte mit dem Shadow Level eine Beschleunigung um dem Faktor 1.2 für den Europa-Graph erreicht werden. Dieser Effekt tritt bei der Berechnung von Alternativgraphen nicht mehr so deutlich auf. Die Berechnung wird lediglich um einen Faktor 1.03 beschleunigt, was etwa 44ms Zeitersparnis entspricht (Tabelle 6). Dies liegt an der vergleichsweise geringen Anzahl an Partitionen, die durch das Shadow Level laufen. In unserer Testmenge wurden maximale 3% der Partitionen im Shadow Level traversiert (siehe Abschnitt 5.6.1).

Das Partitionsplitting in Kombination mit dem Shadow Level kann die Berechnung auf dem Europa-Graph um den Faktor 1.7 beschleunigen, im Vergleich zur Partitionierung AG-MLD-4. Bei der Vorberechnung beträgt der Faktor etwa 1.4. Diese weitere Beschleunigung kann durch den relativen Anteil der zu aktualisierenden Partitionen erklärt werden, die maximal 3% der traversierten Partitionen im Shadow Level können eine Aktualisierung von bis zu 80% der Partitionen auf dem obersten Level erwirken (siehe Abschnitt 5.6.1).

5.5 Konfigurationen für Alternativgraphen

Es muss für sechs Parameter eine gute Wahl getroffen werden. Bader et al. haben für die Berechnung die Konfiguration AG-04 gewählt (Tabelle 7). Für eine Erweiterung der Penalty-Methode (*multi-increase*) wurde AG-03 genutzt. Diese beiden Konfigurationen werden wir nutzen, um die Auswirkung auf unsere Implementierung der Penalty-Methode zu untersuchen. Zuerst suchen wir eine obere Grenze $limitN$ für die letzten $limitN$ Iterationen, in denen sich im Alternativgraphen keine Änderung ergeben hat. Weiter werden mit dem so bestimmten Parameter $limitN$ die Werte für die übrigen Parameter festgelegt. Für das Festlegen der Parameter ist die s-t-Menge auf die ersten 1 000 Routen eingeschränkt. Eine abschließenede Auswertung der Qualität der Alternativgraphen wird auf der kompletten s-t-Menge durchgeführt.

5.5.1 Iterationen

Es wurden drei Testinstanzen mit der oberen Schranke $limitN \in \{5, 10, 15\}$ berechnet. Tabelle 8 zeigt, dass sich die obere Schranke für die Anzahl an Iterationen, in der sich keine Ände-

limitN	*totalDistance*	*averageDistance*	*decisionEdges*	*objectiveFunction*	Zeit [s]
5	2.905	1.047	7	2.858	0.794
10	2.907	1.047	7	2.859	0.803
15	2.908	1.047	7	2.860	0.815

Tabelle 8: Auswirkung von *limitN* auf die Qualität der Alternativgraphen.

rung ergeben hat, nur minimal auf die Qualität auswirkt. Der Qualitätsunterschied zwischen $limitN = 5$ und $limitN = 15$ ist minimal.

Der Zeitunterschied von 0.02 Sekunden von 5 Iterationen zu 15 Iterationen in denen sich keine Änderungen ergeben, entspricht im Schnitt weniger als einer Iteration mehr die durchgeführt werden muss. Der Parameter hat nicht direkt Einfluss auf alle Berechnungen, sondern nur auf die, in denen die Bedingung 2.2 nicht verletzt wird und sich über mehrere ($limitN$) Iterationen keine Änderung im Alternativgraph ergibt.

Die Berechnung für $limitN = 15$ ist geringfügig langsamer, wie für die anderen beiden Konfigurationen, bei diesem Wert wird aber die beste Zielfunktion erreicht. Wir legen $limitN = 15$ fest.

5.5.2 Penalties, lokale und globale Parameter

Tabelle 7 zeigt die Ausgangskonfigurationen, sowie von Bader et al. vorgeschlagen. Für die weiteren Parameter wurden jeweils drei Werte ausgesucht, die direkt aus Tabelle 9 bzw. 10 entommen werden können. In Tabelle 9 sind die Ergebnisse für AG-04 aufgeführt und in Tabelle 10 die für AG-03. Eine Sortierung der Werte nach $max\text{-}global\text{-}factor$ zeigt den Einfluss auf die *totalDistance* und *averageDistance*. Mit einem hohen $max\text{-}global\text{-}factor$ werden längere Teilstrecken erlaubt. Dies führt direkt zu einer höhren *totalDistance* sowie *averageDistance*. Vergleicht man bei festem $max\text{-}global\text{-}factor$ die Zielfunktion in Bezug auf $min\text{-}global\text{-}factor$, dann lässt sich eine klare Dominanz der Klasse $min\text{-}global\text{-}factor = 0.1$ gegenüber der Klasse $min\text{-}global\text{-}factor = 0.01$ feststellen. Der Unterschied zur Klasse $min\text{-}global\text{-}factor = 0.05$ fällt nicht so deutlich aus. Eine Auswirkung der Parameter auf die *decisionEdges* lässt sich nur für AG-04 und $max\text{-}global\text{-}factor = 1.3$ feststellen.

