Fachprojekt Routingalgorithmen SS2022 - Projektbericht

Gajann Sivarajah
Jan Schulte
Phil Seißelberg

ABSTRACT

Dieser Bericht enthält die Resultate unserer Gruppenarbeit, welche im Zuge des Fachprojekts Routingalgorithmen im Sommersemester 2022 stattgefunden hat. Die Projekte basieren dabei jeweils auf dem Paper “Traffic engineering with joint link weight and segment optimization” [**?** , 1] und den zugehörigen Repositories auf GitHub. In Projekt 1 steht dabei die Abwandlung von in Python implementierten Routing-Algorithmen im Vordergrund. In Projekt 2 werden diese Algorithmen dann in Mininet-Experimenten auf Basis des Papers verwendet.

KEYWORDS

JointHeur, waypoints, GreedyWPO

ACM Reference Format:
Gajann Sivarajah, Jan Schulte, and Phil Seißelberg. 2022. Fachprojekt Routingalgorithmen SS2022 - Projektbericht. In *Proceedings of ACM Conference (Conference’17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 GRUNDLAGEN DER PROJEKTARBEIT

Die hier beschriebene Projektarbeit basiert auf dem schon oben genannten Paper “Traffic engineering with joint link weight and segment optimization” [**?**], welches im Jahr 2021 veröffentlicht wurde. Das Paper fokussiert sich dabei auf das Thema Traffic Engineering. Hierbei wird der Routingablauf beinflusst, um eine möglichst geringe Auslastung der Links zu erreichen und “Daten-Staus” zu vermeiden. Dazu schägt das Paper eine Kombination zweier Ansätze vor. Zum einen können die Gewichtungen der Links modifiziert werden. Dies wird durch den HeurOSPF-Algorithmus ermöglicht. Zum anderen können Waypoints verwendet werden, um den Datenfluss über bestimmte Knoten umleiten zu können. Die Generierung der Waypoints erfolgt hierbei durch den GreedyWPO-Algorithmus. Das Zusammenfügen dieser beiden Lösungsansätze erfolgt in dem Paper durch die Einführung des JointHeur-Algorithmus. Dieser Algorithmus ist die Basis aller Projektarbeiten, die in diesem Bericht beschrieben werden. Die Zielfunktion ist dabei, die Maximale Auslastung aller Links(MLU) minimal zu halten.

Conference’17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 PROJEKT 1 - PRAKTISCHE IMPLEMENTIERUNGEN IN PYTHON

2.1 Einführung

Wie im Abstract erwähnt basiert das erste Project auf der Arbeit EINFÜGEN. Wir haben uns dazu entschieden, als Basis für unsere Algorithmen den im genanten Paper beschriebenen JointHeur zu verwenden. Einen besonderen Fokus haben wir auf die Anzahl der generierten Wegpunkte gelegt. Als Zielfunktion haben wir weiterhin (eine möglichst niedrige) MLU gewählt.

2.2 kWPO-JointHeur

Die im Paper beschriebene JointHeur-Routine sieht die Wahl von maximal einem Wegpunkt pro Demand vor. Um diese Einschränkung zu umgehen, definieren wir eine modifizierte Routine mit dem Namen kWPO-JointHeur. Im Gegensatz zur JointHeur soll hier die Routine zur Generierung von Wegpunkten, GreedyWPO, mehrfach wiederholt werden. Dabei soll in jeder Iteration die durch Wegpunkte modifizierte Anfrageliste der letzten Iteration als Eingabe eingespeist werden. Die Anzahl der Iterationen soll dabei durch einen Parameter k angegeben werden können. Für $k = 0$ würde die Ausführung des Algorithmus derer von HeurOSPF entsprechen. Der normale JointHeur-Algorithmus aus dem Paper entspricht der Ausführung mit $k = 1$. Für Werte $k > 1$ erzeugt die k-fache Anwendung maximal $2^k - 1$ Wegpunkte pro Demand.

