Report zum Fachprojekt Routingalgorithmen

Pouria Araghchi
TU Dortmund
Dortmund, Deutschland
pouria.araghchi@tu-dortmund.de

Kai Lukas Ilmenau
TU Dortmund
Dortmund, Deutschland
kai.ilmenau@tu-dortmund.de

Naveed Niazi
TU Dortmund
Dortmund, Deutschland
naveed.niazi@tu-dortmund.de

ABSTRACT

In diesem Report werden die Ergebnisse unserer Arbeit im Rahmen des Fachprojekts "Routingalgorithmen" zusammengefasst. Wir haben die Algorithmen auf die Maximum Link Utilization (MLU) nach [2] untersucht. Beschreibung der Ergebnisse von Pourias Algorithmus Der Algorithmus, der inverse capacity mit centrality Metriken verrechnet, ist meistens schlechter als inverse capacity, da reines inverse capacity öfter zwei kürzste Wege finden um demands gleichmäßiger zu verteilen damit einzelne Links nicht überlastet werden. Es wird jedoch auch gezeigt, dass Topologien gefunden werden können, bei den pures inverse capacity eine schlechtere MLU hat. Beschreibung der Ergebnisse von Naveeds Algorithmus Die Replikation andere Gruppen im ersten Projektteil nicht erfolgreich, und im zweiten schon, jedoch haben wir daraus die Lehre gezogen wie wir unsere Arbeit besser replizierbar machen und das frühe Kommunikation mit den anderen Gruppen wichtig ist.

CCS CONCEPTS

• **Networks** → **Traffic engineering algorithms**.

KEYWORDS

weight optimization, waypoint optimization

1 EINLEITUNG

In der heutigen Zeit werden mehrere Millionen Datenpakete von und zu Knotenpunkten verschickt. Daher ist das *Routing* von diesen Datenpaketen in der heutigen Zeit wichtiger denn je. Gutes Routing kann Datenstaus verringern oder sogar verhindern, Verteilerknoten auslasten oder entlasten und sorgt damit fuer einen reibungslosen Datenverkehr ueberall auf der Welt. Wie gut das Routing innerhalb einer Netzwerktopologie ist, wird durch den darunterliegenden Routingalgorithmus bestimmt. Hierfür gibt es verschiedenste Ansätze mit variierender Komplexität und Erfolgswahrscheinlichkeit.

Im Rahmen des Fachprojekts "Routingalgorithmen" haben wir uns näher mit den verschiedenen Algorithmen beschäftigt und haben eigene Variationen dieser Algorithmen erstellt, um ein tieferes Verständnis der Materie zu erlangen. Zielfunktion war hierbei die Maximum Link Utilization (MLU) nach [2]. In diesem Report fassen wir die Ergebnisse unserer Arbeit zusammen. Dies umfasst eine Beschreibung unserer drei algorithmischen Ideen (Sequential combination aus inverse capacity und demand first waypoints, centrality Metriken mit inverse capacity und Naveeds Algorithmus) und Experimente zu Algorithmen im Stil des Papers[2]. Danach werden die Resultate dieser Experimente eingeordnet und erklärt. Anschließend wird auf die Arbeit einer anderen Gruppe im Fachprojekt eingegangen, dessen Arbeit wir im Rahmen dieses Fachprojekts repliziert haben.

Authors' addresses: Pouria Araghchi, TU Dortmund, Otto-Hahn-Straße 14, Dortmund, Deutschland, pouria.araghchi@tu-dortmund.de; Kai Lukas Ilmenau, TU Dortmund, Otto-Hahn-Straße 14, Dortmund, Deutschland, kai.ilmenau@tu-dortmund.de; Naveed Niazi, TU Dortmund, Otto-Hahn-Straße 14, Dortmund, Deutschland, naveed.niazi@tu-dortmund.de.

2 THEORETISCHE GRUNDLAGEN

Hier wird etwas zum Paper stehen[2]. Die Grundlage zu dieser Arbeit bildet das Paper "Traffic Engineering with Joint Link Weight and Segment Optimization"[2]. In der Artbeit wurde alternativen Routing-Algorithmen gesucht, welche eine optimale *minimal link utilization* (MLU) auf diversen Topologien liefert. Zwei Algorithem und insbesondere deren Verknuepfung haben sich als vorteilhaft erwiesen. Die beiden Algortihmen sind die Link-Weight- und Waypoint-Optimisierung. Die Link-Weight-Optimisierung macht ...
Die Waypoint-Optimisierung hingegen bietet ...
Die Kombination aus beiden fuehrt zu ...
Im Vergleich von vielen Routing-Algorithmen auf verschiedensten Topologien schnitt die JOINT ..... ab, siehe Abbildung ... .

3 UMSETZUNG

In diesem Abschnitt werden unsere Projekte und die dazugehörigen Ideen vorgestellt. Die beiden Projekte sind in Unterabschnitte und die algorithmischen Ideen und Umsetzungen des jeweiligen Gruppenmitglieds in Paragraphen unterteilt.

3.1 Algorithmen zu Projekt 1

In Projekt 1 ging es darum, eine algorithmische Idee auf Basis von [2] herauszuarbeiten und diese mithilfe desselben Git-Prokjekts[1] mit alternativen Routingalgorithmen zu vergleichen.