Je längere Pfade man mit $max\text{-}global\text{-}factor$ zulässt, desto weniger Zeit wird für die Berechnung benötigt. Dies spiegelt sich auch in der Anzahl benötigter Iterationen wider. Durch einen geringeren $max\text{-}global\text{-}factor$ werden mehr Teilpfade nicht mit in den Alternativgraphen aufgenommen und somit werden auch mehr Iterationen benötigt bis eine der Abbruchbedingung verletzt ist.

Ein eindeutiger Einfluss von $min\text{-}global\text{-}factor$ und $max\text{-}local\text{-}factor$ auf die Laufzeit ist nicht erkennbar.

maxGF	minGF	maxLF	*totalDistance*	*averageDistance*	*decisionEdges*	*objectiveFunction*	# Iterationen	Zeit [s]
1.200	0.010	1.200	2.660	1.034	8	2.626	9	1.182
1.200	0.010	1.250	2.661	1.034	8	2.627	9	1.187
1.200	0.010	1.300	2.663	1.034	8	2.628	9	1.185
1.200	0.050	1.200	2.667	1.035	8	2.632	10	1.198
1.200	0.050	1.250	2.668	1.035	8	2.634	10	1.188
1.200	0.050	1.300	2.669	1.035	8	2.634	9	1.185
1.200	0.100	1.200	2.668	1.035	8	2.633	10	1.195
1.200	0.100	1.250	2.669	1.035	8	2.634	10	1.202
1.200	0.100	1.300	2.671	1.035	8	2.636	10	1.200
1.250	0.010	1.200	2.838	1.043	8	2.796	7	0.955
1.250	0.010	1.250	2.841	1.043	8	2.799	7	0.938
1.250	0.010	1.300	2.843	1.043	8	2.800	7	0.939
1.250	0.050	1.200	2.846	1.043	8	2.803	7	0.953
1.250	0.050	1.250	2.852	1.043	8	2.809	7	0.949
1.250	0.050	1.300	2.853	1.043	8	2.810	7	0.948
1.250	0.100	1.200	2.847	1.043	8	2.804	7	0.957
1.250	0.100	1.250	2.852	1.043	8	2.809	7	0.953
1.250	0.100	1.300	2.853	1.043	8	2.810	7	0.950
1.300	0.010	1.200	2.891	1.047	7	2.844	6	0.825
1.300	0.010	1.250	2.894	1.047	7	2.848	6	0.827
1.300	0.010	1.300	2.892	1.047	7	2.845	6	0.811
1.300	0.050	1.200	2.900	1.047	7	2.853	6	0.826
1.300	0.050	1.250	2.905	1.047	7	2.858	6	0.825
1.300	0.050	1.300	2.905	1.047	7	2.858	6	0.824
1.300	0.100	1.200	2.902	1.047	7	2.855	6	0.835
1.300	0.100	1.250	2.907	1.047	7	2.860	6	0.833
1.300	0.100	1.300	2.907	1.047	7	2.860	6	0.826

Tabelle 9: Auswirkung der Bedingung an Teilpfade mit der Konfiguration AG-04.