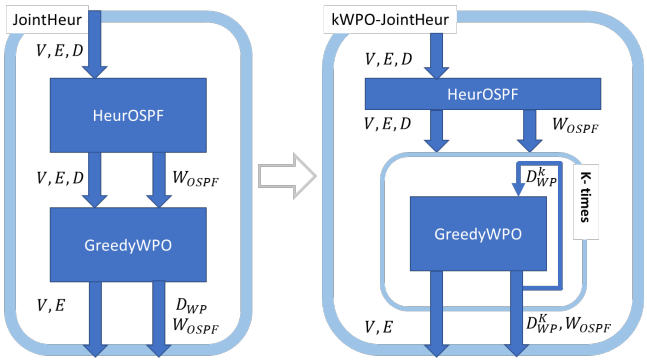

Figure 1: Konzept von kWPO-JointHeur

Zur weiteren Optimierung von kWPO-JointHeur betrachten wir die GreedyWPO-Komponente genauer. In dieser Routine werden die Anfragen zunächst nach ihrem Volumen sortiert und dann in dieser Reihenfolge hinsichtlich der Wegpunkte bearbeitet. Es fällt auf, dass die Reihenfolge einen Einfluss auf die Performance haben muss, da bereits gesetzte Wegpunkte nicht gelöscht werden und unbearbeitete Anfragen nicht berücksichtigt werden. Jedoch berücksichtigt das verwendete Sortierkriterium nicht die zugrunde liegende Topologie. Neben dem Anfragevolumen führen wir das

GreedyWPO(V,E,D,W):

$U_{min} \leftarrow MLU(V,E,D,W)$
$\boldsymbol{D_{WP} \leftarrow D \text{ sorted by demand value}}$
for demand $\psi = (s,t,d) \in D_{WP}$ *do*
  $wp_\psi \leftarrow None$
  for node $w \in V$ *do*
    $D' \leftarrow D_{WP} \backslash \psi \cup \{(s,w,d),(w,t,d)\}$
    $U \leftarrow MLU(V,E,D',W)$
    if $U < U_{min}$ then
      $wp_\psi \leftarrow w$
      $U_{min} \leftarrow U$
for each $wp_\psi \neq None$ *do*
  $D_{WP} \leftarrow D_{WP} \backslash \psi \cup \{(s,wp_\psi,d),(wp_\psi,t,d)\}$
return D_{WP}

Figure 2: Pseudocode zu kWPo-JointHeur

zusätzliche Sortierkriterium der Knotenkapazität ein. Die Knotenkapazität eines Knoten sei durch die Summe der verbundenen Kantenkapazitäten gegeben. Wie definieren die Knotenkapazität C_v wie folgt:

$$c_v = \sum_{e_{v_i} \in E} c(e_{v_i})$$

Im Rahmen von kWPO-JointHeur, sollen die Anfragen nun nach der Summe der Quellen- und Senkenknotenkapazität absteigend sortiert werden. Die Begründung liegt in der Tatsache, dass Knotenpaare mit hoher Kapazität durch alternative Wege hoher Kapazität entsprechend die MLU stärker senken können.

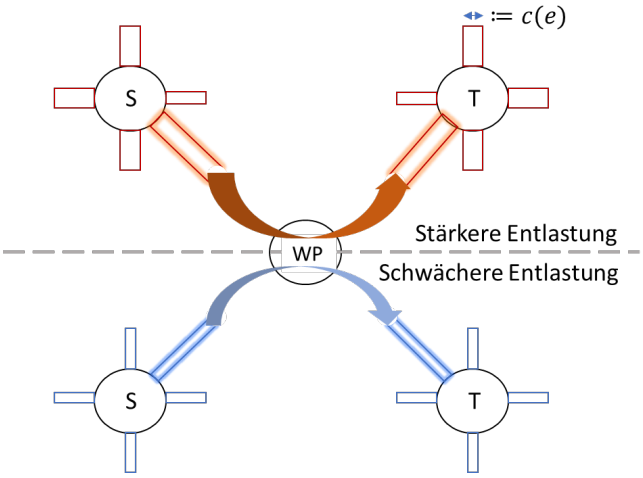

Figure 3: Einfluss der Knotenpakazität auf MLU

Ein erstes Beispiel für die Effektivität dieser Abwandlung kann in Abbildung 3 betrachtet werden. So kann bei einer Sortierung