3.1.1 *Sequential combination aus inverse capacity und demand first waypoints.*

Der Kerngedanke hier war es, einen einen schnellen Algorithmus zu finden, welcher hinreichend gute Ergebnisse in Bezug auf die MLU abliefert. Diese Eigenschaft ist insbesondere für Netzwerke sinvoll, dessen Auslastungen sich regelmäßig während des Betriebs ändern und der ALgorithmus reaktiv sehr schnell alle Gewichte und Wegpunkte neu berechnet.
Die Grundlage dieses Algorithmus war die sequentielle Kombination von dem empirisch häufig verwendetem *inverse_capacity* und dem etwas langsameren aber genaueren *demand_first_waypoints*.
Anmerkung: Die Algorithemn Namen "demand first waypoints" und "greedy waypoints" sind hier austauschbar.
Daraufhin bildeten sich zwei Fragen.

(1) Ist die Kombination genauer als dessen Bestandsteile?
(2) Ist die zusätzliche Rechenzeit gerechtfertigt?

Die Abbildungen 1 und 3 geben eine Antwort auf Frage 1.
Sie zeigen, dass die sequentielle Kombination beider Algorithmen

[1] https://github.com/fruitiestPunch/FaPro_P1, siehe Abschnitt A

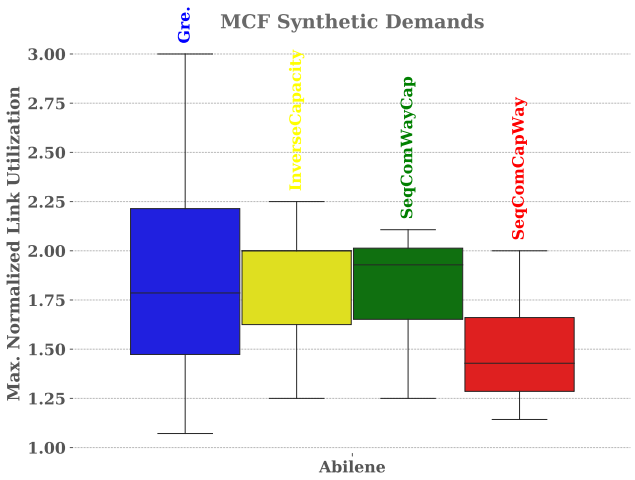

Figure 1: Vergleich von vier Algorithmen mit syntetischen Anforderungen auf der Abilene-Topologie. Legende: greedy waypoints in blau, inverse capacity in gelb, SeqComWayCap in grün, SeqComCapWay in rot.

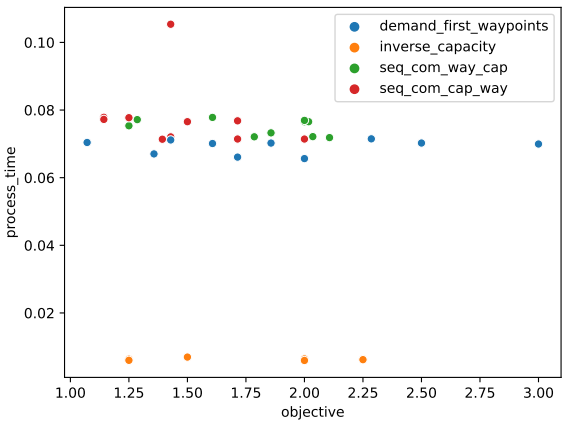

Figure 2: Vergleich von vier Algorithmen mit syntetischen Anforderungen auf der Abilene-Topologie. Legende: greedy waypoints in blau, inverse capacity in gelb, SeqComWayCap in grün, SeqComCapWay in rot.

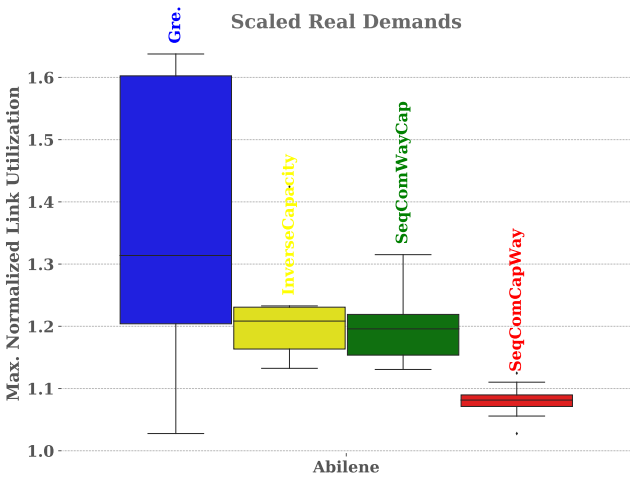

Figure 3: Vergleich von vier Algorithmen mit realen Anforderungen auf der Abilene-Topologie. Legende: greedy waypoints in blau, inverse capacity in gelb, SeqComWayCap in grün, SeqComCapWay in rot.