maxGF	minGF	maxLF	*totalDistance*	*averageDistance*	*decisionEdges*	*objectiveFunction*	# Iterationen	Zeit [s]
1.200	0.010	1.200	2.660	1.032	8	2.628	8	1.057
1.200	0.010	1.250	2.658	1.032	8	2.627	8	1.033
1.200	0.010	1.300	2.659	1.032	8	2.627	8	1.046
1.200	0.050	1.200	2.665	1.032	8	2.634	8	1.048
1.200	0.050	1.250	2.665	1.032	8	2.634	8	1.044
1.200	0.050	1.300	2.666	1.032	8	2.634	8	1.052
1.200	0.100	1.200	2.665	1.032	8	2.634	8	1.057
1.200	0.100	1.250	2.666	1.032	8	2.634	8	1.080
1.200	0.100	1.300	2.666	1.032	8	2.634	8	1.047
1.250	0.010	1.200	2.828	1.038	8	2.790	7	0.873
1.250	0.010	1.250	2.830	1.039	8	2.792	7	0.858
1.250	0.010	1.300	2.830	1.039	8	2.791	7	0.858
1.250	0.050	1.200	2.833	1.039	8	2.795	7	0.873
1.250	0.050	1.250	2.838	1.039	8	2.799	7	0.861
1.250	0.050	1.300	2.839	1.039	8	2.800	7	0.862
1.250	0.100	1.200	2.834	1.039	8	2.796	7	0.867
1.250	0.100	1.250	2.840	1.039	8	2.801	7	0.869
1.250	0.100	1.300	2.840	1.039	8	2.802	7	0.872
1.300	0.010	1.200	2.878	1.042	8	2.835	6	0.803
1.300	0.010	1.250	2.880	1.042	8	2.837	6	0.786
1.300	0.010	1.300	2.876	1.042	8	2.834	6	0.784
1.300	0.050	1.200	2.883	1.042	8	2.841	6	0.816
1.300	0.050	1.250	2.887	1.042	8	2.845	6	0.791
1.300	0.050	1.300	2.886	1.042	8	2.844	6	0.790
1.300	0.100	1.200	2.884	1.042	8	2.842	6	0.795
1.300	0.100	1.250	2.889	1.042	8	2.847	6	0.799
1.300	0.100	1.300	2.889	1.043	8	2.847	6	0.794

Tabelle 10: Auswirkung der Bedingung an Teilpfade mit der Konfiguration AG-03.

Konfiguration	Anfangs-konfiguration	minGF	maxGF	maxLF
AG-Best	AG-04	0.1	1.3	1.3
AG-Fast	AG-03	0.01	1.3	1.3

Tabelle 11: Auswahl an Konfigurationen nach Qualität und Berechnungszeit.

Konfiguration	$totalDist.$	$avg.Dist.$	$dec.Edges$	$objFct.$	Zeit[s]	Iterat.
AG-Best	2.940	1.048	7	2.892	0.829	6
AG-Fast	2.887	1.042	8	2.844	0.804	6

Tabelle 12: Qualitätsmerkmale der Standardkonfiguration mit allen Optimierungen.

5.6 Analyse

Tabelle 11 führt die beste und schnellste Konfiguration auf. Für diese wurden die Qualitätsmerkmale auf der kompletten s-t-Menge berechnet. Die Ergebnisse (Tabelle 12) der beiden Konfigurationen nähern sich ein wenig an. AG-Best liefert allerdings weiterhin das bessere Ergebnis, dafür muss aber ein Zeitverlust von 20ms gegenüber AG-Fast hingenommen werden. Für die weitere Analyse werden wir nur die Konfiguration AG-Best betrachten.

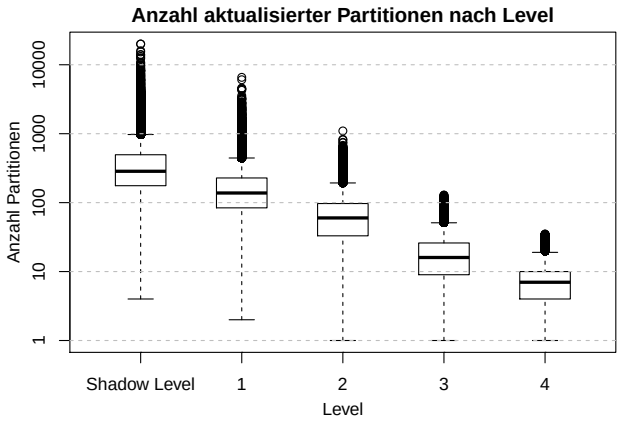
5.6.1 Arbeit pro Level

Die Hälfte aller Pfade aus der s-t-Menge verläuft durch nicht mehr als etwa 500 Partitionen von insgesamt 664 300 Partitionen (vergleiche Tabelle 1 und Abbildung 12). Maximal durchläuft ein Pfad der Testmenge etwa 20 000 Partition, das entspricht etwa 3% aller Partitionen. Die Vater-Partitionen der traversierten Partitionen müssen aktualisiert werden. Die Anzahl an zu aktualisierenden Partitionen nimmt steigendem Level ab. Der relative Anteil steigt auf dem obersten Level allerdings auf bis zu 80%. Dieser Effekt kommt da druch zustande, dass das oberste Level eine starke Vereinfachung des Graphen darstellt, das Shadow Level nähert den ursprünglichen Graph noch sehr gut an.

