

Fachprojekt Routingalgorithmen SS2022 - Projektbericht

Gajann Sivarajah
Jan Schulte
Phil Seielberg

ABSTRACT

Dieser Bericht enthlt 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 zugehrigen 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 Seielberg. 2022. Fachprojekt Routingalgorithmen SS2022 - Projektbericht. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 4 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 verffentlicht wurde. Das Paper fokussiert sich dabei auf das Thema Traffic Engineering. Hierbei wird der Routingablauf beeinflusst, um eine mglichst geringe Auslastung der Links zu erreichen und "Daten-Staus" zu vermeiden. Dazu schtzt das Paper eine Kombination zweier Anstze vor. Zum einen knnen die Gewichtungen der Links modifiziert werden. Dies wird durch den HeurOSPF-Algorithmus ermglicht. Zum anderen knnen Waypoints verwendet werden, um den Datenfluss ber bestimmte Knoten umleiten zu knnen. Die Generierung der Waypoints erfolgt hierbei durch den GreedyWPO-Algorithmus. Das Zusammenfgen dieser beiden Lsungsanstze erfolgt in dem Paper durch die Einfhrung 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.

2 PROJEKT 1 - PRAKTISCHE IMPLEMENTIERUNGEN IN PYTHON

2.1 Einfhrung

Wie im Abstract erwhnt basiert das erste Project auf der Arbeit EINFGEN. Wir haben uns dazu entschieden, als Basis fr unsere Algorithmen den im genannten Paper beschriebenen JointHeur zu verwenden. Einen besonderen Fokus haben wir auf die Anzahl der generierten Wegpunkte gelegt. Als Zielfunktion haben wir weiterhin (eine mglichst niedrige) MLU gewhlt.

2.2 kWPO-JointHeur

Die im Paper beschriebene JointHeur-Routine sieht die Wahl von maximal einem Wegpunkt pro Demand vor. Um diese Einschrnkung 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 knnen. Fr $k = 0$ wrde die Ausfhrung des Algorithmus derer von HeurOSPF entsprechen. Der normale JointHeur-Algorithmus aus dem Paper entspricht der Ausfhrung mit $k = 1$. Fr Werte $k > 1$ erzeugt die k -fache Anwendung maximal $2^k - 1$ Wegpunkte pro Demand. Zur weiteren Optimierung

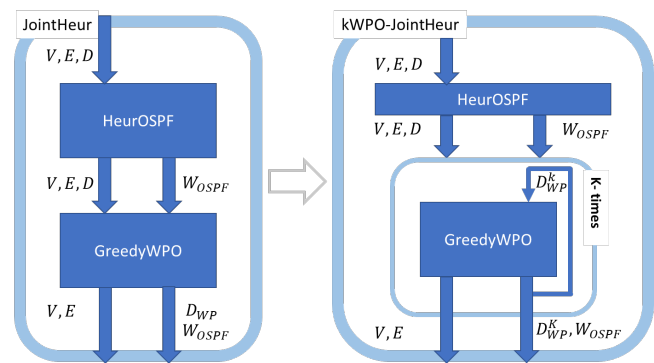


Figure 1: Konzept von kWPO-JointHeur

von kWPO-JointHeur betrachten wir die GreedyWPO-Komponente genauer. In dieser Routine werden die Anfragen zunchst nach ihrem Volumen sortiert und dann in dieser Reihenfolge hinsichtlich der Wegpunkte bearbeitet. Es fllt auf, dass die Reihenfolge einen Einfluss auf die Performance haben muss, da bereits gesetzte Wegpunkte nicht gelscht werden und unbearbeitete Anfragen nicht

berücksichtigt werden. Jedoch berücksichtigt das verwendete Sortierkriterium nicht die zugrunde liegende Topologie. Neben dem Anfragevolumen führen wir das 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.