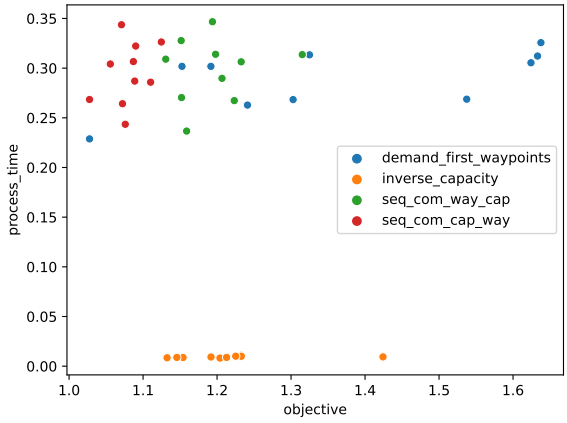

Figure 4: Vergleich von vier Algorithmen mit syntetischen Anforderungen auf der Abilene-Topologie. Legende: greedy waypoints in blau, inverse capacity in gelb, SeqComWayCap in grün, SeqComCapWay in rot.

in beiden Fällen mindestens genauso gut ist wie eines seiner Bestandteile. Im Fall von *SeqComCapWay* ist dieser im Durchschnitt sogar besser als beide Algorithmen. Dies lässt sich darauf zurückführen, dass die Kantengewichte durch den *inverse_capacity*-Algorithmus zu Beginn verbessert werden und anschließend durch den zweiten Algorithmus weiter verbessert werden, falls möglich. Im Folgenden wird daher nur noch auf die sequentielle Kombination *SeqComCapWay* (in rot) eingegangen.

Die Abbildungen 2 und 4 hingegen sind relevant für die Frage 2. Diese zeigen einen etwas genaueren Blick auf die Rechenergebnisse geplottet über die dafür benötigte Zeit. Hier kann auch direkt gesehen werden, dass *inverse_capacity* mit Abstand der schnellste hier verwendete Algorigthmus ist. Dies ist besonders vorteilhaft, da es bedeutet, dass die sequentielle Kombination aus *inverse_capacity* und *greedy_waypoints* kaum mehr Berechnungszeit benötigt als *greedy_waypoints* alleine. Somit konnte empirisch gezeigt werden, dass in diesem "vereinfachten" Beispiel die Kombination der oben genannten Algorithmen nicht nur im Durchschnitt bessere MLU-Ergebnisse liefert, sondern das auch in derselben Zeit erledigt. Und somit lässt sich die Frage der Rechtfertigung auslagern. In allen Fällen, in denen der *greedy_waypoints*-Algorithmus gerechtfertigt

ist, ist auch die oben genannte Kombination gerechtfertigt.
Da es aus Zeitgründen sowie mangelnder Rechenressourcen nicht so einfach möglich war, diese Algorithmen auf sehr großen Tologien zu testen, kann nicht mit Sicherheit gesagt, wie hoch die Berechnungszeit in Topologien mit 50 oder mehr Knoten wirklich ist. Daher ist es schwer, diese Algorithmen (mit Ausnahme von *inverse_capacity* in solchen Topologien dynamisch zu verwenden.

3.1.2 Centrality-Metriken mit inverse capacity. Der Gedanken hier war, den Algorithmus inverse capacity, der statisch auf der Topologie (also unabhängig von den demands) berechnet wird, mit Zentralitätswerten zu erweitern.

Ich habe mich für die Zentralitätswerte closeness-, eigenvector- und betweeness-Zentralität entscheiden. Im Folgenden werden nun die Zentralitätsmetriken definiert (Definitionen aus [1]), und dann erläutet warum diese ausgewählt wurden:

(1) **Adjazenzmatrix** Adjazenzmatrix A wird benutzt um einen Graphen als Matrix darzustellen. Hierfür gilt:

$$a_{ik} = \begin{cases} 1 & \text{wenn Knoten i und k eine Kante verbindet,} \\ 0 & \text{sonst} \end{cases}$$

(2) **closeness-centrality** Wie der Name bereits vermuten lässt, berechnet diese Metrik, wie nah ein Knoten allen anderen ist. Die closeness-centrality c_i des Knoten i:

$$c_i = \frac{1}{\sum_{j \neq i} H(\mathcal{P}_{i \to j})}$$

$\mathcal{P}_{i \to j}$ ist der kürzeste Pfad von Knoten i nach Knoten j.
$H(\mathcal{P}_{i \to j})$ ist die Anzahl an Kanten, die genutzt werden müssen, um von Knoten i nach Knoten j zu kommen.
Je höher die closeness-Metrik, desto zentraler ist der Knoten im Graph.

(3) **betweenness-centrality** Die betweenness-centrality gibt das Verhältnis aller kürzesten Wege, die durch einen Knoten i führen, zur Menge aller kürzesten Wege an.
Die betweenness-centrality eines Knoten i:

$$b_i = \sum_{s,t \in \mathcal{N}} \frac{|\mathcal{P}_{s \to t}(i)|}{|\mathcal{P}_{s \to t}|}$$

wobei $|\mathcal{P}_{s \to t}|$ die Anzahl aller kürzesten Wege von s nach t ist und $|\mathcal{P}_{s \to t}(i)|$ die Anzahl dieser Wege, die durch den Knoten i verlaufen.

(4) **eigenvector-centrality):**
Die eigenvektor-centrality x_i eines Knotens i ist das i-te Element des Eigenvektors, die dem größten Eigenwert λ_1 der Adjazenz-Matrix A entspricht.
Die Formel zur Berechnung lautet:

$$x_i = \frac{1}{\lambda_1} \sum_{k=1}^{N} a_{ik} x_k$$

Hat ein Knoten einen hohen eigenvektor-centrality, lässt sich folgern, dass der Knoten mit anderen wichtigen Knoten verbunden ist.