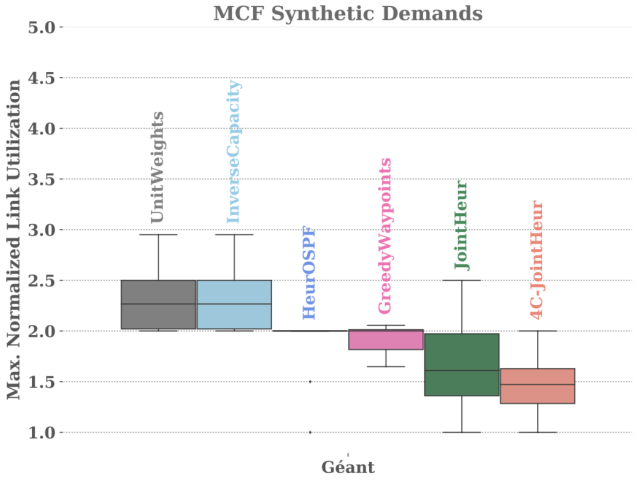

Figure 4: 4C-JointHeur bei Verwendung synthetischer Demands

nach Capacity und mit dem Parameter $k = 4$ auf künstlichen Demands ein besseres Ergebnis erreicht werden als mit dem klassischen JointHeur. In Abbildung 4 werden die Abwandlungen mit den

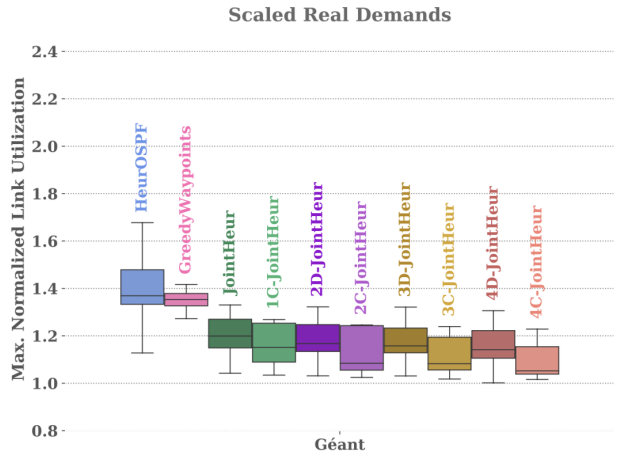

Figure 5: Verschiedene D- und C-JointHeur bei Verwendung tatsächlicher Demands

Parametern 1,2,3 und 4 betrachtet. Zusätzlich wird zu jedem dieser Fälle auch die Sortierung nach Capacity gegenübergestellt. Für das Ergebnis wurden echte Demands verwendet. Neben der Beobachtung, dass die Sortierung nach Capacity durchaus einen Unterschied macht, wird auch deutlich, dass mit höheren Werten für k auch eine bessere MLU einhergeht. In Abbildung 3 wird das Ergebnis für die Abwandlung 4C des JointHeur nochmal auf weiteren Topologien getestet und erreicht fast überall dort bessere Ergebnisse.

2.3 Topo-kWP-JointHeur

Während wir mit der ersten Idee JointHeur um mehr Waypoints je Demand erweitert haben schränken wir mit Topo-kWP-JointHeur

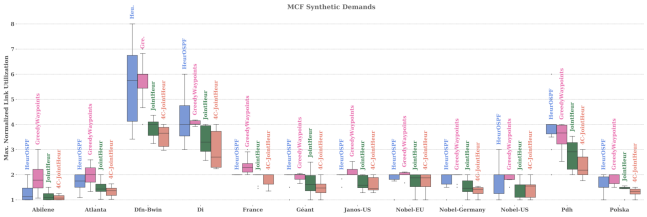

Figure 6: Vergleich von 1D- und 4C-JointHeur auf verschiedenen Topologien

die nutzbaren Waypoints für die gesamte Topologie ein. Durch diese Abwandlung kann der tatsächliche Einfluss der Waypoints auf die Performance von JointHeur besser veranschaulicht werden. Eine Steigerung der Performance, in diesem Fall also der MLU, kann durch diese Abwandlung offensichtlich nicht erwartet werden. Bestenfalls kann, bei entsprechend hohen Werten für die Beschränkung, das gleiche Ergebnis wie beim normalen JointHeur erreicht werden. Daher steht bei dieser Abwandlung eher im Vordergrund, für welche Beschränkungswerte und damit einhergehend wie stark, sich die Performance beeinflussen lässt.