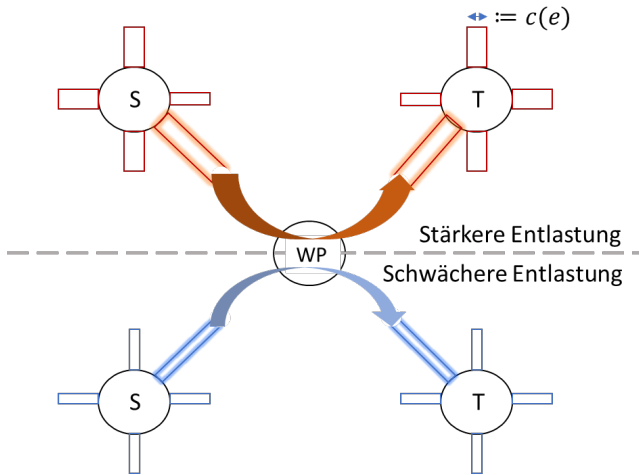


Figure 2: Einfluss der Knotenkapazität auf MLU

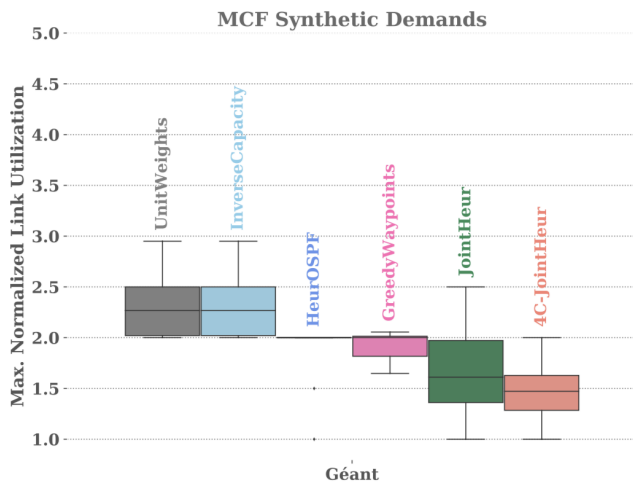


Figure 3: 4C-JointHeur bei Verwendung synthetischer Demands

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

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

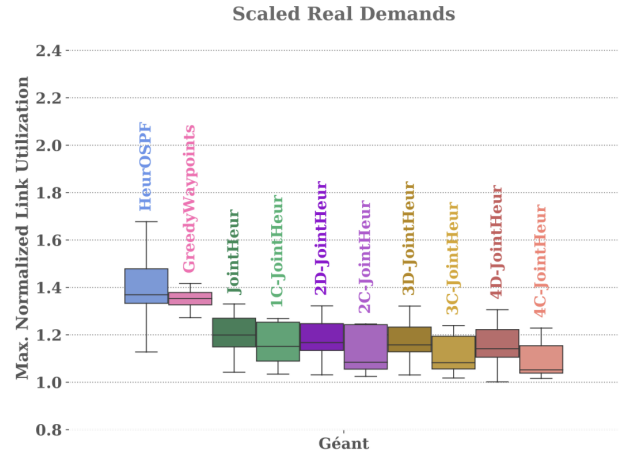


Figure 4: Verschiedene D- und C-JointHeur bei Verwendung tatsächlicher Demands

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.

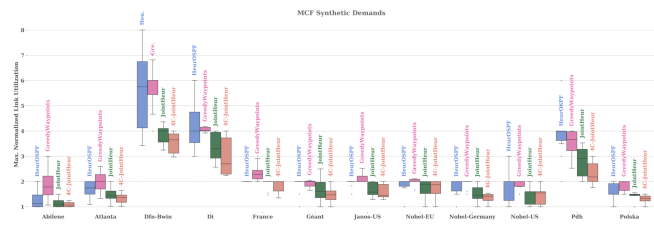


Figure 5: Vergleich von 1D- und 4C-JointHeur auf verschiedenen Topologien

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

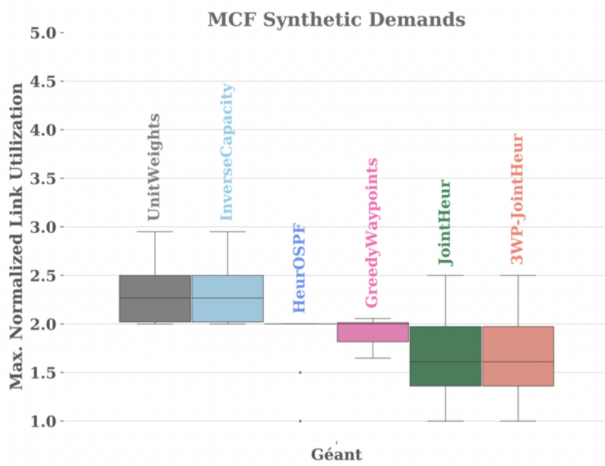


Figure 6: 3WP-JointHeur auf der Topologie Géant mit künstlichen Demands

Unterkapitels zeigt die Resultate von Topo- k WP-JointHeur mit einer Beschränkung von $k = 3$. es würden künstliche Demands und die Topologie Géant 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, 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

