Karlsruhe Institute of Technology

Heuristiken zum Organisieren von Sitzungen

Bachelorarbeit
von

Lukas Hennig

An der Fakultät für Informatik
Institut für Theoretische Informatik

Erstgutachter:	Prof. Dr. Dorothea Wagner
Zweitgutachter:	Prof. Dr. Peter Sanders
Betreuer:	Ben Strasser

Bearbeitungszeit: 1.12.2016 – 31.3.2017

Selbstständigkeitserklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) habe ich befolgt.

Karlsruhe, 31.3.2017

. . . . . .

(**Lukas Hennig**)

Abstract

Nowadays this is probably a well-known situation to many people: You already work together with your colleagues on various topics, but there is constantly someone coming up with an idea for another exciting new project. Unfortunately, there is never enough time to work on all of those. Therefore, one has to make a selection.
This thesis presents the Meetingsproblem: There is a set of people, a set of potential meetings and a limited number of available time slots. For simplicity, we assume that every meeting only takes one time slot. Each meeting has a set of participants and can only take place if all participants are available. Furthermore the meetings have weights associated with them. Every person can only participate at one meeting per time slot to not overburden them. We are asking for a plan which tells us for every person when and at which meetings he is participating in.
First we examine how well this problem can be solved optimally by integer linear programming. Subsequently we develop scalable heuristics and evalute those on randomly generated instances. We tested various greedy heuristics, a heuristic which repeatedly solves smaller subproblems with less time slots, and a heuristic based on local search which can improve a given solution.
We state that a simple and fast greedy heuristic can already provide good results. With some additional processing we further imprvove those results with the local search heuristic.

Deutsche Zusammenfassung

Heutzutage vermutlich vielen Leuten eine bekannte Situation: Man arbeitet gemeinsam mit seinen Kollegen bereits an diversen Themen, aber ständig kommt jemand mit einer Idee für ein weiteres tolles neues Projekt. Leider ist die verfügbare Zeit viel zu knapp, um alle Ideen umzusetzen. Es muss daher eine Auswahl getroffen werden.
In dieser Arbeit wird dieses Problem als das Meetingsproblem vorgestellt: Es gibt eine Menge von Personen, eine Menge potentieller Treffen und eine beschränkte Anzahl verfügbarer Zeitslots. Der Einfachheit halber wird angenommen, dass jedes Treffen genau ein Zeitslot lang ist. Jedes Treffen hat eine Menge von Teilnehmern und kann nur stattfinden, wenn alle Teilnehmer anwesend sind. Außerdem liegt eine Gewichtung der Treffen vor. Um niemanden zu überlasten, kann jede Person an höchstens einem Treffen pro Zeitslot teilnehmen. Gesucht ist ein Plan, der für jede Person festlegt, wann und an welchen Treffen diese Person teilnimmt. Das Ziel besteht darin, die Summe der Gewichte über alle stattfindenden Treffen zu maximieren.
Zunächst wird untersucht, wie gut sich dieses Problem mittels ganzzahliger linearer Programmierung optimal lösen lässt. Anschließend werden skalierbare Heuristiken entworfen und experimentell anhand zufällig generierter Probleminstanzen ausgewertet. Getestet wurden verschiedene Greedy-Heuristiken, eine Heuristik, die das Problem schrittweise für Teilmengen der Zeitslots löst, und eine Heuristik basierend auf lokaler Suche, mit der sich eine gegebene Lösung verbessern lässt.
Es wurde festgestellt, dass bereits eine einfache und schnelle Greedy-Heuristik gute Ergebnisse erzielen kann. Mit etwas Mehraufwand werden diese Lösungen mit lokaler Suche noch verbessert.

Inhaltsverzeichnis

1. Einleitung

Zeit ist kostbar. Man hat nie genug Zeit für all die Dinge, die man erledigen will, oder zu erledigen hat. Ein Termin folgt dem anderen, eine Besprechung der nächsten. Man trifft sich mit Kunden und vereinbart direkt weitere Termine. Die nächste Woche ist bereits durchgeplant. Und ständig kommen Anfragen für neue Projekte dazu. Aber alle Kollegen sind bereits anderweitig ausgelastet. Du hast eine tolle Idee, kannst sie aber nicht alleine umsetzen? Schreib sie auf die To-Do-Liste, zu den 100 anderen Sachen.
Dabei sollte dem modernen Menschen doch mehr Zeit als noch nie zuvor zur Verfügung stehen. Neue Technik soll vieles erleichtern und Zeit sparen. Die Bewegung von A nach B wird immer schneller. Wer heute etwas im Internet bestellt, der kann damit rechnen, es bereits am nächsten Tag in der Hand zu halten. Spontan eine Telefonkonferenz mit jemandem am anderen Ende der Welt führen? Heutzutage kein Problem.

„Wir haben nicht zu wenig Zeit, sondern zu viel zu tun.“[DPDK17]

Die Lösung? Zeitmanagement. Wichtige Tätigkeiten werden priorisiert, andere verworfen.

Diese Arbeit betrachtet das dargestellte Problem in vereinfachter Form. Es gibt eine Menge von Personen, eine Menge potentieller Treffen und eine beschränkte Anzahl verfügbarer Zeitslots. Der Einfachheit halber wird angenommen, dass jedes Treffen genau ein Zeitslot lang ist. Jedes Treffen hat eine Menge von Teilnehmern und kann nur stattfinden, wenn alle Teilnehmer anwesend sind. Außerdem liegt eine Gewichtung der Treffen vor. Um niemanden zu überlasten, kann jede Person an höchstens einem Treffen pro Zeitslot teilnehmen. Gesucht ist ein Plan, der für jede Person festlegt, wann und an welchen Treffen diese Person teilnimmt. Das Ziel besteht darin, die Summe der Gewichte über alle stattfindenden Treffen zu maximieren.

In Kapitel 2 werden zunächst grundlegende Begriffe definiert und das beschriebene Problem formalisiert. Zur verständlichen und kompakten Darstellung werden verschiedene graphentheoretische Repräsentationen vorgestellt. Außerdem wird die Komplexität des Problems analysiert.

Als erstes wird untersucht, wie groß die Probleminstanzen werden können, damit sie sich noch in vertretbarer Zeit mittels ganzzahliger linearer Programmierung optimal lösen lassen. Dabei kommt der kommerzielle Gurobi Löser [GO17a] zum Einsatz. Die Vorgehensweise dazu wird in Kapitel 3.1.2 beschrieben.

Die Laufzeiten für diesen Ansatz schießen sehr schnell in die Höhe. Daher werden in Kapitel 3 auch verschiedene heuristische Ansätze vorgestellt. Darunter sind eine Reihe von einfachen

Greedy-Algorithmen, ein Algorithmus, der schrittweise kleinere Teilprobleme mit weniger Zeitslots löst, und ein Algorithmus, der basierend auf lokaler Suche eine gegebene Startlösung verbessern kann. Die entwickelten Ansätze werden anschließend experimentell evaluiert und bezüglich Laufzeit und Güte der Lösungen ausgewertet. Die Ergebnisse dazu finden sich in Kapitel 4.

Da keine Realwertdaten zur Verfügung standen, wurde das Problem hier sehr allgemein betrachtet. Die vorgestellten Algorithmen werden auf zuvor zufällig generierten Probleminstanzen getestet. Die Erstellung der Testdaten wird in Kapitel 4.2.1 beschrieben. Alle Algorithmen wurden in Java implementiert.

1.1. Verwandte Arbeiten

Das Problem kann als Hypergraph dargestellt werden (siehe Kapitel 2): Die Personen werden zu Knoten und die Treffen zu Hyperkanten zwischen den Knoten der teilnehmenden Personen. Die Auswahl einer Teilmenge von Treffen ohne gemeinsame Teilnehmer mit maximalem Gewicht entspricht dann gerade dem Finden eines Matchings mit maximalem Gewicht. Und die Zuordnung der Treffen zu k Zeitslots entspricht einer Färbung der Kanten mit maximal k Farben. Im allgemeinen Fall ist das Problem NP-schwer.

In [KRSW11] und [DW75] werden nur Treffen zwischen jeweils zwei Gruppen von Personen betrachtet. Das Problem kann dann als bipartiter Graph dargestellt werden und in Polynomialzeit gelöst werden. Und in [CR09] wird eine Approximation für das Finden einer maximal gewichtigen k-Färbung für triangulierte Graphen vorgestellt.

Das hier betrachtete Problem tritt in etwas anderen Formen auch in ähnlichen Kontexten auf. Für das Examination-Scheduling-Problem in [MHH06] und [CT03] gibt es zum Beispiel eine Menge von Studenten, die an diversen Kursen teilnehmen. Die Kurse sollen so auf Zeitslots verteilt werden, dass kein Student an mehr als einem Kurs pro Zeitslot teilnimmt. Die Anzahl an verwendeten Zeitslots soll jetzt aber hier möglichst klein sein. Und häufig muss eine Lösung noch weitere Eigenschaften erfüllen. So kann es zum Beispiel Limits geben für die maximale Anzahl an gleichzeitig stattfindenden Kursen, die minimale und maximale Anzahl an Kursen, die ein Student pro Tag haben darf, oder wie groß die zeitlichen Abstände zwischen Kursen für einen Student sein dürfen.

Für genau einen Zeitslot entspricht das Problem gerade dem Finden einer maximalen Clique oder unabhängigen Menge (siehe Kapitel 3.3). In [Pel09] werden Heuristiken für maximale Cliquen und unabhängige Mengen vorgestellt.

2. Grundlagen

In diesem Kapitel werden die verwendeten Begriffe und Notationen definiert. In Abschnitt 2.2 wird das betrachtete Problem formalisiert und graphentheoretische Darstellungen vorgestellt. Die NP-Schwere des Problems wird in Abschnitt 2.3 nachgewiesen.

2.1. Begrifflichkeiten

Definition 2.1. *Ein* einfacher Graph *ist ein ungerichteter Graph ohne Mehrfachkanten und ohne Schleifen.*

Definition 2.2. *Der* Komplementgraph *eines Graphen* $G = (V, E)$ *ist* $\bar{G} = (V, \bar{E})$ *mit* $\bar{E} \coloneqq \{e \mid e \notin E\}$

Definition 2.3. *Ein* Hypergraph *ist ein Graph, in dem Kanten auch mehr als zwei Knoten miteinander verbinden können. Die Kanten werden dann auch* Hyperkanten *genannt.*

Definition 2.4. *Eine* unabhängige Menge *ist eine Teilmenge von Knoten eines Graphen, in der keine zwei Knoten benachbart sind.*

Definition 2.5. *Eine* Clique *ist eine Teilmenge von Knoten eines Graphen, in der alle Knotenpaare mit einer Kante verbunden sind.*

Definition 2.6. *Eine* k-Kantenfärbung *ordnet jeder Kante eines Graphen eine Farbe zu, sodass keine zwei benachbarten Kanten die gleiche Farbe haben. Es können höchstens k Farben verwendet werden.*

Definition 2.7. *Eine* k-Paarung *auf einem Graphen wählt eine Teilmenge von Knoten, so dass jeder Knoten mit maximal k anderen Knoten in dieser Menge benachbart ist.*

2.2. Problemstellung

Meetingsproblem

Gegeben sind eine Menge P von m Personen, eine Menge T von n gewichteten Teilmengen (Treffen) T_i von P mit ganzzahligen Gewichten $w_i > 0$ und eine maximale Anzahl an Zeitslots t. Jedes Treffen ist genau ein Zeitslot lang. Ein Treffen kann höchstens einmal stattfinden und nur dann, wenn alle Teilnehmer anwesend sind. Keine Person kann an mehr als einem Treffen pro Zeitslot teilnehmen.

Gesucht ist eine Auswahl von Treffen, sowie eine gültige Zuweisung dieser zu den verfügbaren Zeitslots, sodass die Summe ihrer Gewichte maximal wird. Formal ist also eine Menge R von t disjunkten Teilmengen R_k von T gesucht, sodass innerhalb einer Menge R_k keine Person in mehr als einer Menge T_i vorkommt. Die Summe der Gewichte aller Treffen T_i in allen R_k in R soll maximiert werden.

$T(R)$ bezeichnet die Menge aller Treffen innerhalb einer Lösung R, also $T(R) \coloneqq \bigcup_{R_k \in R} R_k$.

$w(T)$ bezeichnet das Gesamtgewicht aller Treffen in einer Menge T, also $w(T) \coloneqq \sum_{T_i \in T} w_i$. Unter anderem bezeichnet daher $w(T(G))$ das Gesamtgewicht, oder auch den *Wert*, einer Lösung R.

Wegen der gelegentlichen Verwendung des Subskripts für die Bezeichnung von Variablen kann anstatt T_i und R_k auch die Notation $T[i]$, beziehungsweise $R[k]$, verwendet werden, um ein Element in T, beziehungsweise in R, zu bezeichnen.

Definition 2.8. *Zwei Treffen stehen genau dann im* Konflikt*, wenn sie mindestens einen Teilnehmer gemeinsam haben. Diese Treffen können also nicht im gleichen Zeitslot stattfinden.*

Definition 2.9. *Eine Menge R ist eine* zulässige *(oder auch* gültige*) Lösung, genau dann, wenn folgende Eigenschaften erfüllt sind:*

- $|R| \leq t$.
- *Die Elemente R_k von R sind paarweise disjunkte Teilmengen von T.*
- $\forall p_j \in P$, $\forall R_k \in R$ *: Person p_j ist in höchstens einer Menge $T_i \in R_k$.*

Graphentheoretische Darstellung

Das obige Problem lässt sich auch als folgendes graphentheoretisches Problem darstellen:

Definition 2.10. Hypergraphrepräsentation
Jede Person entspricht einem Knoten. Jedes Treffen entspricht einer gewichteten Hyperkante zwischen den teilnehmenden Personen.

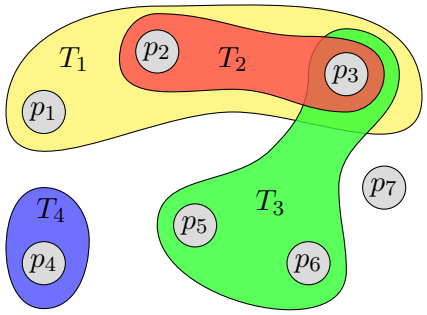


Abbildung 2.1.: Beispiel Hypergraph (basierend auf [WC10])

Auf diesem Hypergraphen ist dann eine maximal-gewichtige k-Paarung mit k-Kantenfärbung gesucht. Die k-Paarung stellt sicher, dass jeder Knoten maximal Grad k hat, also jede Person insgesamt an maximal k Treffen teilnimmt. Eine k-Kantenfärbung auf diesem Teilgraphen ordnet jedem der Treffen einen Zeitslot zu.