Die praktische Abwandlung im Code ist simpel - Es muss lediglich ein Parameter k in die GreedyWPO-Komponente des JointHeur-Algorithmus eingepflegt werden. k fungiert dann als ein runter zählender Counter. Sollte GreedyWPO einen Waypoint wählen, so würde k verringert, bis k am Ende 0 beträgt. Somit kann sichergestellt werden, dass nur die erlaubte Anzahl an Wegpunkten erzeugt wird.
Nach der Implementierung haben wir den abgewandelten Algo-

GreedyWPO(V,E,D,W,K):

$k \leftarrow K$
$U_{min} \leftarrow MLU(V,E,D,W)$
$D_{WP} \leftarrow D$ *sorted by demand value*
for demand $\psi = (s,t,d) \in D_{WP}$ *do*
  $wp_\psi \leftarrow None$
  if $k \leq 0$ ***then break***
  for node $w \in V$ *do*
    $D' \leftarrow D_{WP} \backslash \psi \cup \{(s,w,d),(w,t,d)\}$
    $U \leftarrow MLU(V,E,D',W)$
    if $U < U_{min}$ then
      $wp_\psi \leftarrow w$
      $U_{min} \leftarrow U$
  if $wp_\psi \neq None$ ***then*** $k \leftarrow k-1$
for each $wp_\psi \neq None$ *do*
  $D_{WP} \leftarrow D_{WP} \backslash \psi \cup \{(s,wp_\psi,d),(wp_\psi,t,d)\}$
return D_{WP}

Figure 7: Pseudocode zu Topo-kWP-JointHeur

rithmus für mehrere Werte für k ausgeführt. Die folgenden Plots stellen die interessantesten Werte dar: Die erste Abbildung dieses

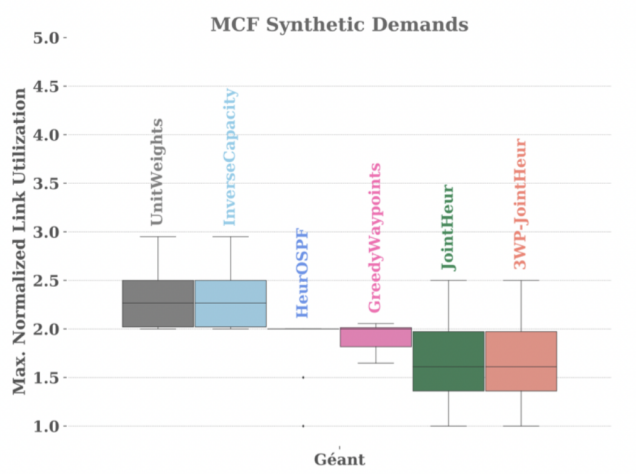

Figure 8: 3WP-JointHeur auf der Topologie Geant mit künstlichen Demands

Unterkapitels zeigt die Resultate von Topo-kWP-JointHeur mit einer Beschränkung von $k = 3$. es würden künstliche Demands und die Topologie Geant verwendet. Offensichtlich kann in diesem Fall mit $k = 3$ genau die Performance von JointHeur arbeiten. Für höhere Werte von k wird genau das selbe Ergebnis erzielt, somit nutzt JointHeur in diesem Fall nicht mehr als 3 Wegpunkte auf dem Weg durch die Topologie.
Die darauf folgende Abbildung zeigt dagegen die Verwendung echter Demands mit mehreren möglichen Werten für k. Offensichtlich verbessert sich die MLU mit jedem weiteren nutzbaren Waypoint. Die Abbildung verdeutlicht den Einfluss der Waypoints auf die MLU-Performance von JointHeur sehr gut. Um herauszufinden,

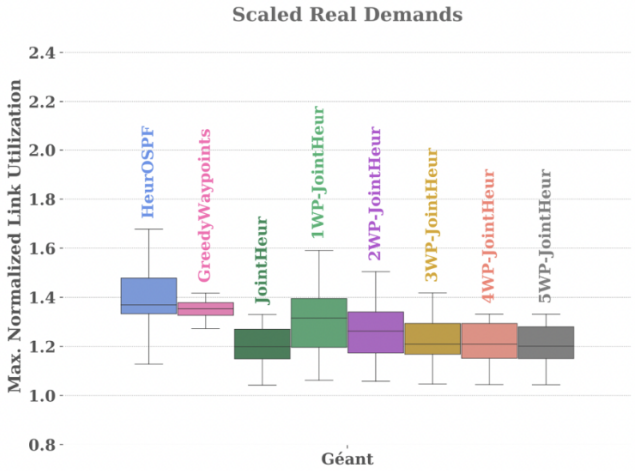

Figure 9: 3WP-JointHeur auf der Topologie Geant mit realen Demands

mit welcher möglichst kleinen Waypoint-Beschränkung eine minimale MLU erreicht werden kann, haben wir die Beschränkungen

