Graph Golf
Betrachtung des Order-Degree-Problems

Diplomarbeit vorgelegt von:

Robert Waniek

Matrikelnummer: 309118

Diplom-Studiengang: Techno- und Wirtschaftsmathematik

Fakultät II – Mathematik und Naturwissenschaften

Technische Universität Berlin

Erstprüfer: Prof. Dr. Thorsten Koch (koch@zib.de)

Zweitprüfer: Prof. Dr. Ralf Borndörfer (borndoerfer@zib.de)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 24.09.2021

Robert Waniek

Inhaltsverzeichnis

1. Einleitung

Das Order Degree Problem (ODP) ist ein graphentheoretisches Minimierungsproblem. Wie in Abschnitt 2 vorgestellt, ist das Ziel des ODP die Bestimmung des minimalen Durchmessers zu einer vorgegebenen Knotenanzahl und einem maximalen Knotengrad.

Während das betrachtete Problem historisch um 1960 als Planungsproblem für ein Netzwerk aus Flugverbindungen entstand, bekommt es seit 2010 einen neuen Praxisbezug durch die Vernetzung von Rechenzentren. Durch technische Weiterentwicklungen in der Zwischenzeit waren klassische Fragestellungen nach Redundanz und Kapazitäten zunehmend praktisch lösbar. Als neuer Aspekt rückten hohe Latenzen durch viele Zwischenknoten bei Netzwerkverbindungen in den Fokus.

Mit dem Begriff „Graph Golf“ wurde das Problem 2015 als jährlicher Wettbewerb und internationaler Praxisaustausch aufgegriffen. Abschnitt 3 gibt einen Überblick zu Publikationen mit Bezug zum Wettbewerb, für den neue Verfahren und Verbesserungen bestehender Verfahren erarbeitet werden.

In Abschnitt 4 werden grundlegende Aspekte zur theoretischen Komplexität des ODP betrachtet, bevor ab Abschnitt 5 die praktische Modellierung und Lösung des ODP als gemischt-ganzzahliges Programm (MIP) und mittels Heuristiken betrachtet wird.

Nach der Betrachtung des Test-Setups, grundlegender Spezialfälle und einer einfachen Startheuristik in Abschnitt 6, wird in Abschnitt 7 eine Annahme zur Struktur guter ODP-Lösungen getroffen, die als Grundlage für weitere Testrechnungen verwendet wird.

Die Ergebnisse der durchgeführten Testrechnungen mit den erarbeiteten Heuristiken und MIP-Modellen werden in Abschnitt 8 vorgestellt. Zum Abschluss werden mögliche Vertiefungen der betrachteten Ideen diskutiert.

Als Auswahl der Probleminstanzen wird die Enumeration der Knotenanzahl bis 99 mit allen möglichen maximalen Knotengraden betrachtet. Des Weiteren erfolgte die Erarbeitung während des Graph-Golf-Wettbewerbs 2021, sodass die dort vorgestellten Instanzen ebenfalls betrachtet werden.

Zum Zeitpunkt der Fertigstellung dieser Arbeit ist der Wettbewerb noch im Gange. Im Zeitraum 27. April bis 25. Juli 2021 konnten Teilnehmende ihre Lösungen verdeckt einreichen. Seit 26. Juli 2021 sind alle eingereichten Lösungen veröffentlicht. Der Wettbewerb dauert noch bis zum 11. Oktober an. (vgl. Graph Golf Rules, 2021)

1.1. Verwendete Abkürzungen

(in alphabetischer Reihenfolge)

AODDEP ASPL Order Degree Diameter Entscheidungs-P.

AODP ASPL Order Degree Problem

APSP All Pairs Shortest Path

ASPL Average Shortest Path Length

DDP Degree Diameter Problem

MCF Multi Commodity Flow

MIP gemischt-ganzzahliges Programm

ODDEP Order Degree Diameter Entscheidungs-P.

ODiP Order Diameter Problem

ODP Order Degree Problem

SSSP Single Source Shortest Path

2. Problemdefinition & Begriffe

Das ODP ist eine Ausprägung eines graphentheoretischen Meta-Problems der Optimierung auf drei konkurrierenden Zielsetzungen:

- order: Maximieren der Knotenanzahl n eines Graphen
- degree: Minimierung des maximalen Knotengrades d eines Graphen
- diameter: Minimierung des Durchmessers k eines Graphen

Durch die Fixierung von jeweils zwei Parametern und Optimierung des dritten Parameters ergeben sich folgende Optimierungsprobleme:

- Order Degree Problem (ODP)
- Degree Diameter Problem (DDP)
- Order Diameter Problem (ODiP)

Die fixierten Parameter sind dabei jeweils namensgebend. Das ODiP gilt als bisher wenig berücksichtigt. Das DDP gilt durch eine Vielzahl an Veröffentlichungen seit 1980 als umfangreich betrachtet (siehe Einleitung von Abschnitt 3).

2.1. Graphentheoretische Formulierung

Definition 1 (Durchmesser)

Der Durchmesser d eines Graphen $G = (V, E)$ ist das Maximum der Längen der kürzesten s-t-Pfade aller Knoten $s, t \in V$ (auch: All Pairs Shortest Path (APSP)). □

Definition 2 (Order Degree Problem (ODP))

Gegeben seien die Knotenanzahl $|V| = n \in \mathbb{N}$ eines ungerichteten Graphen G und der maximale Knotengrad $d \in \mathbb{N}$.

Gesuchte Lösung des ODP ist der minimale Durchmesser $k(n, d) \in \mathbb{N}$ und eine zugehörige Kantenmenge $E \subseteq V \times V$, sodass $G = (V, E)$ ein ungerichteter, einfacher, zusammenhängender Graph ist, für alle Knotenpaare $(s, t) \in V \times V$ der kürzeste s-t-Pfad nicht länger ist als $k(n, d)$ und für alle Knoten $v \in V$ der Knotengrad $\delta(v)$ maximal d beträgt. □

Als Vorbereitung zur Literatureinordnung (Abschnitt 3) und zu den Komplexitätsbetrachtungen (Abschnitt 4) wird das zugehörige Entscheidungsproblem formuliert:

Definition 3 (Order Degree Diameter Entscheidungs-P. (ODDEP))

Gegeben seien $n \in \mathbb{N}$, $d \in \mathbb{N}$ und $k \in \mathbb{N}$. Gesucht ist die Antwort auf folgende Frage:

Existiert ein ungerichteter, einfacher, zusammenhängender Graph $G = (V, E)$ mit Knotenanzahl $n = |V|$ und Kantenmenge $E \subseteq V \times V$, sodass der maximale Knotengrad höchstens d und der Durchmesser höchstens k beträgt? □

2.2. Zielfunktionserweiterung zum Graph-Golf-Wettbewerb

Ausgehend vom Durchmesser eines Graphen als Maximum des APSP als Worst-Case-Betrachtung bietet sich zusätzlich eine Average-Case-Betrachtung anhand des Mittelwerts des APSP als zusätzliches Minimierungsziel an.

Definition 4 (Average Shortest Path Length (ASPL))

Die ASPL l eines Graphen $G = (V, E)$ ist der Durchschnitt der Längen der kürzesten s-t-Pfade aller Knotenpaare $(s, t) \in V \times V$. □

Der Durchschnitt aller Längen der kürzesten s-t-Pfade SP_{st} eines Graphen mit n Knoten summiert über n Knoten jeweils $n - 1$ Pfade, da der Pfad von einem Knoten zu sich selbst mit Länge 0 nicht gezählt wird. Daraus ergibt sich:

$$l := \sum_{s \in V} \sum_{\substack{t \in V \\ s \neq t}} \frac{SP_{st}}{n \cdot (n-1)} \tag{1}$$

Definition 5 (ASPL Order Degree Problem (AODP))

Gegeben sei ein ODP wie in Definition 2. Gesuchte Lösung des AODP ist die minimale ASPL $l(n, d)$ mit zugehöriger Kantenmenge $E \subseteq V \times V$ aus den optimalen Lösungen des zugrundeliegenden ODP. □

Beobachtung: Jede zulässige bzw. optimale Lösung für das AODP ist auch zulässig bzw. optimal für das zugehörige ODP.

AODP-Zielfunktion: Für die gleichzeitige Minimierung von Durchmesser und nachrangigem ASPL wird eine kombinierte Zielfunktion benötigt. Im Hinblick auf die MIP-Modellierung in Abschnitt 5 wird die Summierung als Ansatz gewählt, wobei durch Gewichtung eine Priorisierung realisiert wird. Zur Vermeidung von numerischen Problemen durch eine unnötig hohe Skalierung („Big-M-Formulierung“) erfolgt zur Ermittlung des minimal notwendigen Skalierungsfaktors folgende Abschätzung:

$$\begin{aligned} k &\geq SP_{st} && \forall\, s, t \in V : s \neq t \\ &= \frac{1}{n} \cdot \sum_{s \in V} SP_{st} && \forall\, t \in V : s \neq t \\ &= \frac{1}{n(n-1)} \cdot \sum_{s \in V} \sum_{\substack{t \in V \\ s \neq t}} SP_{st} = l \end{aligned} \tag{2}$$

Aus $k \geq l$ lassen sich mit $a \geq 1$ folgende zulässige Zielfunktionen für das AODP ableiten:

$$\min \text{ AODP}: \quad a \cdot k + l \tag{3}$$

Da die ASPL im Optimum meist kleiner als 10 ist, wird zur leichteren Ablesbarkeit von Durchmesser und ASPL im Zielfunktionswert in Abschnitt 5 zur MIP-Modellierung der Parameter $a = 10$ verwendet.

2.3. Notation kurzgefasst

- Knotenanzahl $n = |V|$
- maximaler Knotengrad d
- Durchmesser k
- ASPL l
- ODP : minimaler Durchmesser $k(n, d)$
- AODP : minimaler Durchmesser $k(n, d)$ & minimaler ASPL $l(n, d)$

AODP kurzgefasst:

$$\begin{aligned}
\text{Zielfunktion} \quad & \min \ 10 \cdot k + l && (4)\\
\text{Durchmesser} \quad & \forall\, s, t \in V : SP_{st} \leq k && (5)\\
\text{ASPL} \quad & \frac{1}{n \cdot (n-1)} \sum_{s \in V} \sum_{\substack{t \in V \\ s \neq t}} SP_{st} = l && (6)\\
\text{APSP} \quad & \forall\, s, t \in V : \ \text{berechne } SP_{st} && (7)\\
\text{Knotengrad} \quad & \forall\, i \in V : \delta(i) \leq d && (8)
\end{aligned}$$

3. Stand der Literatur

Historisch stammt die Fragestellung (laut Chung, 1987) aus dem Jahr 1960 und wurde von Erdős und Rényi, 1963 in einer ungarischen Publikation mit russischer und englischer Zusammenfassung betrachtet. Eine erweiterte Version erschien als englische Publikation von Erdős et al., 1966.

A possible interpretation of determining $F_2(n, k)$ is as follows: we want to establish a network of air connections between n airports, so that the maximal number of airports with which any given airport is connected by a (direct) connection is equal to k, further that it should be possible to travel from any given airport to any other either directly or by changing the plane exactly once; (it is supposed that each plane travels from an airport A to another airport B and back, without landing at intermediate places); the problem is to determine the minimal number of air connections with which such a communication network can be realized.

Abbildung 1: Erdős und Rényi, 1963, S.641

Die mögliche Interpretation (Abbildung 1) diskutiert $F_2(n, k)$ als minimale Kantenanzahl für einen Graphen der Menge $H_2(n, k)$.

Die Zahl 2 steht für den Durchmesser und n für die Anzahl der Knoten. Abweichend von der in dieser Arbeit verwendeten Notation steht k hier für den maximalen Knotengrad. In der Notation von Definition 2 ergibt sich aus den Definitionen von Erdős et al., 1966, S.215:

$$F_k(n, d) = \min_{G_n \in H_k(n,d)} E(G_n)\,,$$

wobei $G_n = (V, E)$ ein Graph mit $|V| = n$ Knoten und $E(G_n) = |E|$ Kanten ist. $H_k(n, d)$ ist der Lösungsraum des in Abschnitt 2 vorgestellte Meta-Problems. Das ODDEP und die daraus abgeleiteten Optimierungsprobleme sind Varianten des ursprünglichen Optimierungsproblems. Das Ursprungsproblem von Erdős und Rényi, 1963 fixiert alle drei Parameter (n, d, k) und minimiert die Kantenanzahl.

Das DDP und ähnliche Probleme werden von Chung, 1986, 1987 als offen betrachtet. Ausgehend von theoretischen Grundlagen zu Moore-Graphen von Hoffman und Singleton, 1960 wurde das DDP in der Folge u.a. von Cerf et al., 1974, McKay et al., 1998, Miller und Slamin, 2000 und Miller und Širáň, 2013 weiter betrachtet, sowie bekannte DDP-Lösungen zusammengetragen (vgl. Combinatorics Wiki, 2019).

Die im letzten Satz von Abbildung 1 angesprochene Realisierung eines Kommunikationsnetzwerks stellt seit ca. 2010 eine praktische Herausforderung für die Vernetzung von Rechenzentren und den Betrieb von Internetanbindungen dar, da die Anzahl der

Zwischenknoten inzwischen signifikante Auswirkungen auf die Gesamtlatenz der Verbindungen zwischen den Endknoten im Netzwerk hat. Dieser Sachverhalt wurde u.a. von Shin et al., 2011 im Vergleich von klassischen Vernetzungsstrukturen mit zufallserweiterten Grundtopologien vertieft.

Durch den praktischen Bedarf und inzwischen vorhandene Rechenkapazitäten erfolgten für das ODP neben graphentheoretischen Überlegungen zu besseren Schranken und Konstruktionsheuristiken für gute (ggf. suboptimale) Lösungen für den praktischen Einsatz außerdem praktische Evaluationen im Vergleich zu bisher genutzten Strukturen (u.a. Besta und Hoefler, 2014; Koibuchi et al., 2012; Singla et al., 2012).

Zur Erhöhung der Aufmerksamkeit wird seit 2015 der Wettbewerb Graph Golf abgehalten (Graph-Golf-Website, 2021).

3.1. Theoretische Betrachtungen

Trivialer Spezialfall für den Durchmesser 1 und maximalen Knotengrad $d \geq n - 1$ ist der vollständige Graph K_n. Bereits in Erdős und Rényi, 1963 wurden Spezialfälle für die Durchmesser 2 und 3 und grundlegenden Abschätzungen für das Meta-Problem betrachtet. Diese Fälle wurden in weiteren Veröffentlichungen zum DDP z. B. von McKay et al., 1998 sowie der historisch vorausgehenden Publikation von Hoffman und Singleton, 1960 betrachtet, die auch den Begriff Moore-Graph als DDP-Optimallösung definiert.

Für $d = 1$ gibt es für $n > 2$ keinen zusammenhängenden Graphen. Der Kreisgraph C_n löst für $d = 2$ das ODP trivial mit dem Durchmesser $\left\lfloor \frac{n}{2} \right\rfloor$. Weitere Spezialfälle werden in Abschnitt 6 im Hinblick auf die Enumeration der ersten Probleminstanzen betrachtet.

Davon ausgehend erfolgte die Betrachtung komplexer Konstruktionen aus Teilstrukturen. Da die Struktur des Lösungsgraphen für das ODP unklar ist, werden verschiedene Strukturansätze verfolgt, z. B. als Heuristik oder für praktische Anwendungen durch Fujimoto und Kobayashi, 2018. Anhand einer Beobachtung bei Optimallösungen wird dieser heuristische Ansatz in Abschnitt 7 aufgegriffen.

3.1.1. Schranken

Aus allgemeinen Abschätzungen und Spezialfällen für Graphenstrukturen/-klassen bestehen Schranken, die zur Verbesserung des Löseprozesses (siehe Abschnitt 6) herangezogen werden. Aufgrund des gemeinsamen Meta-Problems sind viele Erkenntnisse zum DDP auf das ODP übertragbar.

Für die Minimierungsprobleme ODP und AODP ist jede zulässige Lösung eine obere bzw. primale Schranke für die Optimallösung.

Die aus der Betrachtung des DDP bekannte Moore-Schranke gilt laut Miller et al., 2013, S.328 ebenso für das ODP und das AODP als untere Schranke für den Durchmesser:

$$k(n,d) \geq K_{n,d} = \left\lceil \log_{d-1}\left(\frac{(n-1)(d-2)}{d} + 1\right)\right\rceil$$

für $n-1 > d > 2$. (Trivialfälle wie $d \leq 2$ oder $K_{n,d} = 1$ werden in Abschnitt 6 betrachtet.)

Für Instanzen mit $d > 2$ und $n = d^2+1$ oder auch $n = d^2$ zeigten Erdős et al., 1980, dass keine Moore-Graphen existieren. Folglich existieren auch keine Optimallösungen für das ODP bzw. AODP, die die Moore-Schranke mit Gleichheit erfüllen.

Laut Cerf et al., 1974, Korollar 1, S.339 gilt für die ASPL bei ungerichteten, regulären, zusammenhängenden Graphen die untere Schranke:

$$l(n,d) \geq L_{n,d} = \frac{S_{n,d} + K_{n,d}R_{n,d}}{n-1}$$
$$\text{mit } S_{n,d} = \sum_{i=1}^{K_{n,d}-1} i \cdot d \cdot (d-1)^{i-1}$$
$$\text{und } R_{n,d} = n - 1 - \sum_{i=1}^{K_{n,d}-1} d \cdot (d-1)^{i-1}$$

Hinsichtlich der Erreichbarkeit der Schranke erwähnt Cerf et al., 1974, S.341 für Graphen mit 22 Knoten und maximalem Knotengrad 3:

> But even more interesting is the fact that for $N = 22$ and $V = 3$, no graph can achieve the lower bound.

Weiterhin gibt es für $n \cdot d$ ungerade keine d-regulären Graphen, sodass optimale Lösungen für das AODP mit ASPL echt größer als die Cerf-Schranke erwartbar sind.

3.1.2. Komplexität

In Hasegawa und Handa, 2018, S.237 wird das ODP als „known as NP-hard“ bezeichnet, jedoch kein Beleg angeführt. Denkbar ist eine Verwechslung mit der Zugehörigkeit zu NP, die leicht zu zeigen ist (siehe Satz 1 in Abschnitt 4).

Ein ähnlicher Zitierfehler findet sich bei Literatur zum DDP mit Verweis auf Dekker et al., 2012. Dort wird ein Nachweis geführt, dass das Largest Degree-Bounded Subgraph Problem (MaxDDBS) NP-schwer ist, da es bewiesen NP-schwere Spezialfälle enthält. Weiterhin ist das DDP der Spezialfall des MaxDDBS auf einem vollständigen Graphen als Host-Graph. Dies lässt jedoch keine Rückschlüsse zu, ob das DDP NP-schwer ist, was Dekker et al., 2012, S.251 adressieren mit:

> Note that this does not imply that DDP is NP-hard; actually, the complexity of DDP is not known to-date.

Auffällige Gemeinsamkeit der von Dekker et al., 2012 angeführten NP-schweren Spezialfälle des MaxDDBS ist die Suche einer Lösung in einem Subgraphen als Problembestandteil, während Anforderungen an Kontengrad und Durchmesser abgeschwächt werden. In diesem Sinne gleicht die Kantenauswahl im DDP bzw. ODP der Verwendung des vollständigen Graphen als Host-Graph. Durch diese starke Einschränkung sind die Erkenntnisse zum MaxDDBS nicht unmittelbar übertragbar.

Nach Stand der Literatur ist die NP-Vollständigkeit von ODP und DDP offen. Weitere Aspekte zur Komplexität werden in Abschnitt 4 betrachtet.

3.2. Praktische Lösungsverfahren

Lösungsgüte: Da für einen Großteil der Probleminstanzen die unteren Schranken mit Gleichheit erfüllbar sind, können gefundene optimale Lösungen für diese Instanzen unmittelbar erkannt werden. Die Güte heuristisch gefundener Lösungen wird entsprechend anhand der Moore-Schranke für den Durchmesser sowie der Cerf-Schranke für die ASPL (vgl. Abschnitt 3.1.1) in absoluter und relativer Differenz bewertet.

3.2.1. APSP-Implementation

Mit dem Durchmesser als Zielfunktion des ODP muss der Wert einer Lösung durch Berechnung des APSP erfolgen, mindestens am Ende des Löseprozesses zum Vergleich mit anderen Lösungen und Schranken.

Abhängig von der Modellierung und Implementation des jeweils verfolgten Lösungsansatzes werden verschiedene bekannte Algorithmen eingesetzt:

- klassischer All Pairs Shortest Path (APSP): Floyd-Warshall-Algorithmus
- spezieller APSP für ungewichtete Graphen von Seidel, 1995: siehe Abschnitt 5
- klassischer Single Source Shortest Path (SSSP): Dijkstra-Algorithmus $\forall\, v \in V$
- einfache Parallelisierung des Dijkstra-Algorithmus über Quell-Knoten
- Breitensuche-APSP (BFS-APSP): parallelisiert in Nakao et al., 2019, 2020

Klassische APSP-Ansätze haben eine Laufzeit von $O(n^3)$. BFS-APSP nutzen den meist niedrigen Knotengrad d und erreichen eine Laufzeit von $O(n^2 d)$. Durch parallelisierte GPU-Implementationen werden laut Graph-Golf-Erfahrungsberichten (z. B. dem Vortrag von Terao, 2019 und der Publikation von Nakao et al., 2020) signifikant kürzere Laufzeiten erzielt.