Eine reduzierte Darstellung, bei der Personen und ihre Teilnahmen an Treffen auf die dadurch entstehenden Konflikte reduziert werden, ist die des *Konfliktgraphen*.

Definition 2.11. Konfliktgraph
Jedes Treffen entspricht einem gewichteten Knoten. Zwischen zwei Knoten gibt es genau dann eine ungerichtete Kante, falls die entsprechenden Treffen im Konflikt stehen.

In diesem Konfliktgraphen sind dann t paarweise disjunkte unabhängige Mengen gesucht, deren Knotengewichte in der Summe maximal sind.

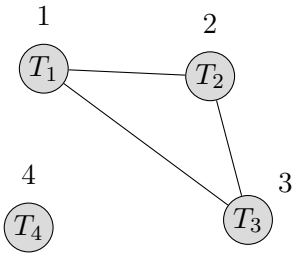


Abbildung 2.2.: Beispiel Konfliktgraph

Alternativ kann auch der dazu komplementäre Graph betrachtet werden.

Definition 2.12. Komplementärer Konfliktgraph
Jedes Treffen entspricht einem gewichteten Knoten. Zwischen zwei Knoten gibt es genau dann eine ungerichtete Kante, falls die entsprechenden Treffen nicht im Konflikt stehen.

Unabhängige Mengen entsprechen Cliquen im Komplementgraphen und andersherum. Daher werden auf diesem Graphen dann k paarweise disjunkte Cliquen gesucht, deren Knotengewichte in der Summe maximal sind.

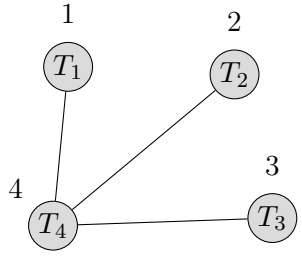


Abbildung 2.3.: Beispiel komplementärer Konfliktgraph

Größe des Lösungsraum

Wenn man nur t Treffen auswählt, dann kann man diese trivial ohne Konflikt den t Zeitslots zuordnen. Bei n Treffen ergeben sich dafür $\binom{n}{k}$ viele Möglichkeiten. Wenn sowohl die Anzahl an Zeitslots als auch die Anzahl an Treffen zunimmt, dann wächst bereits dieser Lösungsteilraum exponentiell und damit auch der gesamte Lösungsraum.

2.3. Komplexitätsanalyse

Komplexitätsklassen

Die Komplexitätsklasse P enthält alle Entscheidungsprobleme, für die es einen deterministischen Algorithmus gibt, der das Problem in polynomieller Laufzeit bezüglich der Eingabegröße löst. Die Komplexitätsklasse NP enthält alle Entscheidungsprobleme, für die es eine nicht-deterministische Turingmaschine gibt, die das Problem in polynomieller Zeit bezüglich der Eingabegröße löst. [Kar72]

Ein Problem heißt NP-schwer, wenn sich sämtliche Probleme aus NP in Polynomialzeit auf dieses Problem reduzieren lassen [Knu74]. Ein Algorithmus, der ein NP-schweres Problems löst, kann über diese Reduktion auch sämtliche Probleme in NP lösen. NP-schwere Probleme sind also mindestens so schwer zu lösen, wie die Probleme in NP. Vermutlich lassen sich diese Probleme im Allgemeinen nicht effizient lösen.

Probleme, die sowohl NP-schwer als auch in NP sind, heißen NP-vollständig [Kar72]. Sie sind damit die schwersten Probleme innerhalb von NP.

NP-Schwere Beweis

Im Folgenden wird die NP-Schwere des vorgestellten Problems durch Polynomialzeitreduktion des NP-vollständigen Clique Entscheidungsproblems gezeigt. Clique ist eines der 21 ursprünglichen Probleme, die von Richard Karp 1972 als NP-vollständig nachgewiesen wurden [Kar72]. Mit der Polynomialzeitreduktion von Clique folgt, dass sich auch sämtliche anderen Probleme in NP in Polynomialzeit auf das hier betrachtete Problem reduzieren lassen. Damit folgt dann die NP-Schwere des betrachteten Problems.

Definition 2.13. Clique Entscheidungsproblem
Gegeben ist ein einfacher Graph $G = (V, E)$ und ein Parameter k. Gefragt ist nun, ob es eine Clique mit mindestens k Knoten gibt.

Polynomialzeitreduktion

Eine Instanz von Clique lässt sich in Polynomialzeit in eine Instanz des vorgestellten Problems überführen.

Beweis. Gegeben ist eine Instanz von Clique mit Graph $G = (V, E)$ mit $V = \{v_1, v_2, \ldots, v_n\}$ und Parameter k. Wir suchen im Folgenden eine größte Clique C im Graphen G, um damit das Clique-Problem zu entscheiden.
Es folgt die Transformation in eine Instanz von Meetingsproblem:
Anzahl Zeitslots $t = 1$
Treffen T: Jeder Knoten v_i in V ergibt ein Treffen T_i in T mit Gewicht $w_i = 1$.
Die Erzeugung der Personen und Zuweisung der Teilnehmer lässt sich am einfachsten mit diesem Pseudocode darstellen:

```
   // Personenmenge ist zunächst leer
1  P = ∅
2  for v_i ∈ V do
3      T_i := Treffen für Knoten v_i
4      for v_j ∈ V do
5          T_j := Treffen für Knoten v_j
6          if Kante (v_i, v_j) ∉ E then
7              P_k := ErzeugeNeuePerson()
               // Füge neue Person in P ein
8              P := P ∪ P_k
               // Füge neue Person als Teilnehmer zu beiden Treffen hinzu
9              T_i := T_i ∪ P_k
10             T_j := T_j ∪ P_k
```

Das Ergebnis ist eine Instanz I von Meetingsproblem. Zwei Treffen T_i und T_j haben genau dann eine gemeinsame Person als Teilnehmer, und damit einen Konflikt, falls die

entsprechenden Knoten v_i und v_j keine gemeinsame Kante in E haben.
Und damit andersrum: Zwei Treffen in I haben genau dann *keinen Konflikt*, falls die entsprechenden Knoten eine *gemeinsame Kante* haben.

Eine Lösung des MEETINGSPROBLEMS I liefert einen Zeitslot R_1 mit maximaler Anzahl von Treffen, die keine Konflikte untereinander haben.
Rücktransformation: Clique $C \coloneqq \{v_i \mid T_i \in R_1\}$

Beweis, dass C eine Clique ist: Die Treffen in R_1 sind im gleichen Zeitslot und können daher keine Konflikte untereinander haben, also paarweise keine gemeinsamen Teilnehmer. Also haben die entsprechenden Knoten in G paarweise eine gemeinsame Kante. Das entspricht der Definition von Clique.

Beweis, dass C maximal ist: Da alle Treffen Gewicht 1 haben, entspricht eine maximal gewichtige Lösung des MEETINGSPROBLEMS einer maximale Anzahl an Treffen. Damit hat auch die aus diesen Treffen abgeleitete Clique C eine maximale Anzahl an Knoten.

Mit der maximalen Clique C lässte sich jetzt das CLIQUE-Problem entscheiden: Falls $|C| \geq k$, dann wird für das CLIQUE-Problem Ja zurück gegeben, ansonsten $Nein$.

Der Aufwand dieser Reduktion liegt in $\mathcal{O}(n^2)$ mit $n =$ Anzahl Knoten in G. □

Folgerung

Mit dem Nachweis der NP-Schwere des betrachteten Problems folgt, dass es sich vermutlich im Allgemeinen, also auf beliebigen Probleminstanzen, nicht effizient optimal lösen lässt. Wir werden daher in dieser Arbeit diverse Heuristiken untersuchen.

3. Algorithmen

In diesem Kapitel werden die betrachteten Lösungsansätze und Algorithmen vorgestellt.

3.1. Ganzzahlige lineare Programmierung

Um die Güte und Laufzeit der betrachteten Heuristiken einschätzen zu können, soll zunächst untersucht werden, wie gut sich das Problem optimal lösen lässt.

Das betrachtete Problem lässt sich im Bereich der kombinatorischen Optimierung [Dom15, S. 128] einordnen: Es soll eine Teilmenge an Treffen ausgewählt und einer begrenzten Anzahl an Zeitslots zugeordnet werden, ohne dass dabei bestimmte Bedingungen verletzt werden. Die Anzahl an Auswahl- und Zuordnungsmöglichkeiten ist offensichtlich beschränkt und damit auch der Lösungsraum. Mit steigender Anzahl an Treffen und Zeitslots wächst dieser Lösungsraum aber exponentiell (siehe 2.2). Es wird daher schnell unmöglich, eine optimale Lösung zu finden, indem man alle Elemente des Lösungsraums auflistet.

Ein gängiger Ansatz ist es, ein solches Problem als ganzzahliges lineares Problem [Dom15, S. 128] zu formulieren. Dazu werden ganzzahlige Variablen definiert, mit denen sich das Problem und die möglichen Lösungen darstellen lassen. Mit diesen Variablen wird eine Zielfunktion aufgestellt, die den Wert einer potentiellen Lösung berechnet. Der Lösungsraum wird durch eine Menge von Nebenbedingungen in Form von linearen Ungleichungen eingeschränkt. Ziel ist es dann, eine Lösung zu finden, die, bei Einhaltung der so definierten Nebenbedingungen, die Zielfunktion maximiert.

Für Probleme in dieser Form gibt es eine Reihe von exakten Lösungsverfahren, wie zum Beispiel Branch-and-Bound, Schnittebenenverfahren, oder eine Kombination dieser Verfahren (Branch-and-Cut) [Dom15, S. 134]. Diese können unter Umständen während der Suche, ohne Auswirkung auf die gefundene optimale Lösung, Teile des Lösungsraums ausschließen und so die Suche beschleunigen.

3.1.1. Ganzzahliges lineares Programm

Das betrachtete Problem lässt sich wie folgt als ganzzahliges lineares Programm formulieren:

$$
\begin{aligned}
m &= \text{Anzahl Personen} \\
n &= \text{Anzahl Treffen} \\
t &= \text{Anzahl Zeitslots} \\
w_i &= \text{Gewicht von Treffen } i \\
z_{ij} &= \begin{cases} 1 & \text{Person } j \text{ nimmt an Treffen } i \text{ teil} \\ 0 & \text{sonst} \end{cases}
\end{aligned}
$$

Entscheidungsvariablen:

$$
x_{ki} = \begin{cases} 1 & \text{Treffen } i \text{ findet in Zeitslot } k \text{ statt} \\ 0 & \text{sonst} \end{cases}
$$

$$
\begin{aligned}
\text{Maximiere } & \sum_{k=1}^{t} \sum_{i=1}^{n} x_{ki} w_i \\
\text{unter den Nebenbedingungen } & \sum_{i=1}^{n} x_{ki} z_{ij} \leq 1, \forall j = 1, \ldots, m, \forall k = 1, \ldots, t \quad \textbf{(1)} \\
& \sum_{k=1}^{t} x_{ki} \leq 1 \quad , \forall i = 1, \ldots, n \quad \textbf{(2)} \\
& x_{ki} \in \{0, 1\} \quad , \forall k = 1, \ldots, t, \forall i = 1, \ldots, n
\end{aligned}
$$

(1) Jede Person kann pro Zeitslot an maximal einem Treffen teilnehmen. Unter der Annahme, dass alle Personen grundsätzlich in jedem Zeitslot verfügbar sind, ist damit implizit garantiert, dass alle Personen für ein stattfindendes Treffen anwesend sind.
(2) Treffen können höchstens einmal stattfinden.

Für dieses ganzzahlige lineare Programm wird eine Belegung der Entscheidungsvariablen x_{ki} gesucht, die einen möglichst großen Zielfunktionswert erreicht und gleichzeitig die Nebenbedingungen erfüllt. Alle anderen Variablen sind durch eine gegebene Probleminstanz festgesetzt.

3.1.2. Gurobi

Da viele Probleme in der Praxis so modelliert werden (Beispiele finden sich in [Dom15, Kap. 6, S. 127]), gibt es Softwarepakete, die sich auf das Lösen von Problemen in dieser Form spezialisiert haben. In dieser Arbeit wird der kommerzielle Gurobi Löser [GO17a] verwendet. Gurobi wirbt mit guter Performance, auch bei Benutzung der Standardparametern, und einfacher Verwendung dank Schnittstellen für viele gängige Programmiersprachen. Außerdem stehen akademische Lizenzen frei zur Verfügung.

Das ganzzahlige lineare Programm aus Abschnitt 3.1.1 wurde mittels der von Gurobi bereitgestellten Java Schnittstelle implementiert.

Beim optimalen Lösen des Problems mit Gurobi interessiert vor allem, wie sich die Laufzeiten in Abhängigkeit von der Größe der Probleminstanzen entwickeln. Ab wann sind die Laufzeiten also so groß, dass es sinnvoller wird, stattdessen lieber eine Heuristik zu verwenden. Außerdem können die in diesem Schritt erhaltenen optimalen Lösungen später genutzt werden, um die Güte der Heuristiken für diese Probleminstanzen zu bewerten.

3.1.3. Varianten

Gurobi stellt eine ganze Reihe optionaler Parameter zur Verfügung, die es prinzipiell ermöglichen, die Ausführung an das Problem anzupassen. Der Einfachheit halber wurden aber die meisten Parameter auf ihren Standardwerten belassen. In dieser Arbeit wurden allerdings verschiedene Ausführungsvarianten des Gurobi-Lösers verglichen, für die einzelne Parameter modifiziert werden mussten. Diese Ausführungsvarianten werden im Folgenden vorgestellt.

Anzahl Threads

Standardmäßig steht es Gurobi frei sämtliche CPU Threads des Systems zu benutzen. Mittels Parameter lässt sich die Anzahl der verwendeten Threads aber limitieren. Da in dieser Arbeit nur serielle Heuristiken getestet wurden, bietet es sich für den direkten Vergleich an, auch Gurobi bei der Ausführung auf einen Thread zu beschränken.
Da sich Gurobi aber mittels Änderung eines einzelnen Parameters einfach parallel ausführen lässt, wurden sowohl die serielle als auch die parallele Ausführung getestet und miteinander verglichen.