Die closeness-centrality wurde gewählt, da die closeness aussagt wie vielen Kanten ein Knoten von allen anderen entfernt ist, daher potenziell viele Datenstörme durch diesen Knoten fließen können. Die betweeness-centrality wurde gewählt, da Knoten mit

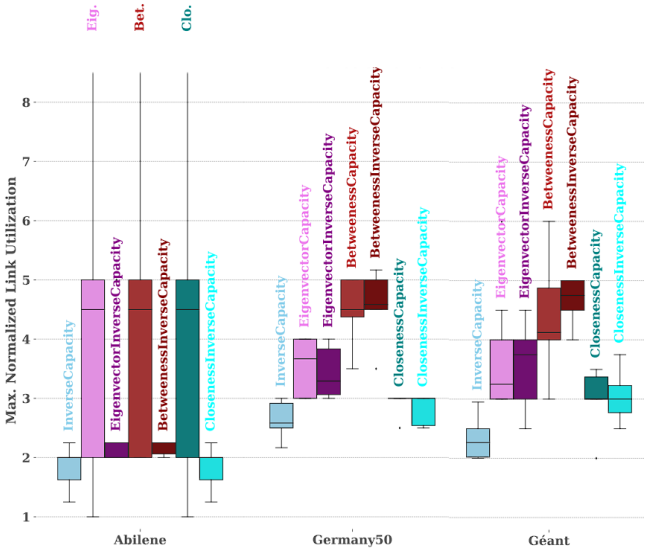

Figure 5: Ergebnisse für centrality-inverse-capacity

hoher betweeness, an vielen kürzesten Wegen beteiligt sind, hier also das Risiko für Link überlastung hoch ist. Die eigenvector-centrality wurde gewählt, da eine hohe eigenvector-centrality darauf schließen lässt, dass der Knoten ein guter "Verteiler"-Knoten im Netz ist.

In der tatsächlichen Implementierung wurden die centrality Berechnungen von NetworkKit benutzt, und dort zusätzlich noch normalisiert.

Um inverse capacity mit den centrality Werten zu verrechnen wird folgende Formel genutzt (für die Kante von i nach j:

$$new\_weight_{ij} = inverse\_capacity\_weight * \frac{CentralityNode_i + CentralityNode_j}{2}$$

In 5 gibt es neben dem $centralityInverseCapacity auch noch den $centralityCapacity, dass ist der Algorithmus ohne InverseCapacity, also:

$$new\_weight_{ij} = weight * \frac{CentralityNode_i + CentralityNode_j}{2}$$

Diese dürften schlecht abschneiden, sind hier jedoch mit dabei, um den Unterschied zu unterstreichen.

Wie man in Abbildung 5 zu sehen ist, ist der neue Algorithmus keine Verbesserung auf den getesteten Topologien. Am schlechtesten schneidet hier der Algorithmus mit der betweeness centrality ab, dicht gefolgt von der eigenvector centrality. Die closeness centrality schneidet mit am besten ab und ist im Beispiel für "Abilene" gleich auf mit inverse capacity. Die Algorithmen ohne inverse capacity sind, schlechter, jedoch (abgesehen von Abilene) nah an den anderen Algorithmen dran.

Die schlechtere Performance kann dadurch erklärt werden, dass die weights der Kanten bei reinem Inverse Capacity, zwei kürzeste Wege zwischen zwei Knoten entstehen lassen, und diese durch die centralities dann, durch die neue Gewichtung, nur noch einen

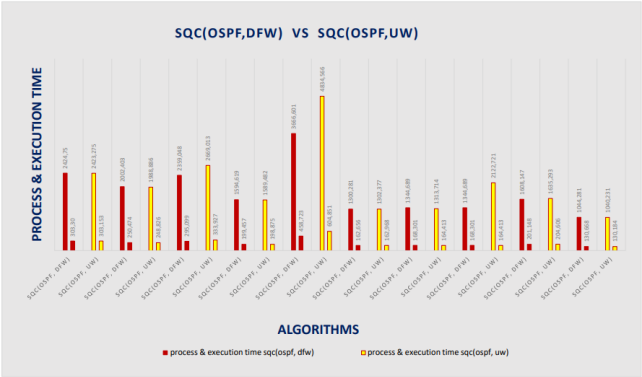

Figure 6: Plotergebnisse

kürzesten Weg haben. Dadurch werden dann einzelne Links überlastet und die MLU wird schlechter.