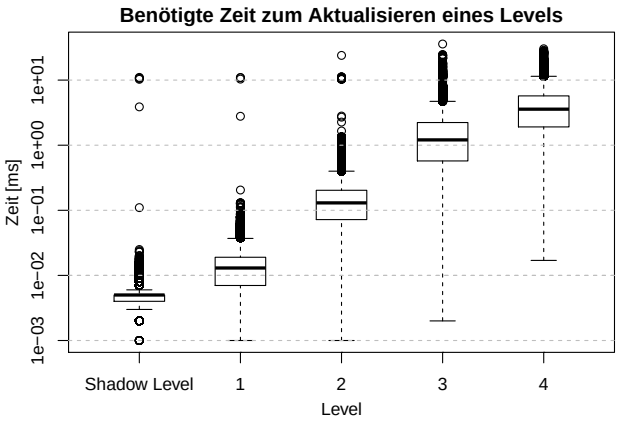
Die Zeit zum Aktualisieren eines Levels steigt hingegen je höher ein Level ist. Bis auf einen Ausreißer lassen sich die ersten drei Levels in weniger als 15ms aktualisieren. Beim Median sind es in allen drei Levels wesentlich weniger. Der Median des dritten Levels benötigt nur wenig mehr als eine 1/10ms. Das Maximum des vierten Levels benötigt insgesamt die meiste Zeit von etwas über 35ms. Das obere Quartil benötigt allerdings immer noch weniger als fünf Millisekunden zum Aktualisieren des Levels. Für das obere Level steigt die Zeit beim Quartil auf knapp über fünf Millisekunden. Für das oberste Level werden auf der Testmenge maximal 30ms zum Aktualisieren benötigt. Dass die benötigte Zeit für die oberen Level wächst, hängt direkt mit der Anzahl an zu aktualisierenden Partitionen pro Level zusammen. Auf dem untersten Level müssen nur etwa 3% den Graphen aktualisiert werden. Die Partitionen auf dem obersten Level entsprechen allerdings 80% des Graphen. Hinzukommt, dass der Suchraum pro Partition mit dem Level steigt (Tabelle 1, Mittlere # Randknoten in gleicher Vater-Partition).

5.6.2 Analyse Dijkstra Rang

Abbildung 13 zeigt Qualitätsmerkmale und Zeiten nach Dijkstra Rang. Ausreißer wurden nicht mit geplottet. Die Analsye im Folgenden bezieht die Ausreißer nicht mit ein.



(a) Anzahl der aktualisierten Partitionen nach Level.



(b) Zeit zum Aktualisieren eines Levels.

Abbildung 12: Anzahl traversierter Partitionen und Zeit zum Aktualisieren für Standardkonfiguration mit der Partitionierung AG-MLD-4-S-Splitted.

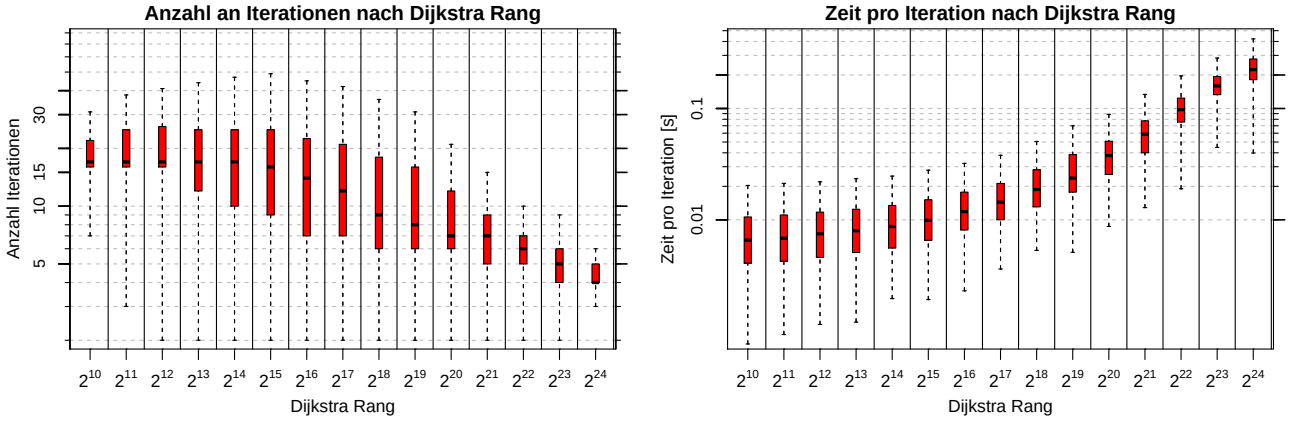
Die Anzahl der Iterationen die nötig ist, bis die Penalty-Methode terminiert, sinkt mit steigender Pfadlänge (Abbildung 13(a)). Wobei die maximale Anzahl an Iterationen erst ein wenig steigt und dann ebenfalls ab 2^{15} monton fällt. Entgegengesetzt steigt die Zeit, die eine Iteration benötigt für das Maximum, die beiden Quartile und den Median monoton(Abbildung 13(b)). Die sinkende Anzahl an Iterationen wird durch schnelleres ansteigen der *averageDistance* zustande kommen. So länger ein Pfad ist, desto eher lassen sich gute Pfade finde, die sich in großen Teilen vom kürzesten Weg unterscheiden, dafür aber auch wesentlich länger werden.