Gurobi verwendet zum Lösen des Problems unter anderem einen Branch-and-Bound Algorithmus in Kombination mit verschiedenen Schnittebenenverfahren [GO17c]. Mit einer größeren Anzahl an Threads kann Gurobi also prinzipiell mehr Knoten im Branch-and-Bound Suchbaum in der selben Zeit durchsuchen.

Begrenzte Laufzeit

Gurobi kann in seiner Laufzeit begrenzt werden und die bis dahin beste gefundene Lösung ausgeben. In der Regel findet Gurobi, unter anderem dank seiner internen Heuristiken, schnell eine gültige Lösung. Anschließend braucht Gurobi dann aber lange, um eine tatsächlich optimale Lösung zu finden, beziehungsweise die Optimalität der besten bisher gefundenen Lösung nachzuweisen. Es bietet sich daher an, Gurobi mit unterschiedlichen Laufzeitbeschränkungen auf großen Probleminstanzen laufen zu lassen, um anschließend zu analysieren, wie sich die Lösungsqualität in Abhängigkeit von der Laufzeit verbessert. Dazu wurde Gurobi neben der unbeschränkten Variante auch beschränkt auf 1 bis 20 Sekunden ausgeführt.

3.2. Greedy

Greedy-Algorithmen zeichnen sich dadurch aus, dass sie in jedem Entscheidungsschritt immer die Wahl treffen, die zu diesem Zeitpunkt das bestmögliche Ergebnis verspricht. Der erwartete Nutzen einer Entscheidung wird durch eine Bewertungsfunktion angegeben. Eine einmal getroffene Wahl steht fest, wird also im späteren Verlauf des Algorithmus nicht noch einmal überdacht. Mit jeder getroffenen Wahl werden die Auswahlmöglichkeiten in den folgenden Entscheidungsschritten weiter eingeschränkt. Greedy-Algorithmen sind häufig sehr schnell, erzeugen im Allgemeinen aber keine optimalen Lösungen. Wegen ihrer meist kurzen Laufzeiten und einfachen Implementierung eignen sie sich dennoch häufig, um für viele Probleme eine ausreichend gute Lösung in angemessener Zeit zu finden.

Für diese Arbeit wurden eine ganze Reihe verschiedener Greedy-Algorithmen ausprobiert und verglichen, die jeweils aus zwei Schritten bestehen: Zunächst wird die Menge der potentiellen Treffen nach einem gegebenen Kriterium sortiert. Im Allgemeinen erhofft man sich von diesem Sortierungskriterium einen möglichst großen Nutzen für den Wert der erzeugten Lösungen. Für den Vergleich wurden aber auch ein paar Sortierungen getestet, von denen eigentlich keine besonders gute Lösung zu erwarten ist.

Anschließend werden die Treffen nacheinander in dieser Reihenfolge gemäß einer ebenfalls vorgegebenen zweiten Regel einem *passenden Zeitslot* zugeordnet. Ein Zeitslot ist passend, wenn er nicht bereits ein anderes Treffen enthält, das mit dem aktuell betrachteten Treffen im Konflikt steht. Wird für ein Treffen kein passender Zeitslot gefunden, dann ist dieses nicht Teil der Lösung. Damit ist garantiert, dass die so konstruierten Lösungen immer zulässig ist.

Alle betrachteten Greedy-Algorithmen sind also eine Kombination aus einer Sortierungsregel und einer Regel für die Auswahl passender Zeitslots. Algorithmus 3.1 stellt den generellen Ablauf dar.

Algorithmus 3.1 : GREEDY

Input : Probleminstanz $I \coloneqq$ (Personen P, Treffen T, Gewichtsfunktion w, Zeitslots t), Sortierungsregel S, Auswahlregel A
Output : Lösung R

```
// Initialisierung: Lösung mit leeren Zeitslots
R := ∅
for k := 0; k < t; k := k + 1 do
    R_k := ∅
    R := R ∪ R_k
// Auswahl und Zuweisung von Treffen
T_sort := Sortiere Treffen T gemäß Sortierungsregel S
foreach T_i ∈ T_sort do
    R_k := Wähle passenden Zeitslot für T_i gemäß Auswahlregel A
    if R_k ≠ null then
        R_k := R_k ∪ T_i          // Treffen T_i wird in Zeitslot k aufgenommen
return R
```

3.2.1. Sortierung der Treffen

Die getesteten Sortierungsregeln werden hier kurz vorgestellt. Falls das primäre Sortierungskriterium keine eindeutige Reihenfolge zwischen zwei Treffen festlegen kann, wird zusätzlich sekundär absteigend nach Gewicht und aufsteigend nach Anzahl der Teilnehmer sortiert. Dadurch sollen in diesen Fällen Treffen mit höherem Gewicht, beziehungsweise kleinerer Teilnehmerzahl, bei der Auswahl bevorzugt werden.

Je nach Wahl der Repräsentation und Datenstruktur der Probleminstanzen liegen die bei der Sortierung benötigten Informationen für jedes Treffen, beziehungsweise jede Person, bereits in konstanter Zeit abrufbar vor, oder eben nicht. Um alle getesteten Algorithmen unter ähnlich guten Bedingungen testen und vergleichen zu können, wurden daher in der Implementierung die benötigten Informationen für die verschiedenen Sortierungskriterien so weit wie möglich vorberechnet. Diese Vorberechnungen gehen nicht in die Laufzeitmessungen mit ein.

Gewicht absteigend (WEIGHT-DESC)

Die Treffen werden absteigend nach Gewicht und sekundär aufsteigend nach Teilnehmerzahl sortiert. Treffen mit hohem Gewicht werden mit höherer Priorität in die Lösung aufgenommen, um eine Lösung mit möglichst großem Wert zu erhalten.

Teilnehmerzahl aufsteigend (Members-Asc)

Die Treffen werden aufsteigend nach Teilnehmerzahl und sekundär absteigend nach Gewicht sortiert. Die Annahme hierbei ist, dass Treffen mit kleiner Teilnehmerzahl auch weniger Konflikte mit anderen Treffen haben, so dass sie sich gut mit anderen Treffen innerhalb eines Zeitslots kombinieren lassen.

Teilnehmerzahl absteigend (Members-Desc)

Die Treffen werden absteigend nach Teilnehmerzahl und sekundär absteigend nach Gewicht sortiert. Dies dient dem Vergleich mit der Sortierungsregel Members-Asc.

Verhältnis Gewicht zu Teilnehmerzahl absteigend (Wtmr-Desc)

Die Treffen werden absteigend nach dem Verhältnis von Gewicht und Teilnehmerzahl und sekundär absteigend nach Gewicht sortiert. Die Idee hierbei ist, die zwei Kriterien Gewicht und Teilnehmerzahl zu kombinieren.

Verhältnis Gewicht zu Teilnehmerzahl absteigend 2 (Wtmr2-Desc)

Die Treffen werden absteigend nach dem Verhältnis von Gewicht und Teilnehmerzahl und sekundär aufsteigend nach Teilnehmerzahl sortiert. Der Unterschied zu Wtmr-Desc liegt im sekundären Sortierungskriterium.

Teilnahmen aufsteigend (Participations-Asc)

Die Treffen werden in der Form sortiert, dass Personen, die an weniger Treffen teilnehmen als andere, ihre Treffen zuerst wählen. Die Treffen einer Person werden absteigend nach Gewicht und aufsteigend nach Teilnehmerzahl sortiert.

Teilnahmen absteigend (Participations-Desc)

Die Treffen werden in der Form sortiert, dass Personen, die an mehr Treffen teilnehmen als andere, ihre Treffen zuerst wählen. Die Treffen einer Person werden absteigend nach Gewicht und aufsteigend nach Teilnehmerzahl sortiert.

Konflikte aufsteigend (Conflicts-Asc)

Die Treffen werden aufsteigend nach der Anzahl an Konflikten mit anderen Treffen und sekundär absteigend nach Gewicht und aufsteigend nach Teilnehmerzahl sortiert. Die Annahme hierbei ist, dass sich Treffen mit weniger Konflikten besser mit anderen Treffen innerhalb eines Zeitslots kombinieren lassen.

Konflikte absteigend (Conflicts-Desc)

Die Treffen werden absteigend nach der Anzahl an Konflikten mit anderen Treffen und sekundär absteigend nach Gewicht und aufsteigend nach Teilnehmerzahl sortiert. Dies dient dem Vergleich mit der Sortierungsregel Conflicts-Asc.

Verhältnis Gewicht zu Konflikte absteigend (Wtcr-Desc)

Die Treffen werden absteigend nach dem Verhältnis von Gewicht und Anzahl Konflikte und sekundär absteigend nach Gewicht und aufsteigend nach Teilnehmerzahl sortiert. Die Idee hierbei ist, die zwei Kriterien Gewicht und Anzahl Konflikte zu kombinieren.

Verhältnis Gewicht zu Konflikte aufsteigend (WTCR-ASC)

Die Treffen werden aufsteigend nach dem Verhältnis von Gewicht und Anzahl Konflikte und sekundär absteigend nach Gewicht und aufsteigend nach Teilnehmerzahl sortiert. Dies dient dem Vergleich mit der Sortierungsregel WTCR-DESC.

Keine / Zufällige Sortierung (NONE)

Werden die Treffen nicht sortiert, dann liegen sie auf Grund der zufälligen Erzeugung der Probleminstanzen in zufälliger Reihenfolge vor (siehe Kapitel 4.2.1). Dieser Fall dient dem Vergleich mit sämtlichen anderen Sortierungsregeln. Sollte ein Algorithmus basierend auf einer der obigen Sortierungsregeln im Durchschnitt ein schlechteres, oder nicht wirklich besseres Ergebnis liefern als der unsortierte Fall, dann kann man sich die Sortierung in diesem Algorithmus auch sparen. Das kann unter anderem noch interessant werden, wenn später betrachtete Algorithmen auf den hier vorgestellten GREEDY-Algorithmen aufbauen.

3.2.2. Zeitslot Auswahl

Nachdem die Treffen sortiert wurden, werden sie in dieser Reihenfolge nacheinander passenden Zeitslots zugeordnet. Für die Auswahl des Zeitslots wurden die folgenden Auswahlregeln getestet.

FIRST-FIT

Ein Treffen wird dem ersten passenden Zeitslot zugeordnet. Die Suche beginnt für jedes Treffen von vorne beim ersten Zeitslot.

NEXT-FIT

Ein Treffen wird dem nächsten passenden Zeitslot zugeordnet, startend bei dem Zeitslot, dem zuletzt ein Treffen hinzugefügt wurde. Die Suche ist zyklisch: Wenn der letzte Zeitslot überprüft wurde, wird die Suche beim ersten Zeitslot fortgesetzt, solange bis alle Zeitslots einmal überprüft wurden.

BEST-FIT

Ein Treffen wird dem passenden Zeitslot zugeordnet, in dem bereits die meisten Personen an einem Treffen teilnehmen.

WORST-FIT

Ein Treffen wird dem passenden Zeitslot zugeordnet, in dem die meisten Personen noch an keinem Treffen teilnehmen.

3.2.3. Zeitkomplexität

Die Laufzeit der GREEDY-Algorithmen setzt sich hauptsächlich zusammen aus der einmaligen Sortierung der Treffen und den im schlimmsten Fall nt Abfragen, ob ein Treffen einem Zeitslot zugeordnet werden kann.

Unter der Annahme, dass die bei der Sortierung verwendeten Informationen in konstanter Zeit zur Verfügung stehen, benötigt die Sortierung aller Treffen $\mathcal{O}(n \log n)$ Zeit.

Für die Überprüfung, ob ein Treffen in einem Zeitslot aufgenommen werden kann, wird für alle Teilnehmer des Treffens überprüft, ob einer davon bereits an einem anderen Treffen im gleichen Zeitslot teilnimmt. Die gewählte Implementierung erlaubt es, in konstanter Zeit herauszufinden, an welchem Treffen eine Person innerhalb eines Zeitslots teilnimmt. Damit

ergibt sich hierfür eine erwartete Gesamtlaufzeit von $\mathcal{O}(nt\mathbb{E}[\text{Anzahl Teilnehmer}])$, wobei $\mathbb{E}[\text{Anzahl Teilnehmer}]$ der durchschnittlichen Anzahl Teilnehmer pro Treffen entspricht. Diese kann je nach verwendeten Parametern bei der Generierung der Probleminstanzen schwanken (siehe Tabelle 4.2).

3.2.4. Beispiel für Nicht-Optimalität

Jeder dieser GREEDY-Algorithmen kann für bestimmte Probleminstanzen keine optimale Lösung finden. So nimmt zum Beispiel der GREEDY-WEIGHT-DESC-Algorithmus die Treffen absteigend nach ihrem Gewicht in die Lösung auf. Das kann selbst bei einfachsten Probleminstanzen zu einer nicht optimalen Lösung führen, wie der Konfliktgraph in Abbildung 3.1 beispielhaft zeigt:

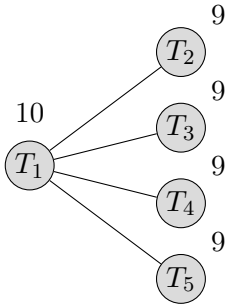


Abbildung 3.1.: Beispiel Konfliktgraph für Nicht-Optimalität von GREEDY

Treffen sind als gewichtete Knoten, Konflikte als Kanten dargestellt. Die Anzahl Zeitslots t sei 1. Der GREEDY-WEIGHT-DESC-Algorithmus nimmt hier immer zuerst das Treffen T_1 mit Gewicht 10 in die Lösung auf. Da die restlichen Treffen mit T_1 im Konflikt stehen, können diese nicht mehr in die Lösung aufgenommen werden. Die optimale Lösung würde jedoch die Treffen T_2, T_3, T_4 und T_5 enthalten und damit auf ein Gesamtgewicht von 36 kommen. Damit ist die gefundene Lösung in diesem Beispiel um einen Faktor $c = 36/10 = 3.6$ schlechter als die optimale Lösung. Im Allgemeinen, bei beliebigen Gewichten und beliebig vielen Konflikten mit T_1, lässt sich für diesen Faktor keine obere Schranke angeben.