3.1.3 Sequential Combination aus OSPF und Uniform Weights. Der zentrale Gedanke dieses Algorithmus bestand darin, im Rahmen des ersten Projekts die ursprüngliche Kombination aus OSPF und Demand First Waypoint(DFW) zu optimieren, indem der DFW durch den Uniform Weights erstzt wurde. Beide Algorithmen haben das Ziel, die Netzwerküberlastung zu minimieren, indem sie die Linkgewichte anpassen. Jedoch unterscheiden in ihrer Herangehensweise an das Traffic Engineering.
Der DFW-Algorithmus fokussiert sich darauf, den Verkehr zu spezifischen Zwischenpunkten im Netzwerk, sogenannten Waypoints, zu leiten, um die Verkehrslast auszugleichen und Stau zu vermeiden. Auf der anderen Seite zielt der UW-Algorithmus darauf ab, den Verkehr gleichmäßig über das gesamte Netzwerk zu verteilen. Das Ziel bestand darin herauszufinden, welche Kombination dieser Algorithmen in Bezug auf die Maximale Link-Auslastung(MLU) die besseren Ergebnisse für den Datenverkehr im Netwerk liefert.
Die Tests[2] für beide Algorithmen wurden erfolgreich durchgeführt, Allerdings gab es Schwierigkeiten beim Plotten der Ergebnisse aufgrund zahlreicher Fehlermeldungen, und aufgrund mangelder Erfahrung mit Python fiel es schwer, diese Fehler zu beheben.
Dennoch wurden einige Ergebnisse wie die Prozesszeit und Ausführungszeit beider Algorithmen manuell zusammengefasst, wie in Abbildung 6 dargetellt. Die Auswertung zeigt, dass in einigen Fällen die Kombination aus OSPF und DFW besser abschneidet, während in anderen Fällen die Kombination aus OSPF und UW bessere Ergebnisse liefert. Im Durchschnitt kann man sagen, dass die Kombination von OSPF und DFW ähnlich gute Leistung erbringt wie OSPF mit UW.
OSPF mit UW bietet folgende Merkmale:

(1) Einfache Konfiguration und Verwaltung des Routers, da alle Verbindungen die gleiche Kostenmetrik haben.
(2) Gleichmäßige Lastverteilung über das Netzwerk, da alle Verbindungen als gleichwertig behandelt warden.
(3) Begrenzte Optimierungsfähigkeit: durch die Verwendung einheitlicher Gewichte kann OSPF mit UW keine spezifischen Leistungsmerkmale oder Netzwerkanforderungen berücksichtigen. Dies kann in einigen Anwendungsfällen zu Suboptimalität in Bezug auf Bandbreite, Verzögerung oder andere wichtige Faktoren führen.
(4) Es ist nicht empfohlen in komplexen Netzwerken mit unterschiedlichen Leistungseigenschaften der Verbindungen, da die einheitliche Gewichte nicht auf die individuellen Anforderungen reagieren können.
(5) Empfohlen in Netzwerken mit homogenen Verbindungen, bei denen keine signifikanten Unterschiede in den Leistungseigenschaften bestehen.

Also wenn eine dynamische Anpassung des Routings, Ressourceneffizienz und die Fähigkeit, auf Veränderungen zu reagieren, wichtig sind, kann OSPF mit DFW geeigent sein. Wenn hingegen eine einfache Konfiguration und gleichmäßige Lastverteilung gewünscht sind, kann OSPF mit UW angemessen sein.

3.2 Experimente zu Projekt 2

In diesem Projekt ging es darum, die eigene algorithmische Idee zu nehmen und in einem virtuellen Netzwerk[3] zu testen. Dieses Projekt wurde ebenfalls mithilfe eines bereits existierenden Repositories[4] bearbeitet.

3.2.1 Sequential combination aus inverse capacity und demand first waypoints. Zum Testen dieses Algorithmus wurde eine vereinfachte Topologie mit wenigen Anforderungen erstellt. In Abbildung 8 ist die finale Topologie zu sehen, wobei hier bereits der *inverse_capacity*-Algorithmus darauf ausgeführt wurde. In der Tabelle 1 sind die beiden Anforderungen sowie deren Start- und Endknoten angegeben. In diesem Projekt kam es zu einigen Problemen, die durch die Rechenstärke der genutzten Hardware verschuldet wurden. Abschnitt B befasst sich stärker mit diesem Punkt und dessen Auswirkungen. Abbildung 7 zeigt dabei die MLU-Werte von zwei Algorithmen im Vergleich. Hierfür wurden der Weights-Algorithmus und der sequentielle Kombinations aus *inverse_capacity* und *demand_first_waypoints* miteinander verglichen.
Anhand der Form der Boxplots ist zu erkennen, dass es keine bis minimale Streuung in den finalen Ergebnissen gab. Im Allgemeinen schneiden beide Algorithmen gleich gut ab, denn bei in beiden Fällen liegt die durchschnittliche MLU bei unter 2. Allerdings ist in Abbildung 7 auch zu erkennen, dass beide Ausreißer enthalten, wobei der Ausreißer vom Weights-Algorithmus wesentlich besser liegt, als der andere. Es wird vermutet, dass die starke Spezialisierung des SC-Algorthmus auf der gegebenen Topologie dazu führt, dass er im Allgemeinen sehr gute MLU-Ergebnisse liefert, aber im *worst-case* nicht so gut abschenidet. Im Vergleich liefert Weights-Algorithmus generell konsistent ähnliche Ergebnisse, sodass er im *worst-case* dennoch nicht ganz schlecht ausgeht.

3.2.2 Centrality-Metriken mit inverse capacity. Die Topologie für die Tests der centrality Metriken mit inverse capacity ist die in Abbildung 9 Es gibt zwei Demands, einmal von 0 nach 4 mit Größe 1, und einmal von 3 nach 4 mit Größe 8. Außerdem wurde hier nur die centrality Metrik closeness benutzt, da diese im ersten Projekt am besten der drei Metriken abgeschlossen hat.

[2] https://github.com/fruitiestPunch/FaPro_P1

[3] https://github.com/nikolaussuess/nanonet, siehe Abschnitt A
[4] https://github.com/fruitiestPunch/FaPro_P2, siehe Abschnitt A

Figure 7: Vergleich von zwei Algorithmen als Boxplotdiagramme.