auch auf mehreren Topologien betrachtet. Dies wird in der nächsten Abbildung dargestellt. Es hat sich dabei herausgestellt, dass mit einer Beschränkung von 3 Wegpunkten in den meisten Fällen die Ergebnisse des unbeschränkten JointHeur erreicht werden können. Es folgt daraus, dass die meisten Topologien höchstens 3 Wegpunkte erzeugen und nutzen. Es gab jedoch auch Topologien, auf welchem 3WP-JointHeur schlechter performt hat, darunter Janos-US, Nobel-Germany oder Polska. In diesem Fall konnten offensichtlich wichtige Wegpunkte aufgrund der Beschränkung nicht erzeugt werden.

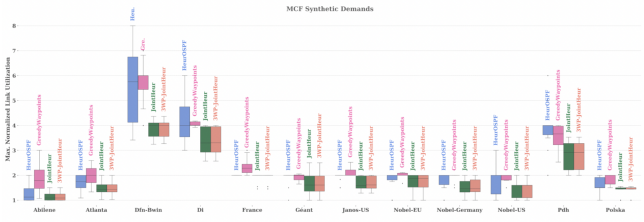

Figure 10: 3WP-JointHeur im Vergleich auf allen Topologien

2.4 Nodes-kWP-JointHeur

In unserer dritten Abwandlung betrachten wir ebenfalls eine Beschränkung der nutzbaren Wegpunkte, dieses Mal jedoch auf einzelne Knoten und nicht auf die gesamte Topologie bezogen. Ein Array K ordnet jedem Knoten v einen Parameter k_v (im Folgenden allgemein als k bezeichnet) zu, welcher angibt, wie oft dieser Knoten als Wegpunkt verwendet werden darf. Hierbei ist insbesondere interessant, welchen Einfluss die verschiedenen Werte für k auf die Performance haben und ob es sinnvoll und durchsetzbar ist, bestimmte Nodes durch $k = 0$ zu sperren, sodass diese nicht als Wegpunkt fungieren können.

Ein Problem bei der Implementierung ist jedoch, geeignete Werte für K zu finden. Welche Werte überhaupt gültig sind, hängt von der Topologie ab und kann somit grundsätzlich nicht verallgemeinert werden. Damit es trotzdem für alle Topologien einheitlich und leicht generierbar ist und weil uns insbesondere das Sperren von Knoten interessiert hat, haben wir uns in der praktischen Umsetzung haben dafür entschieden, jeden c_{gen}-ten (wobei $c_{gen} \in N$) Knoten beliebig oft als Wegpunkt zu erlauben und die anderen Knoten als Wegpunkte zu sperren ($k = 0$). So sind z. B. bei $c_{gen} = 4$ 75% der Knoten gesperrt, während 25% der Knoten beliebig oft als Wegpunkte verwendet werden dürfen. Im Folgenden sprechen wir auch von einer $\frac{1}{c}$-JointHeur.

Wie sich anhand der Durchführung auf der Topologie Geant erkennen lässt, wird, obwohl für ein größeres c_{gen} der Pool an Wegpunkten kleiner wird, das Ergebnis nicht zwingend schlechter. Der Grund dafür ist, dass auch bei einem größeren c_{gen} Wegpunkte nicht gesperrt sein können, die bei einem kleineren c_{gen} gesperrt sind. So sieht man z. B. in Abbildung 10, dass 1/5N-JointHeur besser als 1/4N-JointHeur performt.

In Abbildung 11 wird nochmal deutlich, dass die 1/3N-JointHeur nicht zwingend schlechter als JointHeur. Es hängt auch von der gewählten Topologie ab.