Beweis. Das Gewicht von K_1 sei w_1, das kleinste Gewicht unter den restlichen $(m-1)$ in Konflikt stehenden Treffen sei w_{min}. Damit ist der Faktor $c \geq \dfrac{(m-1)w_{min}}{w_1}$.
Im schlechtesten Fall ist $w_{min} = w_1$ und m sehr groß, also
$\lim_{w_{min} \to w_1} \dfrac{(m-1)w_{min}}{w_1} = (m-1)$ und $\lim_{m \to \infty}(m-1) = \infty$. □

Genauso lassen sich auch für die anderen Sortierungsregeln spezielle Probleminstanzen finden, auf denen die gefundenen Lösungen um einen beliebigen Faktor schlechter als die optimalen Lösungen sind. Es handelt sich bei diesen GREEDY-Algorithmen also nur um Heuristiken.

3.3. Max k disjunkte unabhängige Mengen (MAX-k-DIS)

Das Lösen des Problems entspricht der Suche von t = Anzahl Zeitslots disjunkten unabhängigen Mengen unter den Treffen, deren Gewichte in der Summe möglichst groß sind. Die Idee ist nun, immer nur höchstens $k < t$ disjunkte unabhängige Mengen auf einmal zu suchen. Diese werden in die Lösung aufgenommen und dann beginnt die Suche von vorne auf den noch übrigen Treffen. Dieses Vorgehen wird wiederholt, bis für jeden Zeitslot eine

Algorithmus 3.2 : MAX-k-DIS

Input : Probleminstanz $I \coloneqq$ (Personen P, Treffen T, Gewichtsfunktion w, Zeitslots t), Parameter k, Löser L
Output : Lösung R

$R \coloneqq \emptyset$
$T_{remaining} \coloneqq T$ `// Verbleibende Treffen`
while $|T_{remaining}| > 0 \wedge |R| < t$ **do**
  $t_{next} \coloneqq \min(k, t - |R|)$ `// Minimum von` k `und verbleibenden Zeitslots`
  $I' \coloneqq \text{Probleminstanz}(P, T_{remaining}, w, t_{next})$ `// Kleinere Probleminstanz`
  $R' \coloneqq$ Löse I' mittels Löser L
  for $R'_k \in R'$ **do**
    `// Filtere leere Zeitslots aus der Lösung`
    if $R'_k \neq \emptyset$ **then**
      $R \coloneqq R \cup R'_k$ `// Aktualisiere bisherige Lösung`
  $T_{remaining} \coloneqq T_{remaining} \setminus T(R')$ `// Aktualisiere verbleibende Treffen`
`// Fülle restliche Zeitslot mit leeren Mengen`
for $k \coloneqq |R|; k < t; k \coloneqq k + 1$ **do**
  $R_k \coloneqq \emptyset$
return *Lösung* R

unabhängige Menge gefunden wurde, oder keine Treffen mehr übrig sind. Das Vorgehen wird in Algorithmus 3.2 beschrieben.

k ist dabei ein fest gewählter Parameter. Es wurden die Werte $k = 1, 2, 4, 8$ getestet. Für den Fall $k = 1$ entspricht das in jeder Iteration dem Lösen des MAXIMUM-WEIGHT-INDEPENDENT-SET Problems, also der Suche nach einer unabhängigen Menge mit maximalem Gewicht. Das MAXIMUM-WEIGHT-INDEPENDENT-SET Problem ist zwar selbst NP-schwer (und sogar schwer zu approximieren [FGL+91]), aber verglichen zu dem hier gegebenen Problem deutlich schneller lösbar.

Daher kann zum Lösen der kleineren Teilprobleme direkt der Gurobi-Löser verwendet werden. Die Experimente in 4.5 werden Auskunft darüber geben, wie gut dieser Ansatz im Vergleich zum optimalen Lösen des Ausgangsproblems mittels Gurobi skaliert. Alternativ kann auch eine der anderen vorgestellten Heuristiken verwendet werden. Getestet wurden hier, neben dem Gurobi-Löser, die GREEDY-BEST-FIT-WEIGHT-DESC-Heuristik und eine der im Folgenden noch vorgestellten INTERCHANGE-FILL-Heuristiken.

An dieser Stelle wird auf einen formalen Beweis für die NP-Schwere von MAXIMUM-WEIGHT-INDEPENDENT-SET verzichtet. Diese erschließt sich aber auch aus folgender Argumentation: Auf Grund der NP-Schwere von CLIQUE (siehe Abschnitt 2.3) muss auch die Suche nach einer Clique mit größtmöglichem Gewicht NP-schwer sein. Ansonsten könnte man einfach die Clique mit größtem Gewicht und damit größter Kardinalität auf einem Graphen bestimmen, auf dem alle Knoten Gewicht 1 haben. Mit der größten Clique ließe sich dann das NP-schwere CLIQUE Problem entscheiden. Da unabhängige Mengen gerade Cliquen im Komplementärgraphen entsprechen (siehe Abschnitt 2.2), muss auch MAXIMUM-WEIGHT-INDEPENDENT-SET NP-schwer sein.

3.3.1. Beispiel für Nicht-Optimalität

Mit Wahl des Parameters k verkleinert sich der Lösungsraum. Unter Umständen beinhaltet dieser verkleinerte Lösungsraum allerdings nicht mehr die optimale Lösung. In Abbildung

3.2 wird der Konfliktgraph einer Probleminstanz mit $t = 2$ dargestellt, für die dieser Algorithmus basierend auf Gurobi mit $k = 1$ keine optimale Lösung findet. MAX-k-DIS wählt zunächst die Treffen T_1 und T_4, da diese eine maximale unabhängige Menge mit Gewicht 4 bilden. Da T_2 und T_3 miteinander im Konflikt stehen, also keine unabhängige Menge im Konfliktgraphen bilden, kann für den zweiten Zeitslot nur noch eines dieser beiden Treffen gewählt werden. Die gefundene Lösung hat damit Gewicht 5. Die optimale Lösung würde stattdessen T_1 und T_3, sowie T_2 und T_4 in jeweils einen Zeitslot packen und hätte damit Gewicht 6.

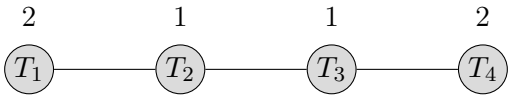


Abbildung 3.2.: Beispiel Konfliktgraph für Nicht-Optimalität von MAX-k-DIS, $k = 1$, $t = 2$

Max k disjunkte Cliquen

In Kapitel 2 wurde bereits festgestellt, dass unabhängigen Mengen gerade Cliquen auf dem Komplementgraphen entsprechen und umgekehrt. Auf dem komplementären Konfliktgraphen betrachtet entspricht MAX-k-DIS also gerade der wiederholten Suche nach den in der Summe schwersten k disjunkten Cliquen. In Abbildung 3.3 findet sich der komplementäre Konfliktgraph für das obige Beispiel. In dieser Darstellung des Problems ist gut erkennbar, wie die schwere Clique aus T_1 und T_4 die beiden jeweils für sich leichteren, aber in der Summe schwereren Cliquen $\{T_1, T_3\}$ und $\{T_2, T_4\}$ verhindert.

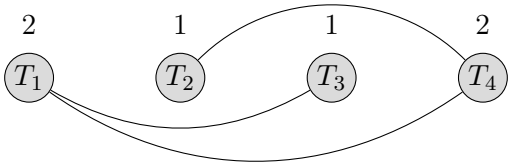


Abbildung 3.3.: Beispiel komplementärer Konfliktgraph, $k = 1$, $t = 2$

3.4. Lokale Suche: Austausch mit Auffüllen

Die in diesem Abschnitt vorgestellte INTERCHANGE-FILL-Heuristik basiert auf dem Prinzip der lokalen Suche.

3.4.1. Lokale Suche

An dieser Stelle wird das generelle Prinzip von lokaler Suche kurz beschrieben. Für eine gegebene Startlösung wird eine Menge von *benachbarten Lösungen* definiert. Mögliche Definitionen von Nachbarschaften basieren auf lokalen Veränderungen der gegebenen Lösung, wie zum Beispiel dem Hinzufügen, Entfernen, Umsortieren oder Austauschen von Elementen innerhalb und außerhalb der Lösung. In der so definierten Nachbarschaft der Startlösung wird nach einer besseren Lösung gesucht. Je nach Strategie wird dabei entweder die komplette, oder nur ein Teil der Nachbarschaft durchsucht. Dieser Vorgang wird solange mit der besten bisher gefundenen Lösung wiederholt, bis entweder keine bessere Lösung mehr gefunden wird, oder ein Abbruchkriterium erfüllt ist. Das könnte zum Beispiel eine begrenzte Laufzeit, oder eine Beschränkungen in der Anzahl von Iterationen oder betrachteten Lösungen sein.

3.4.2. Strategien zur Wahl einer neuen Lösung

Es wurden die folgenden zwei Strategien zur Auswahl einer neuen Lösung innerhalb der Nachbarschaft getestet und bezüglich Lösungsqualität und Laufzeit verglichen.

Steilster Anstieg (SteepestAscent (SA))

Es wird die beste Lösung innerhalb der kompletten Nachbarschaft gewählt. Dazu muss in jeder Iteration die gesamte Nachbarschaft durchsucht werden.

Erste Verbesserung (FirstImprovement (FI))

Es wird die erste gefundene Lösung gewählt, die besser als die momentan beste bekannte Lösung ist. Unter Umständen muss daher nicht in jeder Iteration die komplette Nachbarschaft durchsucht werden.

3.4.3. Austausch mit Auffüllen (Interchange-Fill)

Startlösung

Zum Erzeugen der Startlösung wurden sowohl die in Kapitel 3.2 vorgestellten Greedy-Heuristiken, als auch die Max-k-Dis-Heuristiken aus Kapitel 3.3 getestet. Für die mit diesen Heuristiken erzeugten Startlösungen gilt, dass, falls es ein Treffen gibt, das keinem Zeitslot zugeordnet wurde, dann enthält jeder der t Zeitslots bereits mindestens ein Treffen.

Beweis. Fallunterscheidung für die beiden Heuristiken:

Greedy durchläuft alle Treffen. Das nicht zugeordnete Treffen wurde also mindestens einmal betrachtet. Und auch die Auswahlregeln aus 3.2.2 überprüfen alle verfügbaren Zeitslots mindestens einmal, bevor ein Treffen verworfen wird. Falls es einen leeren Zeitslot gegeben hätte, dann wäre dieser für das Treffen überprüft und das Treffen diesem zugeordnet worden. Da Greedy einmal zugeordnete Treffen nicht wieder aus der Lösung entfernt, folgt der Widerspruch.

Max-k-Dis läuft solange, bis keine Treffen mehr verbleibend sind, oder die Lösung für jeden Zeitslot eine Menge von Treffen enthält. Max-k-Dis startet mit einer leeren Lösung und stellt sicher, dass nur nichtleere Zeitslots aus den Lösungen der Teilprobleme in die Lösung des Gesamtproblems aufgenommen werden. Daher kann also nur mit verbleibenden Treffen abgebrochen werden, falls die Lösung für jeden Zeitslot eine nichtleere Menge an Treffen enthält. Daraus folgt der Widerspruch. □

Mit dieser Feststellung fällt die Beschreibung des später vorgestellten Interchange-Fill Algorithmus etwas leichter aus, denn der Spezialfall, dass die Startlösung weniger als t Teilmengen enthält und dennoch Treffen nicht zugeordnet wurden, muss nicht betrachtet werden.

Austausch

Da alle Treffen nur positive Gewichte haben, lässt sich bei gegebener Startlösung leicht erkennen, dass jede bessere Lösung mindestens eines der Treffen enthalten muss, das nicht bereits Teil der Startlösung ist. Die Idee ist daher nun, zu überprüfen, ob sich die gegebene Lösung durch Hinzunahme eines der bisher nicht zugewiesenen Treffen verbessert.

Für jedes verbleibende Treffen gibt es t mögliche Zeitslots, denen das Treffen zugeordnet werden kann. Es ergeben sich daher $t \cdot (n - T(R))$ mögliche Lösungen, die sich auf diese Weise aus der gegebenen Startlösung R konstruieren lassen. Hierbei bezeichnet $n - T(R)$ die Anzahl der Treffen, die nicht Teil der Startlösung sind. Damit eine so erhaltene Lösung weiterhin zulässig bleibt, müssen alle im Konflikt stehenden Treffen im gewählten Zeitslot aus der Lösung entfernt werden. Es ergibt sich damit die folgende Definition einer Nachbarschaft.

Definition 3.1. *Die 1-Austausch Nachbarschaft einer gegebenen Lösung R ist die Menge aller Lösungen, die man erhält, indem man ein Treffen außerhalb von R einem Zeitslot zuweist und alle im Konflikt stehenden Treffen in diesem Zeitslot aus R entfernt.*

Auffüllen

Die bereits in Kapitel 3.2 betrachteten Greedy-Heuristiken eignen sich auch, um eine gegebene Lösung durch Auffüllen mit den verbleibenden Treffen schnell möglichst gut zu verbessern. Dabei werden die noch nicht zugewiesenen Treffen in der Reihenfolge entsprechend der Sortierung der verwendeten Greedy-Heuristik einem passenden Zeitslot zugewiesen. Für die hier verwendeten Startlösungen gilt, dass die Treffen außerhalb der Startlösung nicht konfliktfrei einem Zeitslot zugewiesen werden können. Die Startlösungen lassen sich daher nicht durch Auffüllen verbessern.

Auf einen vollständigen Beweis hierzu wird an dieser Stelle verzichtet. Aber falls die Startlösung durch eine der vorgestellten Greedy-Heuristiken gefunden wurde, dann ist bereits für sämtliche verbleibenden Treffen überprüft worden, dass sich diese keinem Zeitslot konfliktfrei zuordnen lassen. Und falls eine Max-k-Dis-Heuristik basierend auf Gurobi verwendet wurde, die jedem Zeitslot eine maximale unabhängige Menge zuordnet, dann folgt aus der Maximalität ebenfalls, dass kein Zeitslot mit einem der verbleibenden Treffen konfliktfrei erweitert werden kann. Für den kompletten Beweis müsste an dieser Stelle noch auf die Varianten von Max-k-Dis basierend auf Greedy und basierend auf dem hier erst noch vorgestellten Interchange-Fill-Algorithmus eingegangen werden.