3.2.2. Lösungsheuristiken

Zufällige reguläre Graphen: Ein elementarer Ansatz ist die Erzeugung eines zufälligen regulären Graphen mit den ODP-Parametern. Mit dieser Methode werden von den

Veranstaltern des Graph-Golf-Wettbewerbs initiale Referenzlösungen bereitgestellt (vgl. Graph Golf Problem statement, 2021).

Singla et al., 2012 nutzen diesen Ansatz zur Rack-basierten Vernetzung, sodass innerhalb eines Server-Racks einfach und direkt zwischen Switch und Server verkabelt wird, während die Switches untereinander durch einen regulärer Zufallsgraphen verbunden werden. Die Racks bzw. deren Switches sind hierbei Knoten des ODP.

Da es für ungerades $n \cdot d$ keine d-regulären Graphen gibt, ist diese Heuristik nicht anwendbar. Eine Erweiterung dieses Ansatzes wird in Abschnitt 5.1 betrachtet.

Zur Einordnung der Lösungsgüte von anderen Heuristiken werden oft d-reguläre Zufallsgraphen als Referenz verwendet. Als weitere Referenzen dienen klassische Strukturen der Vernetzung und Verkabelung großer Rechenzentren.

Erweiterung von DDP-Lösungen: Optimallösungen des DDP werden von Hoffman und Singleton, 1960 als Moore-Graphen bezeichnet. Für einige DDP-Instanzen sind optimale Lösungen bekannt und in Sammlungen erfasst, zusammen mit besten bekannten Lösungen für andere Instanzen (vgl. Combinatorics Wiki, 2019).

Für jede ODP-Instanz gibt es einen nächstkleineren Moore-Graphen oder eine nächstkleinere beste bekannte DDP-Lösung. Ausgehend von diesen Graphen fügen Koibuchi et al., 2016 Knoten hinzu und entfernen und ergänzen anschließend zufällig Kanten, bis eine zulässige ODP-Lösung entsteht. Hierbei wird die Güte der DDP-Lösung genutzt, um in der Nähe gute ODP-Lösungen zu erzielen.

Analog nutzen Besta und Hoefler, 2014 diesen Ansatz für eine komplexere Konstruktion ausgehend von guten DDP-Lösungen von McKay et al., 1998, zu denen unter Berücksichtigung weiterer Kriterien (Latenz, Bandbreite, Ausfallsicherheit, Kosten und Stromverbrauch) Knoten und Kanten ergänzt werden.

Symmetrische Ergänzung üblicher Vernetzungsstrukturen: In Rechenzentren werden zur klassischen Vernetzung auch Torus-Graphen verwendet, sodass Singla et al., 2012 diese als Vergleichsreferenz nutzen. Daran angelehnt konstruieren Fujimoto und Kobayashi, 2018 vertex-transitive Graphen aus einem 2D-Torus als Basisgraphen mittels gleichmäßigem Hinzufügen von Kanten. Durch diese Symmetriekonstruktion muss die Berechnung der kürzesten Wege (APSP) zur Bestimmung von Durchmesser und ASPL nur für einen Quellknoten (SSSP) erfolgen.

Symmetrie durch Cayley-Graphen: Um durch Symmetrie den SSSP anstelle des APSP als dominierenden Laufzeitaspekt zu nutzen, werden von Kitasuka et al., 2018 vertex-transitive Cayley-Graphen konstruiert. Durch geeignete Parameterwahl (s, t, r) werden Graphen mit $n = s \cdot t$ Knoten erzeugt, deren Kanten im semidirekten Produkt $\mathbb{Z}_s \rtimes_r \mathbb{Z}_t$ von endlichen zyklischen Gruppen über Voltage-Mengen ausgewählt werden, die sich zu den jeweiligen Parametern ergeben. Die Wahl der Parameter ist in der Konstruk-

tion durch die Anforderungen $\gcd(\phi(s), t) > 1$ und $r^t \equiv 1 \mod s$ eingeschränkt[1], sodass Cayley-Graphen erzeugt werden, die nicht die gesuchte Knotenanzahl haben. Hierfür wurden nachgelagert Zufallsheuristiken zum Einfügen, Entfernen oder Auftrennen von Knoten verwendet.

In einem Vortrag zur Konferenz CANDAR'17 (vgl. Graph Golf Events, 2021) beschreibt Kawamata, 2017 ein ähnliches Verfahren anhand von Voltage-Graphen.

Verkettung kleiner Graphen: Ausgehend vom erwarteten Durchmesser $k = 3$ und aus der Beobachtung kleiner Probleminstanzen konstruieren Kitasuka und Iida, 2016 große Graphen aus $(2k-1)$-Kreisen, also Kreisen der Länge 5. Konstruktionsgrundlage ist der Petersen-Graph als Lösung der ODP-Instanz $(10, 3)$ und historisch umfangreich analysierte DDP-Instanz. Die Konstruktion verkettet in der Reihenfolge benachbarte Petersen-Graphen mit teilweise kreuzenden Kanten. Zur Berücksichtigung aller Knotenanzahlen wird $(n \mod 10)$-mal anstelle eines Petersen-Graphen ein spezieller 11-Knoten-Graph verwendet. Auf Knoten mit Grad kleiner 5 wird anschließend per Greedy-Algorithmus die Anzahl von 5-Kreisen im Graphen erhöht, indem Kanten zwischen Knoten mit Abstand 4 hinzugefügt werden.

Mizuno und Ishida, 2016 präsentieren zwei Vorgehensweisen zur Konstruktion von Graphen mit Durchmesser $k = 2$: Der erste Ansatz erweitert eine Konstruktion von Brown, 1966 für weitere Parameter-Paare (n, d). Im zweiten Ansatz wird das Stern-Produkt für mehr als zwei Graphen verallgemeinert und zur Verkettung eines speziellen Graphen mit $n = 8$ Knoten und maximalem Knotengrad $d = 3$ mit vollständigen Graphen verschiedener Knotenanzahlen verwendet.

Simulated Annealing: Das Simulated-Annealing-Verfahren (SA) wurde als Framework zur Lösung des AODP von Wettbewerbsteilnehmern mit diversen Varianten für Start- und Verbesserungsverfahren angewendet.

Shimizu und Mori, 2016 betrachten nur Graphen mit Durchmesser 3 mit stochastischer lokaler Suche („stochastic local search“ (SLS)) und nutzen als Bewertungsfunktion statt der ASPL die Anzahl der Dreiecke und Vierecke im Graphen. Ausgehend von Zufallsgraphen werden per SLS Nachbarschaften des Graphen für das SA generiert und nach Anzahl der Dreiecke und Vierecke evaluiert. Mit dieser SA-Implementation wurden in 60 Tagen Rechenzeit die besten Lösungen für die Instanzen $(10.000, 60)$ und $(10.000, 64)$ für den Graph-Golf-Wettbewerb 2016 erzielt.

Ausgehend von der Kombination von Petersen-Graphen definieren Kitasuka und Iida, 2016 eine Edge-Importance-Funktion zur Bewertung der Auswirkung einer Kante auf den Gesamtgraphen bzw. dessen ASPL und Durchmesser. Anhand dieser Funktion werden Kanten sortiert und per 2-Opt-SLS getauscht. Per Multiple-2-Opt erfolgt nur alle 50 Kantentausche eine Berechnung von ASPL und Durchmesser, wobei im Update-

[1] Eulersche Phi-Funktion: $\phi(s)$ = Anzahl teilerfremder natürlicher Zahlen $\leq s$ (genannt: Totient)

Verfahren für von Veränderungen unberührte Knotenpaare die Werte der Distanz-Matrix aus der früheren Berechnung ohne Neuberechnung übernommen werden.

Durch Einsatz eines Kürzeste-Wege-Kernels wenden Hasegawa und Handa, 2018 die stochastische Methode „Estimation of Distribution Algorithms with Graph Kernels“ (EDA-GK) aus dem Machine Learning auf das AODP an. Für die per 2-Opt-SLS erzeugten Graphen werden Wahrscheinlichkeiten für die Lösungsgüte geschätzt. Eine aufwändige exakte Berechnung des APSP (vgl. Abschnitt 3.2.1) erfolgt deutlich seltener, wodurch die Laufzeit zur Erzeugung von Lösungskandidaten reduziert wird.

Kombination von Methoden: Zur AODP-Instanz (n, d) verwenden Nakao et al., 2019 g Kopien eines zufälligen regulären Graphen mit $\frac{n}{g}$ Knoten und Knotengrad d, von denen jeweils die gleiche Kante entfernt wird. Unter Einhaltung des maximalen Knotengrads wird für jeden Teilgraph eine Kante zwischen dem ersten Knoten der entfernten Kante und dem zweiten Knoten der entfernten Kante eines anderen Teilgraphen hinzugefügt. Durch diese konstruierte Symmetrie muss der APSP nur für einen Teilgraphen berechnet werden, was beim BFS-APSP eine Laufzeit von $O(n^2 d/g)$ ergibt. Beim anschließend verwendeten Simulated Annealing werden Veränderungen mittels 2-Opt für alle Teilgraphen identisch durchgeführt, sodass die Symmetrie erhalten bleibt.

4. Komplexität

Da in der berücksichtigten Literatur keine ausformulierten Komplexitätsaussagen zum ODP identifiziert wurden, werden nachfolgend die grundlegenden NP-Sachverhalte zum ODP und AODP betrachtet. Hasegawa und Handa, 2018 formulieren in der Einleitung ihres Artikel, dass das ODP bekannt NP-schwer ist. Zu dieser Aussage ist kein Beweis und keine Referenz auf einen Beweis angegeben.

Aufgrund der Ähnlichkeit zwischen ODP und DDP ist eine Äquivalenz naheliegend. Formal ausgeführt wurde diese in der gesichteten Literatur jedoch nicht gefunden. Da sich im Zuge der Literaturrecherche Aussagen, die das DDP als NP-schwer bezeichnen, als Zitatfehler herausstellten (siehe Abschnitt 3.1.2), bleibt auch mit der nachfolgend gezeigten Äquivalenz offen, ob das ODP NP-schwer ist.

Satz 1

Das ODDEP liegt in NP.

Beweis

- Aufgrund der graphentheoretischen Problemformulierung in Abschnitt 2.1 ist die Menge der möglichen Lösungen eine Teilmenge der Menge aller möglichen Kantenmengen, also der Potenzmenge aller Knotenpaare $\mathcal{P}(V \times V)$.

  Mit gegebener, endlicher Knotenanzahl $|V| = n \in \mathbb{N}$ ist auch $\mathcal{P}(V \times V)$ endlich. Die Menge der möglicher Lösungen ist also beschränkt mit der Mächtigkeit: $|\mathcal{P}(V \times V)| = 2^{|V \times V|} = 2^{n^2}$. Damit ist auch die Laufzeit zum Ausprobieren aller möglichen Kantenmengen mit $O\left(2^{n^2}\right)$ beschränkt und folglich endlich.

- Der APSP kann für jeden Lösungskandidaten in polynomialer Laufzeit ermittelt werden, z. B. mittels des Floyd-Warshall-Algorithmus in $\mathcal{O}(n^3)$.

  Der Knotengrad für jeden Knoten kann in $\mathcal{O}(n)$ ermittelt werden, also das Maximum über alle n Knoten in $\mathcal{O}(n^2)$, z. B. durch Iteration über eine Knotenliste sowie eine Nachbarschaftsliste an jedem Knoten oder über eine Adjazenzmatrix mit Dimension $n \times n$.

  Die Ermittlung der Knotenanzahl ist in $\mathcal{O}(n)$ möglich, z. B. durch über die Spaltenanzahl der Adjazenzmatrix oder durch Iteration über eine Knotenliste. ∎

Satz 2 (Äquivalenz zur NP-Vollständigkeit)

Folgende Aussagen sind äquivalent:

1. *Das ODP ist NP-schwer.*
2. *Das DDP ist NP-schwer.*
3. *Das ODiP ist NP-schwer.*

Beweis

ODP, DDP und ODiP haben mit dem ODDEP ein gemeinsames zugehöriges Entscheidungsproblem. Daraus folgt: Ist das ODDEP NP-vollständig, sind ODP, DDP und ODiP NP-schwer.

Als zugehöriges Entscheidungsproblem ist das ODDEP per Definition genau dann NP-vollständig, wenn ODP, DDP oder ODiP NP-schwer sind. ■

Anhand des nachfolgend definierten Entscheidungsproblems lassen sich die Komplexitätsaspekte des ODP auf das AODP übertragen:

Definition 6 (ASPL Order Degree Diameter Entscheidungs-P. (AODDEP))

Gegeben seien $n \in \mathbb{N}$, $d \in \mathbb{N}$, $k \in \mathbb{N}$ und $l \in \mathbb{Q}$. Gesucht ist die Antwort auf folgende Frage:

Existiert ein ungerichteter, einfacher, zusammenhängender Graph $G = (V, E)$ mit Knotenanzahl $n = |V|$ und Kantenmenge $E \subseteq V \times V$, sodass der maximale Knotengrad höchstens d, der Durchmesser höchstens k und die ASPL höchstens l beträgt? □

Satz 3

Das AODDEP liegt in NP.

Beweis

Die Abschätzungen für das ODDEP aus Satz 1 gelten auch für das AODDEP. Die Berechnung der ASPL hat durch Bildung des Mittelwerts aller Ergebnisse des APSP die gleiche Laufzeitkomplexität wie die Berechnung des Durchmessers als deren Maximum.■

Satz 4 (Komplexitätszusammenhang von AODP und ODP)

1. *Das AODDEP ist NP-vollständig, wenn das ODDEP NP-vollständig ist.*
2. *Das AODP ist NP-schwer, wenn das ODP NP-schwer ist.*

Beweis

1. Jede Lösung für das AODDEP ist auch eine Lösung für das ODDEP.
2. Das AODP ist NP-schwer genau dann, wenn das AODDEP NP-vollständig ist. Nach 1. ist das AODDEP NP-vollständig, wenn das ODDEP NP-vollständig ist. Das ODDEP ist genau dann NP-vollständig, wenn das ODP NP-schwer ist. ■

Auch unter Berücksichtigung weiterer naheliegender Literatur wurde kein zielführender Beweisansatz für die NP-Vollständigkeit des ODDEP gefunden.

Bei der Suche wurden die klassischen NP-vollständigen Probleme von Garey und Johnson, 1979, S. 190 ff. betrachtet, sowie aufgrund von Ähnlichkeiten in der Problemformulierung Veröffentlichungen zum Edge Deletion Problem, Minimum Diameter Edge

Addition Problem, Maximum Diameter Edge Deletion Problem und Minimum Diameter Problem.

Das ODP lässt sich in der Methodik von Garey und Johnson, 1979, S. 81, Abb. 4.2 als offenes Problem einordnen, abgegrenzt einerseits durch die Äquivalenz zum DDP mit dem MaxDDBS aus Abschnitt 3.1.2 als NP-schweres Oberproblem und andererseits durch Subprobleme, die bei Relaxierung von Nebenbedingungen entstehen und in polynomialer Laufzeit lösbar sind.

Für verschiedene bekannte NP-schwere Probleme ist aufgrund struktureller Aspekte bisher keine polynomiale Transformation gelungen. Häufigster Aspekt sind Subgraph-Formulierungen, z. B. von Garey und Johnson, 1979 notiert als „Does G contain“ für einen gegebenen Host-Graphen G. Dem entgegen erfolgt die Kantenauswahl des ODP ausgehend vom vollständigen Graphen als Host-Graphen. Durch Hinzufügen einer Subgraph-Nebenbedingung zum ODP entsteht ein NP-schweres Problem, da bereits das „Degree-Bounded Connected Subgraph“-Problem als ein Spezialfall laut Garey und Johnson, 1979, S. 196 (GT26) NP-vollständig ist.

Bei Überlegungen zur Transformation auf Probleme mit Budget-Nebenbedingung wurde durch das das einheitliche Kantengewicht für den einfachen, zusammenhängenden Graph im ODP kein zielführender Ansatz gefunden.

Entfernt man bei der Suche nach einem NP-vollständigen Spezialfall des ODDEP einen der Parameter, sind die resultierenden Probleme polynomial lösbar. Wie in Abschnitt 6.1 gezeigt, lassen sich auch einige Spezialfälle des ODP in polynomialer Zeit optimal lösen. Für den allgemeinen Fall konnte daraus keine Aussage abgeleitet werden.

5. Modellierung

Ausgehend von der graphentheoretischen Problemformulierung in Abschnitt 2 ergibt sich ein exponentiell großer Suchraum möglicher Graphen für zulässigen ODP-Lösungen. Daher werden nachfolgend elementare Möglichkeiten betrachtet werden, diesen Suchraum nach einer Optimallösung aus minimalem Durchmesser und minimaler ASPL für den Löseprozess stärker einzuschränken.

Anschließend wird eine MIP-Implementation vorgestellt, die zusammen mit den diskutierten Reduzierungen in Abschnitt 6 für Modellrechnungen verwendet wird.

Als alternativer Ansatz zur vollständig integrierten Problemmodellierung wird die Dekomposition nach Benders, 1962 betrachtet, um den Löseprozess des im ODP enthaltenen APSP zu entkoppeln.

5.1. Allgemeine Reduzierung des Suchraums

Grundlage der Betrachtung zur Reduzierung der Modellgröße ist der Ansatz, bei einer bekannten ODP-Lösung nur noch Graphen zu betrachten, die einen geringeren Durchmesser haben. Analog sucht man für AODP-Lösungen nur Graphen, die keinen größeren Durchmesser sowie eine kleinere ASPL haben.

Abhängig vom Lösungsverfahren lassen sich bekannte primale und duale Schranken zur Beschleunigung des Löseprozesses oder zur Konstruktion von Graphengeneratoren für einen vorgegebenen Durchmesser (vgl. Abschnitt 3) nutzen.

Beschränkung des APSP: Mit einer bekannten oberen Schranke für den Durchmesser k^* lässt sich auch die Laufzeit des APSP beschränken, indem dieser als SSSP von jedem Knoten ausgehend nur bis zu einer Tiefe von k^* durchgeführt wird. Wird der Zielknoten nicht erreicht, ist die Weglänge größer als k^* und damit der Durchmesser des Graphen größer als die primalen Schranke, wodurch die Lösung verworfen wird, weil sie schlechter ist als die beste bisher bekannte Lösung. Daraus ergibt sich eine Beschränkung der Laufzeit des APSP auf $O(n^2 \cdot k^*)$ für alle Graphen mit Durchmesser $\leq k^*$.

Nutzung von unteren Schranken: Wie im Abschnitt 3.2 zur Lösungsgüte erläutert, findet sich in der Literatur nur die Bewertung von Lösungen durch ihren Abstand zur Moore-Schranke für den Durchmesser und der Cerf-Schranke für die ASPL. Deren Kombination über die Zielfunktion zur initialen dualen Schranke bietet ein häufig erreichtes Optiumalitäts- und damit signifikantes Abbruchkriterium für den MIP-Löseprozess.

Umgekehrt lassen sich aus einer MIP-Optimallösung mit Zielfunktionswert oberhalb der initialen dualen Schranke im Regelfall Rückschlüsse auf höhere untere Schranken für den Durchmesser oder die ASPL der betrachteten Instanz ziehen.

Nutzung von oberen Schranken: Übliche MIP-Löser verwenden einen kombinierten Löseprozess aus LP-Relaxierung und Methoden für die erforderliche Ganzzahligkeit. Um diesen Löseprozess zu beschleunigen, kann eine Startlösung übergeben werden, deren Zielfunktionswert als primale Schranke verwendet wird, um z. B. im Branch-and-Bound-Verfahren Verzweigungen mit schlechterer LP-Relaxierung zu verwerfen.

Darüber hinaus können obere Schranken für den Durchmesser und die ASPL explizit vorgegeben werden, die über die Zielfunktion als primalen Schranke verwendet werden. Werden diese Schranken aus einer bekannten Lösung abgeleitet, wird die Lösungssuche auf bessere, also inhaltlich interessante Lösungen eingeschränkt.

Weiterhin können obere Schranken heuristisch gesetzt werden, z. B. auf den Wert der zugehörigen unteren Schranke, wodurch nur noch in dieser Eigenschaft optimale Lösungen gesucht werden. Wird das Optimierungsmodell durch das Setzen der Schranken unzulässig, gibt es keine Lösung mit dem vorgegebenen Wert. Auf diese Weise lässt sich prüfen, ob die Moore-Schranke für den Durchmesser erreichbar ist oder die untere Schranke für den Durchmesser um 1 angehoben werden kann.

Um nicht für jede Instanz obere Schranken für den Durchmesser und die ASPL z. B. durch eine Heuristik berechnen zu müssen, lässt sich folgende Beobachtung nutzen:

Korollar 1 (Monotonie bezüglich des Knotengrades)

(n, d_1) und (n, d_2) seien Parameter von AODP-Instanzen und (k_1, l_1) eine zulässige Lösung und somit obere Schranke für die AODP-Instanz (n, d_1).