GreedyWPO(V,E,D,W,K):

$U_{min} \leftarrow MLU(V,E,D,W)$
$D_{WP} \leftarrow D$ *sorted by demand value*
for demand $\psi = (s,t,d) \in D_{WP}$ *do*
  $wp_\psi \leftarrow None$
  for node $w \in V$ *do*
    if $\boldsymbol{K[w] \leq 0}$ ***then continue***
    $D' \leftarrow D_{WP} \backslash \psi \cup \{(s,w,d),(w,t,d)\}$
    $U \leftarrow MLU(V,E,D',W)$
    if $U < U_{min}$ then
      $wp_\psi \leftarrow w$
      $U_{min} \leftarrow U$
  if $\boldsymbol{wp_\psi \neq None}$ ***then*** $\boldsymbol{K[wp_\psi] \leftarrow K[wp_\psi] - 1}$
for each $wp_\psi \neq None$ *do*
  $D_{WP} \leftarrow D_{WP} \backslash \psi \cup \{(s,wp_\psi,d),(wp_\psi,t,d)\}$
return D_{WP}

Figure 11: Pseudocode zu Nodes-kWP-JointHeur

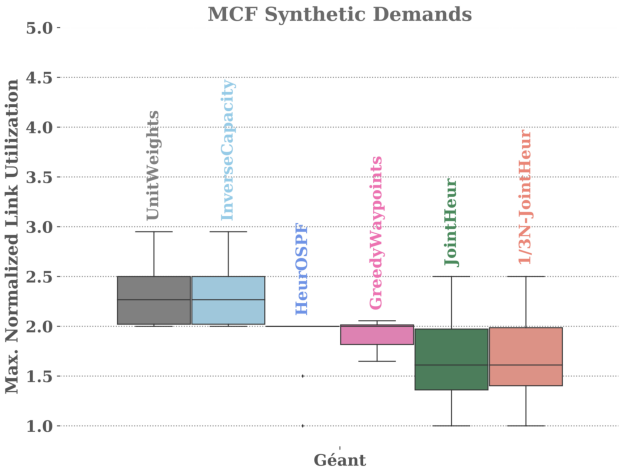

Figure 12: 1/3N-JointHeur auf künstlichen Demands im Vergleich

3 PROJEKT 2 - EXPERIMENTE IN MININET

3.1 Einführung

Auch die zweite Reihe von Experimenten basiert auf dem genanten Paper. Allerdings stand hier nicht primär die Implementierung in Python im Vordergrund. Ziel der Experimente war es stattdessen, unsere Abwandlungen aus Projekt 1 möglichst unter möglichst echten Bedingungen zu testen. Für die Simulation des Netzes haben wir keine tatsächliche Hardware verwendet. Stattdessen haben wir ein in Nanonet implementiertes Simulationsframework genutzt, um im Linux-Kernel ein realistisches Netzwerk abzubilden und das Verhalten der Algorithmen unter realen Bedingungen zu testen.

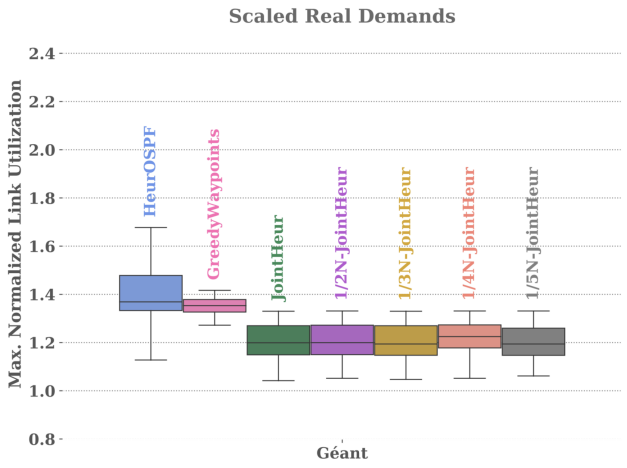

Figure 13: 1/k-JointHeur für verschiedene Werte und realen demands

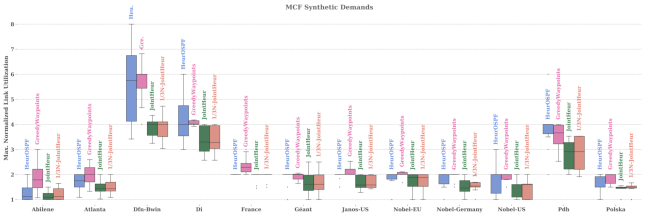

Figure 14: 1/3N-JointHeur auf verschiedenen Topologien getestet

Ziel war es zudem, unsere Ideen durch geeignete Skalierungen möglichst gut dastehen zu lassen.