In einem Austauschschritt wird ein Treffen in die Lösung aufgenommen und alle im Konflikt stehenden Treffen im gleichen Zeitslot aus der Lösung entfernt. Durch diese Veränderung ergeben sich zwei Möglichkeiten, die so erhaltene neue Lösung durch Auffüllen basierend auf einer Greedy-Heuristik mit wenig Aufwand zu verbessern.
So kann zum einen für die restlichen Treffen außerhalb der Lösung überprüft werden, ob sie sich nun konfliktfrei ebenfalls dem veränderten Zeitslot zuweisen lassen. Für ein möglichst gutes Ergebnis in kurzer Laufzeit erfolgt dies hier in der Reihenfolge entsprechend der Weight-Desc Sortierung aus Kapitel 3.2.
Und es kann überprüft werden, ob die aus diesem Zeitslot entfernten Treffen stattdessen konfliktfrei anderen Zeitslots zugewiesen werden können. Da die entfernten Treffen zuvor dem gleichen Zeitslot zugewiesen waren, können sie untereinander keine Konflikte aufweisen. Die Auswahl eines passenden Zeitslots für eines dieser Treffen hat also keinen Einfluss auf die Auswahlmöglichkeiten für die restlichen Treffen. Daher erfolgt die Auswahl passender Zeitslots hier mittels der schnellen First-Fit Auswahlregel und ohne vorherige Sortierung der Treffen.

Es ergibt sich die folgende Definition der im Interchange-Fill-Algorithmus verwendeten Nachbarschaft. Das komplette Vorgehen von Interchange-Fill wird in Algorithmus 3.3 beschrieben.

Definition 3.2. *Die 1-Austausch-mit-Auffüllen Nachbarschaft einer gegebenen Lösung R ist die Menge aller Lösungen, die man wie folgt erhält: Ein Treffen außerhalb von R wird einem Zeitslot zugewiesen. Alle im Konflikt stehenden Treffen in diesem Zeitslot werden aus R entfernt. Der veränderte Zeitslot wird mit den restlichen Treffen außerhalb von R aufgefüllt. Alle anderen Zeitslots werden mit den zuvor entfernten Treffen aufgefüllt.*

Dieses zusätzliche Auffüllen nach dem Austausch kann eine erhebliche Verbesserung bedeuten, wie die Beispiele aus den Abschnitten 3.2.4 und 3.3.1 zeigen: Ohne Auffüllen würde für beide Beispiele keine Verbesserung gefunden werden. Mit Auffüllen hingegen werden für beide Fälle sogar die optimalen Lösungen gefunden. Allerdings gibt es auch für den Interchange-Fill-Algorithmus wieder Probleminstanzen, die sich nicht optimal Lösen lassen. Ein Beispiel dafür wird in Abschnitt 3.4.4 gegeben.

Algorithmus 3.3 : INTERCHANGE-FILL

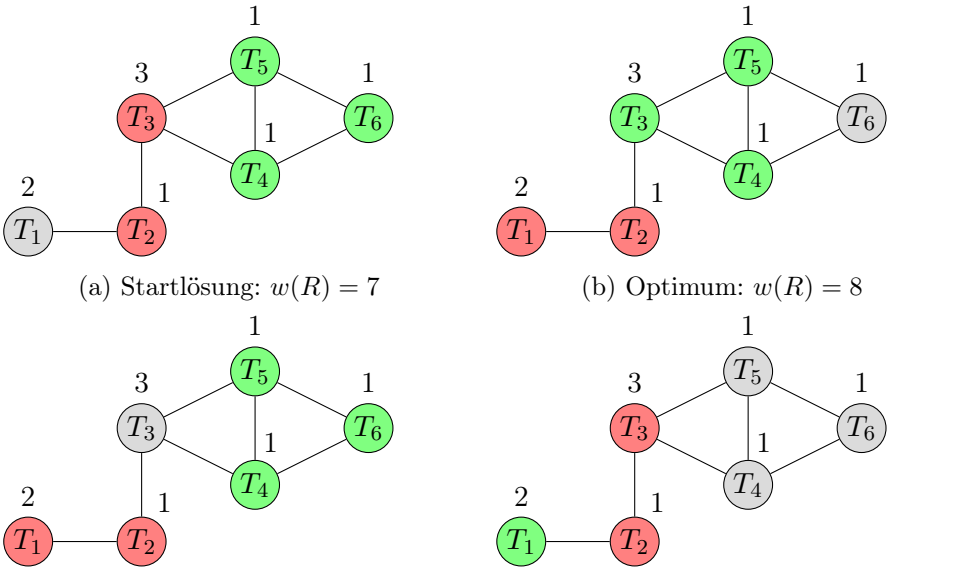
Input : Probleminstanz $I \coloneqq$ (Personen P, Treffen T, Gewichtsfunktion w, Zeitslots t), Löser L, Strategie S
Output : Lösung R

```
1  T_sorted ≔ Sortiere T absteigend nach Gewicht        // WEIGHT-DESC Sortierung
2  R ≔ Löse I mittels Löser L                  // Startlösung, bisher beste Lösung
3  b_NewSolution ≔ true        // Indikator: Suche läuft eine weitere Iteration
   // Suche solange, bis keine bessere Lösung mehr gefunden wurde
4  while b_NewSolution = true do
5      b_NewSolution ≔ false                        // Setze Indikatorvariable zurück
6      T_remaining ≔ (T_sorted \ T(R))              // Verbleibende Treffen sortiert
       // Schleife über jedes verbleibende Treffen und jeden Zeitslot
7      foreach T_i ∈ T_remaining do
8          for k ≔ 0; k < t; k ≔ k + 1 do
9              R_new ≔ Kopie von R
10             T_conflicting ≔ Mit T_i im Konflikt stehende Treffen in R_k
               // Austausch: Entferne im Konflikt stehende Treffen
               // und füge stattdessen T_i in Zeitslot k ein
11             R_new[k] ≔ R_new[k] \ T_conflicting
12             R_new[k] ≔ R_new[k] ∪ T_i
               // Auffüllen von R_new[k] mit restlichen verbleibenden Treffen
13             foreach T_j ∈ T_remaining do
14                 if T_j ≠ T_i ∧ T_j has no conflict in R_new[k] then
15                     R_new[k] ≔ R_new[k] ∪ T_j            // Füge Treffen T_j ein
               // Auffüllen der anderen Zeitslots
               // mit zuvor entfernten Treffen
16             foreach T_c ∈ T_conflicting do
17                 foreach R_new[l] ∈ R_new do
18                     if R_new[l] ≠ R_new[k] ∧ T_c has no conflict in R_new[l] then
19                         R_new[l] ≔ R_new[l] ∪ T_c        // Füge Treffen T_c ein
               // Überprüfe, ob wir eine neue beste Lösung gefunden haben
20             if w(R_new) > w(R) then
21                 R ≔ R_new                        // Ersetze bisherige beste Lösung
22                 b_NewSolution = true // Führe Suche in nächster Iteration fort
                   // Behandlung der First-Improvement-Strategie
23                 if Strategy S = FIRSTIMPROVEMENT then
24                     Springe zu Zeile 4          // Springe zur nächsten Iteration
25 return Lösung R
```

3.4.4. Beispiel für Nicht-Optimalität

In diesem Abschnitt wird die Funktionsweise von Interchange-Fill Schritt-für-Schritt an einem Beispiel vorgestellt. Außerdem wird demonstriert, dass auch dieser Algorithmus bestimmte Typen von Probleminstanzen nicht optimal lösen kann.

Die Abbildungen in 3.4 zeigen eine gegebene Startlösung, die optimale Lösung und zwei Zwischenergebnisse des Interchange-Fill-Algorithmus für eine einfache Probleminstanz dargestellt als komplementärer Konfliktgraph. Zwei Knoten haben in dieser Darstellung also genau dann eine gemeinsame Kante, wenn sie dem gleichen Zeitslot zugewiesen werden können. Die Farbe eines Knoten repräsentiert seine Zuordnung zu einem Zeitslot. In diesem Beispiel gibt es zwei Zeitslots, dargestellt mit den Farben Rot und Grün. Graue Knoten sind keinem Zeitslot zugewiesen, also nicht Teil der Lösung.



(a) Startlösung: $w(R) = 7$ (b) Optimum: $w(R) = 8$

(c) Austausch 1: T_1 zu Rot, $w(R) = 6$ (d) Austausch 2: T_1 zu Grün, $w(R) = 6$

Abbildung 3.4.: Beispiel für Ablauf und Nicht-Optimalität von Interchange-Fill, Darstellung als komplementäre Konfliktgraphen

Wir gehen an dieser Stelle davon aus, dass eine Heuristik uns die Startlösung aus Abbildung 3.4a geliefert hat. Wir befinden uns damit in Zeile 2 des Interchange-Fill-Algorithmus. Die Startlösung hat einen Wert von 7 und einen Knoten T_1, der keinem Zeitslot zugewiesen ist. Die optimale Lösung ist in 3.4b abgebildet und hat einen Wert von 8.

Interchange-Fill betrachtet in jeder Iteration nacheinander alle Knoten, die nicht Teil der aktuell besten Lösung sind, und versucht diese einem Zeitslot zuzuweisen. Im ersten Austauschschritt ordnet der Algorithmus Treffen T_1 dem Zeitslot Rot zu. Da aber T_1 mit T_3 im Konflikt steht, muss T_3 aus Zeitslot Rot entfernt werden, um eine zulässige neue Lösung zu erhalten. Beim anschließenden Auffüllen versucht der Algorithmus den soeben entfernten Knoten T_3 einem anderen Zeitslot zuzuweisen. Da T_3 aber mit T_6 in Konflikt steht, kann er nicht im Zeitslot Grün aufgenommen werden. Damit ergibt sich die erste Zwischenlösung aus Abbildung 3.4c.

Da die neue Lösung mit Wert 7 schlechter als die Startlösung ist, verwirft der Algorithmus diese Lösung und sucht weiter. Wir befinden uns in Zeile 8 des Algorithmus in der zweiten Iteration der for-Schleife. Im nächsten Austauschschritt wird Knoten T_1 dem Zeitslot Grün zugewiesen. Da die Treffen T_4, T_5 und T_6 mit T_1 im Konflikt stehen, werden diese aus Zeitslot Grün entfernt. Da diese drei Treffen auch alle mit T_2 im Konflikt stehen, kann

beim Auffüllen keines davon in Zeitslot Rot aufgenommen werden. Es ergibt sich die Zwischenlösung aus Abbildung 3.4d.

Da auch diese Lösung mit Wert 6 schlechter als die Startlösung ist und es neben T_1 keine anderen Treffen außerhalb der Startlösung gibt, bricht Interchange-Fill an dieser Stelle die Suche ab und gibt die Startlösung als Endergebnis zurück. Die optimale Lösung wurde für diese Probleminstanz und die gegebene Startlösung also nicht gefunden.

4. Experimentelle Evaluierung

Die vorgestellten Algorithmen wurden experimentell evaluiert. Die Algorithmen und ihre Variationen wurden in Java implementiert und auf zuvor generierten Probleminstanzen ausgeführt. Sie wurden bezüglich Laufzeit und Qualität der Lösungen untersucht und miteinander verglichen. In diesem Kapitel werden die Ergebnisse davon vorgestellt. Zunächst aber wird der Versuchsaufbau und die Generierung der Testdaten beschrieben.

4.1. Versuchsaufbau

Einen nicht unerheblichen Teil dieser Arbeit stellte die Entwicklung eines einfachen Softwaregerüsts in Java dar, in dessen Rahmen die entwickelten Algorithmen implementiert, ausgeführt und bewertet wurden. Das entstandene Programm bietet im Wesentlichen die folgenden Features:

- Modellierung des betrachteten Problems
- Einfacher Instanzgenerator mit konfigurierbaren Parametern
- Unterstützt die Gruppierung von Probleminstanzen und dazugehörigen Ergebnissen
- Einfaches Hinzufügen neuer Löser
- Ausführung ausgewählter Löser auf ausgewählten Daten
- Auswertung: Metriken für Probleminstanzen, Metriken für Löser und deren Ergebnisse
- Validierung von Parametern, Probleminstanzen und Lösungen
- Speichern und Laden von Probleminstanzen, Lösungen und Auswertungsergebnissen
- Export von Ergebnissen in wahlweise JSON oder CSV-Tabellenformat
- Gruppierter Export der Ergebnisse für bestimmte Löser
- Bedienung mittels Kommandozeilenparameter
- Ausführliche Ausgabe von Statusmeldungen und Fehlerbehandlung

4.1.1. Auswertung

Wegen der Begrenzung der Laufzeiten und dem Vergleich von paralleler und sequentieller Ausführung ergeben sich insgesamt 22 Variationen des Gurobi-Lösers aus Kapitel 3.1.2. Durch die Kombination von Sortierungs- und Auswahlregeln ergeben sich insgesamt 48 Variationen des Greedy-Algorithmus aus Kapitel 3.2. Für den Max-k-Dis-Algorithmus aus Kapitel 3.3 ergeben mit den verschiedenen Werten für den Parameter k und den unterschiedlichen Lösern für die Teilprobleme 12 Variationen. Mit unterschiedlicher Wahl der Startlöser und der Unterscheidung zwischen FirstImprovement und SteepestAscent Strategie ergeben sich für den in Kapitel 3.4 vorgestellten Interchange-Fill-Algorithmus 96 Variationen, die auf den Greedy-Algorithmen basieren, und weitere 24 Varianten basierend auf den Max-k-Dis-Algorithmen.
Insgesamt ergeben sich damit *202* zu testende Löser. In Kapitel 4.2 werden 9 unterschiedliche Variationen der Probleminstanzen (Problemgruppen) vorgestellt, für die diese Löser getestet wurden. Unter Beachtung der dort genannten Besonderheiten für die Problemgruppe Huge, ergibt sich damit insgesamt ein Datensatz mit *1797* Einträgen.

Für jeden Algorithmus und jede Problemgruppe wurden jeweils Minimum, Maximum und Durchschnitt über die Ergebnisse für alle Probleminstanzen für die folgenden Metriken erfasst:

- Laufzeit
- Speichernutzung direkt nach Ausführung
- Werte der Lösungen
- Anzahl Treffen in den Lösungen
- Anzahl Treffen außerhalb der Lösungen

In die Bewertung und den Vergleich der Algorithmen flossen aber nur die Laufzeit und die Werte der Lösungen ein. Für die Interchange-Fill-Algorithmen wurde außerdem die Anzahl an Iterationen erfasst.