Für $d_2 > d_1$ ist (k_1, l_1) auch eine obere Schranke für die AODP-Instanz (n, d_2), da der Durchmesser und die ASPL monoton sinken, wenn der maximale Knotengrad steigt.

Beweis

Seien (k_1^*, l_1^*) und (k_2^*, l_2^*) die zugehörigen Optimallösungen. Für die zugehörigen Lösungsgraphen beträgt die maximale Kantenanzahl $\left\lfloor \frac{n \cdot d_1}{2} \right\rfloor \leq \left\lfloor \frac{n \cdot d_2}{2} \right\rfloor$. Mit Fujimoto und Kobayashi, 2018, S.2269, Proposition 2 sinkt die ASPL , wenn Kanten hinzugefügt werden.

Analog kann sich der Durchmesser durch Hinzufügen einer Kante nicht erhöhen. Da die Optimallösung (k_1^*, l_1^*) zur AODP-Instanz (n, d_1) auch eine zulässige Lösung für die AODP-Instanz (n, d_2) ist, folgt mit $k_1 \geq k_1^* \geq k_2^*$ und $l_1 \geq l_1^* \geq l_2^*$ die Behauptung. ∎

Sobald für eine ODP-Instanz (n, d) eine Lösung mit Durchmesser $k = 2$ bekannt ist, folgt mit dieser Abschätzung für alle ODP-Instanzen (n, d') mit $n - 1 > d' > d$, dass deren Optimallösungen ebenfalls den Durchmesser $k' = 2$ haben.[2] Bei der Enumeration der Instanzen bis 99 Knoten in Abschnitt 6 haben die Optimallösungen von mindestens 3553 der 4465 Instanzen den Durchmesser 2.

Hierbei gilt für den kleinsten Knotengrad $\bar{d}$ mit $\bar{k} = 2$ stets $n \gg \bar{d}$. Für solche dünnbesetzten Graphen (sparse graphs) existieren effiziente Algorithmen zur Berechnung des APSP und zur heuristischen Generierung von zulässigen Lösungen.

[2]Der Trivialfall $d \geq n - 1$ mit Durchmesser $k = 1$ wird in Abschnitt 6 erläutert.

5.2. Implementation als MIP

Zusätzlich zur Notation gemäß Abschnitt 2.3 seien zur Kantenauswahl z_{ij} Binärvariablen mit $z_{ij} = 1$ für $(i, j) \in E$, sonst $z_{ij} = 0$.

Dann gilt für den maximalen Knotengrad aus Zeile (8):

$$\forall\, i \in V : \sum_{j \in V} z_{ij} \leq d$$

Weiterhin seien SP_{st} ganzzahlige Variablen (Integervariablen), deren Wert der Länge des kürzesten s-t-Pfades entspricht.

5.2.1. Klassischer MCF-APSP

In Vorlesungen zur Algorithmischen Diskreten Mathematik (ADM) wird die MIP-Implementation des APSP üblicherweise als Multi Commodity Flow (MCF) realisiert. Hierfür seien x_{stij} binäre Hilfsvariablen, sodass $x_{stij} = 1$ genau dann, wenn der kürzeste s-t-Pfad über die Kante $(i, j) \in E$ führt. Durch Implementation des APSP in Zeile (7) als MCF ergibt sich das MIP-Modell:

$$\begin{aligned}
\text{Zielfunktion} \quad & \min\ 10 \cdot k + l \\
\text{Durchmesser} \quad & \forall\, s, t \in V, s \neq t : SP_{st} \leq k \\
\text{ASPL} \quad & \frac{1}{n \cdot (n-1)} \sum_{s \in V} \sum_{\substack{t \in V \\ s \neq t}} SP_{st} = l \\
\text{APSP (7)} \quad & \forall\, s, t \in V, s \neq t : SP_{st} = \sum_{i \in V} \sum_{\substack{j \in V \\ i \neq j}} x_{stij} \\
\text{Pfaderhaltung (7)} \quad & \forall\, s, t \in V, s \neq t : \forall\, i \in V \backslash \{s, t\} : \sum_{j \in V \backslash \{s,t,i\}} (x_{stij} - x_{stji}) = 0 \\
\text{Pfadquelle (7)} \quad & \forall\, s, t \in V, s \neq t : \sum_{i \in V \backslash \{s,t\}} (x_{stsi} - x_{stis}) = 1 \\
\text{Pfadziel (7)} \quad & \forall\, s, t \in V, s \neq t : \sum_{i \in V \backslash \{s,t\}} (x_{stit} - x_{stti}) = -1 \\
z\text{-}x\text{-Koppelung (7)} \quad & \forall\, s, t, i, j \in V, s \neq t, i \neq j : x_{stij} \leq z_{ij} \\
\text{Knotengrad} \quad & \forall\, i \in V : \sum_{\substack{j \in V \\ i \neq j}} z_{ij} \leq d \\
& \forall\, i, j \in V, i \neq j : z_{ij} \in \{0, 1\} \\
& \forall\, s, t \in V, s \neq t : SP_{st} \in \mathbb{N} \\
& \forall\, s, t, i, j \in V, s \neq t, i \neq j : x_{stij} \in \{0, 1\}
\end{aligned}$$

Aus dieser Modellierung entsteht ein Speicherbedarf von $O(n^4)$ durch die Variablen x_{stij}, was die praktische Optimierung auch auf leistungsstarken Systemen nur für kleine Graphen bzw. Knotenanzahlen ermöglicht.

5.2.2. Quadratischer Seidel-APSP

Der von Seidel, 1995 vorgestellte Algorithmus löst den APSP für ungerichtete, ungewichtete, zusammenhängende Graphen. Für die verwendete Adjazenzmatrix $A \in \{0,1\}^{n\times n}$ wird in Claim 1 als Kernelement für das Quadrat $Z = A \cdot A$ formuliert: Für $Z = (w_{st})$ ist $w_{st} = 1$ genau dann, wenn es einen Pfad der Länge 2 zwischen den Knoten s und t gibt. Hinsichtlich des ODP folgt daraus für den Durchmesser: $k = \min\{i \in \mathbb{N} \mid A^i = \mathbb{1}_{n,n}\}$.

Da für die Berechnung der ASPL die Länge der einzelnen Pfade gemittelt wird, müssen die Matrizen A^i zu jeder Distanz als Zwischenergebnisse der Ermittlung des Durchmessers ebenfalls gespeichert werden.

Als Hilfsvariablen seien $dist_{stj} = 1$ bei Existenz eines s-t-Pfades mit Länge $\leq j$, sonst 0. Offensichtlich sind $z_{st} = dist_{st1}$ für alle $s,t \in V$. Für $s,t \in V$ und eine Distanz j soll $dist_{st(j+1)} = 1$ sein genau dann, wenn $dist_{stj} = 1$ oder es ein $u \in V\backslash\{s,t\}$ gibt, sodass $dist_{suj} \cdot dist_{ut1} = 1$. Daraus ergibt sich folgender Ansatz zur quadratischen Modellierung eines s-t-Pfades:

$$SP_{st} = 1 + \sum_{j=1}^{n}(1 - dist_{stj}) \tag{9}$$

$$dist_{st(j+1)} \leq dist_{stj} + \sum_{\substack{u\in V\\ s\neq u\neq t}} dist_{suj} \cdot dist_{ut1} \tag{10}$$

$$\forall\, u \in V\backslash\{s,t\} : dist_{st(j+1)} \geq dist_{suj} \cdot dist_{ut1} \tag{11}$$

$$dist_{st(j+1)} \geq dist_{stj} \tag{12}$$

Anhand der Optimierungsrichtung ergibt sich folgende Beobachtung: Durch die Minimierung von Durchmesser und ASPL müssen in der Modellierung die SP_{st} nur nach unten beschränkt werden. Durch das negative Vorzeichen bei der Summierung (9) müssen die $dist_{stj}$ entsprechend nach oben beschränkt werden, hier durch (10).

Da die Ungleichungen (11) und (12) das MIP-Polyeder nicht in Zielfunktionsrichtung beschränken, werden diese im Folgenden nicht weiter berücksichtigt.

Mit (9) und (10) ergibt sich folgende quadratische AODP-Modellierung:

$$
\begin{aligned}
\text{Zielfunktion} \quad & \min\ 10 \cdot k + l \\
\text{Durchmesser} \quad & \forall\, s,t \in V : SP_{st} \leq k \\
\text{ASPL} \quad & \frac{1}{n \cdot (n-1)} \sum_{s \in V} \sum_{\substack{t \in V \\ s \neq t}} SP_{st} = l \\
SP\text{-Summierung} \quad & \forall\, s,t \in V : SP_{st} = 1 + \sum_{j=1}^{n} (1 - dist_{stj}) \\
dist\text{-Berechnung} \quad & \forall\, j \in \{1, \ldots, n-1\} : \forall\, s,t \in V : \\
& dist_{st(j+1)} \leq dist_{stj} + \sum_{\substack{u \in V \\ s \neq u \neq t}} dist_{suj} \cdot dist_{ut1} \\
\text{Knotengrad} \quad & \forall\, i \in V : \sum_{\substack{j \in V \\ i \neq j}} dist_{ij1} \leq d \\
& \forall\, s,t \in V, s \neq t : SP_{st} \in \mathbb{N} \\
& \forall\, s,t \in V, s \neq t : \forall\, j \in \{1, \ldots, n\} : dist_{stj} \in \{0,1\}
\end{aligned}
$$

Aus der Modellierung entsteht ein Speicherbedarf von $O(n^3)$, z. B. durch die Variablen $dist_{stj}$, also um den Faktor n kleiner als mit MCF-APSP.

Reduzierung der Modellgröße durch Beschränkung des Durchmessers: Für eine bekannte Lösung mit Durchmesser k^* gilt aufgrund dessen Eigenschaft als obere Schranke der SP_{st} für alle $j > k^*$ stets $dist_{stj} = 1$. Im Sinne der minimierenden Zielfunktion sind alle Lösungen mit Durchmesser größer als k^* uninteressant.

Unterlässt man die Unterscheidung, ob ein Graph einen Durchmesser von $k^* + 1$ oder größer hat, weil diese mögliche Lösung ohnehin als schlechter als die bisher bekannte verworfen wird, muss $dist_{stj}$ nur für $j \in \{1, \ldots, k^*\}$ berechnet werden. Entsprechend sinkt der Speicherbedarf auf $O(n^2 \cdot k^*)$.

5.2.3. Linearisierung des quadratischen Modells

Da der Simplex-Algorithmus und daran angelehnte Verfahren der Optimierung zur Löschung gemischt-ganzzahliger Probleme (MIP) für lineare Problemformulierungen ausgelegt sind, wird nachfolgend der quadratische Modellierungsteil linearisiert.

Zur Linearisierung sei die Binärvariable $y_{stuj} := dist_{suj} \cdot dist_{ut1}$. Folglich ist y_{stuj} genau dann 1, wenn $dist_{suj} = 1$ und $dist_{ut1} = 1$, sonst 0. Daraus folgt als lineare MIP-Formulierung:

$$y_{stuj} \leq dist_{suj} \tag{13}$$
$$y_{stuj} \leq dist_{ut1} \tag{14}$$
$$y_{stuj} \geq dist_{suj} + dist_{ut1} - 1 \tag{15}$$

Aufgrund der Koppelung an $dist_{stj}$ müssen auch die y_{stuj} im Sinne der Optimierungsrichtung nach oben beschränkt werden, sodass (15) im Weiteren nicht mehr berücksichtigt wird. Mit (13) und (14) ergibt sich die linearisierte *dist*-Berechnung:

$$\begin{aligned}
\textit{dist}\text{-Berechnung } & \forall\, s,t \in V, s \neq t : \forall\, j \in \{1,\ldots,n-1\} : \\
& dist_{st(j+1)} \leq dist_{stj} + \sum_{\substack{u \in V \\ s \neq u \neq t}} y_{stuj} \\
\text{Linearisierung 1 } & \forall\, s,t \in V, s \neq t : \forall\, j \in \{1,\ldots,n-1\} : \\
& \forall\, u \in V \backslash \{s,t\} : y_{stuj} \leq dist_{suj} \\
\text{Linearisierung 2 } & \forall\, s,t \in V, s \neq t : \forall\, j \in \{1,\ldots,n-1\} : \\
& \forall\, u \in V \backslash \{s,t\} : y_{stuj} \leq dist_{ut1} \\
\text{Linearisierungsvariable } & \forall\, s,t \in V, s \neq t : \forall\, j \in \{1,\ldots,n-1\} : \\
& \forall\, u \in V \backslash \{s,t\} : y_{stuj} \in \{0,1\}
\end{aligned}$$

Aus der Modellierung entsteht ein Speicherbedarf von $O(n^4)$ durch die Variablen y_{stuj}, sodass der Vorteil im Speicherbedarf gegenüber dem MCF-APSP zunächst verloren geht.

Reduzierung der Modellgröße durch Beschränkung des Durchmessers: Analog zum quadratischen Modell sinkt auch beim linearisierten Modell der Speicherbedarf, wenn man die Unterscheidung unterlässt, ob ein Graph einen Durchmesser von $k^* + 1$ oder größer hat. Neben $dist_{stj}$ müssen auch y_{stuj} nur für $j \in \{1,\ldots,k^*\}$ berechnet werden. Entsprechend sinkt der Speicherbedarf auf $O(n^3 \cdot k^*)$.

Wie in den Abschnitten 6 und 7 vorgestellt, lässt sich für eine beliebige Knotenanzahl n ein $k^* \ll n$ mit geringem Aufwand ermitteln, sodass effektiv der Speicherbedarf in der Praxis für das linearisierte Modell mit n^3 und für das quadratische Modell mit n^2 wächst.

5.3. Praktische Reduzierungen

Nutzung der Modellsymmetrie: Aufgrund der vorhandenen Symmetrie im ungerichteten Graphen werden nur Knotenpaare (a,b) mit $a < b$ modelliert, also z. B. Kanten- und Pfadvariablen zur Reduzierung der Anzahl der Variablen stets als $(\min(a,b), \max(a,b))$ betrachtet. Analog werden z. B. MIP-Nebenbedingungen nur für $a < b$ erzeugt.

Für das Modell mit MCF-APSP ist dies nur teilweise möglich, da die Flusserhaltung eine gerichtete Flussmodellierung erfordert, sodass die x_{stij} zwar nur für $s < t$, jedoch für alle $i \neq j$ notwendig sind.

Dekompositionsansatz: Da das ODP eine Komposition von Nebenbedingungen ist, liegt eine Dekomposition nach Benders, 1962 in die Kantenauswahl im MIP und eine kombinatorische APSP-Berechnung nahe. Insbesondere für dünn besetzte Graphen mit $n \ll k$ sind laufzeiteffiziente Implementationen mit geringem Speicherbedarf wünschenswert, z. B. mittels Dijkstra-Algorithmus.

Allerdings beruht die Zielfunktion ausschließlich auf dem APSP-Ergebnis durch Ermittlung von Durchmesser und ASPL, sodass die Richtung des Optimierungsprozesses für die zugehörige LP-Relaxierung verloren geht.

Wie im Abschnitt 4 erwähnt, wird das ODP bei Weglassen jeweils einer Klasse von Nebenbedingungen in polynomialer Laufzeit lösbar, sodass die Komplexität des Problems auf der Koppelung der Nebenbedingungen basiert.

Ohne eine effektive Koppelung des APSP an die LP-Relaxierung als Kernelement des MIP-Löseprozesses gehen dessen Vorteile und Funktionsweisen weitgehend verloren. Hierfür konnte bisher keine praktisch wirksame Vorgehensweise identifiziert werden.

5.4. Einfache Fixierungen

Fixierung auf die maximale Kantenanzahl: Laut Fujimoto und Kobayashi, 2018 sinkt die ASPL mit dem Hinzufügen von Kanten. Umgekehrt steigen ASPL und Durchmesser bei Hinzufügen von Kanten nicht, sodass es stets eine Optimallösung mit maximaler Kantenanzahl gibt. Zur Reduzierung des Lösungsraums auf Graphen mit maximaler Kantenanzahl sei daher: $|E| = \left\lfloor \frac{n \cdot d}{2} \right\rfloor$

Fixierungen am ersten Knoten: O. B. d. A. seien die Knoten mit 1 bis n bezeichnet. Da alle Kanten ungewichtet sind und der Knotengrad auch für den ersten Knoten maximal sein soll, seien o. B. d. A. die Kanten zwischen dem Knoten 1 und den Knoten 2 bis $d + 1$ als ausgewählt fixiert.

Davon ausgehend lässt sich eine Baumstruktur vermuten, wobei die Tiefe des Baumes sowie die Zulässigkeit der zugehörigen Fixierung an dieser Stelle offen bleibt. Der Ansatz wird in Abschnitt 7 aufgegriffen.

Fixierung der Knotengrade: Die Fixierung auf die maximale Kantenanzahl impliziert für $n \cdot d$ gerade, dass alle Knotengerade ausgeschöpft werden. Für $n \cdot d$ ungerade hat abweichend genau ein Knoten den Grad $d - 1$, wofür in Abschnitt 6 der letzte Knoten gewählt wird. Durch diese Festlegung können die Knotengrad-Nebenbedingungen in der MIP-Formulierung als Gleichungen modelliert werden:

$$\forall\, i \in \{1, \ldots, n-1\} : \sum_{j \in V} z_{ij} = d$$

$$\sum_{j \in V} z_{nj} = \begin{cases} d & \text{für } n \cdot d \text{ gerade} \\ d-1 & \text{für } n \cdot d \text{ ungerade} \end{cases}$$

Werden alle Knotengrade fixiert, ist die Fixierung der Kantenanzahl als Summierung der Knotengrade redundant.

6. Modellrechnungen

Da jede Lösung des AODP ebenfalls eine Lösung des ODP ist und Optimalität für das AODP auch Optimalität für das ODP impliziert, wird im Hinblick auf den Graph-Golf-Wettbewerb nachfolgend primär das AODP betrachtet.

Als Überblick werden anhand der praktischen Modellierungen alle Instanzen mit bis zu 49 Knoten für das Modell mit MCF-APSP sowie mit bis zu 99 Knoten für die Modelle mit Seidel-APSP betrachtet. Hierbei werden zuerst einfache Spezialfälle ausgeschlossen und anschließend das Vorgehen per MIP-Löser vorgestellt.

Neben der praktischen Umsetzung der Modellierung aus Abschnitt 5 wird die heuristische Erzeugung von Startlösungen für alle Instanzen betrachtet, da die üblichen zufälligen regulären Graphen nur für den Fall $n \cdot d$ gerade existieren.

Als zweiter Ausgangspunkt zur Auswahl von Instanzen bietet sich der Graph-Golf-Wettbewerb an, sodass die Instanzen des diesjährigen Wettbewerbs (2021) sowie früherer Wettbewerbsjahre (2015-2019) betrachtet werden.

6.1. Einfache Spezialfälle

Da bei der nachfolgenden Enumeration alle maximalen Knotengrade $d \in \mathbb{N}$ zu einer Knotenanzahl $n \in \mathbb{N}$ berücksichtigt werden sollen, werden nachfolgend einige elementare Fälle betrachtet:

In einem einfachen Graphen mit $n = 1$ Knoten gibt es keine Kanten. Folglich sind dessen Durchmesser und ASPL trivial 0.

Für einen einfachen, zusammenhängenden Graphen mit $n = 2$ Knoten gibt es für $d \geq 1$ nur den vollständigen Graphen K_2, dessen Durchmesser und ASPL trivial 1 sind.

Ein Graph mit maximalem Knotengrad $d \leq 1$ ist für $n > 2$ nicht zusammenhängend.

Bei einem maximalen Knotengrad $d = 2$ sind keine Verzweigungen des Graphen möglich. Für einen zusammenhängenden Lösungsgraphen gibt es dadurch zwei zulässige Lösungstypen zur Knotenanzahl n:

- Ein beliebiger Hamiltonpfad hat den Durchmesser $k = n - 1$.
- Ein beliebiger Hamiltonkreis hat den Durchmesser $k = \left\lfloor \frac{n}{2} \right\rfloor$.

Für $n > 2$ ist mit $\left\lfloor \frac{n}{2} \right\rfloor < n - 1$ ein beliebiger Hamiltonkreis optimale Lösung der Probleminstanz $(n, 2)$.

Bei maximalem Knotengrad $d \geq n - 1$ ist der vollständige Graph K_n mit Durchmesser $k = 1$ und ASPL $l = 1$ die optimale Lösung für jede Knotenanzahl n.

Für dichte Graphen mit maximalem Knotengrad $d \approx n$ ist die Berechnung des APSP aufwändig, weshalb nachfolgend exemplarisch auf Grundlage des vollständigen Graphen eine Optimallösung für $d = n - 2$ konstruiert wird. Eine ggf. induktive Erweiterung

dieses Ansatzes ist wünschenswert, aber an dieser Stelle nicht direkt absehbar.