Insgesamt ergibt sich für kürzere Strecken bis zu Rang 2^{13} eine maximale Berechnungszeit von 0.5s (Abbildung 13(c)). Die maximale Zeit steigt bis auf zwei Sekunden bei Rang 2^{24} an. Beim Median bleibt die Berechnungszeit bei etwa einer Sekunde. Die längere Berechnungszeit für die höheren Ränge lässt sich erneut auf das Ungleichgewicht der zu aktualisierenden Partitionen auf dem untersten und obersten Level zurückführen. Gerade bei hohen Rängen wird hier das Verhältnis 3% zu 80% aus Abschnitt 5.6.1 erreicht.

Für die Zielfunktion lässt sich ebenfalls eine Steigerung mit zunehmender Pfadlänge feststellen (Abbildung 13(d)). Hierbei werden maximale Werte von über fünf erreicht. Die Qualität der Mediane steigt von unter 1.5 bis über 3 für den Dijkstra Rang 2^{23} an. Für Rang 2^{24} fällt der Wert für den Median auf unter 3 ab. Das Maximum bleibt für diesen Rang unter 4. Die Quartilabstände sind für die meisten Ränge über dem Wert eins. Nur die beiden oberen Dijkstra Ränge haben einen kleineren Quartilsabstand. Wobei auch hier der Abstand dazu tendiert, mit längerer Pfadlänge abzunehmen. Das Verhalten lässt darauf schließen, dass es lokal nur wenige Alternativen zum kürzesten Weg gibt. Mit steigender Länge finden sich immer mehr Alternativen. Der Einbruch bei 2^{24} ist auf eine geringe Anzahl an Stichproben zurückzuführen. Es muss nicht zu jeder Suchanfrage ein Knoten mit diesem Rang geben. Für fast alle Ränge lassen Pfade finden für die keine Alternativen existieren, daraus resultiert $totalDistance$=1 und $averageDistance$= 1 und somit folgt $objectiveFunction$=1.

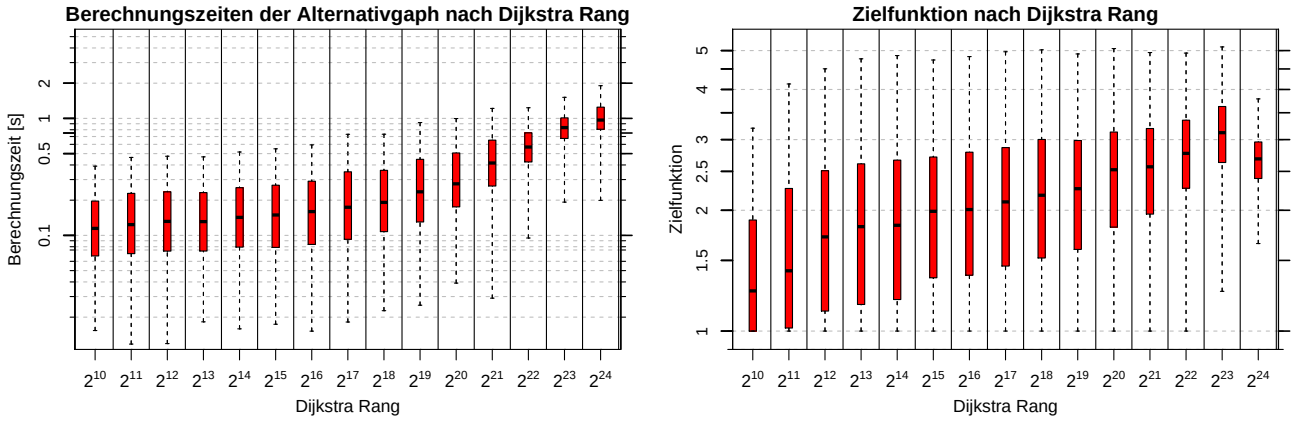
5.7 Vergleich zu Bader et al.