4.1.2. Laufzeitmessung

Zum Teil wurden viele Algorithmen direkt hintereinander im selben Prozess ausgeführt und ausgewertet. Es hat sich gezeigt, dass die zuerst ausgeführten Algorithmen unabhängig vom Algorithmus deutlich schlechtere Laufzeiten erzielten. Vermutlich lässt sich das damit erklären, dass die Java VM Klassen dynamisch während der Ausführung bei der ersten Verwendung lädt und initialisiert [TFGA15].
Um diese Fehler bei der Laufzeitmessung zu minimieren, werden daher zunächst sämtliche benötigten Probleminstanzen geladen und die zu testenden Algorithmen durchlaufen zwei Aufwärmphasen. Dazu werden alle Algorithmen einmal anfangs und ein weiteres mal direkt vor ihrer jeweiligen bewerteten Ausführung auf einer Probleminstanz unbewertet ausgeführt. Erst nach diesen Aufwärmphasen erfolgen die tatsächlich bewerteten Ausführungen.

Testsystem

Die Ausführung und Laufzeitmessung erfolgte auf einem 64-Bit Windows 10 System mit einem Intel Core i7-4790K mit 8 Threads (4 Kerne mit Hyperthreading) bei 4.4 GHz und 12 GB Ram. Verwendet wurden Java Version 1.8.0_121-b13 und die Java HotSpot 64-Bit Server VM Version 25.121-b13. Die Java virtuelle Maschine wurde bei der Ausführung auf 8 GB Speicher beschränkt. Die gemessene Speichernutzung lag aber mit maximal ~3 GB immer deutlich darunter.

4.2. Probleminstanzen

4.2.1. Generator

Da keine Daten für das betrachtete Problem in dieser Form vorlagen, wurden die Algorithmen auf zufällig generierten Probleminstanzen getestet. Der für diesen Zweck erstellte Generator verfügt über eine Reihe von Parametern, mit denen sich die Generierung der Probleminstanzen steuern lässt. Der Generator wählt zufällige Werte innerhalb konfigurierbarer Unter- und Obergrenzen für folgende Parameter:

- Anzahl Zeitslots
- Anzahl Personen
- Anzahl Treffen
- Anzahl Teilnehmer pro Treffen
- Gewichtung der Treffen

Die Teilnehmer eines Treffens werden zufällig aus der Menge von Personen gewählt. Es kann daher prinzipiell vorkommen, dass einzelne Personen an keinem Treffen teilnehmen. Diese werden aus der erzeugten Probleminstanz entfernt. Die Probleminstanzen können dementsprechend letztendlich etwas weniger Personen enthalten, als durch die Parameter angegeben wurde.

Über die Kombination der Parameter 'Anzahl Personen' und 'Anzahl Teilnehmer pro Treffen' lässt sich indirekt die durchschnittliche Anzahl von Teilnahmen an Treffen pro Person beeinflussen. Bei gleichbleibender Anzahl Treffen lässt sich damit auch die durchschnittliche Anzahl an Konflikten zwischen Treffen variieren.

4.2.2. Problemgruppen

Um zu untersuchen, ob sich bestimmte Algorithmen für bestimmte Typen von Probleminstanzen unterschiedlich gut eignen, wurden die Algorithmen auf verschiedenen Gruppen von Probleminstanzen getestet. Diese Gruppen ergeben sich durch unterschiedliche Wahl der Parameter bei der Generation der Probleminstanzen. Die verwendeten Parameter werden in Tabelle 4.1 aufgelistet. Für jede Gruppe wurden jeweils 10 Probleminstanzen generiert. In der Auswertung der Algorithmen wird dann in den meisten Fällen die Laufzeit und der Lösungswert im Durchschnitt über alle 10 Probleminstanzen betrachtet.

Tabelle 4.1.: Problemgruppen Parameter

Gruppe	$\|t\|$	$\|P\|$	$\|T\|$	min $\|T_i\|$	max $\|T_i\|$	min w_i	max w_i
Normal	10	10	40	1	9	1	9
DenseWeights	10	10	40	1	9	1	5
Small	10	10	20	1	9	1	9
Large	10	10	80	1	9	1	9
Short	5	10	20	1	9	1	9
Long	20	10	80	1	9	1	9
Sparse	10	10	40	1	4	1	9
Dense	10	10	40	6	9	1	9
Huge	40	20	320	1	9	1	9

Mit der Parameterwahl ergeben sich für jede Problemgruppe bestimmte durchschnittliche Beschaffenheiten der Probleminstanzen. Diese können Tabelle 4.2 entnommen werden.

Tabelle 4.2.: Problemgruppen Metriken

Gruppe	avg $\frac{Treffen}{Person}$	avg $\frac{Treffen/Person}{Zeitslot}$	avg $\lvert T_i \rvert$	avg $\frac{Konflikte}{Treffen}$	avg w_i
Normal	20.39	2.04	5.10	33.83	5.13
DenseWeights	20.39	2.04	5.10	33.83	3.05
Small	10.25	1.02	5.13	16.75	4.90
Large	39.66	3.97	4.96	66.59	4.91
Short	10.25	2.05	5.13	16.75	4.90
Long	39.66	1.98	4.96	66.59	4.91
Sparse	10.10	1.01	2.50	19.12	5.13
Dense	30.00	3.00	7.50	39.00	5.13
Huge	79.88	2.00	4.99	210.41	5.02

Die Problemgruppe Normal stellt die gewählten Standardwerte der Parameter dar. Die Anzahl an Personen wurde für alle Probleminstanzen (außer für Huge) auf 10 festgesetzt. Die restlichen Parameter wurden für jede Problemgruppe gezielt so angepasst, dass sich im Durchschnitt für die erzeugten Probleminstanzen im Vergleich zur Problemgruppe Normal eine bestimmte Änderung ergibt.

Normal

Die Parameterwerte für die Problemgruppe Normal wurden so gewählt, dass sich auch die darauf aufbauenden Problemgruppen mit größeren Probleminstanzen (mit Ausnahme von Huge) noch innerhalb vertretbarer Laufzeiten optimal lösen lassen. Außerdem weisen die Probleminstanzen sämtlicher Problemgruppen ausreichend Treffen und Konflikte auf, sodass selbst optimale Lösungen nicht alle Treffen enthalten können. Es muss also immer eine Auswahl stattfinden.

Bei der gegebenen Wahl der Parameter für Normal haben die Treffen im Durchschnitt ~5 Teilnehmer und ~34 Konflikte. Es ergibt sich pro Person eine durchschnittliche Anzahl von ~20 Teilnahmen an Treffen. Bei 10 Zeitslots folgen daraus durchschnittlich ~2 potentielle Treffen pro Person und Zeitslot.

DenseWeights

Um zu untersuchen, ob sich eine kleinere Abweichung der Gewichte untereinander auf die Qualität der Greedy-Algorithmen auswirkt, wurde in DenseWeights das maximale Gewicht heruntergesetzt. Die maximale absolute Abweichung der Gewichte vom Durchschnitt wurde damit von 4 auf 2 reduziert.

Small

In Small wurde die Anzahl an Treffen halbiert. Bei gleichbleibender durchschnittlicher Anzahl von Teilnehmern pro Treffen, hat sich damit auch die durchschnittliche Anzahl von Teilnahmen an Treffen pro Person und die Anzahl an Konflikten pro Treffen in etwa halbiert.

Large

In Large wurde die Anzahl an Treffen verdoppelt, womit sich auch die durchschnittliche Anzahl von Teilnahmen an Treffen pro Person und die Anzahl an Konflikten pro Treffen in etwa verdoppelt hat.

Short

Die Problemgruppe Short simuliert einen verkleinerten Planungszeitraum bei gleichbleibender Auslastung der Personen. Die Anzahl an Zeitslots wurde halbiert. Um die durchschnittliche Anzahl an Treffen pro Person pro Zeitslot aufrecht zu erhalten, wurde entsprechend auch die Anzahl an Treffen halbiert. Wie für die Problemgruppe Small hat sich damit auch die durchschnittliche Anzahl an Konflikten pro Treffen in etwa halbiert.

Long

Die Problemgruppe Long simuliert einen vergrößerten Planungszeitraum bei gleichbleibender Auslastung der Personen. Die Anzahl an Zeitslots wurde verdoppelt. Um die durchschnittliche Anzahl an Treffen pro Person pro Zeitslot aufrecht zu erhalten, wurde entsprechend auch die Anzahl an Treffen verdoppelt. Wie für die Problemgruppe Large hat sich damit auch die durchschnittliche Anzahl an Konflikten pro Treffen in etwa verdoppelt.

Die Probleminstanzen dieser Problemgruppe sind im Vergleich zu denen der restlichen Problemgruppen (außer Huge) am schwersten optimal zu lösen. Durch gleichzeitige Vergrößerung der Anzahl an Treffen und Zeitslots ergibt sich für Long die größte Erweiterung des Lösungsraums.

Sparse

Um die Auswirkung unterschiedlicher durchschnittlicher Anzahl an Konflikten zwischen Treffen bei gleichbleibender Anzahl von Treffen zu untersuchen, wurde für Sparse die durchschnittliche Anzahl an Teilnehmern pro Treffen von ~5 auf ~2.5 reduziert. Damit reduziert sich die durchschnittliche Anzahl an Konflikten pro Treffen um etwa 44%.

Dense

Ähnlich zu Sparse wurde in Dense die Anzahl an Teilnehmern pro Treffen von ~5 auf ~7.5 erhöht. Damit sind im Durchschnitt fast sämtliche Treffen im Konflikt mit allen anderen Treffen. Im Durchschnitt kann also höchstens ein Treffen pro Zeitslot in eine Lösung aufgenommen werden.

Huge

Die Problemgruppe Huge dient zum Austesten der Skalierbarkeit der Heuristiken. Im Vergleich zu Normal wurde die Anzahl an Zeitslots vervierfacht und die Anzahl an Personen verdoppelt. Zum Beibehalten der durchschnittlichen Anzahl von 2 Treffen pro Person pro Zeitslot wurde die Anzahl an Treffen entsprechend verachtfacht.

Die Probleminstanzen in dieser Problemgruppe sind so groß, dass sie sich nicht mehr innerhalb vertretbarer Zeit mit Gurobi optimal lösen lassen. Daher wurden für diese Problemgruppe nur die Varianten von Gurobi getestet, die in ihrer Laufzeit auf 20 Sekunden oder weniger beschränkt waren. Und auch für die Max-k-Dis-, und die darauf basierenden Interchange-Fill-Algorithmen, wurde die Variante basierend auf dem Gurobi-Löser für $k = 8$ nicht betrachtet.

4.3. Ganzzahlige lineare Programmierung (Gurobi)

In diesem Abschnitt werden die Ergebnisse für den Gurobi-Löser vorgestellt. Im Anhang in Tabelle A.1 finden sich für alle Problemgruppen die durchschnittlichen Lösungswerte der von Gurobi gefundenen optimalen Lösungen. Diese werden später mit den Lösungswerten der verschiedenen Heuristiken verglichen. In Tabelle 4.3 finden sich die minimalen, maximalen

und durchschnittlichen Laufzeiten für Gurobi bei serieller und paralleler Ausführung für alle Problemgruppen, sowie der Speedup für die durchschnittlichen Laufzeiten.

Erwartungsgemäß steigen die durchschnittlichen Laufzeiten für die Problemgruppen Large und Long unverhältnismäßig stark an, verglichen zu der Steigerung an der Anzahl von Treffen und Zeitslots.

Auffällig sind aber auch die zum Teil großen Unterschiede zwischen minimaler und maximaler Laufzeit von Probleminstanzen innerhalb der gleichen Problemgruppe. Der Aufwand zum optimalen Lösen einer Probleminstanz ist also nicht nur abhängig von der Anzahl an Treffen, Zeitslots und Personen, sondern auch stark von den zufällig erzeugten Konflikten zwischen den Treffen.

Tabelle 4.3.: Gurobi Laufzeiten (ms) bei 8 Threads

Gruppe	Seriell			Parallel			Speedup
	min	max	avg	min	max	avg	für avg
Normal	15	313	69.0	16	302	71.8	0.96
DenseWeights	16	331	69.2	16	195	67.5	1.03
Small	10	16	13.9	13	16	13.7	1.01
Large	144	1 873	626.3	77	3 277	732.0	0.86
Short	11	13	11.8	11	14	12.1	0.98
Long	1 284	55 515	13 458.9	1 231	74 931	14 991.6	0.90
Sparse	17	88	28.9	16	125	32.1	0.90
Dense	14	16	14.9	14	16	14.8	1.01
Huge	-	-	-	-	-	-	-

Vergleich von serieller und paralleler Ausführung

In 3.1.3 stellte sich zunächst die Frage, inwieweit sich die Anzahl verfügbarer Threads auf die Laufzeiten von Gurobi auswirkt.

Mit Blick auf die Ergebnisse in Tabelle 4.3 scheint die parallele Ausführung hier keine Verbesserung mit sich zu bringen, in einigen Fällen sogar eher langsamer zu sein. Auch für die Problemgruppen Large und Long mit größeren Probleminstanzen und größerem Branch-and-Bound Suchbaum ergeben sich im Durchschnitt längere Laufzeiten.

Sehr auffällig sind dabei die zum Teil deutlich schlechteren Laufzeiten für einzelne Probleminstanzen, die sich nicht mit dem generellen Overhead zum Starten oder der Kommunikation zwischen 8 Threads erklären lassen. In Tabelle 4.4 finden sich die Laufzeiten für die Probleminstanzen aus der Problemgruppe Long. Für Probleminstanz 3 zum Beispiel braucht Gurobi mit 8 Threads etwa doppelt so lange, wie mit nur einem Thread.
Die Ausgabe von Gurobi [GO17b] offenbart, dass sich die Anzahl der untersuchten Knoten des Branch-and-Bound Suchbaums für ein und dieselbe Probleminstanz deutlich unterscheiden kann. Die Ergebnisse dazu sind ebenfalls in Tabelle 4.4 zu finden. Zum Beispiel werden für Probleminstanz 3 bei der Ausführung mit nur einem Thread 740 Knoten untersucht, während es bei der parallelen Ausführung mit 5185 Knoten etwa 7 mal so viele sind, bevor Gurobi abbricht und ein Ergebnis ausgibt. Und auch die Laufzeiten für das Untersuchen eines Knoten im Suchbaum können stark schwanken.

Gurobi schränkt den Suchbaum durch verschiedene Schnittverfahren schrittweise ein und bricht die Suche ab, sobald der Suchraum vollständig durchsucht wurde, oder der relative Abstand zwischen der aktuell besten Lösung und der kleinsten bekannten oberen Grenze für den Zielfunktionswert eine festgelegte Schwelle unterschreitet [GO17c].