Korollar 2 (Konstruktion für maximalen Knotengrad $d = n - 2$)

O. B. d. A. seien die Knoten mit 1 *bis* n *bezeichnet. Ausgehend vom vollständigen Graphen werden einige Pfadlängen von* 1 *auf* 2 *steigen. Offensichtlich suboptimal hinsichtliches des Durchmessers wäre die Verlängerung eines Pfades auf Länge* 3*.*

Durch Entfernen der Kanten $(2i-1, 2i)$ *für* $i \in \{1, \ldots, \frac{n}{2}\}$ *entsteht für gerade Knotenanzahl* n *ein Graph zur optimalen AODP-Lösung* $\left(2, \frac{n}{n-1}\right)$*.*

Für ungerade Knotenanzahl n *entsteht durch Entfernen der Kanten* $(2i-1, 2i)$ *für* $i \in \left\{1, \ldots, \frac{n-1}{2}\right\}$ *und der Kante* $(1, n)$ *ein Graph zur optimalen AODP-Lösung* $\left(2, \frac{n^2+1}{n^2-n}\right)$*.*

Beweis

Für die konstruierten Graphen ist die Kantenanzahl maximal und die Knotengrade sind zulässig. Für alle Knotenpaare, deren Kante zugunsten des zulässigen Knotengrads entfernt werden musste, erhöht sich die Pfadlänge auf 2.

Bei ungerader Knotenanzahl $n \geq 5$ hat das Knotenpaar $(1, n)$ ebenfalls die Pfadlänge 2, da beide Knoten mit dem Knoten 3 verbunden sind.

Damit hat der Graph einen Durchmesser $k = 2$ und für die ASPL gilt:

- n gerade: Es werden $\frac{n}{2}$ Kanten entfernt, also steigen n Pfadlängen von 1 auf 2. In der Summierung in Formel (1) gehen diese mit Vorfaktor $\frac{1}{n(n-1)}$ ein. Somit steigt die ASPL um $\frac{1}{n-1}$ und beträgt: $1 + \frac{1}{n-1} = \frac{n}{n-1}$.
- n ungerade: Es werden $\frac{n+1}{2}$ Kanten entfernt, also steigen $n + 1$ Pfadlängen von 1 auf 2. Somit steigt die ASPL um $\frac{n+1}{n\cdot(n-1)}$ und beträgt: $1 + \frac{n+1}{n\cdot(n-1)} = \frac{n^2+1}{n^2-n}$. ∎

Für einfache, zusammenhängende Graphen mit $n = 3$, $n = 4$ und $n = 5$ Knoten gibt es keine Probleminstanzen, die nicht einem der betrachteten Fälle entsprechen.

Daher werden nachfolgend nur Probleminstanzen mit $n \geq 6$ Knoten und maximalem Knotengrad $2 \leq d \leq n - 3$ betrachtet.

6.2. Startheuristik

Von den Veranstaltern des Graph-Golf-Wettbewerbs wurden Referenzlösungen bereitgestellt, die mittels eines Generators für reguläre Zufallsgraphen erzeugt werden. (vgl. Graph Golf Problem statement, 2021) Davon ausgehend wurde eine Verallgemeinerung für beliebige Parameter (n, d) erarbeitet, da Zufallsgraphen häufig gute Lösungen liefern.

Für $n \cdot d$ gerade ruft das von den Veranstaltern des Graph-Golf-Wettbewerbs bereitgestellte Python-Skript `create-random.py` die Funktion „random regular graph“ der Python-Bibliothek NetworkX auf. Diese implementiert einen Algorithmus von Steger

und Wormald, 1999 und erzeugt so einen Kandidatengraphen mit n Knoten und Knotengrad d. Anschließend wird überprüft, ob der Graph zusammenhängend ist, was für $d \geq 3$ meistens der Fall ist. Da der Methodenaufruf nicht deterministisch ist, reicht in den seltenen Ausnahmefällen ein erneuter Skriptaufruf mit den gleichen Parametern aus.

Weil für den zu ergänzenden Fall mit $n \cdot d$ ungerade kein regulärer Graph existiert, wird stattdessen ein zufälliger regulärer Graph für $(n-1, d-1)$ verwendet. Ein $(d-1)$-regulärer Graph mit $n-1$ Knoten existiert, da $(n-1) \cdot (d-1)$ gerade ist. Anschließend wird ein Knoten mit Kanten zu den ersten d Knoten hinzugefügt. Der Durchmesser des resultierenden Graphen ist maximal um 1 höher als der erzeugte Zufallsgraph.

Die alternative Erzeugung eines zufälligen regulärer Graphen für $(n, d-1)$ führt zu einer zulässigen Startlösung, jedoch mit geringerer Kantenanzahl, sodass eine höhere ASPL wahrscheinlich ist:

$$\frac{n(d-1)}{2} < \frac{(n-1)(d-1)+2d}{2} = \frac{n(d-1)+d+1}{2}$$

Als Beispiel erhält man für $(n, d) = (17, 5)$ mit dem beschrieben Vorgehen die Lösung $(k; l) = (3; 1, 949)$, mit der Alternative mittels $(n, d) = (17, 4)$ nur $(k; l) = (3; 2, 007)$.

6.2.1. Kreis-Stern-Heuristik für $d = 3$

Für ungerade Knotenanzahl n und maximalen Knotengrad $d = 3$ entsteht mit dem beschriebenen Vorgehen bestenfalls ein Kreis mit Durchmesser $\frac{n-1}{2}$, an den ein Knoten angefügt wird, sodass ein Graph mit Durchmesser $k = \frac{n+1}{2}$ entsteht. Häufig wird kein zusammenhängender regulärer Graph für $d = 2$ gefunden, wodurch diese Erzeugung einer Startlösung nicht zuverlässig funktioniert.

Nachfolgende Heuristik kombiniert den Kreisgraphen der Länge $\frac{n+1}{2}$ mit einem Sterngraphen der Größe $\frac{n-1}{2}$, wobei verschiedene Anzahlen von übersprungenen Knoten für das Sternmuster ausprobiert werden. Optisch erinnert die Konstruktion an den Petersen-Graphen (siehe Abbildung 2).

Ansatz ist die Verkettung zweier Kreise der Länge $\lceil \frac{n}{2} \rceil = \frac{n+1}{2}$ und $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$, wobei zur Verkürzung des Durchmesser die Knotenreihenfolge eines Kreises zum Stern permutiert wird, um weiter entfernte Knoten des ersten Kreises möglichst kurz über den zweiten Kreis zu verbinden und umgekehrt.

Die Sternkonstruktion erfolgt durch Überspringen von i Knoten mit $i \in \{1, \ldots, \lfloor \frac{n-1}{4} \rfloor\}$, sodass für $n \geq 9$ verschiedene mögliche Sterne erzeugt werden und der beste Kandidat hinsichtlich Durchmesser und ASPL verwendet wird.

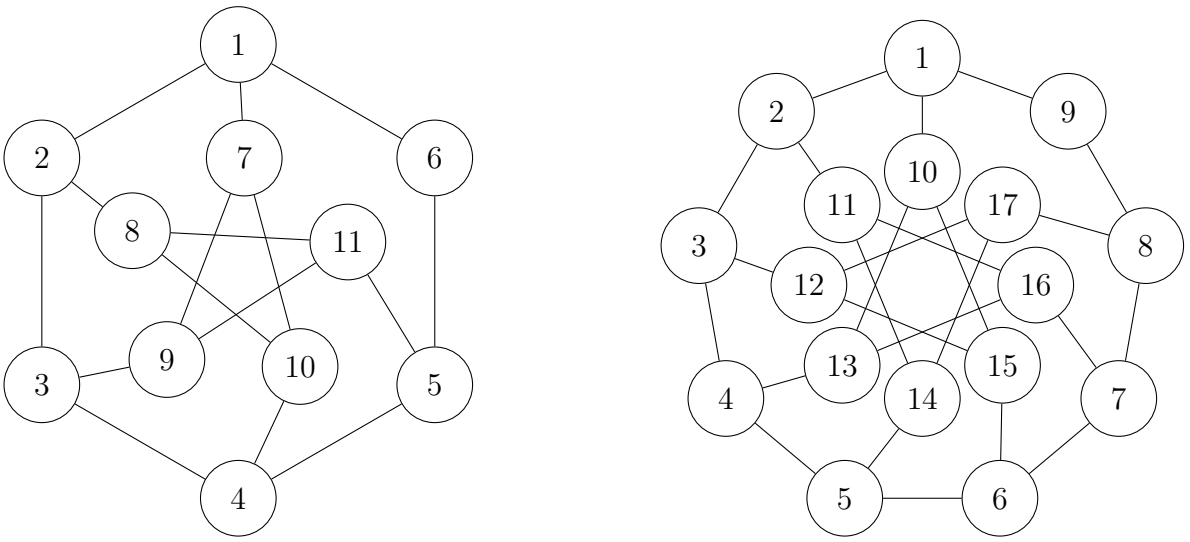

Abbildung 2: Ergebnisse der Kreis-Stern-Heuristik für $n = 11$ und $n = 17$

Der Ansatz ist methodisch mit der Verkettung kleiner Graphen aus Abschnitt 3.2.2 verwandt. Die Notation für die verketteten Graphen lässt sich am Schläfli-Symbol orientieren, sodass das Beispiel für $n = 11$ durch Kombination von $\{6\}$ mit $\{5/2\}$ mit Durchmesser $k = 3$ und ASPL $l = 1,8\overline{54}$ entstanden ist.

Eine Verallgemeinerung auf höhere Knotengrade wurde nicht vertieft, da Ansätze mit mehreren Sternen deutlich schlechteren Lösungen als die obige Vorgehensweise ergaben. Auch die Ausweitung dieses Ansatzes auf gerade Knotenanzahlen lieferte schlechtere Lösungen als die Generierung zufälliger regulärer Graphen.

Die Heuristik erzeugt für den betrachteten Spezialfall zuverlässig Lösungen mit Durchmesser und ASPL vergleichbar zu denen regulärer Zufallsgraphen der benachbarten geraden Knotenanzahlen. Eine Python-Implementation ist im Code-Listing 5 festgehalten.

6.3. Test-Setup

Die Modelle aus Abschnitt 5 wurden mit der Modellierungssprache ZIMPL von Koch, 2004 implementiert und sind als Code-Listings 1, 2 und 3 mit optionaler Erweiterung für die Fixierungen von Kanten als Code-Listing 4 im Anhang B beigelegt.

Für die Modelle mit Seidel-APSP wurden für die Implementierung folgende Aspekte ergänzt:

- primale und duale Schranken für Durchmesser und ASPL
- Beschränkung der maximalen Pfadlänge im Seidel-APSP
- Fixierung der Knotengrade (vgl. Abschnitt 5.4)

Für alle Modelle erfolgte die Nutzung von einfacher Symmetrie (vgl. Abschnitt 5.3) zur

Reduzierung der Modellgröße.

Die Modellrechnungen wurden mit der Software Gurobi in der Version 9.1.2 auf einem Testsystem mit folgenden Leistungsdaten durchgeführt:

- Prozessor: AMD Ryzen 7 PRO 3700 mit 8 Kernen je 3,6 GHz (16 Threads)
- Arbeitsspeicher: 64 GB
- Betriebssystem: Ubuntu 20.04 LTS

Für jede Instanz (n, d) werden die unteren Schranken für den Durchmesser und die ASPL berechnet. Weiterhin wird eine heuristische Lösung gemäß Abschnitt 6.2 oder Abschnitt 7 erzeugt und deren Durchmesser und ASPL als obere Schranken berechnet und verwendet. Mit diesen Schranken wird MIP-Modell in ZIMPL generiert und nach Umwandlung in eine LP-Datei in Gurobi eingelesen.

Da eine Lösung erst bei exaktem Erreichen der dualen Schranke optimal ist, wird der Parameter „MIPGap=0.0“ (statt „1e-4“) gesetzt. Anstelle der gleichzeitigen Nutzung mehrere Prozessorkerne für eine Probleminstanz werden mehrere Probleminstanzen gleichzeitig mit jeweils einem Thread berechnet, wofür der Parameter „Threads=1“ (statt Anzahl der Prozessor-Threads) gesetzt wird. Da jenseits der Modellierung keine Erkenntnisse zum Anheben der duale Schranke bekannt sind, ist dies auch im Löseprozess wenig erfolgversprechend, sodass mit dem Parameter „ImproveStartTime=0.0“ der Schwerpunkt der Optimierung auf die Lösungsverbesserung und nicht auf die Verbesserung der dualen Schranke gesetzt wird.

Technische Limitierungen

Da das Modell mit MCF-APSP durch die $O(n^4)$ Variablen deutlich mehr Arbeitsspeicher benötigt, wurden für dieses Modell nur die 990 Instanzen bis 49 Knoten betrachtet.

Aufgrund der Modellgröße reichte das verwendete Zeitlimit von 1 Stunde für das Lösen der Wurzel-LP-Relaxierung bereits für die AODP-Instanz $(27, 3)$ nicht aus, was bei den Seidel-APSP erst ab der AODP-Instanz $(48, 5)$ der Fall war. Insofern erscheint das Modell auch wenig zielführend zum Einsatz auf einem leistungsstärkeren Testsystem.

Für die Seidel-APSP-Modelle konnte der Löseprozess der Wettbewerbsinstanz $(432, 12)$ aufgrund der unzureichenden Menge an Arbeitsspeicher nicht erfolgreich gestartet werden. Testrechnungen auf einem leistungsstärkeren Testsystem sind für einen späteren Zeitpunkt geplant.

6.4. Instanzen des Graph-Golf-Wettbewerbs

Für die Wertung im Graph-Golf-Wettbewerb werden seit 2015 jedes Jahr Instanzen (n, d) aus Knotenanzahl und maximalem Knotengrad ausgewählt (siehe Graph Golf

Rules, 2021). Für 2021 sind das in der Kategorie „General Graph“ die Instanzen $(40, 5)$, $(432, 12)$, $(512, 18)$, $(1024, 5)$, $(3602, 24)$, $(65536, 64)$, $(100000, 128)$ und $(158976, 10)$.

Aufgrund der Überlappung zur Enumeration der Instanzen bis 99 Knoten ist die Instanz $(40, 5)$ das Schwerpunktbeispiel für folgende Betrachtungen.

Auffällig ist, dass in 2021 für sämtliche Instanzen $n \cdot d$ gerade ist, was jedoch nicht auf alle Wettbewerbsjahre zutrifft.

6.4.1. Schranken der Wettbewerbsinstanzen 2021

Durch die Veranstalter wurden für die Wettbewerbsinstanzen zufällige Startlösungen anhand des Skripts aus Abschnitt 6.2 mit den zugehörigen Schranken bereitgestellt:

n	d	k_{LB}	k_{RR}	Δk	l_{LB}	l_{RR}	Δl
40	5	3	4	1	2,23076	2,37179	0,14103
432	12	3	4	1	2,63805	2,70395	0,06590
512	18	3	4	1	2,33072	2,48283	0,15211
1024	5	5	7	2	4,45259	4,71975	0,26716
3602	24	3	4	1	2,83338	2,86897	0,03559
65536	64	3	4	1	2,93652	2,95774	0,02122
158976	10	6	7	1	5,47772	5,58714	0,10942
100000	128	3	4	1	2,83488	2,84742	0,01254

Tabelle 1: Schranken für die Wettbewerbsinstanzen 2021

Zur Knotenanzahl n und zum maximalem Knotengrad d gehören jeweils:

- untere Schranke k_{LB} für den Durchmesser k (Moore-Schranke)
- obere Schranke k_{RR} für den Durchmesser k (Startheuristik)
- Gap für den Durchmesser: $\Delta k := k_{RR} - k_{LB}$
- untere Schranke l_{LB} für die ASPL l (Cerf-Schranke)
- obere Schranke l_{RR} für die ASPL l (Startheuristik)
- Gap für die ASPL : $\Delta l := l_{RR} - l_{LB}$

Die erzeugten Zufallsgraphen haben mit den Werten 1 und 2 für Δk bereits gute Durchmesser. Ob eine Verbesserung des Durchmessers möglich ist, bleibt ohne Wissen über die Erreichbarkeit der Moore-Schranke offen.

Die kleinen Werte für Δl sind ein Anzeichen für mögliche numerische Probleme. Der Faktor $\frac{1}{n \cdot (n-1)}$ in der Berechnung der ASPL (siehe Formel (1) in Abschnitt 2.2) unterschreitet für große Knotenanzahlen übliche Genauigkeiten bzw. numerische Toleranzen von MIP-Lösern, z. B. 10^{-6} für $n = 1024$ oder 10^{-9} für $n = 65536$. Da Testrechnungen

mit MIP-Löser nur für Instanzen mit $n \ll 1000$ durchgeführt wurden, wird dieser Aspekt nachfolgend nicht weiter betrachtet.

6.4.2. Frühere Wettbewerbsinstanzen

In 2015-2019 wurden verschiedene Schwerpunkte ausgewählt, wobei teilweise Strukturen und stets eine Vielfalt in Größe, Knotengrad und Teilbarkeit der Parameter erkennbar sind. 2020 ist der Wettbewerb ausgefallen.

Da die Kriterien zur Auswahl der Instanzen auf der Website nicht dokumentiert sind, werden aufgrund der Menge der Instanzen diese nachfolgend verkürzt gruppiert und primär in Reihenfolge der Knotenanzahl aufgelistet.

- 2015: $\{16, 64, 256, 4096, 10000\} \times \{3, 4, 16, 23, 60, 64\}$
- 2016: $\{36, 96, 384, 1024\} \times \{3\}$, $\{64, 256, 512, 1024\} \times \{8\}$, $\{300, 1800\} \times \{7\}$, $\{1024\} \times \{11, 32\}$, $(1560, 40)$, $(3250, 57)$, $\{10000, 100000\} \times \{7, 11, 20\}$
- 2017: $(32, 5)$, $(256, 18)$, $\{576, 1344\} \times \{30\}$, $(4896, 24)$, $(88128, 12)$, $\{9344, 98304\} \times \{10\}$, $\{100000\} \times \{32, 64\}$
- 2018: $(72, 4)$, $\{256\}\times\{5, 10\}$, $(2300, 10)$, $\{3019, 4855\}\times\{30\}$, $(12000, 7)$, $(20000, 11)$, $\{40000, 132000\} \times \{8\}$, $(77000, 6)$, $\{200000\} \times \{32, 64\}$, $(400000, 32)$
- 2019: $\{50, 512, 1024\} \times \{4\}$, $\{512, 9344, 65536\} \times \{6\}$, $(1726, 30)$, $(4855, 15)$, $(100000, 8)$, $\{1000000\} \times \{16, 32\}$

Die Ergebnisse sind für alle früheren Wettbewerbsjahre unter Graph Golf Rankings and solutions, 2021 verlinkt. Auffällig ist, dass nach 2015 nur noch wenige Instanzen mit Erreichen der unteren Schranke gelöst wurden. Im Kontext von Abschnitt 3.1.1 bleibt oft unklar, ob Graphen existieren, der die untere Schranke erreichen.

7. Heuristische Baumstruktur

Für das Schwerpunktbeispiel $(40, 5)$ erfolgten mehrere Testrechnungen. Als Startlösung wurde jeweils die beste Lösung der letzten Testrechnung in den Löseprozess übergeben.

Als erster Wettbewerbserfolg wurde für das MIP-Modell mit quadratischem Seidel-APSP mit Fixierung der Kantenanzahl nach 20 Stunden Laufzeit auf 8 Threads eine Lösung gefunden, die den optimalen Durchmesser $k_{LB} = 3$ statt $k_{RR} = 4$ und eine ASPL $l = 2,2\overline{3}$ statt $l_{RR} = 2,37$ hat. (Diese Testrechnungen erfolgten außerhalb des definierten Test-Setups auf einem Testsystem mit einem Intel Core i7-9700K Prozessor mit 8 Kernen je 3,6 GHz, 64 GB Arbeitsspeicher und Ubuntu 18.04 LTS.)

Nach Anwendung der Fixierungen aus Abschnitt 5.4 wurde die verbliebene ASPL-Gap von $0,00256$ nach weiteren 66 Stunden auf 8 Threads geschlossen, sodass eine bewiesen optimale Lösung gefunden und beim Graph-Golf-Wettbewerb eingereicht wurde.