3.2 kWPO-JointHeur

Der Algorithmus kWPO-JointHeur versucht einer Anfrage mehrere Wegpunkte zuzuweisen, um damit die Auslastung zu verbessern. Daher suchen wir eine Topologie, die von mehreren Wegpunkten profitieren kann. Dazu modifizieren wir die Joint-Topologie, die bereits im ursprünglichen Paper verwendet wurde. Durch die Duplizierung eines Teilbereiches dieser Topologie, entsteht eine zweiter Bereich, welcher für Wegpunkte empfänglich wird.

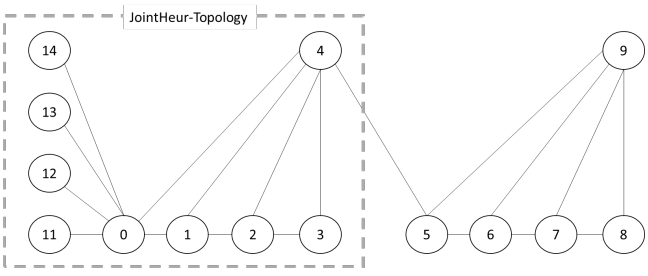

Figure 15: Topologie für kWPO-JointHeur-Experiment

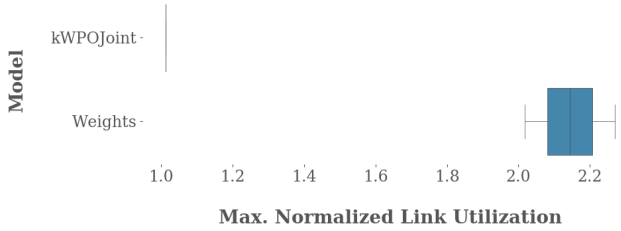

Figure 16: Ergebnis des Experiments

In Abbildung 13 können wir sehen, dass die Konfiguration mit 2 Wegpunkten einen sehr viel besseren MLU-Wert erzeugt, als die Steuerung über die reine Gewichtung von Kanten.

3.3 Topo-kWP-JointHeur

Unsere zweite Abwandlung konnte recht einfach implementiert werden. Zur Erinnerung: bei der zweiten Idee geht es um die Beschränkung der nutzbaren Waypoints für die gesamte Topologie. Da diese Abwandlung niemals zu einer besseren MLU führen kann, wird das Resultat dieses Algorithmus nicht besonders gut dargestellt, sondern eher schlecht. Für die Implementierung dieser Idee haben wir die Joint-Topologie verwendet, welche im zugehörigen Repository des Base-Paper gegeben war. In dieser haben wir dann einen Waypoint unbrauchbar gemacht, um eine Beschränkung zu simulieren. Das Resultat kann auf folgender Abbildung betrachtet werden:

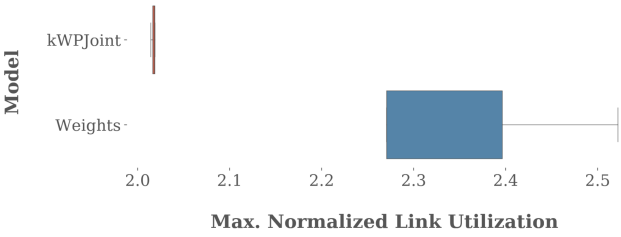

Figure 17: Resultat von kWP-JointHeur

Wie in den Abbildungen 9 und 10 erkennbar Offensichtlich führt die schon das entfernen eines Waypoints zu einer deutlichen Rechtsverschiebung im Plot und somit zu einer Verschlechterung der MLU auf über 2.0.

3.4 Nodes-kWP-JointHeur

Ausgehend von den Resultaten zur dritten Abwandlung aus Projekt 1 haben wir uns als Leitfrage gestellt: Welche Knoten sollen gesperrt werden?

4 REPLIKATION DER ERGEBNISSE EINER ANDEREN GRUPPE

Zu Ende der Bearbeitungszeit fand jeweils ein sogenannter Code-Freeze statt, bei dem die teilnehmenden Gruppen ihre Arbeit untereinander bewerten und replizieren sollten.