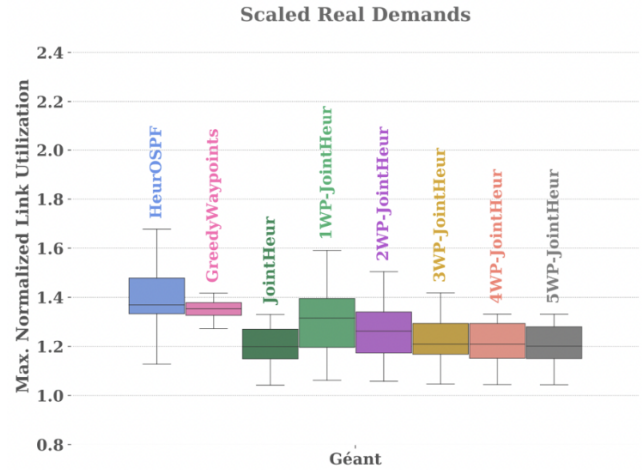


Figure 7: 3WP-JointHeur auf der Topologie Géant mit realen Demands

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.

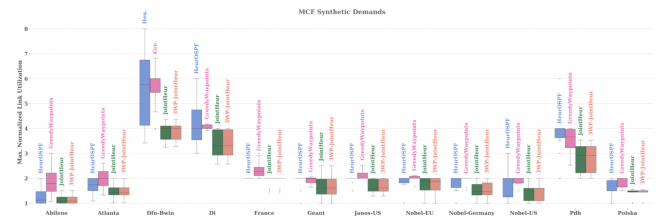


Figure 8: 3WP-JointHeur im Vergleich auf allen Topologien

2.4 Nodes- k WP-JointHeur

In unserer dritten Abwandlung betrachten wir ebenfalls eine Beschränkung der nutzbaren Wegpunkte, dieses mal jedoch auf einzelne Nodes und nicht auf die gesamte Topologie bezogen. Dazu führen wir für jede Node einen Counter ein und werten das Ergebnis für verschiedene Werte von k und verschiedene Topologien aus. Hierbei ist insbesondere interessant, welcher Einfluss die verschiedenen Werte für k auf die Performance haben und ob es sinnvoll und durchsetzbar ist, bestimmte Nodes durch $k = 0$ zu meiden (und somit das gegenteilige eines Waypoints zu erzeugen). 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. Daher muss man allgemeinere Regeln für k verwenden und anhand derer die konkret verwendeten Werte abgeleitet werden.