Aus der MIP-Lösung wurde der zugehörige Graph als Kantenliste gespeichert, zur weiteren Analyse mit Graphviz visualisiert und anschließend manuell eingefärbt:

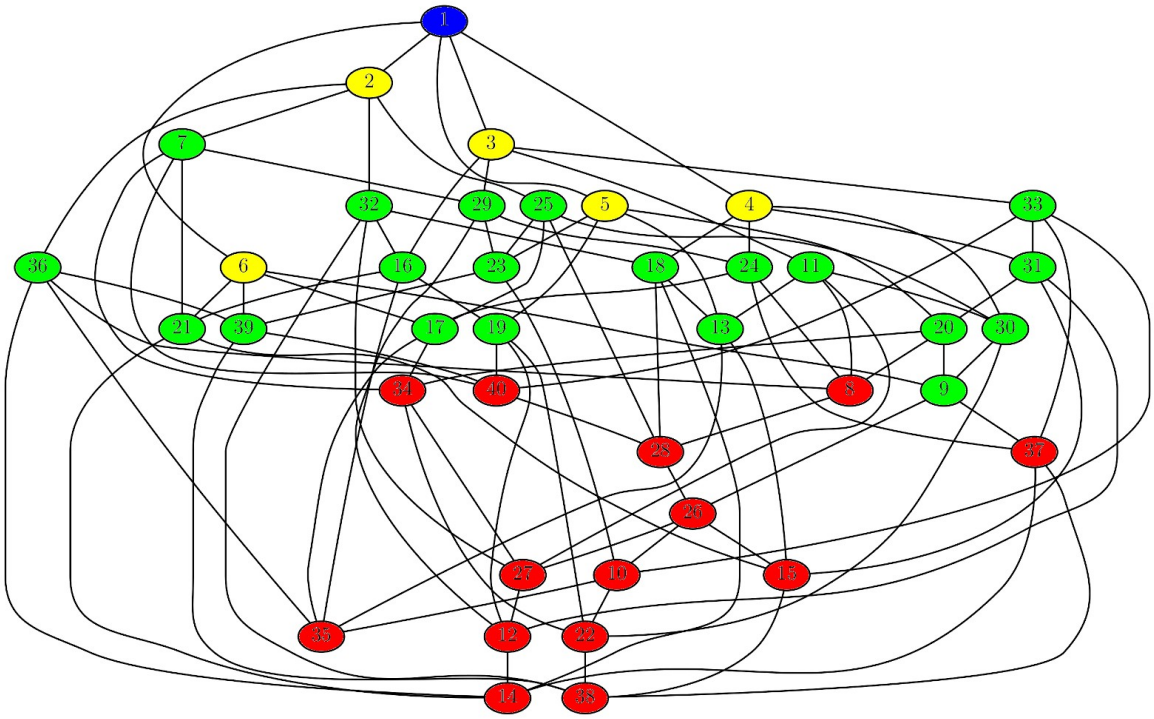

Abbildung 3: Optimallösung für $(40, 5)$

Die farbige Markierung der Abbildung 3 beginnt an der Wurzel (blauer Knoten). Die Nachbarn der Wurzel wurden gelb, sowie deren Nachbarn außer der Wurzel grün markiert. Alle dann noch unmarkierten Konten wurden rot markiert. Diese Farbmarkierung

visualisiert eine in der optimalen Lösung enthaltene Baumstruktur:

1. Die Wurzel (blau) und deren Nachbarn (gelb) sind aufgrund der Fixierungen aus Abschnitt 5.4 offensichtlich.
2. Die Wurzelnachbarn (gelb) sind untereinander nicht verbunden.
3. Die grünen Knoten sind mit genau einem Wurzelnachbarn (gelb) verbunden.

Zwischen roten und grünen Knoten ist auf den ersten Blick keine feste Struktur oder Regelung erkennbar, da Verbindungen sowohl innerhalb als auch zwischen den Färbungen erfolgen.

Die Punkte 2 und 3 sind insofern auffällig, als dass die Verbindungen zwischen den gelb und grün markierten Knoten keiner Fixierung unterlagen. Diese Beobachtung wird als Ansatz für eine Erweiterung der Fixierungen aus Abschnitt 5.4 genutzt.

7.1. Strukturannahme AODP-Baum

Kürzeste-Wege-Algorithmen wie der Dijkstra-Algorithmus und der BFS-APSP (vgl. Abschnitt 3.2.1) arbeiten für ungewichtete Graphen eine Baumstruktur in Breitensuche ab.

Die von Cerf et al., 1974 angegebene Formel für die untere Schranke für die ASPL gibt mit $d \cdot (d-1)^{i-1}$ eine maximale Anzahl von Knoten in der i-ten Ebene eines Baumes mit Grad d an, dessen Wurzel d Nachbarn und alle weiteren Baumknoten $d-1$ Nachbarn in der nächsten Ebene haben.

Da diese Herleitung keineswegs den Anforderungen an einen Zulässigkeitsbeweis für eine Fixierung genügt, wird nachfolgend eine heuristische Annahme getroffen, um einen Baum als Grundstruktur für eine Lösungssuche zu fixieren.

Für jede AODP-Instanz (n, d) gibt es einen nächstkleineren Baum mit n_B Knoten, dessen innere Knoten den Grad d haben. n_B lässt sich wie folgt über die Höhe des Baumes h_B ermitteln:

$$n_B(n,d) = \sum_{i=0}^{h_B(n,d)} (d-1)^i \quad \text{mit} \quad h_B(n,d) = \max\left\{ j \in \mathbb{N} \,\middle|\, \sum_{i=0}^{j} (d-1)^i \leq n \right\}$$

Im Beispiel $(40, 5)$ ist $n_B(40, 5) = 26$ mit $h_B(40, 5) = 2$. In den Farben von Abbildung 3 bilden die Farben blau, gelb und grün den einfachen AODP-Baum.

Für die ersten Knotengrade und Baumtiefen ergeben sich als n_B:

$d \setminus h_B$	1	2	3	4	5	6	7
3	4	10	22	46	94	190	382
4	5	17	53	161	485	1457	4373
5	6	26	106	426	1706	6826	27306
6	7	37	187	937	4687	23437	117187
7	8	50	302	1814	10886	65318	391910

Tabelle 2: Knotenanzahlen im einfachen AODP-Baum

Da zwischen den möglichen Werten für n_B große Abstände bestehen, werden die verbleibenden $n - n_B$ Knoten (im Beispiel rot) durch Anhängen an die Blätter des Baumes (im Beispiel grün) erweitert.

O. B. d. A. seien die Knoten des einfachen AODP-Baumes von 1 bis n_B ebenenweise aufsteigend nummeriert, also von der Wurzel 1 über deren Nachbarn 2 bis $d+1$ usw. bis zu den Blättern. Die verbleibende Knoten seien entsprechend mit $n_B + 1$ bis n nummeriert.

Die in Abbildung 3 deutlich erkennbaren Querverbindungen zwischen grünen und roten Knoten haben bestimmenden Einfluss auf den Durchmesser und die ASPL. Daher werden die hinzugefügten Knoten (rot) gleichmäßig verteilt an die bisherigen Blätter (grün) mit jeweils einer Kante angehängt, sodass am Ende zwei Blätter (grün) des einfachen AODP-Baumes maximal einen um 1 unterschiedlichen Knotengrad haben.

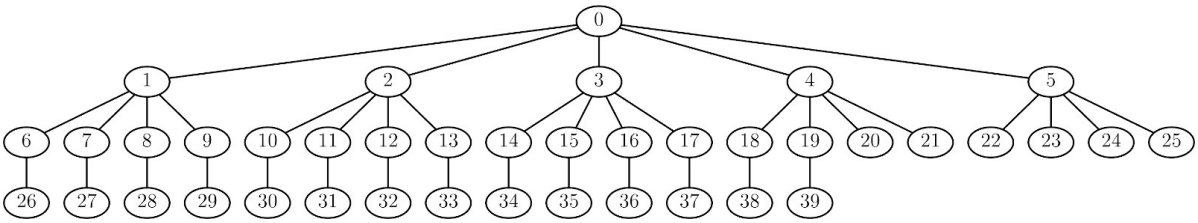

Abbildung 4: erweiterter AODP-Baum für $(40, 5)$

Der einfache Baum zur Instanz $(40, 5)$ hat $n_B(40, 5) = 26$ Knoten, davon 20 Blätter. Die verbleibenden $40 - 26 = 14$ Knoten können entsprechend an die Knoten 6 bis 19 direkt angehängt werden.

Für die Instanz $(40, 3)$ hat der einfache AODP-Baum $n_B(40, 3) = 22$ Knoten, davon 12 Blätter, sodass 6 der verbleibenden $40 - 21 = 18$ Knoten an Blätter gehängt werden, die schon einen verbleibenden Knoten haben.

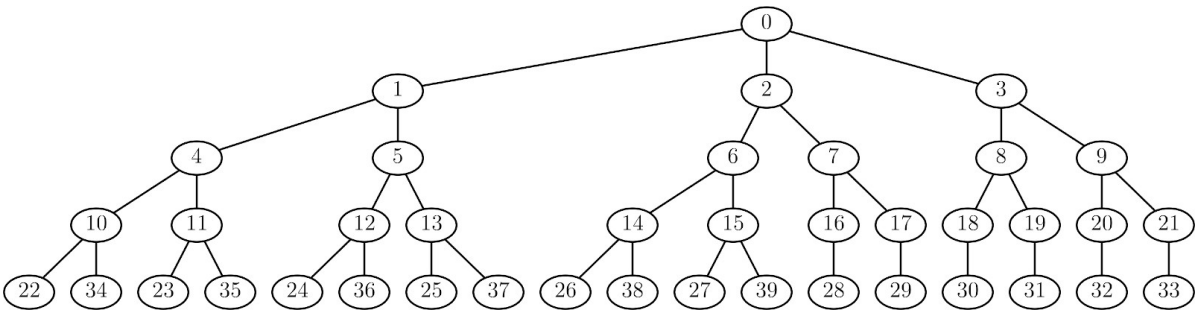

Abbildung 5: erweiterter AODP-Baum für $(40, 3)$

Da n per Definition von n_B kleiner ist als die Knotenanzahl des nächstgrößeren Baumes, existiert diese Konstruktion des erweiterten AODP-Baumes. Im Folgenden wird der einfache AODP-Baum nicht weiter betrachtet, sodass bei Erwähnung des AODP-Baumes stets der erweiterte AODP-Baum gemeint ist.

Der AODP-Baum liefert eine zulässige AODP-Lösung, da er zusammenhängend ist und die Knotengrad-Nebenbedingung des AODP erfüllt.

Die Konstruktion ist als Python-Implementation im Code-Listing 6 festgehalten. Aus Implementationsgründen erfolgt die Nummerierung der Knoten in allen Quellcodes von 0 bis $n - 1$ statt von 1 bis n.

7.1.1. Verwendung im MIP-Modell als Fixierung

Ausgangspunkt der Strukturannahme ist die Zielsetzung die einfachen Fixierungen aus Abschnitt 5.4 auf deutlich mehr Kanten, also Modellvariablen, zu erweitern. So soll einerseits die Modellgröße im Löseprozess reduziert und andererseits der Löseprozess durch Beschränkung des Lösungsraums durch starke Reduzierung von Symmetrieeffekten beschleunigt werden.

Hierfür wird die optionale Erweiterung des MIP-Modells für die Fixierungen von Kantenvariablen (siehe ZIMPL-Code-Listing 4) verwendet, um Kanten des AODP-Baumes als ausgewählt zu setzen.

Diese Fixierung erfordert Startlösungen, die ebenfalls den AODP-Baum enthalten, sodass Lösungen aus der allgemeinen Startheuristik aus Abschnitt 6.2 nicht als Startlösung verwendbar sind. Nachfolgend wird daher ein Greedy-Algorithmus entworfen, um solche Startlösungen zu erzeugen.

Neben der Erzeugung des MIP-Modells mit AODP-Baum als Fixierung für die Optimierung wurde auch ein Hilfsmodell erzeugt, für das alle Kantenvariablen anhand der Startlösung fixiert wurden, um eine Lösungsdatei für den MIP-Löser zu generieren.

7.2. Greedy-Algorithmus

Da die Struktur der optimalen Lösung bisher unklar ist, fehlt es an definitiven Erkenntnissen, welcher kombinatorische Ansatz zuverlässig zur Optimallösung führt. Das Generieren eines zufälligen d-regulären Graphen und der MIP-Löseprozess stellen hierbei eher Aufwandsextreme dar.

Auf Grundlage der Strukturannahme aus Abschnitt 7.1 wird nachfolgend ein Greedy-Algorithmus definiert. Zielsetzung ist ein systematischer, kombinatorischer Algorithmus zur Erzeugung guter AODP-Lösungen.

1. Erzeuge den zugehörigen AODP-Baum als Ausgangsgraphen.
2. Bestimme den minimalen Knotengrad im Graphen.
3. Iteriere über die Knoten mit minimalem Knotengrad (Quelle):
   a) Iteriere über die Knoten mit nicht ausgeschöpftem Knotengrad:
   Wenn der Knoten keine Kante zur Quelle hat und nicht die Quelle ist, füge den Knoten zur Liste der möglichen Ziele hinzu.
   b) Iteriere über die Liste der möglichen Ziele:
      i. Berechne den kürzesten Weg zwischen Quelle und Ziel.
      ii. Ist der Weg nicht länger als der bisher bekannte beste Weg, speichere Distanz, Quelle und Ziel als bestmögliche Kante.
   c) Wenn eine bestmögliche Kante existiert:
      i. Füge diese Kante zum Graphen hinzu.
      ii. Wenn $|E| < \left\lfloor \frac{n \cdot d}{2} \right\rfloor$, gehe zu 2.

   Sonst gehe zu 5.
4. Wenn $|E| = \left\lfloor \frac{n \cdot d}{2} \right\rfloor$, STOP: Gib den aktuellen Graphen aus.
5. Variiere den Ablauf wie folgt:
   Variation 1: Kehre zwischen Schritt 3a und 3b die Reihenfolge der Liste um.
   Variation 2: Speichere in Schritt 3(b)ii nur bei kürzerem Weg.
   Variation 3: Kombiniere Variation 1 und 2.
   Variation 4: Randomisiere die Liste zwischen Schritt 3a und 3b.

Die Variationen versuchen bestimmte Konstellationen von kürzesten Wegen und Knotengraden zu vermeiden, die als Seiteneffekte das Erreichen der maximalen Kantenanzahl verhindern. Für die Instanzen bis 99 Knoten war für die 4 Instanzen $(13, 4)$, $(21, 4)$, $(25, 5)$ und $(73, 7)$ die Nutzung von Variation 1 erforderlich. Alle anderen Instanzen konnten ohne Variation erzeugt werden.

Durch Variation 4 terminiert der Algorithmus nicht mit Sicherheit. Ohne Variation 4 terminiert der Algorithmus nicht immer mit Ausgabe einer Lösung.

Entfernt man das Kriterium der maximalen Kantenanzahl und gibt den Graphen aus,

sobald keine Kante mehr hinzugefügt werden kann, terminiert der Algorithmus immer mit einer zulässigen Lösung, da bereits der AODP-Baum eine zulässige, wenn auch extrem schlechte Lösung darstellt.

Da der Algorithmus zur Erzeugung von guten, zulässigen Startlösungen für die MIP-Modelle mit der in Abschnitt 7.1.1 vorgestellten Fixierung von Variablen entworfen wurde, wurden mögliche Variationen und deren Lösungsgüte nicht weiter vertieft.

Die Konstruktion ist als Python-Implementation im Code-Listing 7 festgehalten.

7.2.1. AODP-Baum mit Zufall

Da die häufige Berechnung von kürzesten Wegen für große Probleminstanzen laufzeitintensiv ist, lässt sich der Greedy-Algorithmus dahingehend abschwächen, dass Quelle und Ziel einer neuen Kante geeignet gewählt werden und anstelle der Berechnung des kürzesten Weges der Zufall gestellt wird.

Die deutlich geringere Laufzeit geht mit schlechteren Ergebnissen einher, die mit denen zufälliger regulärer Graphen vergleichbar sind. Ziel des Algorithmus ist es, zulässige Startlösungen für die MIP-Modelle mit der Fixierung aus Abschnitt 7.1.1 schneller als der Greedy-Algorithmus zu finden und dabei die maximale Kantenanzahl zu wahren.

1. Erzeuge den zugehörigen AODP-Baum als Ausgangsgraphen.
2. Berechne den minimalen Knotengraph $\delta_{\min}$ im Graphen.
3. Erzeuge eine Liste von Knoten mit minimalem Knotengrad (Quellen).
4. Erzeuge eine Liste von Knoten mit Knotengrad $< d$ und $\leq \delta_{\min} + 1$ (Ziele).
5. Randomisieren: Bringe die Listen jeweils in eine zufällige Reihenfolge.
6. Iteriere aufsteigend über die Liste der Quellen u:
   a) Iteriere absteigend über die Liste der Ziele v:
      i. Wenn $u \neq v$ und $(u, v) \notin E$: Füge die Kante (u, v) hinzu.
7. Wenn in Schritt 6(a)i eine Kante hinzugefügt wurde und $|E| < \left\lfloor \frac{n \cdot d}{2} \right\rfloor$, gehe zu 2.
8. Wenn $|E| = \left\lfloor \frac{n \cdot d}{2} \right\rfloor$, STOP: Gib den aktuellen Graphen aus.
9. Gehe zu 1.

Aufgrund des Zufallsprinzips ist ein Terminieren in keinem Durchlauf gesichert. Bei der Enumeration aller Instanzen bis 99 Konten konnten für 48 Instanzen mit 72 bis 99 Knoten auch nach einer Vielzahl von Iterationen keine Lösungen gefunden werden.

Die Konstruktion ist als Python-Implementation im Code-Listing 8 festgehalten.

7.2.2. AODP-Baum mit Zufall und submaximaler Kantenanzahl

Der systematische Greedy-Algorithmus und das gemischte Vorgehen aus systematischem Auswählen von Knotenkandidaten und zufälligem Hinzufügen von Kanten verfolgen beide das Ziel eines Graphen mit maximaler Kantenanzahl $\left\lfloor \frac{n \cdot d}{2} \right\rfloor$.

Insbesondere für große Instanzen aus dem Graph-Golf-Wettbewerb wurde abseits davon auch das Vorgehen implementiert, zum AODP-Baum über die Knoten mit nicht ausgereiztem Knotengrad zu iterieren und eine Kante zu einem zufälligen Zielknoten einzufügen, wenn dessen Knotengrad ebenfalls nicht maximal ist. Ist es für eine Iteration nicht möglich, eine weitere Kante hinzuzufügen, bricht der Algorithmus ab und gibt die zulässige AODP-Lösung aus.

Die Konstruktion ist als Python-Implementation im Code-Listing 9 festgehalten.

Auf diese Weise konnten für die Wettbewerbsinstanzen $(65536, 64)$, $(100000, 128)$ und $(158976, 10)$ zulässige Lösungen mit AODP-Baumstruktur generiert werden. Die zugehörigen Lösungen sind zwar geringfügig schlechter als die vom Veranstalter vorgegebenen Lösungen durch zufällige reguläre Graphen, aber dafür als Startlösung bei Verwendung der Fixierung aus Abschnitt 7.1.1 zulässig.

Die weitere Suche nach einem Mittelweg zwischen seltener Berechnung des APSP und der Generierung guter Lösungen mit größeren Knotenzahlen wurde an dieser Stelle nicht weiter vertieft. Die Kombination der Strukturannahme mit Methoden aus Abschnitt 3.2.2 wird als zielführender vermutet.

8. Auswertung der Rechenergebnisse

Für betrachtete Instanzen wurden Lösungen mit folgenden Verfahren berechnet:

- Heuristische Algorithmen:
  - RR: Random-Regular-Heuristik aus Abschnitt 6.2
  - TG: Tree-Greedy-Heuristik aus Abschnitt 7.2
  - TGR: Tree-Greedy-Random-Heuristik aus Abschnitt 7.2.1
  - TRS: Tree-Random-Sloppy-Heuristik aus Abschnitt 7.2.2
- MIP-Modell-Optimierung:
  - MCF: MCF-APSP mit einfachen Fixierungen aus Abschnitt 5.2.1:
    1 Stunde Zeitlimit, keine Startlösung, RR-Werte als Schranke
  - MCF-TG: MCF-APSP mit einfachen Fixierungen aus Abschnitt 5.2.1:
    1 Stunde Zeitlimit, TG-Startlösung, TG-Werte als Schranke
  - SAQ-TG: Quadratischer Seidel-APSP aus Abschnitt 5.2.2
    1 Stunde Zeitlimit, TG-Startlösung, TG-Werte als Schranke
  - SAL-TG: Linearisierter Seidel-APSP aus Abschnitt 5.2.3
    1 Stunde Zeitlimit, TG-Startlösung, TG-Werte als Schranke

Die Verfahren werden nachfolgend mit ihren Abkürzungen bezeichnet.

Gruppierung betrachteter Instanzen: Die Enumeration (vgl. Abschnitt 6) der ersten Instanzen mit 6 bis 99 Knoten umfasst 4465 Instanzen. Diese werden anhand ihrer praktischen Lösbarkeit wie folgt gruppiert:

- leicht: Optimallösung durch Gleichheit zur unteren Schranke durch Heuristiken
- mittel: Optimallösung mit MIP-Löser innerhalb 1 Stunde
- schwer: Die aufgelisteten Löser haben keine bewiesene Optimallösung gefunden.

Durch die in Abschnitt 6.3 beschriebenen technischen Einschränkungen für das MCF-APSP-MIP-Modell erfolgt die Auswertung zweigeteilt in kleine Instanzen bis 49 Knoten und große Instanzen ab 50 Knoten. Entsprechend werden 990 kleine Instanzen mit 6 bis 49 Knoten und 3475 große Instanzen mit 50 bis 99 Knoten betrachtet.

8.1. Ergebnisse der Heuristiken

Als erste Einschätzung zur Schwere der Instanzen und zur Effektivität der Heuristiken wird die Anzahl der generierten Lösungen betrachtet, die durch Gleichheit von Durchmesser und ASPL mit den unteren Schranken nachweisbar optimal sind:

Knotenanzahl	Instanzen	gelöst	RR	TG	TGR	TRS
6 bis 49	990	539	507	448	518	1
50 bis 99	3475	1951	1924	1526	1881	0
6 bis 99	4465	2490	2431	1974	2399	1

Tabelle 3: Optimallösungen von Heuristiken

Die Spalte „gelöst“ enthält die Anzahl der leichten Instanzen, also 539 der 990 kleinen, 1951 der 3475 großen und 2490 der 4465 AODP-Instanzen insgesamt sind leicht.