Tabelle 4.4.: Gurobi Laufzeiten für LONG bei 8 Threads

Instanz	Laufzeit (ms)		Speedup	# BB-Knoten	
	Seriell	Parallel		Seriell	Parallel
1	7 972	7 446	1.07	1 382	3 242
2	1 284	1 231	1.04	48	73
3	5 688	11 466	0.50	740	5 185
4	3 808	2 725	1.40	1 033	922
5	55 515	74 931	0.74	10 359	31 404
6	9 329	8 530	1.09	1 191	1 676
7	8 325	5 600	1.49	1 617	2 684
8	24 708	17 295	1.43	1 217	2 766
9	10 966	12 629	0.87	553	1 080
10	6 994	8 063	0.87	713	1 129

Meine Vermutung ist, dass die Laufzeiten für dieses Problem stark davon abhängen, welche Knoten des Suchbaums in welcher Reihenfolge untersucht, und welche Schnitte angewandt werden. Die parallele Ausführung kann diese Entscheidungen beeinflussen, und damit für ein und dieselbe Probleminstanz sowohl zu besseren Laufzeiten (siehe Instanz 7 von LONG), als auch schlechteren Laufzeiten (siehe Instanz 3 von LONG) führen.

Ähnliche Beobachtungen, dass scheinbar kleine Änderungen, die eigentlich keinen Einfluss auf die Performance des Problems haben sollten, dennoch die Laufzeiten eines Löser für (gemischt-) ganzzahlige Programme stark beeinflussen können, wurden bereits in der Vergangenheit gemacht. In [FLM+16] wird ein Überblick über verwandte Arbeiten zu diesem Thema gegeben. Außerdem wird darin versucht, einen Algorithmus zu entwerfen, der diese Performance Unterschiede zwischen verschiedenen Ausführungen zur Beschleunigung ausnutzen kann.

In dieser Arbeit werde ich nicht näher auf dieses Thema eingehen und stattdessen nur die Ergebnisse der seriellen Ausführung von Gurobi weiter betrachten.

Begrenzte Laufzeit

In 3.1.3 stellte sich außerdem die Frage, wie gut die gefundenen Lösungen sind, wenn Gurobi in seiner Laufzeit beschränkt wird. Die Vermutung dabei ist, dass Gurobi häufig schon früh eine gute oder sogar optimale Lösung findet, dann aber lange braucht, um den restlichen Lösungsraum zu durchsuchen und die Optimalität nachzuweisen.

Da nur Laufzeitbeschränkungen von einer Sekunde und höher getestet wurden, werden hier nur die Ergebnisse für die größeren Problemgruppen LONG und HUGE näher vorgestellt. Für alle anderen Problemgruppen fand Gurobi für alle Probleminstanzen bereits nach einer Sekunde Laufzeit eine optimale Lösung.

In Tabelle 4.5 finden sich die Lösungswerte für die Problemgruppe LONG nach unterschiedlichen Laufzeiten, sowie die relativen Abweichungen zu den optimalen Lösungswerten. Die ersten Vorkommen optimaler Lösungen sind **fett** markiert.
Nach einer Sekunde Laufzeit wurde bereits für vier Probleminstanzen eine optimale Lösung gefunden, nach zwei Sekunden für drei weitere und nach nur drei Sekunden auch für die restlichen Probleminstanzen. Die relativen Abweichungen zwischen dem Lösungswert nach einer Sekunde und dem optimalen Lösungswert sind maximal 1.95%. Die größte relative Abweichung für die Lösungswerte nach zwei Sekunden liegt bei 0.41%. Die durchschnittliche relative Abweichung für eine Probleminstanz ist nach einer Sekunde Laufzeit 0.59% und nach zwei Sekunden 0.11%.

Die Unterschiede zu den Laufzeiten von Gurobi in der unbeschränkten Variante sind beachtlich (siehe serielle Laufzeiten in Tabelle 4.3): Die minimale Laufzeit beträgt im unbeschränkten Fall über 1.2 Sekunden, die maximale Laufzeit sogar über 55.5 Sekunden und die durchschnittliche Laufzeit ~13.5 Sekunden.

Tabelle 4.5.: Gurobi Lösungswerte für LONG in Abhängigkeit von der Laufzeit

Instanz	$w(R)$			$\Delta w(R)$ (%)	
	1s	2s	3s	1s	2s
1	**283**	283	283	0.00%	0.00%
2	**263**	263	263	0.00%	0.00%
3	288	**290**	290	0.69%	0.00%
4	**280**	280	280	0.00%	0.00%
5	245	245	**246**	0.41%	0.41%
6	270	**271**	271	0.37%	0.00%
7	289	**292**	292	1.03%	0.00%
8	301	306	**307**	1.95%	0.33%
9	278	281	**282**	1.42%	0.35%
10	**264**	264	264	0.00%	0.00%

Die ersten Vorkommen optimaler Lösungen sind **fett** markiert.

Für die Problemgruppe HUGE sind die Unterschiede zwischen den Lösungswerten nach den ersten paar Sekunden und denen nach 20 Sekunden schon deutlich größer. In Abbildung 4.1 findet sich eine Darstellung der durchschnittlichen Lösungswerte für die verschiedenen Laufzeiten. Die genauen Werte finden sich auch im Anhang in Tabelle A.2. Diese werden später mit den Lösungswerten der Heuristiken verglichen.

Es ist zu erkennen, dass sich die Lösungswerte im Durchschnitt erst nach etwa 6 Sekunden verbessern. Die Ausgabe von Gurobi [GO17b] verrät uns, dass in den ersten 6 bis 7 Sekunden zunächst eine heuristische Startlösung gefunden, das gegebene Problem vereinfacht und der Wurzel Knoten untersucht werden. Erst danach geht Gurobi in die Branch-and-Cut Suche über, in der dann nach und nach bessere Lösungen gefunden werden. Die Lösungswerte in den ersten 6 Sekunden entsprechen also den von Gurobi anfangs gefundenen heuristischen Lösungen.
Im Anhang in Tabelle A.3 finden sich die Ergebnisse der verschiedenen GREEDY-Algorithmen für die Problemgruppe HUGE. Vergleicht man die durchschnittlichen Lösungswerte der GREEDY-Algorithmen mit dem Wert, den Gurobi in den ersten 6 Sekunden liefert, dann fällt auf, dass dieser gerade dem Wert von FIRST-FIT-NONE entspricht. Vermutlich verwendet Gurobi also anfangs eine dazu ähnliche Heuristik zum Finden der Startlösungen.

4.4. GREEDY

In diesem Kapitel werden die Ergebnisse für die GREEDY-Algorithmen aus Kapitel 3.2 vorgestellt. Insgesamt wurden 48 Varianten von GREEDY auf allen 9 Problemgruppen ausgeführt. Da die Laufzeiten aller getesteten GREEDY-Algorithmen vergleichsweise klein sind, interessieren vor allem die Werte der gefundenen Lösungen.

Wenn man sich für jede Problemgruppe die GREEDY-Algorithmen mit den besten durchschnittlichen Lösungswerten anschaut, dann stellt man fest, dass die GREEDY-Algorithmen basierend auf den Sortierungen WEIGHT-DESC und WTCR-DESC in Kombination mit den Zeitslot Auswahlregeln BEST-FIT und FIRST-FIT für alle Problemgruppen im Durchschnitt die besten Lösungen liefern.

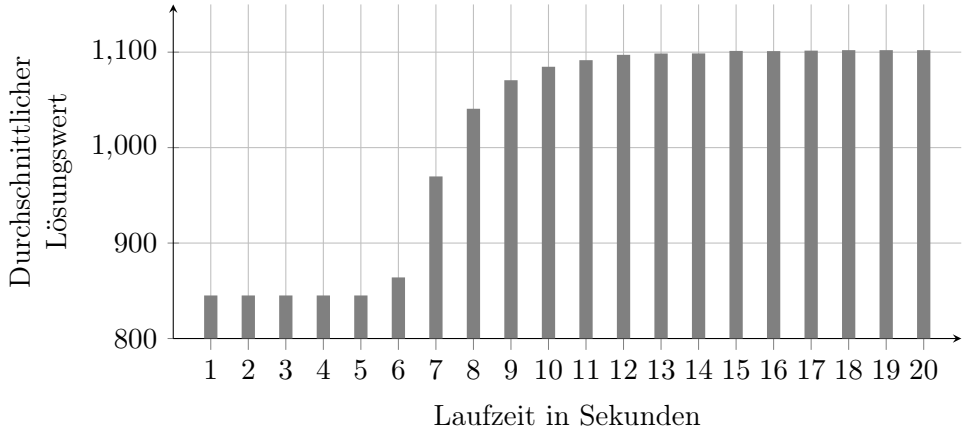


Abbildung 4.1.: Gurobi durchschnittliche Lösungswerte für Huge in Abhängigkeit von der Laufzeit

Für die Greedy-Algorithmen basierend auf den Sortierungen Wtcr-Asc, Conflicts-Desc, Members-Desc und None ergeben sich (erwartungsgemäß) für alle Problemgruppen durchschnittlich die schlechtesten Lösungswerte.

Im Anhang in Tabelle A.3 findet sich exemplarisch eine Übersicht über die Laufzeiten und Lösungswerte aller getesteten Greedy-Algorithmen für die Problemgruppe Huge.

Für die Problemgruppen DenseWeights und Normal liefert Greedy-Best-Fit-Wtcr-Desc den besten durchschnittlichen Lösungswert, für alle anderen Problemgruppen Greedy-Best-Fit-Weight-Desc. Für diese beiden Problemgruppen liegt der relative Abstand zwischen den durchschnittlichen Lösungswerten der beiden Greedy-Algorithmen unter 0.5%.

Vergleich von Best-Fit und First-Fit

In Tabelle 4.6 werden die Laufzeiten und Lösungswerte von Greedy-Best-Fit-Weight-Desc und Greedy-First-Fit-Weight-Desc verglichen. Greedy-First-Fit-Weight-Desc ist im Durchschnitt maximal ∼40% schneller (für Normal), minimal ∼4% langsamer (für Short) und durchschnittlich über alle Problemgruppen etwa 21% schneller als Greedy-Best-Fit-Weight-Desc. Die durchschnittlichen Lösungswerte von Greedy-First-Fit-Weight-Desc sind hingegen maximal ∼1% schlechter (für Huge).

Tabelle 4.6.: Vergleich von Greedy-Best-Fit-Weight-Desc (BF-WD) und Greedy-First-Fit-Weight-Desc (FF-WD)

Problemgruppe	avg Laufzeit (ms)		avg $w(R)$		ΔLaufzeit	$\Delta w(R)$
	BF-WD	FF-WD	BF-WD	FF-WD	(%)	(%)
Normal	9.3	5.5	124.6	123.4	40.86%	0.96%
DenseWeights	9.5	5.7	72.1	71.4	40%	0.97%
Small	8.7	8.4	85.7	85.5	3.45%	0.23%
Large	10.8	9.7	176	174.5	10.19%	0.85%
Short	7.6	7.9	56.9	56.5	-3.95%	0.70%
Long	12.7	9.7	263.8	262.5	23.62%	0.49%
Sparse	8.5	6.4	186.4	184.7	24.71%	0.91%
Dense	10.5	7.9	84.0	84.0	24.76%	0.00%
Huge	33.8	25.9	1 086.1	1 073.8	23.37%	1.13%

Skalierbarkeit und Vergleich mit Gurobi

In Tabelle 4.7 finden sich für alle Problemgruppen die durchschnittlichen Laufzeiten und Lösungswerte der Greedy-Best-Fit-Weight-Desc-Heuristik im Vergleich mit Gurobi. Für die Problemgruppe Huge liefert Gurobi ab etwa 11 Sekunden Laufzeit im Durchschnitt bessere Lösungswerte als Greedy-Best-Fit-Weight-Desc. Daher wurde hier sowohl mit den Werten von Gurobi bei einer Laufzeit von 20 Sekunden, als auch bei 11 Sekunden verglichen.

An den Laufzeiten für die Problemgruppen Large, Long, und Huge lässt sich erkennen, dass Greedy-Best-Fit-Weight-Desc deutlich besser skaliert als Gurobi. Für Huge liegen die Laufzeiten immer noch im Bereich von wenigen Millisekunden, während Gurobi bereits für die Probleminstanzen in der Problemgruppe Long durchschnittlich über 13 Sekunden lang rechnet. Die Lösungswerte von Greedy-Best-Fit-Weight-Desc weichen im Durchschnitt nur maximal ~5% von den optimalen Werten ab.

Tabelle 4.7.: Vergleich von Greedy-Best-Fit-Weight-Desc (BF-WD) und Gurobi

Gruppe	avg Laufzeit (ms)		avg $w(R)$		Laufzeit	$w(R)$
	BF-WD	Gurobi	BF-WD	Gurobi	(%)	(%)
Normal	9.3	69.0	124.6	129.2	13.48%	96.44%
DenseWeights	9.5	69.2	72.1	76.1	13.73%	94.74%
Small	8.7	13.9	85.7	86.5	62.59%	99.08%
Large	10.8	626.3	176.0	193.1	1.72%	91.14%
Short	7.6	11.8	56.9	58.0	64.41%	98.10%
Long	12.7	13 458.9	263.8	277.8	0.09%	94.96%
Sparse	8.5	28.9	186.4	192.4	29.41%	96.88%
Dense	10.5	14.9	84.0	84.0	70.47%	100.00%
Huge (11s)	33.8	11 082.9	1 086.1	1 091.6	0.30%	99.50%
Huge (20s)	33.8	20 090.7	1 086.1	1 102.1	0.17%	98.55%

4.5. Max-k-Dis

In diesem Kapitel werden die Ergebnisse für die Max-k-Dis-Algorithmen aus Kapitel 3.3 vorgestellt.

Vergleich mit Greedy-Best-Fit-Weight-Desc

Zunächst hat sich bei der Auswertung gezeigt, dass die Max-k-Dis-Algorithmen, die zum Lösen der Teilprobleme auf der Greedy-Best-Fit-Weight-Desc-Heuristik basieren, gegenüber der direkten Verwendung der Greedy-Best-Fit-Weight-Desc-Heuristik keine Vorteile mit sich bringen. Die durchschnittlichen Lösungswerte waren für alle Problemgruppen kleiner oder gleich denen von Greedy-Best-Fit-Weight-Desc und die durchschnittlichen Laufzeiten nicht erheblich kleiner, in einigen Fällen sogar etwas größer.