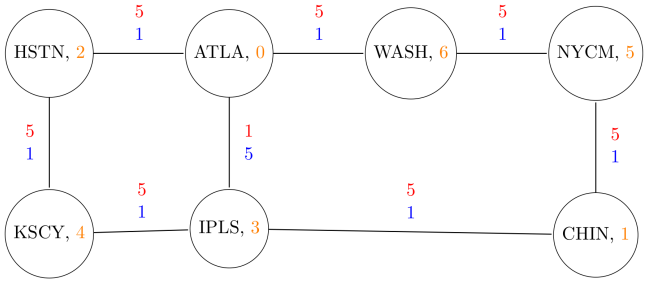

Figure 8: Vereinfachte Netzwerktopologie [Anlehnung an Abilene]. Legende: Kapazitäten in rot, Gewichte in blau.

Table 1: Vereinfachte Anforderungstabelle. Vollständige Tabelle unter https://github.com/fruitiestPunch/FaPro_P2/tree/master/pouria/network_origin.pdf

↓ von, nach →	IPLS	WASH	···
ATLA	5		
IPLS		7	
⋮			

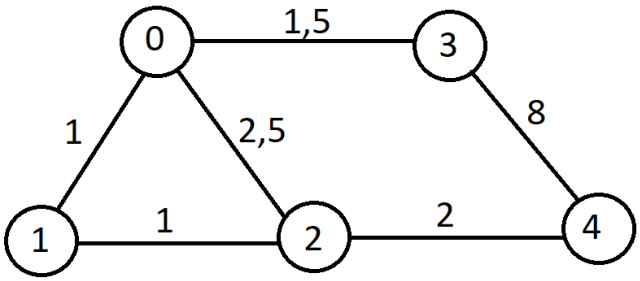

Figure 9: Topologie für die Tests des Algorithmus centrality Metriken mit inverse capacity

Warum diese Topologie genutzt wurde, wird klar wenn man sich die Ergebnisse anschaut. Wie man in Abbildung 11 sehen kann, performt auf dieser Topologie der Algorithmus besser als inverse capacity. Der Grund dafür wird in Abbildung 10 verdeutlicht. Inverse capacity schickt beide Demands über die Kante zwischen

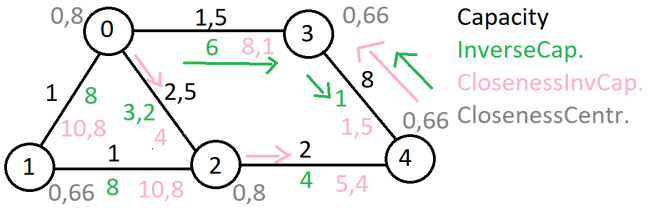

Figure 10: Topologie mit ausgerechneten Werte, Pfeile geben die jeweils genommenen Pfade an

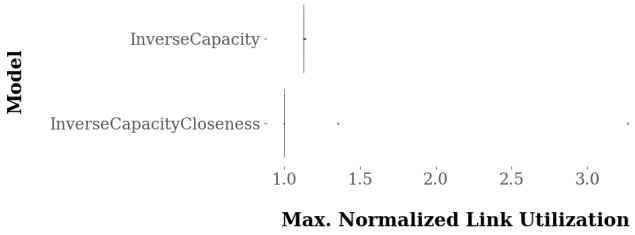

Figure 11: Ergebnisse für Algorithmus centrality Metriken mit inverse capacity

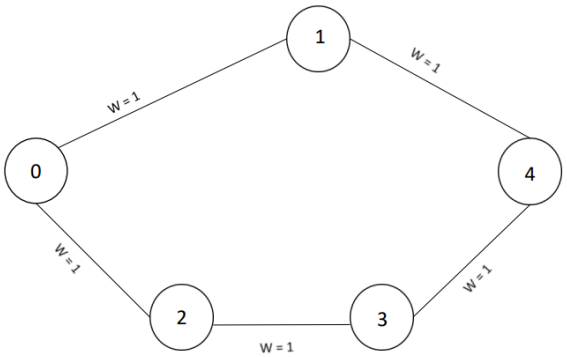

Figure 12: Plotergebnisse

3 und 4, dadurch wird diese überlastet. Inverse capacity mit der closeness Metrik schick ein Demand von 3 nach 4 und den anderen über 0 nach 2 nach 4 und verhindert dadurch die überlastung.

3.2.3 Sequential Combination aus OSPF und Uniform Weights. Um den Algorithmus im Nanonet zu testen, wurde eine Topologie1 erstellt, die 5 Knoten umfasst, wie in Abbildung 12 dargestellt. Dabei wurden allen Kanten gleichmäßige Gewichtungen zugewiesen. Anschließend wurden Demands von Knoten 0 zu Knoten 4 gesendet. Um sicherzustellen, dass alle verfügbaren Wege von 0 zu Knoten 4 genutzt werden, wurden Knoten 1 und 3 als sogenannte Waypoints hinzugefügt. In Abbildung 13 ist der Vergleich zwischen JoinUW (oben) und der Sequential Combination aus OSPF und UW (unten) zu sehen.