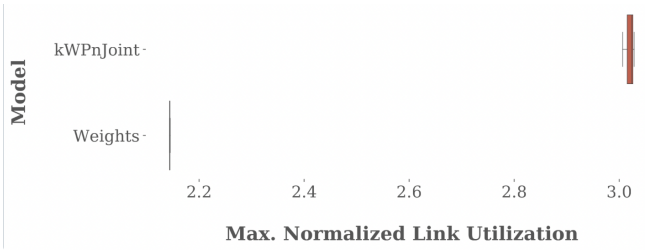

Figure 18: Resultat von kWPn-JointHeur

4.1 Code-Freeze - Projekt 1

Während der Bearbeitungszeit von Projekt 1 würde das folgende Repository repliziert: EINFÜGEN. Die Gruppe bot dabei zwei Wege an, ihre Experimente zu replizieren. Neben der Bereitstellungsform, in welchem auch der Code des Base-Papers bereitgestellt wird, ist auch eine vorkonfigurierte virtuelle Maschine verfügbar. Die gegebenen Dateien konnten ohne Probleme ausgeführt werden. Typischerweise muss allerdings für die Ausführung auf durchschnittlicher Hardware viel Zeit eingeplant werden. So dauerte die Ausführung des Experiments, welches mehrere Topologien nutzt, mehr als einen Tag. Die aus den Datei gewonnen Plots entsprachen denen, welche die Gruppe selbst angegeben hatte. Zusätzlich zu den bestehenden Plotter wurde auch zusätzlich ein weiteres Diagramm erzeugt, welches die tatsächliche Ausführungszeit der Algorithmen darstellt. Allerdings hätte die Darstellung für die Ausführung bei künstlichen Demands besser skaliert sein können, da einige in der Legende genannten Algorithmen im Plot nicht zu sehen sind, siehe beispielsweise IndependentPathWaypoints.

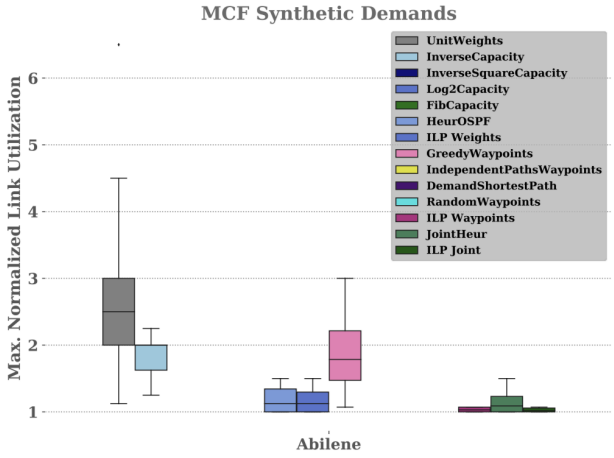

Figure 19: Beispielplot aus der Replikation der anderen Gruppe

4.2 Code-Freeze - Projekt 2

Auch das zweite Projekt der anderen Gruppe basierte auf den Grundideen von Projekt 1 und wurde in Form von Experimenten mit Nanonet durchgeführt. Auch hier ist wieder positiv anzumerken, dass der Code wieder zusätzlich zum bestehenden Weg auch über ein vorbereitetes VM-Image direkt ausgeführt werden kann. Die zur Ausführung nötigen Schritte sind zudem nochmal beschrieben. Bei der reinen Ausführung des Codes hatten wir keine Probleme. Allerdings waren nachträglich einige Modifikationen an der Datei nötig, welche zum Plotten der Ergebnisse benötigt wird. Nach den Modifikationen haben wir folgenden Boxplot erhalten:

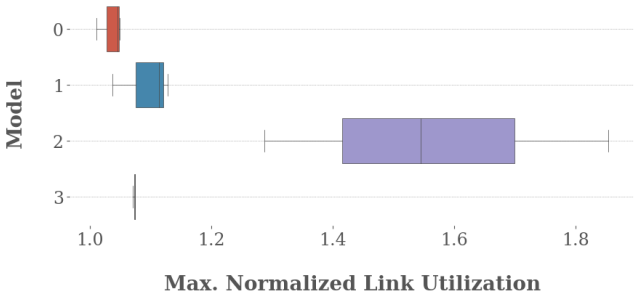

Figure 20: Erzeugter Boxplot aus der zweiten Replikation