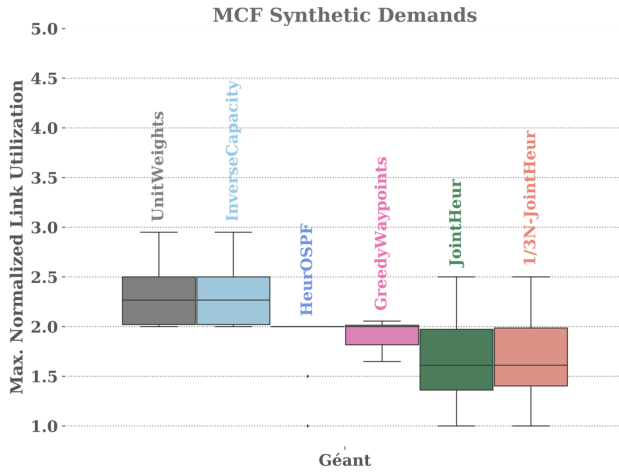


Figure 9: 1/3N-JointHeur auf künstlichen Demands im Vergleich

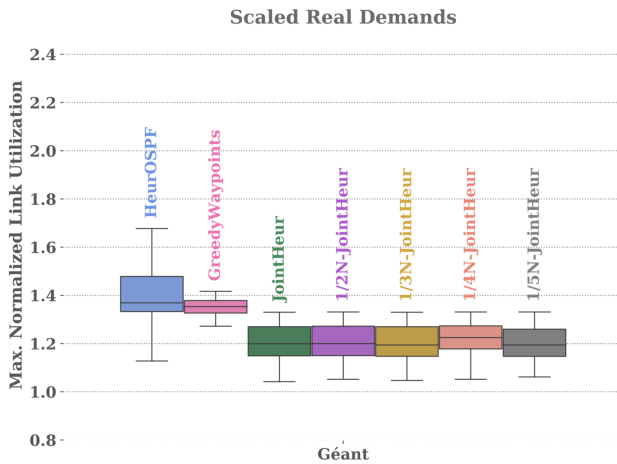


Figure 10: 1/k-JointHeur für verschiedene Werte und realen demands

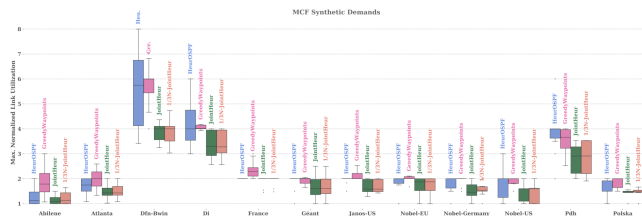


Figure 11: 1/3N-JointHeur auf verschiedenen Topologien getestet

3 PROJEKT 2 - EXPERIMENTE IN MININET

3.1 Einführung

Auch die zweite Reihe von Experimenten basiert auf dem genannten 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. 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.

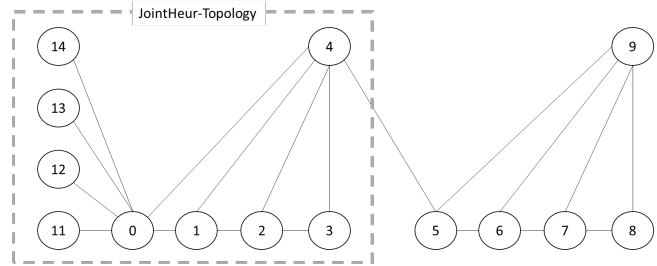


Figure 12: Topologie für kWPO-JointHeur-Experiment

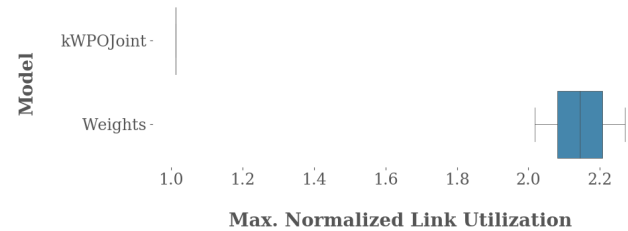


Figure 13: Ergebnis des Experiments

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

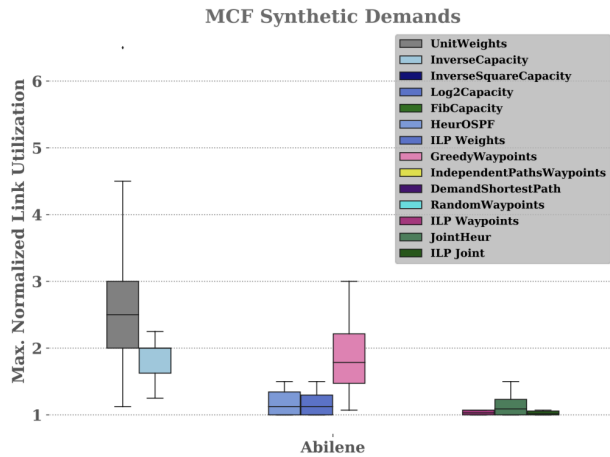


Figure 15: Beispielplot aus der Replikation der anderen Gruppe

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:

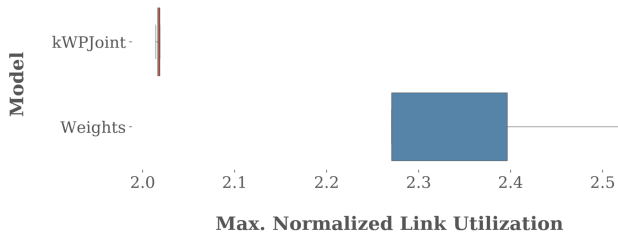


Figure 14: Resultat von kWP-JointHeur

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.

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. In diesem Abschnitt werden die implementierten Ideen der von uns betrachteten Gruppe kurz erklärt. Danach wird auf weitere Beobachtungen eingegangen.

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 Bereitstellung, 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. 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.

4.2 Code-Freeze - Projekt 2