Da TGR auffällig mehr Optimallösungen als TG findet, ist die Greedy-Auswahl von Kanten zwischen maximal entfernten Knoten zwar naheliegend, jedoch wenig effektiv.

Zur weiteren Einschätzung der erzeugten Lösungsgüte betrachten wir die Gap als Differenz aus unteren Schranken und Zielfunktionswert $10 \cdot k + l$. Mit „nicht leicht“ seien hier die Gruppen mittel und schwer zusammengefasst.

Gruppe	RR	TG	TGR	TRS
leicht	$0,24$	$2,07$	$0,20$	$0,47$
nicht leicht	$7,21$	$6,21$	$6,99$	$7,94$
gesamt	$3,32$	$3,90$	$3,23$	$3,78$

Tabelle 4: Durchschnittliche Gap von Heuristiken

Aufgrund der geringen Anzahl an Instanzen ohne TGR-Lösung (43 leichte, 5 nicht leichte) wird deren Auswirkung auf die Auswertung hier vernachlässigt, da die zugehörigen Instanzen von RR und TG optimal oder mit Gap $\ll 1$ gelöst werden.

Da TRS kaum Optimallösungen findet, keine besondere Lösungsgüte liefert und im Löseprozess nicht ideal einsetzbar ist (siehe Abschnitt 7.2.2), wird diese Heuristik in nachfolgenden nicht tiefer betrachtet.

Im Vergleich zwischen RR und TGR mit TG fällt auf, dass TG zwar weniger Optimallösungen findet, jedoch die erzeugten Lösungen für mittlere und schwere Instanzen im Mittel eine deutlich geringere Gap haben.

Analog zur Methodenvielfalt in der Literatur (siehe Abschnitt 3) bietet sich auch die Kombination von Methoden an, da die Lösungen unmittelbar vergleichbar sind und für TG und TGR auch als Startlösungen für den MIP-Löseprozess verwendbar sind.

8.2. Ergebnisse der MIP-Modelle

Zur Beschleunigung des Löseprozesses wurden für die MIP-Modelle mit Seidel-APSP zusätzlich die heuristischen Fixierungen aus Abschnitt 7.1.1 angewendet. Da deren Zulässigkeit nicht bewiesen ist, gibt ein Terminieren des MIP-Lösers mit Optimallösung,

z. B. durch abgeschlossenes Branch-and-Bound-Verfahren, ohne Erreichen der unteren Schranken keinen Optimalitätsnachweis. Für eine Optimalitätsprüfung müsste ein Testlauf ohne heuristische Fixierungen mit der vermuteten Optimallösung als Startlösung erfolgen.

Für die bessere Auswertbarkeit werden optimale Lösungen aller MIP-Modelle als Optimallösungen analog zu Lösungen behandelt, die die unteren Schranken erreichen. (Diese Vereinfachung ersetzt nicht den offenen Zulässigkeitsnachweis.)

Gruppe	Instanzen	gelöst	MCF	MCF-TG	SAQ-TG	SAL-TG
leicht 6 bis 49	539	539	466	527	539	539
nicht leicht 6 bis 49	451	334	34	224	331	332
gesamt 6 bis 49	990	873	500	751	870	871
leicht 50 bis 99	1951	1934	–	–	1932	1926
nicht leicht 50 bis 99	1524	767	–	–	760	759
gesamt 50 bis 99	3475	2701	–	–	2692	2685
leicht 6 bis 99	2490	2473	–	–	2471	2465
nicht leicht 6 bis 99	1975	1101	–	–	1091	1091
gesamt 6 bis 99	4465	3574	–	–	3562	3556

Tabelle 5: Optimallösungen der MIP-Modelle (1h Zeitlimit)

Da durch TG weniger Instanzen optimal gelöst werden als durch RR und TGR, bleiben einige leichte Instanzen auch im MIP-Prozess mit TG-Startlösung ohne Optimallösung.

Die Spalte „gelöst“ zu den „nicht leicht“ Zeilen enthält die Anzahl der mittleren Instanzen. Die verbleibenden Instanzen werden als schwer bezeichnet.

Analog zu Tabelle 4 betrachten wir in der fertigen Gruppierung die Gap zur durchschnittlichen Abschätzung der Lösungsgüte. (Da mit MCF nur für 682 von 990 Instanzen eine zulässige Lösung gefunden wurde, existiert für 308 Instanzen keine sinnvoll Gap, sodass MCF in dieser Auswertung nicht berücksichtigt wird.)

Gruppe	Anzahl	MCF-TG	SAQ-TG	SAL-TG
leicht 6 bis 49	539	0,22	0,00	0,00
mittel 6 bis 49	334	4,46	2,30	2,30
schwer 6 bis 49	117	8,63	8,36	8,36
gesamt 6 bis 49	990	2,16	1,05	1,05
leicht 50 bis 99	1951	–	0,10	0,13
mittel 50 bis 99	767	–	4,79	4,79
schwer 50 bis 99	757	–	9,54	9,54
gesamt 50 bis 99	3475	–	2,15	2,17
leicht 6 bis 99	2490	–	0,08	0,10
mittel 6 bis 99	1101	–	4,22	4,22
schwer 6 bis 99	874	–	9,38	9,38
gesamt 6 bis 99	4465	–	1,91	1,92

Tabelle 6: Durchschnittliche Gap der MIP-Modelle (1h Zeitlimit)

Das Modell mit MCF-APSP zeigt bereits bei kleinen Instanzen deutliche Schwächen in Speicherbedarf und Laufzeit im MIP-Löseprozess. Die Modelle mit Seidel-APSP unterscheiden sich nicht signifikant im Lösungsverhalten.

Erwartbarerweise gibt es bei höheren Knotenanzahlen mehr schwere Instanzen, sodass oberhalb von 100 Knoten das Verhältnis von mittleren und schweren Instanzen sich aufgrund der Modellgröße weiter in Richtung der schweren Instanzen verschieben wird.

Bei kleinen Knotenanzahlen sind schwere Instanzen selten und mittlere Instanzen häufen sich für ungerades $n \cdot d$, da dieses oft mit einer Optimallösung ungleich den unteren Schranken einhergeht.

Ab 30 Knoten häufen sich schwere Instanzen mit maximalem Knotengrad $d < \frac{n}{4}$, deren Anzahl entsprechend mit der Knotenanzahl für große Instanzen steigt.

Für hohe Knotenanzahlen (z. B. $n \gg 100$) mit moderatem Durchmesser (z. B. $d \ll 100$) wie in den Wettbewerbsinstanzen erscheinen so auch hohe Rechenaufwände (z. B. von 60 Tagen, vgl. Shimizu und Mori, 2016) vertretbar.

Für Knotengrade $d > \frac{n}{2}$ sind die zugehörigen Instanzen häufig leicht lösbar und haben bereits in der heuristischen Startlösung den Durchmesser $k_{RR} = 2$. Zu diesen Instanzen gehören leider selten praxisrelevante Fragestellungen, sodass leicht lösbare AODP-Instanzen mit hohem Knotengrad oder geringer Knotenanzahl praktisch eher uninteressant erscheinen.

8.3. Visualisierung von Lösungen

Zum ersten Verständnis für die AODP-Baum-Heuristik wurde in Abschnitt 7 eine Lösung in Ebenen ausgehend von einem Wurzelknoten betrachtet. Nach Umsetzung der

Fixierungen mit nachweislich optimalen Lösungen bleibt offen, ob daraus direkt optimale Lösungen konstruiert werden können.

Da bei schweren Instanzen der Knotengrad 3 häufig auftritt, werden nachfolgend optimale Lösungen der mittleren Instanzen mit geringster Knotenanzahl $(7, 3)$ und $(9, 3)$ zur weiteren Betrachtung visualisiert. Um hierbei analytische Fehler durch heuristische Fixierungen auszuschließen, wurden beide Optimallösungen wurden ohne diese berechnet.

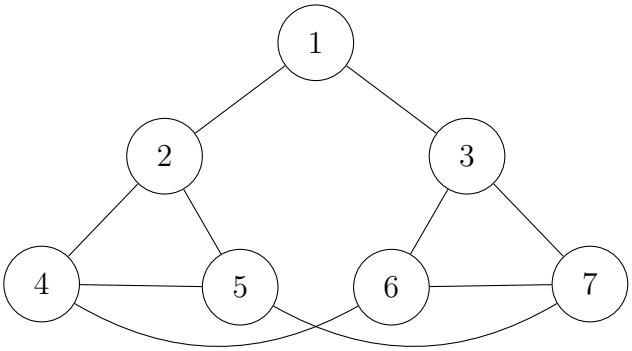

Abbildung 6: optimaler Lösungsgraph für $(7, 3)$

Die Lösung erfüllt die Moore-Schranke für den Durchmesser $k = 2$. Die Cerf-Schranke für die ASPL ist mit $1,52381 > 1,5$ nicht erreicht. Durch die Optimalität der Lösung ist gezeigt, dass die Cerf-Schranke für diese Instanz nicht erreichbar ist.

Die Struktur der Lösung besteht aus den AODP-Baum und einem 4-Kreis der Blätter. Aufgrund der geringen Instanzgröße ließen sich diverse Hypothesen formulieren. Das nachfolgende Beispiel zeigt bereits eine stärkere Kreuzung in den unteren beiden Ebenen:

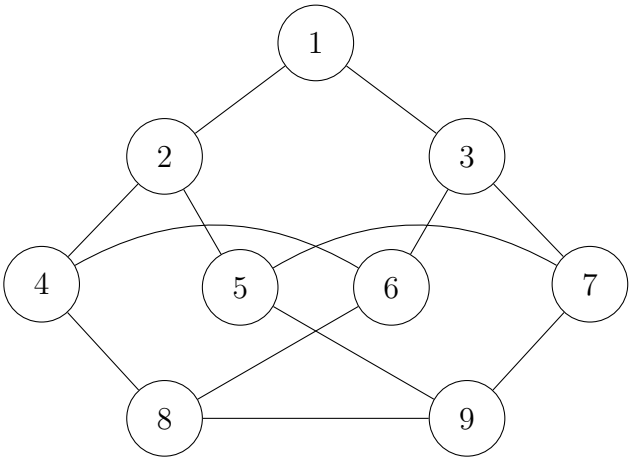

Abbildung 7: optimaler Lösungsgraph für $(9, 3)$

Die betrachtete Struktur besteht aus drei Bestandteilen:

- Kanten innerhalb der Blattebene des einfachen AODP-Baumes
- Kanten innerhalb der Ebene der angehängten Knoten
- Kanten zwischen diesen beiden Ebenen

Eine entsprechende Burgzinnen-Heuristik könnte so deutlich strukturiertere Lösungen konstruieren. Um zwischen Durchmesser und Baumhöhe eine nutzbare Koppelung aufzubauen, erzeugen die Querkanten ab der Blattebene letztendlich Kreise analog zu den $(2k-1)$-Kreisen von Kitasuka und Iida, 2016.

8.4. Ergebnisse für Wettbewerbsinstanzen 2021

Neben der in Abschnitt 7 vorgestellten Optimallösung für die Wettbewerbs-Instanz $(40, 5)$ konnte auch die von den Veranstaltern außerhalb der Wertung eingestellte Instanz $(30, 4)$ mit $k = 3$ und $l = 2,31\ldots$ optimal gelöst werden. Mit dem Verfahren SAQ-TG ließen sich beide Instanzen innerhalb einer Stunde auf dem in Abschnitt 7 benannten Testsystem mit 8 Threads lösen.

Aufgrund der fehlenden Optimierungsrichtung konnten mit der in Abschnitt 5.3 angesprochenen Benders-Dekomposition keine Erfolge verzeichnet werden. Die damit gefundenen Lösungen waren meist schlechter als die bereitgestellte Zufallslösung, aber letztendlich vor allem schlechter als die Lösungen der in Abschnitt 7 vorgestellten Heuristiken auf Basis des AODP-Baumes.

Anknüpfend an Tabelle 1 von Seite 31 folgt eine Übersicht der bisherigen Ergebnisse zu den Wettbewerbsinstanzen 2021:

Instanz	Quelle	Status	k	$\Delta\, k$	l	$\Delta\, l$
$(40, 5)$	SAQ	optimal	1	$2,23076$	$2,37179$	$0,14103$
$(432, 12)$	TG	<RR	1	$2,63805$	$2,70395$	$0,06590$
$(512, 18)$	TG	<RR	1	$2,33072$	$2,48283$	$0,15211$
$(1024, 5)$	TG	<RR	2	$4,45259$	$4,71975$	$0,26716$
$(3602, 24)$	TGR	<RR	1	$2,83338$	$2,86897$	$0,03559$
$(65536, 64)$	TRS	>RR	1	$2,93652$	$2,95774$	$0,02122$
$(158976, 10)$	TRS	>RR	1	$5,47772$	$5,58714$	$0,10942$
$(100000, 128)$	TRS	>RR	1	$2,83488$	$2,84742$	$0,01254$

Tabelle 7: Lösungen für die Wettbewerbsinstanzen 2021

Die Lösungen zu den letzten drei Instanzen wurden nur erzeugt, um die Instanzen überhaupt betrachtet zu haben, da außer RR und TRS keine andere Methode eine Lösung innerhalb weniger Tage lieferte.

Für $(3602, 24)$ wurde die Erzeugung per TG nach mehreren Tagen Laufzeit ohne Ergebnis abgebrochen. Stattdessen wurde mit TGR eine Lösung erzeugt, die minimal besser als

die RR-Lösung ist.

Für $(40, 5)$ funktionierten alle bisher beschrieben Vorgehensweisen problemlos und bis auf das MCF-APSP-MIP-Modell in geringer bis moderater Laufzeit.

Für $(432, 12)$, $(512, 18)$ und $(1024, 5)$ konnte per TG eine Lösung erzeugt werden, die besser als RR ist. Für diese Instanzen wurden mit TGR und TRS keine besseren Lösungen erzeugt. Weiterhin ist für diese Instanzen auch die Verwendung des Gurobi-MIP-Lösers mit einem der Seidel-APSP-MIP-Modelle auf einem leistungsstärkeren System geplant.

8.4.1. Greedy-Ansatz für obere Schranken

Für die Instanz $(432, 12)$ ist (Stand: 03.09.2021) auf der Graph-Golf-Website (Graph Golf Rankings and solutions, 2021) die Optimallösung $(3; 2, 63805)$ gelistet. Entsprechend könnte man den Greedy-Ansatz auch auf die oberen Schranken im MIP-Modell anwenden und das Modell ohne Startlösung mit Fixierung des Durchmessers auf $k = 3$ durchrechnen. Während für die Instanz $(512, 18)$ bereits Lösungen mit minimalem Durchmesser bekannt sind, gibt es für $(1024, 5)$ Lösungen für den Durchmesser 6 im Wettbewerb, jedoch bisher keine für den Durchmesser $k_{LB} = 5$.

Entsprechend könnte man versuchen, auch ohne Kenntnis der Zulässigkeit den Suchraum und die Tiefe des APSP auf einen vorgegebenen Durchmesser zu beschränken. Auf diese Weise kann ggf. nachweisbar sein, dass es keinen Graphen mit diesem Durchmesser gibt. Das alternative Ergebnis einer besseren Lösung ist als Erkenntnis ebenfalls interessant.

Nimmt man die Baumstruktur als zulässig an, lässt sich für die kleinste schwere Instanz $(17, 4)$ die Existenz einer Lösung mit Durchmesser $k_{LB} = 2$ in unter 10 Sekunden Laufzeit verneinen. Mit Gurobi auf 16 Threads ist diese Aussage mit einfachen Fixierungen aus Abschnitt 5.4 in unter 1 Minute Laufzeit und ohne Fixierung in unter 2 Minuten möglich.

8.5. Ergebnisse früherer Wettbewerbsinstanzen

Durch Überschneidung zur Enumeration lassen sich für einige Wettbewerbsinstanzen die nachfolgenden Lösungen dokumentieren:

Instance	Jahr	Wettbewerb (mit Gap)	MIP-Lösung (mit Gap)
$(16, 3)$	2015	$32, 20$ (optimal)	$32, 20$ (optimal)
$(16, 4)$	2015	$31, 75$ (optimal)	$31, 75$ (optimal)
$(32, 5)$	2017	$32, 03$ (optimal)	$32, 04$ $(0, 01)$
$(36, 3)$	2016	$43, 07$ $(0, 01)$	$43, 08$ $(0, 02)$
$(50, 4)$	2019	$42, 64$ $(10, 05)$	$42, 70$ $(10, 11)$
$(64, 3)$	2015	$53, 77$ $(0, 01)$	$73, 86$ $(20, 10)$
$(64, 4)$	2015	$42, 87$ $(0, 01)$	$52, 98$ $(10, 12)$
$(64, 8)$	2016	$31, 93$ $(10, 05)$	$32, 06$ $(10, 19)$
$(64, 16)$	2015	$21, 75$ (optimal)	$21, 75$ (optimal)
$(64, 23)$	2015	$21, 64$ (optimal)	$21, 63$ (optimal)
$(64, 60)$	2015	$21, 05$ (optimal)	$21, 05$ (optimal)
$(72, 4)$	2018	$42, 99$ (optimal)	$53, 07$ $(10, 08)$
$(96, 3)$	2016	$64, 27$ $(0, 07)$	$74, 37$ $(10, 17)$

Tabelle 8: Lösungen für Instanzen früherer Wettbewerbsjahre

Die Optimallösungen wurden jeweils in unter 1 Stunde Laufzeit auf 1 Thread gefunden. Für die anderen Instanzen wurde der Löseprozess im Vorhinein nicht begrenzt und manuell nach 10 Stunden bis 5 Tagen abgebrochen.

Für die Instanz $(36, 3)$ war die Fixierung des Durchmessers auf $k = 4$ erfolgreich: Die Lösung wurde mit einem Thread nach 58.785s für das quadratische und 80.907s für das linearisierte Seidel-APSP-MIP-Modell erhalten, während der MIP-Löseprozess mit $k =$ 5-Startlösung auch nach ca. 250.000s mit unverändertem Durchmesser und moderater ASPL-Verbesserung abgebrochen wurde. Für die anderen Instanzen wurden auf diesem Wege keine Lösungen gefunden.

Bei Versuchen mit großen Instanzen wurde für die Instanz $(256, 64)$ aus dem Wettbewerbsjahr 2015 wurde mit dem linearisierten Seidel-APSP-MIP-Modell in 85.619s die Optimallösung mit dem Zielfunktionswert $21, 74$ gefunden. Hierbei wurde durch eine TG-Lösung mit Zielfunktionswert $31, 75$ der Durchmesser auf $k \leq 3$ beschränkt.

Da diese Instanzen nicht im Fokus der Betrachtung standen, wurde nur mit moderatem Aufwand versucht, diese besser oder optimal zu lösen. Als analytischer Kontext bieten diese jedoch gute Referenzwerte auf gute oder optimale Lösungen der Instanzen sowie das bisher technisch Machbare hinsichtlich der Erreichbarkeit der unteren Schranken.

9. Abschluss und Ausblick

In der Betrachtung des ODP bleibt offen, inwiefern sich aus der Problemformulierung eine Lösungsstruktur ableiten lässt. Ebenso wie beim DDP bleibt unklar, ob es einen Algorithmus mit polynomialer Laufzeit gibt, der zu einer Instanz (n, d) den minimalen Durchmesser $k(n, d)$ sowie für das AODP die minimalen ASPL $l(n, d)$ liefert. Die gezeigte Äquivalenz macht eventuelle zukünftige Erkenntnisse, dass das DDP NP-schwer ist, auf das ODP übertragbar.

Die MIP-Modellierung bietet einen systematischen Ansatz zur Problemlösung, jedoch mit praktischen Grenzen in der Laufzeit und im Speicherbedarf. Zusätzlich zu den Standardmethoden üblicher Lösersoftware lässt sich der Lösungsraum anhand der bekannten Schranken und heuristisch erzeugter Lösungen stark einschränken, um die Suche nach guten Lösungen zu erleichtern.

Aufgrund der Symmetrie des ODP wurde ausgehend von der Analyse optimaler Lösungen ein heuristischer Ansatz zur Fixierung einer baumförmigen Grundstruktur vorgestellt. Auf Grundlage dieser Baumstruktur wurden Startlösungen im Greedy-Verfahren erzeugt.

Der Bogen aus Erkenntnissen der betrachteten Literatur zum ODP und Grundlagen zum DDP, bewiesenen Komplexitätsbetrachtungen, einer Problem-Modellierung als MIP und praktischen Modellrechnungen wird abschließend ergänzt mit dem pragmatischen Heuristik-Ansatz.

Denkbar ist eine weitere Vertiefung der heuristischen Fixierung dahingehend, ob diese aufgrund einer strukturellen Ähnlichkeit zur Theorie der Moore- und Cerf-Schranke generell zulässig ist, also keine Optimallösung vom Lösungsraum-MIP-Polyeder abschneidet. Einen weiteren Ansatz bietet die Koppelung von erfolgreichen Verfahren der Literatur mit der heuristischen Fixierung, um mehr gute und ggf. optimale Lösungen mit Verfahren zu erzeugen, die polynomiale Laufzeit haben und den Einblick in mögliche Lösungsstrukturen vertiefen.