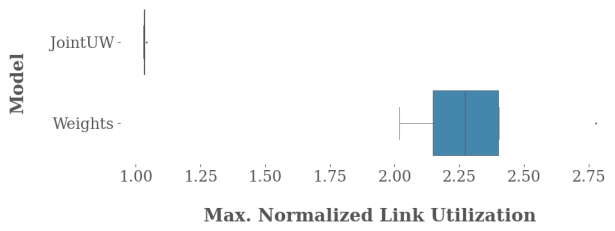

Figure 13: Topologie

```
MCF Synthetic Demands - all_algorithms
Mean objective over all topologies:
                                                    plot_results.py:247: RuntimeWarning: Mean of empty slice.
  mean = np.mean(df_x["objective"].values.mean())

  ret = ret.dtype.type(ret / rcount)
                nan: nan

Plot files:
Traceback (most recent call last):
  File '                                            \plot_results.py", line 330, in <module>
    prepare_data_and_plot(df_i, title_i, plot_type_i)
  File '                                            \plot_results.py", line 282, in prepare_data_and_plot
    create_box_plot(df, "topology_name", "objective", "algorithm_complete", plot_file, x_label="",
  File "                                            \plot_results.py", line 143, in create_box_plot
    add_vertical_algorithm_labels(box_plot.axes)
  File "                                            \plot_results.py", line 103, in add_vertical_algorithm_la
bels
    lines_per_box = int(len(lines) / len(boxes))
                    ~~~~~~~~~~~~^~~~~~~~~~~~~~~
ZeroDivisionError: division by zero
```

Figure 14: Plotting-Fehlermeldung im Terminal

4 REPLIKATION

Im Folgenden werden die Ergebnisse und Einsichten der Replikationen von Gruppe 2 vorgestellt.

4.1 Replikation zu Projekt 1

Die Replikation von diesem Projekt[5] war zu Beginn leider etwas schwierig, weil in beiden Fällen Abhängigkeiten fehlten. Ein großes Problem beim Replizieren des Deep-Learning-Projekts war, dass sich der Prozess nach einer Weile selbstständig beendet hat. Die Vermutung liegt nahe, dass der Rechenaufwand so hoch war, dass der Rechner ab einem bestimmten Zeitpunkt den Prozess selbst beendet hat. Ein weiteres Problem beim Replizieren waren die Fehlermeldungen und Abbrüche beim Plotten. Diesem und ähnlichen Fehlern sind wir bei unser Bearbeitung ebenfalls begegnet und die Behebung hat sich als außerordentlich schwierig erwiesen, weil dieser "Plottererror" innerhalb einer Python-Bibliothek liegt. Abbildung 14 zeigt einen solchen Fehler, welcher sich hartnäckig auch nach einigen Anpassungsversuchen weiterhin erhalten hat.

4.2 Replikation zu Projekt 2

Die Replikation des Codes[6] aus Gruppe 2 lief im Allgemeinen problemlos, wie die Abbildungen 15 und 16 zeigen. Ein Problem, welches auch innerhalb unserer Gruppe aufgetaucht ist, waren die identischen Werte nach Beendigung der *nanonet_batch.py*-Datei. Wie bereits in Abschnitt 3.2.1 erwähnt, liegt dieses Problem vermutlich an mangelnder Rechenleistung während des Berechnungsprozesses. Nach wiederholtem Ausführen des Codes konnte dieses Problem jedoch behoben werden.

[5]https://github.com/kohlbold/FpRouting
[6]https://github.com/kohlbold/FPRouting2