Die Implementierung von Bader et al. [24] liefert bei der Konfiguration AG-04 für die Zielfunktion einen Wert von 3.21. Wir erreichen hier nur einen Wert von 2.89. Allerdings wurde von Bader et al. nur eine Auswertung auf 100 zufälligen Routen auf einem Europa-Graph (18 Millionen Knoten, 42 Millionen Kanten) durchgeführt. Des Weiteren wird bei ihnen nicht genau



(a) Anzahl an Iteratitionen

(b) Zeit pro Iteration



(c) Zeit zum Berechnen eines Alternativgraphens

(d) Entwicklung der Zielfunktion

Abbildung 13: Zeiten, Iterationen und Qualität nach Dijkstra Rang.

beschrieben unter welchen Umständen ein Pfad in den Alternativgraph aufgenommen wird. Somit ist es nicht möglich beide Verfahren abschließend zu vergleichen.

Um den Laufzeitvorteil unseres Verfahren herauszustellen, haben wir auf unserer s-t-Menge die Zeit gemessen, die ein paralleler bidirektionaler Dijkstra zur Berechnung der kürzesten Wege benötigt. Eine Anfrage kann im Mittel in 1.6 Sekunden beantwortet werden. Die benötigte Zeit zur Berechnung von Alternativgraphen einer naiven Implementierung des Penalty-Verfahrens, kann mit sechs Iterationen im Mittel und dieser Zeit abgeschätzt werden. Die naive Implementierung benötigt mindestens 9.6 Sekunden zur Berechnung. Damit ist unsere Implementierung um mindestens ein Faktor 12 schneller als eine naive Implementierung des Verfahrens.

Dass unsere Implementierung nicht nur nach den theoretischen Qualitätsmerkmalen gute Werte liefert, sondern auch visuell zufriedenstellend ist, kann im Anhang nachvollzogen werden. Dort sind Abbildungen von zufällig ausgewählten Alternativgraphen dargestellt. Es werden ausreichend viele Alternativen gefunden, die sich jeweils stark voneinander unterscheiden.

6 Zusammenfassung

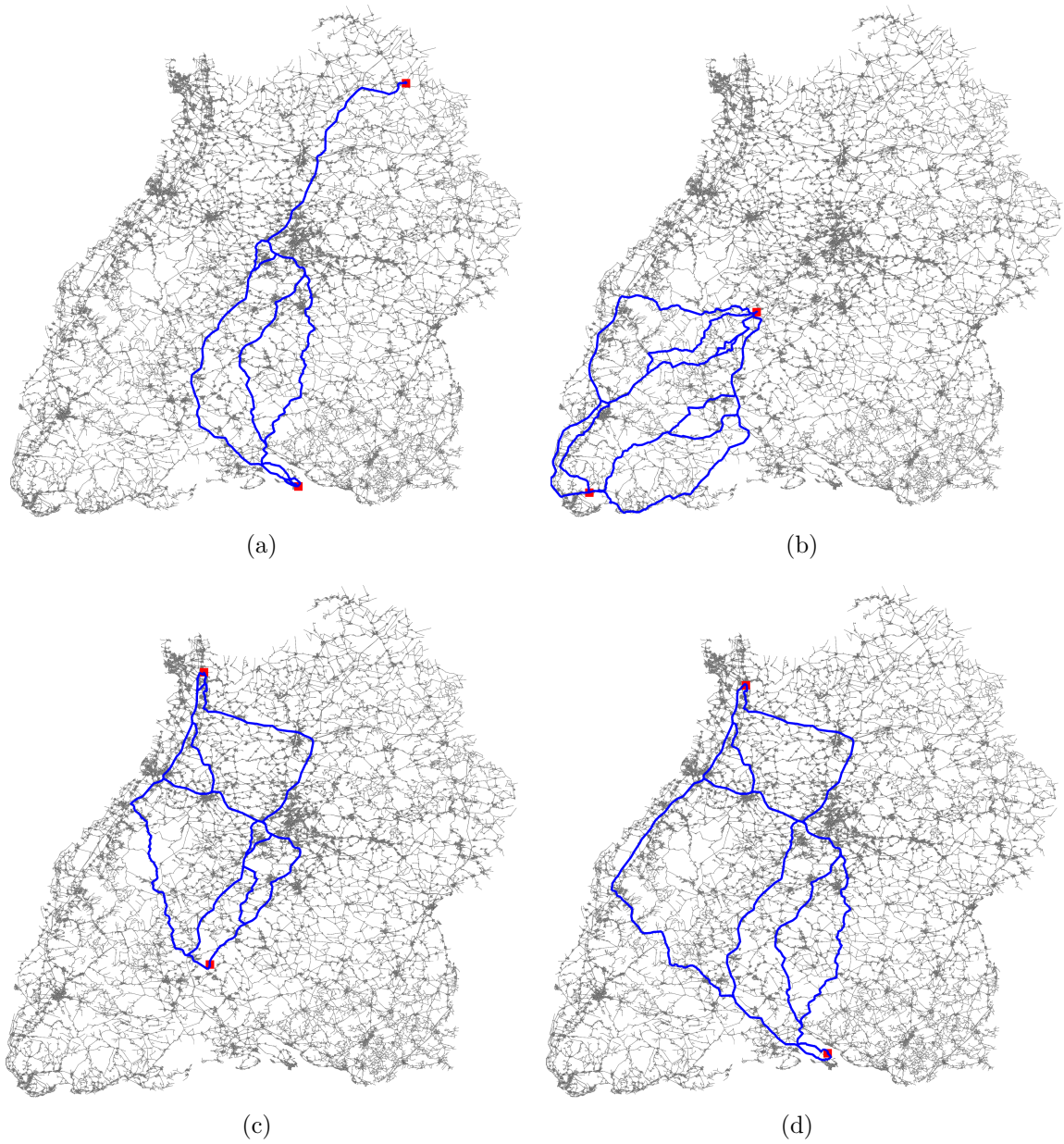
Mit unserer Implementierung des Penalty-Verfahren mit MLD als Grundlage sind wir in der Lage Alternativgraphen schnell zu berechnen. Mit dieser Kombination der beiden Verfahren konnten wir erstmals eine Beschleunigung von über einer Größenordnung gegenüber dem klassischen Ansatz erzielen.