Über die MIP-Methodik könnten die bereits vorhandenen Erfolge im Graph-Golf-Wettbewerb auf größere Instanzen und bessere Lösungen ausgeweitet werden. Fachlich könnte hier die Aufmerksamkeit von zufallsbasierten Lösungsgenerierungen und -verbesserungen stärker auf systematische Vorgehensweisen orientiert werden.

Aufgrund der generischen Erweiterungsmöglichkeiten der MIP-Methodik lassen sich weitere Nebenbedingungen ergänzen oder verändern, z. B. Strukturbedingungen wie der Grid-Graph vergangener Graph-Golf-Wettbewerbsjahre oder der Host-Switch-Graph im aktuellen Wettbewerb.

Mit dem Host-Switch-Graph wird eine stärkere Nähe des abstrakter angelegten ODP zur Praxisanwendung der Vernetzung in Rechenzentren hergestellt: Durch die Unterscheidung zwischen Endgeräten (Host) mit nur einer Netzwerkverbindung und Verteilungen (Switches) mit mehreren Netzwerkverbindungen entsteht eine flexiblere Nutzung der

verfügbaren Knotengrade. Darüber hinaus ist eine feingranularere Definition der Knotengrade denkbar, z. B. durch die Unterscheidung verschiedener Switch-Größen, wodurch jedoch die strukturelle Nähe zum klassischen ODP und den zugehörigen Überlegungen und Erkenntnissen abnimmt.

Ebenso wie das ODP eine theoretische Vereinfachung der üblichen komplexen Praxis im Rechenzentrumsalltag darstellt, sind in der Praxis suboptimale Lösungen mit abschätzbarer Güte und vertretbarem Rechenaufwand durchaus verwendbar. Der Ansatz der Beschränkung des Durchmessers durch eine obere Schranke, also auf ausreichend gute und damit interessante Lösungen, wendet umgekehrt den Pragmatismus der Praxis auf die theoretische Suche an.

Hier bietet die MIP-Modellierung den theoretischen wie praktischen Vorteil der Integration von theoretischen Erkenntnissen und praktischen Methoden in einen strukturierten und gerichteten Löseprozess.

A. Literaturverzeichnis

Benders, J. F. (1962). Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, *4*(1), 238–252. DOI: 10.1007/bf01386316

Besta, M. & Hoefler, T. (2014). Slim Fly: A Cost Effective Low-Diameter Network Topology, In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, 348-359, IEEE. DOI: 10.1109/sc.2014.34

Brown, W. G. (1966). On Graphs that do not Contain a Thomsen Graph. *Canadian Mathematical Bulletin*, *9*(3), 281–285. DOI: 10.4153/cmb-1966-036-2

Cerf, V. G., Cowan, D. D., Mullin, R. C. & Stanton, R. G. (1974). A lower bound on the average shortest path length in regular graphs. *Networks*, *4*(4), 335–342. DOI: 10.1002/net.3230040405

Chung, F. R. K. (1986). Diameters of communication networks. *AMS Proceedings of Symposia in Applied Mathematics*, *34*, 1–18. DOI: 10.1090/psapm/034/846852

Chung, F. R. K. (1987). Diameters of graphs: Old problems and new results. *Congressus Numerantium*, *60*(2), 295–317.

Combinatorics Wiki. (2019). *The Degree Diameter Problem for General Graphs*. Verfügbar 3. September 2021 unter http://www.combinatoricswiki.org/wiki/The_Degree_Diameter_Problem_for_General_Graphs

Dekker, A., Pérez-Rosés, H., Pineda-Villavicencio, G. & Watters, P. (2012). The Maximum Degree & Diameter-Bounded Subgraph and its Applications. *Journal of Mathematical Modelling and Algorithms*, *11*(3), 249–268. DOI: 10.1007/s10852-012-9182-8

Erdős, P. & Rényi, A. (1963). On a problem in the theory of graphs [Hungarian]. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences, Volume VII, Series B/4*, 623–641.

Erdős, P., Rényi, A. & Sós, V. (1966). On a Problem of Graph Theory. *Studia Scientiarum Mathematicarum Hungarica*, *1*, 215–235.

Erdős, P., Fajtlowicz, S. & Hoffman, A. J. (1980). Maximum degree in graphs of diameter 2. *Networks*, *10*(1), 87–90. DOI: 10.1002/net.3230100109

Fujimoto, N. & Kobayashi, H. (2018). Some Torus-embedded Graphs with Regular Structure Have the Minimum Diameter and the Minimum Average Shortest Path Length, In *TENCON 2018 - 2018 IEEE Region 10 Conference*, 2264-2269, IEEE. DOI: 10.1109/tencon.2018.8650461

Garey, M. R. & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA, W. H. Freeman & Co.

Graph Golf Events. (2021). *Events-Seite der Graph-Golf-Website*. Verfügbar 3. September 2021 unter http://research.nii.ac.jp/graphgolf/events.html

Graph Golf Problem statement. (2021). *Problembeschreibungs-Seite der Graph-Golf-Website (Problem statement)*. Verfügbar 3. September 2021 unter http://research.nii.ac.jp/graphgolf/problem.html

Graph Golf Rankings and solutions. (2021). *Ranking-Seite der Graph-Golf-Website (Rankings and solutions 2021)*. Verfügbar 3. September 2021 unter http://research.nii.ac.jp/graphgolf/ranking.html

Graph Golf Rules. (2021). *Rules-Seite der Graph-Golf-Website*. Verfügbar 3. September 2021 unter http://research.nii.ac.jp/graphgolf/rules.html

Graph-Golf-Website. (2021). *Graph Golf The Order/degree Problem Competition*. Verfügbar 3. September 2021 unter http://research.nii.ac.jp/graphgolf/

Hasegawa, R. & Handa, H. (2018). Solving Order/Degree Problems by Using EDA-GK with a Novel Sampling Method. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, *22*(2), 236–241. DOI: 10.20965/jaciii.2018.p0236

Hoffman, A. J. & Singleton, R. R. (1960). On Moore Graphs with Diameters 2 and 3. *IBM Journal of Research and Development*, *4*(5), 497–504. DOI: 10.1147/rd.45.0497

Kawamata, I. (2017). *Vortrag beim CANDAR2017 Graph Golf Workshop: Approximate evaluation and voltage assignment for order/degree problem*. Verfügbar 3. September 2021 unter http://research.nii.ac.jp/graphgolf/2017/candar17/graphgolf2017-kawamata.pdf

Kitasuka, T. & Iida, M. (2016). A heuristic method of generating diameter 3 graphs for order/degree problem (invited paper), In *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, IEEE. DOI: 10.1109/nocs.2016.7579334

Kitasuka, T., Matsuzaki, T. & Iida, M. (2018). Order Adjustment Approach Using Cayley Graphs for the Order/Degree Problem. *IEICE Transactions on Information and Systems*, *E101.D*(12), 2908–2915. DOI: 10.1587/transinf.2018pap0008

Koch, T. (2004). *Rapid Mathematical Prototyping* (Diss.). Technische Universität Berlin. DOI: 10.14279/depositonce-975

Koibuchi, M., Matsutani, H., Amano, H., Hsu, D. F. & Casanova, H. (2012). A case for random shortcut topologies for HPC interconnects, In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 177-188, IEEE. DOI: 10.1109/isca.2012.6237016

Koibuchi, M., Fujiwara, I., Chaix, F. & Casanova, H. (2016). Towards Ideal Hop Counts in Interconnection Networks with Arbitrary Size, In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, 188-194, IEEE. DOI: 10.1109/candar.2016.0042

McKay, B. D., Miller, M. & Širáň, J. (1998). A Note on Large Graphs of Diameter Two and Given Maximum Degree. *Journal of Combinatorial Theory, Series B*, *74*(1), 110–118. DOI: 10.1006/jctb.1998.1828

Miller, M. & Slamin. (2000). On the Monotonicity of Minimum Diameter with Respect to Order and Maximum Out-Degree. In *Computing and Combinatorics. COCOON 2000. Lecture Notes in Computer Science*. 193-201, Springer Berlin Heidelberg. DOI: 10.1007/3-540-44968-x_19

Miller, M., Slamin, Ryan, J. & Baskoro, E. T. (2013). Construction Techniques for Digraphs with Minimum Diameter. In *Combinatorial Algorithms. IWOCA 2013. Lecture Notes in Computer Science*. 327-336, Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-45278-9_28

Miller, M. & Širáň, J. (2013). Moore graphs and beyond: A survey of the degree/diameter problem (Update). *The Electronic Journal of Combinatorics*, *DS14*, 1–92. DOI: 10.37236/35

Mizuno, R. & Ishida, Y. (2016). Constructing large-scale low-latency network from small optimal networks, In *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 1-5, IEEE. DOI: 10.1109/nocs.2016.7579336

Nakao, M., Murai, H. & Sato, M. (2019). A Method for Order/Degree Problem Based on Graph Symmetry and Simulated Annealing with MPI/OpenMP Parallelization, In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 128-137, ACM. DOI: 10.1145/3293320.3293325

Nakao, M., Murai, H. & Sato, M. (2020). Parallelization of All-Pairs-Shortest-Path Algorithms in Unweighted Graph, In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 63-72, ACM. DOI: 10.1145/3368474.3368478

Seidel, R. (1995). On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs. *Journal of Computer and System Sciences*, *51*(3), 400–403. DOI: 10.1006/jcss.1995.1078

Shimizu, N. & Mori, R. (2016). Average shortest path length of graphs of diameter 3, In *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 1-6, IEEE. DOI: 10.1109/nocs.2016.7579335

Shin, J.-Y., Wong, B. & Sirer, E. G. (2011). Small-world datacenters, In *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*, 1-6, ACM Press. DOI: 10.1145/2038916.2038918

Singla, A., Hong, C.-Y., Popa, L. & Godfrey, P. B. (2012). Jellyfish: Networking Data Centers Randomly, In *NSDI'12: Proceedings of the 9th USENIX conference on Networked Systems Design and ImplementationApril 2012*, 1-17, 1USENIX Association. DOI: 10.5555/2228298.2228322

Steger, A. & Wormald, N. (1999). Generating Random Regular Graphs Quickly. *Combinatorics, Probability and Computing*, *8*(4), 377–396. DOI: 10.1017/s0963548399003867

Terao, H. (2019). *Vortrag beim CANDAR2019 Graph Golf Workshop: Construction of Small Diameter/ASPL Graph with GPU.* Verfügbar 3. September 2021 unter http://research.nii.ac.jp/graphgolf/2019/candar19/graphgolf2019-terao.pdf

B. Code-Listings

ZIMPL: Flussmodellierung

```
param nodes := 40;
param degree := 5;

set N := {0..nodes-1};
set E := {<i,j> in N * N with i != j};
set F := {<i,j> in N * N with i < j};

var diameter; var ASPL;
var SP[F]; var z[F] binary;
var x[F * E] binary;

minimize AODP: 10 * diameter + ASPL;

subto diameter: forall <s,t> in F: SP[s,t] <= diameter;

subto ASPL:
ASPL == 2/(nodes*(nodes-1)) * sum <s,t> in F: SP[s,t];

subto APSP: forall <s,t> in F:
SP[s,t] == sum <s,t,i,j> in F * E: x[s,t,i,j];

subto SPtransit:
forall <s,t> in F: forall <i> in N \ {s,t}:
sum <s,t,i,j> in F * E: x[s,t,i,j]
- sum <s,t,j,i> in F * E: x[s,t,j,i] == 0;

subto SPsource: forall <s,t> in F:
sum <s,t,s,j> in F * E: x[s,t,s,j]
- sum <s,t,j,s> in F * E: x[s,t,j,s] == 1;

subto SPtarget: forall <s,t> in F:
sum <s,t,t,j> in F * E: x[s,t,t,j]
- sum <s,t,j,t> in F * E: x[s,t,j,t] == -1;

subto degree: forall <i> in N:
sum <i,j> in F: z[i,j] + sum <j,i> in F: z[j,i] <= degree;

subto ZXlink: forall <s,t,i,j> in F*F:
z[i,j] >= x[s,t,i,j] and z[i,j] >= x[s,t,j,i];
```

Listing 1: ZIMPL Modell mit Flussmodellierung für AODP $(40, 5)$

ZIMPL: quadratischer Seidel-APSP

```
param nodes := 40;
param degree := 5;
param minDiameter := 3;
param minASPL := 2.230769230769231;
param maxDiameter := 4;
param maxASPL := 2.373076923076923;

set N := {0..nodes-1};
set F := {<s,t> in N * N with s < t};
set D := {0..maxDiameter-1};

var diameter integer >= minDiameter <= maxDiameter;
var ASPL real >= minASPL <= maxASPL;
var dist[F*D] binary;

minimize AODP: 10 * diameter + ASPL;

subto diameter: forall <s,t> in F:
1 + sum <d> in D: (1 - dist[s,t,d]) <= diameter;

subto ASPL:
ASPL == 2/(nodes*(nodes-1)) * sum <s,t> in F:
(1 + sum <d> in D: (1 - dist[s,t,d]));

subto DistCalc:
forall <d> in D without {maxDiameter-1}: forall <s,t> in F:
dist[s,t,d+1] <= dist[s,t,d] + sum <k> in N without {s,t}:
dist[min(s,k),max(s,k),d] * dist[min(k,t),max(k,t),0];

subto degreeButLast: forall <s> in N without {nodes-1}:
sum <t> in N without {s}: dist[min(s,t),max(s,t),0] == degree;
subto degreeLast:
sum <t> in N without {nodes-1}: dist[t,nodes-1,0]
== if (nodes mod 2 == 0) then degree else degree-1 end;
```

Listing 2: ZIMPL-Modell mit quadratischem Seidel-APSP für AODP $(40, 5)$

ZIMPL: linearisierter Seidel-APSP

```
param nodes := 40;
param degree := 5;
param minDiameter := 3;
param minASPL := 2.230769230769231;
param maxDiameter := 4;
param maxASPL := 2.373076923076923;

set N := {0..nodes-1};
set F := {<s,t> in N * N with s < t};
set D := {0..maxDiameter-1};

var diameter integer >= minDiameter <= maxDiameter;
var ASPL real >= minASPL <= maxASPL;
var dist[F*D] binary;
var y[F*N*D] binary;

minimize AODP: 10 * diameter + ASPL;

subto diameter: forall <s,t> in F:
1 + sum <d> in D: (1 - dist[s,t,d]) <= diameter;

subto ASPL:
ASPL == 2/(nodes*(nodes-1)) * sum <s,t> in F:
(1 + sum <d> in D: (1 - dist[s,t,d]));

subto DistCalc: forall <d> in D without {maxDiameter-1}:
forall <s,t> in F: dist[s,t,d+1] <= dist[s,t,d]
+ sum <k> in N without {s,t}: y[s,t,k,d];

subto DistLinearize:
forall <d> in D without {maxDiameter-1}:
forall <s,t> in F: forall <k> in N without {s,t}:
y[s,t,k,d] <= dist[min(s,k),max(s,k),d]
and y[s,t,k,d] <= dist[min(k,t),max(k,t),0];

subto degreeButLast: forall <s> in N without {nodes-1}:
sum <t> in N without {s}: dist[min(s,t),max(s,t),0] == degree;
subto degreeLast:
sum <t> in N without {nodes-1}: dist[t,nodes-1,0]
== if (nodes mod 2 == 0) then degree else degree-1 end;
```

Listing 3: ZIMPL-Modell mit linearisiertem Seidel-APSP für AODP $(40, 5)$

ZIMPL: Fixierungen aus Edges-Datei

```
set T := {read "AODP-Tree_40_05.edges" as "<1n,2n>"};

subto Fixings:
forall <i,j> in T:
forall <d> in D without {maxDiameter-1}:
dist[min(i,j),max(i,j),d] == 1;
```

Listing 4: Erweiterung für ZIMPL-Modelle für Fixierungen aus Edges-Datei

Anstelle der Fixierung der Baumstruktur wurde für die Erzeugung von Startlösungen folgende Code-Zeile verwendet:

```
set T := {read "TreeGreedy_40_05.edges" as "<1n,2n>"};
```

Python: Einfache Heuristik: `create-heur-randCS.py`

```
#!/usr/bin/env python3
# coding: utf-8
#=======================================================================
#
# generate heuristic solution to order degree problem
# by Robert Waniek
# 2021
# "create-heur-randCS.py" is licensed under a
# Creative Commons Attribution 4.0 International License.
# http://creativecommons.org/licenses/by/4.0/
#
# based on create-random.py (Create a random graph)
# by Ikki Fujiwara, National Institute of Informatics
# 2018-11-12
# "create-random.py" is licensed under a
# Creative Commons Attribution 4.0 International License.
# http://creativecommons.org/licenses/by/4.0/
#
# CHANGELOG:
# separated script features (creation, statistics, rendering)
# minor feature: added naive support for uneven nodes*degree
# improved feature: support for uneven nodes*3 via cycle-star-graph
# minor feature: handle trivial cases
# minor use case: order-degree-assertion on generated graph
#
#=======================================================================

import networkx
import argparse
import math
import numpy
import sys

########## BASICS ##########

argumentparser = argparse.ArgumentParser()
argumentparser.add_argument('nodes', type=int)
argumentparser.add_argument('degree', type=int)

def main(args):

########## PARSE PARAMETERS ##########

  nodes = args.nodes
  degree = args.degree

########## HANDLE TRIVIAL CASES ##########

  if nodes < 1:
    sys.exit("trivial solution: no graph")
```

```
  if degree < 2:
    sys.exit("trivial solution: no connected graph available")
  if nodes == 2:
    sys.exit("trivial solution: 0 -- 1")
  if degree == 2:
    sys.exit("trivial solution: cycle or not connected")
  if nodes <= degree+1:
    sys.exit("trivial solution: complete graph")
  assert nodes >= 5
  assert degree >= 3

########## GENERATE GRAPH ##########

  #basic case: standard random regular graph exists
  if degree * nodes % 2 == 0:
    graph = networkx.random_regular_graph(degree, nodes, 0)
  #edge case heuristic: nodes*3 % 2 == 1
  elif degree == 3: # and degree * nodes % 2 == 1
    graph = generate_cyclestar(nodes)
  #use (existing) regular graph with nodes-1 and degree-1
  else: # degree > 3 and degree * nodes % 2 == 1
    graph = networkx.random_regular_graph(degree - 1, nodes - 1, 0)
    #add missing node with full degree
    for i in range(0, degree - 1):
      graph.add_edge(i, nodes - 1)

########## GENERATE EDGES FILE ##########

  #verify feasibility of generated graph
  assert graph.number_of_nodes() == nodes
  assert graph.number_of_edges() <= math.floor((degree*nodes)/2)
  assert max([d for n, d in graph.degree()]) == degree
  assert networkx.is_connected(graph)

  basename = "n{}d{}.randomCS".format(nodes,degree)
  networkx.write_edgelist(graph, basename+".edges", data=False)
  return

########## FUNCTIONS ##########

def generate_cyclestar(nodes):
  starsize = math.floor(nodes / 2)
  cyclesize = nodes - starsize
  skipmax = math.floor(starsize / 2)
  assert cyclesize + starsize == nodes
  #creating basic cycle
  base = networkx.cycle_graph(cyclesize)
  #adding star connectors
  for star in range(0, 1):
    for i in range(starsize):
      base.add_edge(i, cyclesize + star * starsize + i)
  #prepare star-tries
```

```
  mindia = nodes
  skip = 2
  done = False
  #try stars
  while not done:
    graph = base.copy()
    i = 0
    j = -1
    #add star
    while j != 0:
      j = (skip+i)%starsize
      graph.add_edge(cyclesize+i,cyclesize+j)
      i = j
    #check result
    if networkx.is_connected(graph):
      assert max([d for n, d in graph.degree()]) == 3
      hops = networkx.shortest_path_length(graph, weight=None)
      diam, aspl = max_avg_for_matrix(hops)
      if diam < mindia or (diam == mindia and aspl < minaspl):
        mingraph = graph.copy()
        mindia = diam
        minaspl = aspl
    #prepare next star
    skip = skip + 1
    done = (skip > skipmax)
  return mingraph

def max_avg_for_matrix(data):
  cnt = 0
  sum = 0.0
  max = 0.0
  for i, row in data:
    for j, val in row.items():
      if i != j:
        cnt += 1
        sum += val
        if max < val:
          max = val
  return max, sum / cnt

if __name__ == '__main__':
  main(argumentparser.parse_args())
```

Listing 5: Erweiterung der Zufallsheuristik (Python-Implementation)