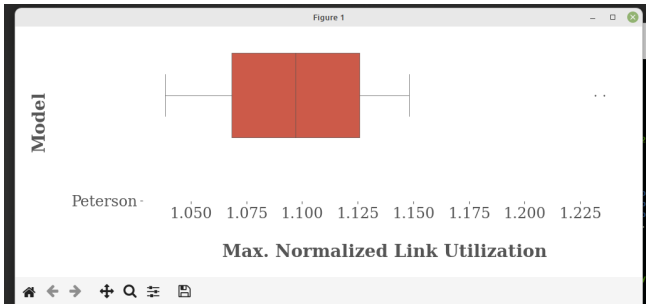

Figure 15: Ergebnisdiagramm des Codes von Gruppe 2

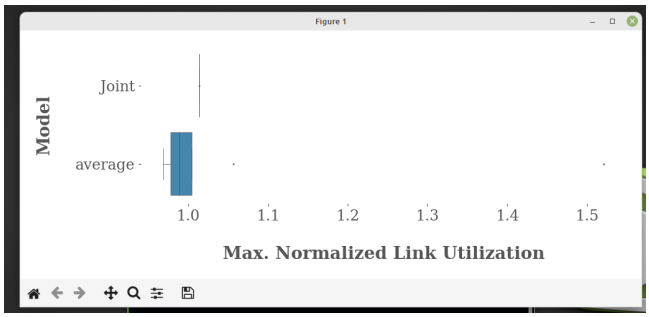

Figure 16: Ergebnisdiagramm des Codes von Gruppe 2

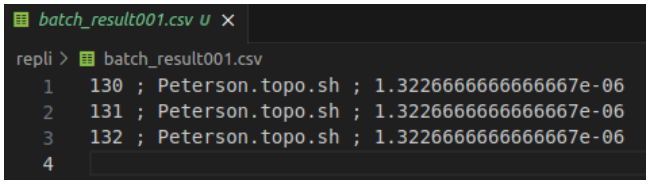

Figure 17: Ergebnis-CSV-Datei mit identischen MLU-Werten in jeder Zeile

5 ZUSAMMENFASSUNG

Es kann gesagt werden, dass wir während des Fachprojekts vieles lernen konnten, insbesondere Dinge, die nicht notwendigerweise im Stundenplan stehen, aber gerade auf der akademischen Laufbahn als Standard angesehen werden. Das Nachstellen einer wissenschaftlichen Arbeit hat sich als nicht-trivial erwiesen und das obwohl diese Arbeit für ihre hohe Reproduzierbarkeit ausgezeichnet wurde. Dies lässt leider nur auf eine Folgerung schließen; viele andere wissenschaftlichen Arbeiten lassen sich viel schwerer oder gar nicht nachstellen. Während des Replizierens der Projekte der anderen Gruppen war es ebenfall überraschend, das selbst wenn unsere Gruppen alle an denselbem Hauptprojekt gearbeitet haben und theoretisch alle dieselben Pakete verwendeten, es manchmal dennoch zu Replikationsschwierigkeiten kam. Allerdings war es auch eine schöne Erfahrung in einer Gruppe an so einem Projekt zu sitzen und die Probleme intern sowie mit den anderen Gruppen besprechen zu können. Das hat nicht nur zu einem größeren Zusammengehörigkeitsgefühl geführt, sondern wir haben auch gelernt, kollaborativ gemeinsame Schwierigkeiten zu lösen.

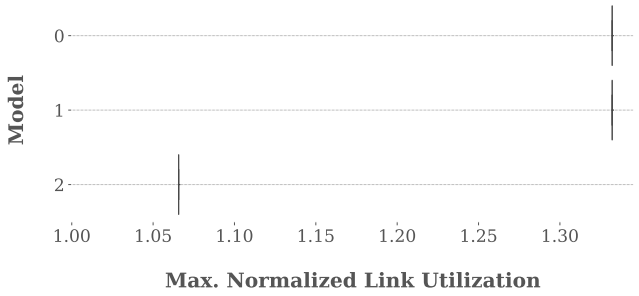

Figure 18: Vergleich von drei Algorithmen als Boxplotdiagramme. Legende: 0 = Joint, 1 = Weights, 2 = Pouria.

6 AUSBLICK

Aufgrund der relativ kurzen Bearbeitungszeit und den begrenzten Rechenkapazitäten unserer eigenen Computer war es etwas schwer, unsere Algorithmen, insbesondere in Projekt 2, auf großen Topologien zu testen. Dies kann dazu führen, dass bestimmt Effekte wie Datenstaus, welche gerade in großen Topologien auftauchen, in kleinen kaum oder gar nicht vorkommen.

ACKNOWLEDGMENTS

Vielen Dank an Marvin für seine Hilfe und Geduld mit unseren Problemen. Ohne seine Hilfe wäre das alles in der kurzen Zeit sehr viel schwieriger gewesen.

REFERENCES

[1] Hale Cetinay, Carmen Mas-Machuca, Jose L. Marzo, Robert Kooij, and Piet Van Mieghem. 2020. *Comparing Destructive Strategies for Attacking Networks.* Springer International Publishing, Cham, 117–140. https://doi.org/10.1007/978-3-030-44685-7_5

[2] Mahmoud Parham, Thomas Fenz, Nikolaus Süss, Klaus-Tycho Foerster, and Stefan Schmid. 2021. Traffic Engineering with Joint Link Weight and Segment Optimization. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies* (Virtual Event, Germany) *(CoNEXT '21)*. Association for Computing Machinery, New York, NY, USA, 313–327. https://doi.org/10.1145/3485983.3494846

A ONLINERESSOURCEN

Im Rahmen dieses Fachprojekts wurden drei öffentliche Repositories als Hauptquellen verwendet. Die ersten beiden Repositories https://github.com/fruitiestPunch/FaPro_P1 und https://github.com/fruitiestPunch/FaPro_P2, welche *forks* von https://github.com/tfenz/TE_SR_WAN_simulation und https://github.com/nikolaussuess/TE_SR_experiments_2021 sind. Die dritte Hauptquelle stellt https://github.com/nikolaussuess/nanonet dar, welche zur Erstellung von vom Projekt les- und interpretierbaren Netzwerktopologiedaten verwendet wurde.

B ALTERNATIVER BOXPLOT

Die Timeouts in den Experimenten von Projekt 2 haben zu einigen Problemen geführt. Speziell im Falle von Abschnitt 3.2.1 war der Rechner, auf dem der dortige Algorithmus getestet wurde, wesentlich leistungsschwächer (4GB RAM) als die anderen Rechner. Daher wird vermutet, dass die Timeouts [7] [8]

```
at now+2min
```

und

```
sleep(8 * 60)
```

nicht ausgereicht haben, sodass einige der Prozesse frühzeitig abgerbochen wurden. Abbildung 18 zeigt dabei die MLU-Werte aller drei Algorithmen im Vergleich. Die Tatsache, dass die Boxplots hier nur als Striche dargestellt werden, weist daraufhin, dass es keine Streuung in den finalen Ergebnissen gab, was auf das frühzeitige Abbrechen einiger Rechenprozesse zurückgeführt werden kann.

Received 6. August 2023; überarbeitet ausstehend; akzeptiert 15. August 2023

[7] https://github.com/nikolaussuess/nanonet, siehe Abschnitt A

[8] *nanonet_batch.py* aus https://github.com/fruitiestPunch/FaPro_P2, siehe Abschnitt A