6.1 Offene Punkte

Die Problemstellung ermöglicht weitere Ansätze die eine weitere Beschleunigungen versprechen. In Abschnitt 5.4.2 konnte der Einfluss der Partitionierung auf die Laufzeit gezeigt werden. Andere Partitionierungen wie Punch könnten einen Vorteil versprechen, indem sie die Varianz der Berechnungszeit glätten und Ausreißer reduzieren.

Die statische Struktur der Eingabe könnte stärker ausgenutzt werden und im Voraus über die optimale Updatetechnik entschieden werden.

Darüber hinaus halten wir es für möglich, die Kosten der Aktualisierung auf höheren Leveln deutlich reduzieren zu können. Die, wie in Abschnitt 5.6.1. gesehen, hohe Anzahl an betroffenen Partitionen spiegelt in unseren Augen nicht die Zahl direkt betroffener Pfade wieder. Durch lokalisiertere Updatetechniken sollte eine deutliche Reduzierung des Aufwands möglich sein.



(a) (b)

(c) (d)

Abbildung 14: Ergebnisse die unsere Implementierung auf Baden-Württemberg liefert.

7 Anhang

Die Alternativgraphen in Abbildung 14 wurden mit der AG-Best Konfiguration (Tabelle 11) auf dem Straßennetz von Baden-Württemberg [19], das von OpenStreetMap [18] erstellt wurde, berechnet.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] Andrew V. Goldberg and Chris Harrelson: *Computing the Shortest Path: A* Search Meets Graph Theory.* In: *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, Seiten 156–165. SIAM, 2005.

[2] Camil Demetrescu and Andrew V. Goldberg and David S. Johnson (Herausgeber): *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, Band 74 der Reihe *DIMACS Book*. American Mathematical Society, 2009.

[3] CAMVIT: *Choice routing.* http://www.camvit.com, 2009.

[4] Daniel Delling and Andrew V. Goldberg and Ilya Razenshteyn and Renato F. Werneck: *Graph Partitioning with Natural Cuts.* In: *25th International Parallel*

and Distributed Processing Symposium (IPDPS'11), Seiten 1135–1146. IEEE Computer Society, 2011.

[5] Daniel Delling and Andrew V. Goldberg and Thomas Pajor and Renato F. Werneck: *Customizable Route Planning*. In: Pardalos, Panos M. und Steffen Rebennack (Herausgeber): *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, Band 6630 der Reihe *Lecture Notes in Computer Science*, Seiten 376–387. Springer, 2011.

[6] Daniel Delling and Dorothea Wagner: *Pareto Paths with SHARC*. In: Vahrenhold, Jan (Herausgeber): *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, Band 5526 der Reihe *Lecture Notes in Computer Science*, Seiten 125–136. Springer, June 2009.

[7] Daniel Delling and Moritz Kobitzsch and Dennis Luxen and Renato F. Werneck: *Robust Mobile Route Planning with Limited Connectivity*. In: *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, Seiten 150–159. SIAM, 2012.