Python: Baumstruktur für Fixierung: AODP-Tree.py

```
#!/usr/bin/env python3
import argparse
import logging
import sys
import networkx
import math

########## BASICS ##########

argumentparser = argparse.ArgumentParser()
argumentparser.add_argument('nodes', type=int)
argumentparser.add_argument('degree', type=int)

logger = logging.getLogger(__name__)
logging.basicConfig(format="%(message)s")
logger.setLevel("INFO")
#logger.setLevel("DEBUG")

def main(args):

########## PARSE PARAMETERS ##########

  nodes = args.nodes
  degree = args.degree

########## HANDLE TRIVIAL CASES ##########

  if nodes < 1:
    sys.exit("trivial solution: no graph")
  if degree < 2:
    sys.exit("trivial solution: no connected graph")
  if nodes == 2:
    sys.exit("trivial solution: 0 -- 1")
  if degree == 2:
    sys.exit("trivial solution: cycle")
  if nodes <= degree+1:
    sys.exit("trivial solution: complete graph")
  assert nodes >= 5

########## PLAN TREE ##########

  assert nodes > degree

  graph = networkx.empty_graph(nodes)
  levelsize = {}
  levelsize[0] = 1
  nodecount = 1
  levelsize[1] = degree
  maxlevel = 0
  while nodecount+levelsize[maxlevel+1] < nodes:
```

```
    nodecount += levelsize[maxlevel+1]
    maxlevel += 1
    levelsize[maxlevel+1] = levelsize[maxlevel] * (degree-1)
  logger.debug("DEBUG: maxlevel {} levelsize {}".format(maxlevel, levelsize))

########## GENERATE TREE ##########

  logger.info("generate tree for fixings")

  #root
  logger.debug("root")
  soffset = 0
  source = 0
  ssize = 1
  tsize = degree
  toffset = soffset + ssize + source*tsize
  for target in range(tsize):
    logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
        {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
        toffset + target))
    graph.add_edge(soffset + source, toffset + target)
  nodecount = 1+degree

  #root neighbors
  ssize = degree
  tsize = degree-1
  soffset = 1
  if maxlevel > 1:
    logger.debug("root neighbors")
    assert nodes >= 1+degree+degree*(degree-1)
    for source in range(ssize):
      toffset = soffset + ssize + source*tsize
      for target in range(tsize):
        logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
            {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
            toffset + target))
        graph.add_edge(soffset + source,toffset + target)
        nodecount = nodecount + 1
    soffset = 1+degree
    ssize = ssize*(degree-1)
  else:
    logger.debug("skipping root neighbors")

  #further levels
  for level in range(2,maxlevel):
    logger.debug("level: "+str(level))
    for source in range(ssize):
      toffset = soffset + ssize + source*tsize
      for target in range(tsize):
        logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
            {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
            toffset + target))
```

```
        graph.add_edge(soffset + source,toffset + target)
        nodecount = nodecount + 1
    soffset = soffset+degree*pow(degree-1,level-1)
    ssize = ssize*(degree-1)

  #append nodes to leaves
  logger.debug("append nodes to leaves")
  source = 0
  target = 0
  toffset = soffset + ssize + source*tsize
  logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {}".format(
      ssize, soffset, source, toffset, target))
  while toffset + target < nodes:
    if source >= ssize:
      source = source - ssize
    logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
        {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
        toffset + target))
    graph.add_edge(soffset + source,toffset + target)
    nodecount = nodecount + 1
    source = source + 1
    target = target + 1

  assert nodecount == nodes

########## GENERATE EDGES FILE ##########

  #verify feasibility of generated graph
  assert graph.number_of_nodes() == nodes
  assert graph.number_of_edges() <= math.floor((degree*nodes)/2)
  assert max([d for n, d in graph.degree()]) == degree
  assert networkx.is_connected(graph)

  logger.info("generate edges file")
  basename = "AODP-Tree_{:02d}_{:02d}".format(nodes,degree)
  networkx.write_edgelist(graph, basename+".edges", data=False)

########## FUNCTIONS ##########

if __name__ == '__main__':
  main(argumentparser.parse_args())
```

Listing 6: Generator für Baumstruktur-Fixierung (Python-Implementation)

Python: Greedy-Heuristik mit Baum-Fixierung: `TreeGreedy.py`

```
#!/usr/bin/env python3
import argparse
import logging
import sys
import networkx
import math
import numpy

########## BASICS ##########

argumentparser = argparse.ArgumentParser()
argumentparser.add_argument('nodes', type=int)
argumentparser.add_argument('degree', type=int)

logger = logging.getLogger(__name__)
logging.basicConfig(format="%(message)s")
#logger.setLevel("WARNING")
logger.setLevel("INFO")
#logger.setLevel("DEBUG")

def main(args):

########## PARSE PARAMETERS ##########

  nodes = args.nodes
  degree = args.degree

########## HANDLE TRIVIAL CASES ##########

  if nodes < 1:
    sys.exit("trivial solution: no graph")
  if degree < 2:
    sys.exit("trivial solution: no connected graph")
  if nodes == 2:
    sys.exit("trivial solution: 0 -- 1")
  if degree == 2:
    sys.exit("trivial solution: cycle")
  if nodes <= degree+1:
    sys.exit("trivial solution: complete graph")
  assert nodes >= 5

########## PLAN TREE ##########

  assert nodes > degree

  graph = networkx.empty_graph(nodes)
  levelsize = {}
  levelsize[0] = 1
  nodecount = 1
  levelsize[1] = degree
```

```
  maxlevel = 0
  while nodecount+levelsize[maxlevel+1] < nodes:
    nodecount += levelsize[maxlevel+1]
    maxlevel += 1
    levelsize[maxlevel+1] = levelsize[maxlevel] * (degree-1)
  logger.debug("DEBUG: maxlevel {} levelsize {}".format(maxlevel, levelsize))

########## GENERATE TREE ##########

  logger.info("generate tree for fixings")

  #root
  logger.debug("root")
  soffset = 0
  source = 0
  ssize = 1
  tsize = degree
  toffset = soffset + ssize + source*tsize
  for target in range(tsize):
    logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
        {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
        toffset + target))
    graph.add_edge(soffset + source, toffset + target)
  nodecount = 1+degree

  #root neighbors
  ssize = degree
  tsize = degree-1
  soffset = 1
  if maxlevel > 1:
    logger.debug("root neighbors")
    assert nodes >= 1+degree+degree*(degree-1)
    for source in range(ssize):
      toffset = soffset + ssize + source*tsize
      for target in range(tsize):
        logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
            {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
            toffset + target))
        graph.add_edge(soffset + source,toffset + target)
        nodecount = nodecount + 1
    soffset = 1+degree
    ssize = ssize*(degree-1)
  else:
    logger.debug("skipping root neighbors")

  #further levels
  for level in range(2,maxlevel):
    logger.debug("level: "+str(level))
    for source in range(ssize):
      toffset = soffset + ssize + source*tsize
      for target in range(tsize):
        logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
```

```
            {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
            toffset + target))
        graph.add_edge(soffset + source,toffset + target)
        nodecount = nodecount + 1
    soffset = soffset+degree*pow(degree-1,level-1)
    ssize = ssize*(degree-1)

  #append nodes to leaves
  logger.debug("append nodes to leaves")
  source = 0
  target = 0
  toffset = soffset + ssize + source*tsize
  logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {}".format(
      ssize, soffset, source, toffset, target))
  while toffset + target < nodes:
    if source >= ssize:
      source = source - ssize
    logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
        {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
        toffset + target))
    graph.add_edge(soffset + source,toffset + target)
    nodecount = nodecount + 1
    source = source + 1
    target = target + 1

  assert nodecount == nodes

########## ADD GREEDY EDGES ##########

  tree = graph.copy()
  success = False

  #1) straight
  #2) reverse
  #3) straight strict
  #4) reverse strict
  #5+) shuffle
  strict=False
  reverse=False
  shuffle=False

  logger.info("adding greedy edges")

  while not success:
    addedEdge = True
    while addedEdge and graph.number_of_edges() < math.floor((degree*nodes)/2):
      addedEdge = False
      maxdist = -1
      mindegree = min([d for n, d in graph.degree()])
      for source in range(nodes):
        #only choose source with lowest degree
        if graph.degree[source] == mindegree:
```

```
      assert graph.degree[source] < degree
      #enforce degree-1 for last node for odd nodes*degree
      if (nodes*degree % 2 == 1) and (source == nodes-1) and (graph.degree[source]
          == degree-1):
        continue
      targets = {}
      #generate array with available targets
      for target in range(nodes):
        if (source != target) and (graph.degree[target] < degree) and (not graph.
            has_edge(min(source, target), max(source, target))):
          targets[target] = target
      if reverse:
        targets = reversed(targets)
      elif shuffle:
        numpy.random.shuffle(targets)
      for target in targets:
        #enforce degree-1 for last node for odd nodes*degree
        if (nodes*degree % 2 == 1) and (target == nodes-1) and (graph.degree[
            target] == degree-1):
          continue
        #choose target with largest shortest path
        dist = networkx.shortest_path_length(graph, source=source, target=target,
            weight=None)
        if ((dist > maxdist) or (not strict) and (dist >= maxdist)):
          maxsource = source
          maxtarget = target
          maxdist = dist
  if maxdist > 0:
    assert not graph.has_edge(min(maxsource, maxtarget), max(maxsource, maxtarget)
        )
    graph.add_edge(min(maxsource, maxtarget), max(maxsource, maxtarget))
    addedEdge = True
    logger.debug("DEBUG: adding edge {}-{} maxdist {}".format(min(maxsource,
        maxtarget), max(maxsource, maxtarget), maxdist))
  else: #creation failed
    break
if not addedEdge: #creation failed - prepare retry
  warning = "WARN: graph creation failed for {} {}".format(nodes,degree)
  graph = tree.copy()
  if reverse or strict or shuffle: warning = warning + " with"
  if reverse: warning = warning + " reverse"
  if strict: warning = warning + " strict"
  if shuffle: warning = warning + " shuffle"
  warning = warning + " - retrying"
  logger.warning(warning)
  if not shuffle: #5) => 5)
    if strict and reverse: #4) => 5)
      reverse = False
      strict = False
      shuffle = True
    elif reverse: #2) => 3)
      strict = True
```

```
        reverse = False
      else: #if strict: 3) => 4) else: 1) => 2)
        reverse = True
  else:
    success = True

########## GENERATE EDGES FILE ##########

  #verify feasibility of generated graph
  assert graph.number_of_nodes() == nodes
  assert graph.number_of_edges() == math.floor((nodes*degree)/2)
  assert max([d for n, d in graph.degree()]) == degree
  assert networkx.is_connected(graph)

  #for odd nodes*degree: degree-1 for last node
  assert (nodes*degree % 2 == 0) or (graph.degree(nodes-1) == degree-1)

  logger.info("generate edges file")
  basename = "TreeGreedy_{:02d}_{:02d}".format(nodes,degree)
  networkx.write_edgelist(graph, basename+".edges", data=False)

########## FUNCTIONS ##########

if __name__ == '__main__':
  main(argumentparser.parse_args())
```

Listing 7: Greedy-Heuristik auf Baumstruktur (Python-Implementation)

Python: Zufalls-Greedy-Heuristik mit Baum-Fixierung: TreeGreedyRandom.py

```
#!/usr/bin/env python3
import argparse
import logging
import sys
import networkx
import math
import numpy

########## BASICS ##########

argumentparser = argparse.ArgumentParser()
argumentparser.add_argument('nodes', type=int)
argumentparser.add_argument('degree', type=int)

logger = logging.getLogger(__name__)
logging.basicConfig(format="%(message)s")
logger.setLevel("INFO")
#logger.setLevel("DEBUG")
```

```
def main(args):

########## PARSE PARAMETERS ##########

  nodes = args.nodes
  degree = args.degree

########## HANDLE TRIVIAL CASES ##########

  if nodes < 1:
    sys.exit("trivial solution: no graph")
  if degree < 2:
    sys.exit("trivial solution: no connected graph")
  if nodes == 2:
    sys.exit("trivial solution: 0 -- 1")
  if degree == 2:
    sys.exit("trivial solution: cycle")
  if nodes <= degree+1:
    sys.exit("trivial solution: complete graph")
  assert nodes >= 5

########## PLAN TREE ##########

  assert nodes > degree

  graph = networkx.empty_graph(nodes)
  levelsize = {}
  levelsize[0] = 1
  nodecount = 1
  levelsize[1] = degree
  maxlevel = 0
  while nodecount+levelsize[maxlevel+1] < nodes:
    nodecount += levelsize[maxlevel+1]
    maxlevel += 1
    levelsize[maxlevel+1] = levelsize[maxlevel] * (degree-1)
  logger.debug("DEBUG: maxlevel {} levelsize {}".format(maxlevel, levelsize))

########## GENERATE TREE ##########

  logger.info("generate tree for fixings")

  #root
  logger.debug("root")
  soffset = 0
  source = 0
  ssize = 1
  tsize = degree
  toffset = soffset + ssize + source*tsize
  for target in range(tsize):
    logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
        {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
```

```
      toffset + target))
  graph.add_edge(soffset + source, toffset + target)
nodecount = 1+degree

#root neighbors
ssize = degree
tsize = degree-1
soffset = 1
if maxlevel > 1:
  logger.debug("root neighbors")
  assert nodes >= 1+degree+degree*(degree-1)
  for source in range(ssize):
    toffset = soffset + ssize + source*tsize
    for target in range(tsize):
      logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
          {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
          toffset + target))
      graph.add_edge(soffset + source,toffset + target)
      nodecount = nodecount + 1
  soffset = 1+degree
  ssize = ssize*(degree-1)
else:
  logger.debug("skipping root neighbors")

#further levels
for level in range(2,maxlevel):
  logger.debug("level: "+str(level))
  for source in range(ssize):
    toffset = soffset + ssize + source*tsize
    for target in range(tsize):
      logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
          {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
          toffset + target))
      graph.add_edge(soffset + source,toffset + target)
      nodecount = nodecount + 1
  soffset = soffset+degree*pow(degree-1,level-1)
  ssize = ssize*(degree-1)

#append nodes to leaves
logger.debug("append")
source = 0
target = 0
toffset = soffset + ssize + source*tsize
logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {}".format(
    ssize, soffset, source, toffset, target))
while toffset + target < nodes:
  if source >= ssize:
    source = source - ssize
  logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
      {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
      toffset + target))
  graph.add_edge(soffset + source,toffset + target)
```

```
    nodecount = nodecount + 1
    source = source + 1
    target = target + 1

  assert nodecount == nodes

########## ADD GREEDY EDGES ##########

  tree = graph.copy()
  logger.info("adding greedy edges")
  while graph.number_of_edges() < math.floor((degree*nodes)/2):
    addedEdge = True
    while addedEdge and graph.number_of_edges() < math.floor((degree*nodes)/2):
      addedEdge = False
      mindegree = min([d for n, d in graph.degree()])
      sources = {} #only use sources with minimum degree
      targets = {} #only use targets with minimum degree + 1 (and < degree)
      nsources = 0
      ntargets = 0
      candidate = 0
      while candidate < nodes:
        if graph.degree[candidate] == mindegree:
          sources[nsources] = candidate
          nsources = nsources + 1
        if graph.degree[candidate] <= mindegree+1 and graph.degree[candidate] < degree
            :
          targets[ntargets] = candidate
          ntargets = ntargets + 1
        candidate = candidate + 1
      numpy.random.shuffle(sources)
      numpy.random.shuffle(targets)
      source = 0
      target = ntargets - 1
      while not addedEdge and source < nsources and target >= 0:
        while target >= 0 and (sources[source] == targets[target] or graph.has_edge(
            min(sources[source], targets[target]), max(sources[source], targets[target
            ]))):
          target = target - 1
        if target >= 0:
          assert not graph.has_edge(min(sources[source], targets[target]),max(sources[
              source], targets[target]))
          logger.debug("DEBUG: adding edge {}-{} with degrees {} & {}".format(min(
              sources[source], targets[target]),max(sources[source], targets[target]),
              graph.degree[min(sources[source], targets[target])],graph.degree[max(
              sources[source], targets[target])]))
          graph.add_edge(min(sources[source], targets[target]),max(sources[source],
              targets[target]))
          addedEdge = True
        else:
          source = source + 1
          target = ntargets - 1
    if graph.number_of_edges() < math.floor((degree*nodes)/2):
```

```
      logger.info("try failed - retrying")
      graph = tree.copy()

########## GENERATE EDGES FILE ##########

  #verify feasibility of generated graph
  assert graph.number_of_nodes() == nodes
  assert graph.number_of_edges() == math.floor((degree*nodes)/2)
  assert max([d for n, d in graph.degree()]) == degree
  assert networkx.is_connected(graph)

  logger.info("success - generate edges file")
  basename = "TreeGreedyRandom_{:02d}-{:02d}".format(nodes,degree)
  networkx.write_edgelist(graph, basename+".edges", data=False)
  return

########## FUNCTIONS ##########

if __name__ == '__main__':
  main(argumentparser.parse_args())
```

Listing 8: Zufalls-Greedy-Heuristik auf Baumstruktur (Python-Implementation)

Python: Zufalls-Heuristik mit Baum-Fixierung:

TreeRandomSloppy.py

```
#!/usr/bin/env python3
import argparse
import logging
import sys
import networkx
import math
import random

########## BASICS ##########

argumentparser = argparse.ArgumentParser()
argumentparser.add_argument('nodes', type=int)
argumentparser.add_argument('degree', type=int)

logger = logging.getLogger(__name__)
logging.basicConfig(format="%(message)s")
logger.setLevel("INFO")
#logger.setLevel("DEBUG")

def main(args):

########## PARSE PARAMETERS ##########
```

```
  nodes = args.nodes
  degree = args.degree

########## HANDLE TRIVIAL CASES ##########

  if nodes < 1:
    sys.exit("trivial solution: no graph")
  if degree < 2:
    sys.exit("trivial solution: no connected graph")
  if nodes == 2:
    sys.exit("trivial solution: 0 -- 1")
  if degree == 2:
    sys.exit("trivial solution: cycle")
  if nodes <= degree+1:
    sys.exit("trivial solution: complete graph")
  assert nodes >= 5

########## PLAN TREE ##########

  assert nodes > degree

  graph = networkx.empty_graph(nodes)
  levelsize = {}
  levelsize[0] = 1
  nodecount = 1
  levelsize[1] = degree
  maxlevel = 0
  while nodecount+levelsize[maxlevel+1] < nodes:
    nodecount += levelsize[maxlevel+1]
    maxlevel += 1
    levelsize[maxlevel+1] = levelsize[maxlevel] * (degree-1)
  logger.debug("DEBUG: maxlevel {} levelsize {}".format(maxlevel, levelsize))

########## GENERATE TREE ##########

  logger.info("generate tree for fixings")

  #root
  logger.debug("root")
  soffset = 0
  source = 0
  ssize = 1
  tsize = degree
  toffset = soffset + ssize + source*tsize
  for target in range(tsize):
    logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
        {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
        toffset + target))
    graph.add_edge(soffset + source, toffset + target)
  nodecount = 1+degree

  #root neighbors
```

```
ssize = degree
tsize = degree-1
soffset = 1
if maxlevel > 1:
  logger.debug("root neighbors")
  assert nodes >= 1+degree+degree*(degree-1)
  for source in range(ssize):
    toffset = soffset + ssize + source*tsize
    for target in range(tsize):
      logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
          {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
          toffset + target))
      graph.add_edge(soffset + source,toffset + target)
      nodecount = nodecount + 1
  soffset = 1+degree
  ssize = ssize*(degree-1)
else:
  logger.debug("skipping root neighbors")

#further levels
for level in range(2,maxlevel):
  logger.debug("level: "+str(level))
  for source in range(ssize):
    toffset = soffset + ssize + source*tsize
    for target in range(tsize):
      logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
          {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
          toffset + target))
      graph.add_edge(soffset + source,toffset + target)
      nodecount = nodecount + 1
  soffset = soffset+degree*pow(degree-1,level-1)
  ssize = ssize*(degree-1)

#append nodes to leaves
logger.debug("append")
source = 0
target = 0
toffset = soffset + ssize + source*tsize
logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {}".format(
    ssize, soffset, source, toffset, target))
while toffset + target < nodes:
  if source >= ssize:
    source = source - ssize
  logger.debug("DEBUG: ssize {} soffset {} source {} toffset {} target {} edge
      {}-{}".format(ssize, soffset, source, toffset, target, soffset + source,
      toffset + target))
  graph.add_edge(soffset + source,toffset + target)
  nodecount = nodecount + 1
  source = source + 1
  target = target + 1

assert nodecount == nodes
```

```
########## ADD RANDOM EDGES ##########

  logger.info("adding random edges")

  full = False
  while not full:
    source = soffset
    full = True
    while source < nodes:
      target = random.randint(soffset, nodes-1)
      if (source != target) and (not graph.has_edge(min(source, target),max(source,
          target))) and (graph.degree[source] < degree) and (graph.degree[target] <
          degree):
        full = False
        graph.add_edge(min(source, target),max(source, target))
        logger.debug("DEBUG: adding edge {}-{}".format(min(source, target),max(source,
            target)))
      source = source + 1

########## GENERATE EDGES FILE ##########

  #verify feasibility of generated graph
  assert graph.number_of_nodes() == nodes
  assert graph.number_of_edges() <= math.floor((degree*nodes)/2)
  assert max([d for n, d in graph.degree()]) == degree
  assert networkx.is_connected(graph)

  logger.info("success - generate edges file")
  basename = "TreeRandomSloppy_{:02d}-{:02d}".format(nodes,degree)
  networkx.write_edgelist(graph, basename+".edges", data=False)
  return

########## FUNCTIONS ##########

if __name__ == '__main__':
  main(argumentparser.parse_args())
```

Listing 9: Zufalls-Heuristik auf Baumstruktur (Python-Implementation)