[8] Daniel Delling and Peter Sanders and Dominik Schultes and Dorothea Wagner: *Engineering Route Planning Algorithms*. In: Lerner, Jürgen, Dorothea Wagner und Katharina A. Zweig (Herausgeber): *Algorithmics of Large and Complex Networks*, Band 5515 der Reihe *Lecture Notes in Computer Science*, Seiten 117–139. Springer, 2009.

[9] David Eppstein: *Finding the k shortest paths*. In: *Proc. 35th Symp. Foundations of Computer Science*, Seiten 154–165. IEEE, November 1994.

[10] Dennis Luxen and Dennis Schieferdecker: *Candidate Sets for Alternative Routes in Road Networks*. In: *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, Band 7276 der Reihe *Lecture Notes in Computer Science*, Seiten 260–270. Springer, 2012.

[11] Ernesto Queiros Martins: *On a Multicriteria Shortest Path Problem*. European Journal of Operational Research, 26(3):236–245, 1984.

[12] Frank Schulz and Dorothea Wagner and Christos Zaroliagis: *Using Multi-Level Graphs for Timetable Information in Railway Systems*. In: *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, Band 2409 der Reihe *Lecture Notes in Computer Science*, Seiten 43–59. Springer, 2002.

[13] George Karypis and Vipin Kumar: *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*. SIAM Journal on Scientific Computing, 20(1):359–392, 1998.

[14] Ittai Abraham and Daniel Delling and Andrew V. Goldberg and Renato F. Werneck: *Alternative Routes in Road Networks*. In: Festa, Paola (Herausgeber): *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, Band 6049 der Reihe *Lecture Notes in Computer Science*, Seiten 23–34. Springer, May 2010.

[15] Jin Y. Yen: *Finding the K Shortest Loopless Paths in a Network*. Management Science, 17(11):712–716, 1971.

[16] Martin Holzer and Frank Schulz and Dorothea Wagner: *Engineering MultiLevel Overlay Graphs for Shortest-Path Queries*. ACM Journal of Experimental Algorithmics, 13(2.5):1–26, December 2008.

[17] Misa, Thomas J. and Frana, Philip L.: *An interview with Edsger W. Dijkstra*. Commun. ACM, 53(8):41–47, August 2010.

[18] OpenStreetMap: *OpenStreetMap*. http://www.openstreetmap.org/.

[19] OpenStreetMap: *OSM Karte Baden-Würrttemberg*. http://download.geofabrik.de/openstreetmap/europe/germany/baden-wuerttemberg.osm.pbf.

[20] PTV AG: *PTV AG*. http://www.ptvgroup.com/.

[21] Reinhard Bauer and Daniel Delling and Peter Sanders and Dennis Schieferdecker and Dominik Schultes and Dorothea Wagner: *Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm*. ACM Journal of Experimental Algorithmics, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.

[22] Robert Geisberger and Moritz Kobitzsch and Peter Sanders: *Route Planning with Flexible Objective Functions*. In: *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, Seiten 124–137. SIAM, 2010.

[23] Robert Geisberger and Peter Sanders and Dominik Schultes and Christian Vetter: *Exact Routing in Large Road Networks Using Contraction Hierarchies*. Transportation Science, 46(3):388–404, August 2012.

[24] Roland Bader and Jonathan Dees and Robert Geisberger and Peter Sanders: *Alternative Route Graphs in Road Networks*. In: Marchetti-Spaccamela, Alberto und Michael Segal (Herausgeber): *Proceedings of the 1st International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems (TAPAS'11)*, Band 6595 der Reihe *Lecture Notes in Computer Science*, Seiten 21–32. Springer, 2011.

[25] Rolf H. Möhring and Heiko Schilling and Birk Schütz and Dorothea Wagner and Thomas Willhalm: *Partitioning Graphs to Speedup Dijkstra's Algorithm*. ACM Journal of Experimental Algorithmics, 11(2.8):1–29, 2006.

[26] Sungwon Jung and Sakti Pramanik: *An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps*. IEEE Transactions on Knowledge and Data Engineering, 14(5):1029–1046, September 2002.

[27] Yanyan Chen and Michael G. H. Bell and Klaus Bogenberger: *Reliable Pretrip Multipath Planning and Dynamic Adaptation for a Centralized Road Navigation System*. IEEE Transactions on Intelligent Transportation Systems, 8(1):14–20, March 2007.

[28] Yun-Wu Huang and Ning Jing and Elke A. Rundensteiner: *Effective Graph Clustering for Path Queries in Digital Maps*. In: *Proceedings of the 5th International Conference on Information and Knowledge Management*, Seiten 215–222. ACM Press, 1996.