Das ist auch nicht besonders verwunderlich: Die Kernidee für Max-k-Dis ist es, durch Unterteilung des Problems in kleinere Teilprobleme mit weniger Zeitslots die Laufzeit zu reduzieren. Da Greedy-Best-Fit-Weight-Desc aber bereits sehr gut skaliert (siehe Tabelle 4.7: Durchschnittliche Laufzeit von ~33ms für Huge), hat die Unterteilung in kürzere Teilprobleme auf den hier betrachteten Probleminstanzen keinen entscheidenden Nutzen. Prinzipiell müssen wegen der Best-Fit Auswahlregel für jedes Treffen in jedem Teilproblem zwar weniger Zeitslots überprüft werden, da aber die nicht zugeordneten Treffen im nächsten Teilproblem ein weiteres mal betrachtet werden müssen, kann hier auch ein Mehraufwand entstehen. Diese Variante von Max-k-Dis wird hier daher nicht weiter betrachtet.

Vergleich mit Gurobi

Die Vorteile von Max-k-Dis gegenüber Gurobi zeigen sich vor allem für größere Probleminstanzen mit vielen Zeitslots. Die Ergebnisse der Max-k-Dis-Algorithmen mit verschiedenen Werten für k für die Problemgruppe Long, sowie der Vergleich mit den entsprechenden Ergebnissen von Gurobi, finden sich in Tabelle 4.8. Es wird zum einen Gurobi zum Lösen der Teilprobleme verwendet und zum anderen eine Interchange-Fill-Heuristik mit SteepestAscent-Strategie und Greedy-Best-Fit-Weight-Desc zum Finden der Startlösung.

Tabelle 4.8.: Max-k-Dis Ergebnisse für Long und Vergleich mit Gurobi

Algorithmus	Laufzeit (ms)			avg $w(R)$	Laufzeit (%)	$w(R)$ (%)
	min	max	avg			
Gurobi	1 284	55 515	13 458.9	277.8	100.00%	100.00%
8-Gurobi	98	692	317.8	271.7	2.36%	97.80%
8-Interchange-Fill	34	40	37.8	267.9	0.28%	96.44%
4-Gurobi	59	118	83.3	266.2	0.62%	95.82%
4-Interchange-Fill	27	36	30.7	263.7	0.23%	94.92%
2-Gurobi	44	85	59.9	262.9	0.45%	94.64%
2-Interchange-Fill	21	28	24.0	262.9	0.18%	94.64%
1-Gurobi	37	61	46.4	262.4	0.34%	94.46%
1-Interchange-Fill	20	28	22.7	262.3	0.17%	94.42%

Für die Problemgruppe Long liefert 8-Gurobi durchschnittlich 2.2% schlechtere Lösungen bei 97.6% kürzeren Laufzeiten. Und 1-Gurobi kommt im Schnitt auf 5.6% schlechtere Lösungen bei 99.7% kürzeren Laufzeiten. Die auf Interchange-Fill basierenden Max-k-Dis-Algorithmen liefern ähnlich gute Ergebnisse, sind aber im Vergleich mit den auf Gurobi basierten Algorithmen etwas schneller. Außerdem weisen sie deutlich kleinere Differenzen zwischen minimalen und maximalen Laufzeiten auf.

Mit größer werdenden Probleminstanzen werden diese Unterschiede zwischen beiden Varianten noch größer. Die Ergebnisse für die Problemgruppe Huge finden sich in Tabelle 4.9. Während hier auf die Ausführung von 8-Gurobi komplett verzichtet werden musste und auch 4-Gurobi bereits eine durchschnittliche Laufzeit von 20 Sekunden aufweist, findet die auf Interchange-Fill basierende Variante durchschnittlich sogar bessere Lösungen als Gurobi in 20 Sekunden und läuft immer noch für alle Probleminstanzen in unter ~1.7 Sekunden. Mit größeren Werten für k steigt hier aber die durchschnittliche Laufzeit etwas an. Der Unterschied zwischen $k = 1$ und $k = 8$ liegt für die Problemgruppe Huge bei ~1.1 Sekunden. Das entspricht einer relativen Steigerung von etwa 296%. Für die restlichen Problemgruppen fällt dieser Unterschied in der Laufzeit weniger stark auf: Für Long liegt der Unterschied bei 15 Millisekunden (Steigerung um 67%) und für Large bei 10 Millisekunden (Steigerung um 46%).

4.6. Interchange-Fill

In diesem Kapitel werden die Ergebnisse für die Interchange-Fill-Algorithmen aus Kapitel 3.4 vorgestellt. Es wurden insgesamt 120 verschiedene Varianten von Interchange-Fill auf allen 9 Problemgruppen ausgeführt. Die Varianten unterscheiden sich zum einen in der Strategie zur Wahl neuer Lösungen zwischen SteepestAscent (SA) und FirstImprovement (FI) und zum anderen in der verwendeten Heuristik zum Finden der Startlösung. 96 Varianten basieren auf den Greedy-Algorithmen aus Kapitel 3.2 und die restlichen 24 auf den Max-k-Dis-Algorithmen aus 3.3.

Tabelle 4.9.: MAX-k-DIS Ergebnisse für HUGE und Vergleich mit Gurobi

Algorithmus	Laufzeit (ms)			avg $w(R)$	Laufzeit (%)	$w(R)$ (%)
	min	max	avg			
GUROBI (20S)	-	-	20 090.7	1102.1	100.00%	100.00%
8-INTERCHANGE-FILL	1 113	1 701	1 455.0	1 106.9	7.24	100.44
4-GUROBI	6 278	41 304	19 860.4	1 106.1	98.85	100.36
4-INTERCHANGE-FILL	784	1 226	900.6	1 106.1	4.48	100.36
1-INTERCHANGE-FILL	355	389	367.7	1 102.7	1.83	100.05
2-INTERCHANGE-FILL	478	641	547.8	1 102.6	2.73	100.05
2-GUROBI	2 105	3 493	2 905.3	1 101.7	14.46	99.96
1-GUROBI	627	1 096	807.0	1 100.6	4.017	99.86

Bei der Auswertung hat sich gezeigt, dass die durchschnittlichen Lösungswerte aller INTERCHANGE-FILL-Varianten für sämtliche Problemgruppen sehr nahe beieinander liegen. Die Laufzeiten hingegen können sich, besonders auf den Problemgruppen mit großen Probleminstanzen, stark unterscheiden. Tabelle 4.10 listet für alle Problemgruppen die minimalen und maximalen durchschnittlichen Laufzeiten und Lösungswerte über alle INTERCHANGE-FILL-Varianten auf. Während sich die Lösungswerte im Schnitt höchstens 3.5% vom besten Wert unterschieden, liegt der maximale Unterschied bei den durchschnittlichen Laufzeiten bei 95.5%.

Tabelle 4.10.: Unterschiede zwischen den INTERCHANGE-FILL Varianten

Gruppe	avg Laufzeit (ms)		ΔLaufzeit (%)	avg $w(R)$		$\Delta w(R)$ (%)
	min	max		min	max	
NORMAL	11.3	70.3	83.93%	126.3	129.2	2.24%
DENSEWEIGHTS	11.5	96.4	88.07%	73.5	76.0	3.29%
SMALL	7.2	24.3	70.37%	86.1	86.5	0.46%
LARGE	22.2	284.0	92.18%	185.5	192.2	3.49%
SHORT	6.1	16.8	63.69%	57.3	58.0	1.21%
LONG	32.4	347.3	90.67%	268.7	274.4	2.08%
SPARSE	8.9	44.4	79.95%	187.3	191.9	2.40%
DENSE	8.5	36.0	76.39%	84.0	84.0	0.00%
HUGE	1 020.9	22 506.9	95.46%	1 109.2	1 136.4	2.39%

Im Anhang in Tabelle A.4 werden für jede Problemgruppe die Ergebnisse des INTERCHANGE-FILL-Algorithmus mit den jeweils besten durchschnittlichen Lösungswerten vorgestellt und mit den entsprechenden Werten von Gurobi verglichen. Falls es für eine Problemgruppe mehrere Varianten von INTERCHANGE-FILL mit dem besten durchschnittlichen Lösungswert gibt, dann wurde jeweils der Algorithmus mit der kleineren durchschnittlichen Laufzeit gewählt.

Da die vollständigen Namen der Algorithmen für diese Tabelle zu lang wären, werden folgende Abkürzungen verwendet: BEST-FIT (BF), FIRST-FIT (FF), NEXT-FIT (NF), WORST-FIT (WF), STEEPESTASCENT (SA), FIRSTIMPROVEMENT (FI), MAX-k-DIS basierend auf Gurobi (k-GUROBI), MAX-k-DIS basierend auf INTERCHANGE-FILL mit STEEPESTASCENT Strategie und GREEDY-BEST-FIT-WEIGHT-DESC-Startheuristik (k-IF). Außerdem wurde DENSEWEIGHTS zu DENSEWEIGHT. Mit *opt (%)* wird der relative Vergleich mit den Werten von Gurobi bezeichnet. Für die Problemgruppe HUGE wurde mit den Werten von Gurobi bei 20 Sekunden Laufzeit verglichen.

An dem Ergebnis für die Problemgruppe Huge lässt sich erkennen, dass Interchange-Fill auch mit schlechten Startlösungen bessere durchschnittliche Lösungswerte aufweisen kann, als für alle anderen getesteten Startlösungen. Allerdings ist hier die durchschnittliche Laufzeit nicht sonderlich gut. Über alle Problemgruppen hinweg lässt sich keine einzelne Startheuristik bestimmen, für die Interchange-Fill in allen Fällen die durchschnittlich besten Lösungen ausgibt. Häufig gibt es aber Interchange-Fill-Varianten, die nur leicht schlechtere Lösungen erzeugen und dafür deutlich geringere Laufzeiten aufweisen. Daher wird hier genauer auf die Unterschiede in den Laufzeiten eingegangen.

Vergleich von SteepestAscent und FirstImprovement

Die Laufzeit von Interchange-Fill hängt von der Anzahl an durchlaufener Iterationen, sowie der verwendeten Strategie ab. Für die größeren Problemgruppen Large, Long und Huge wurden die Unterschiede zwischen der SteepestAscent- und FirstImprovement-Strategie besonders deutlich: Unabhängig von den verwendeten Startlösungen sind hier alle Interchange-Fill-Algorithmen mit FirstImprovement-Strategie durchgängig schneller, als die Varianten mit SteepestAscent. In Tabelle 4.11 finden sich die minimalen und maximalen durchschnittlichen Laufzeiten für diese beiden Gruppen von Interchange-Fill-Algorithmen für Greedy-Startheuristiken.

Auf die durchschnittlichen Lösungswerte hatte die Wahl der Strategie hingegen keinen erkennbaren Einfluss. Für die gleiche Startheuristik lieferte Interchange-Fill mit SteepestAscent-Strategie für manche Problemgruppen etwas bessere Lösungswerte und für andere Problemgruppen tat dies hingegen Interchange-Fill mit FirstImprovement-Strategie. Sowohl für gute als auch schlechte Startlösungen konnte kein Zusammenhang zwischen gewählter Strategie und Lösungswerten festgestellt werden.

Tabelle 4.11.: Vergleich der Laufzeiten von Interchange-Fill mit SteepestAscent- (SA) und FirstImprovement-Strategie (FI)

Gruppe	avg Laufzeit (ms) SA		avg Laufzeit (ms) FI	
	min	max	min	max
Large	34.9	85.4	22.2	27.6
Long	53.1	243.3	32.4	58.9
Huge	5 074.1	16 661.4	1 020.9	1 586.0

Startlösung mit Max-k-Dis

Beim Vergleich der durchschnittlichen Laufzeiten hat sich über alle Problemgruppen gezeigt, dass die Interchange-Fill-Varianten, die für ihre Startlösung eine Max-k-Dis-Heuristik basierend auf Greedy-Best-Fit-Weight-Desc verwenden, keine bessere Laufzeiten aufweisen, als wenn die Startlösung direkt mit Greedy-Best-Fit-Weight-Desc gefunden wird. Auch die durchschnittlichen Lösungswerte waren nicht konsistent besser. Entsprechend wird diese Variante hier nicht weiter betrachtet.

Für Problemgruppen mit kleinen Instanzen findet Max-k-Dis basierend auf Gurobi bereits von sich aus gute Lösungen. Für $k = 8$ sind diese auch häufig direkt optimal. Für die Problemgruppe Normal und $k = 1$ findet Interchange-Fill im Schnitt nur 0.8 Verbesserungen für eine Probleminstanz. Auch die gewählte Strategie macht bei durchschnittlich weniger als einer Verbesserung keinen Unterschied für die Laufzeiten. Für größere Problemgruppen wachsen die Laufzeiten alleine für das Finden der Startlösung, wie bereits in 4.5 gesehen, für Max-k-Dis basierend auf Gurobi unverhältnismäßig stark an.

Für Startlösungen, die mit Max-k-Dis basierend auf der in 4.5 spezifizierten Interchange-Fill-Heuristik gefunden wurden, konnte kein Zusammenhang zwischen den resultierenden Lösungswerten und Parameter k hergestellt werden. Die Laufzeiten hingegen steigen, wie ebenfalls bereits in 4.5 festgestellt wurde, mit größeren Werten für k an. Die im Folgenden betrachteten Greedy-Startheuristiken führen zu genauso guten Lösungswerten, weisen aber kleinere Laufzeiten auf.

Startlösung mit Greedy

Oben wurde bereits in Tabelle 4.10 gezeigt, dass die Startlösungen keinen großen Einfluss auf den durchschnittlichen Wert der Lösungen haben. Es bietet sich daher an, die besonders einfache und schnelle Greedy-First-Fit-None-Heuristik näher zu betrachten. Da in Kapitel 4.4 bereits festgestellt wurde, dass die ebenfalls schnelle Greedy-Best-Fit-Weight-Desc-Heuristik besonders gute Lösungswerte liefert, werden die Ergebnisse von Interchange-Fill für diese beiden Startheuristiken miteinander verglichen. Die durchschnittlichen Laufzeiten, Iterationen pro Probleminstanz und Lösungswerte finden sich in Tabelle 4.12.

Tabelle 4.12.: Vergleich von Interchange-Fill mit FirstImprovement-Strategie und Greedy-First-Fit-None- (FF-N), beziehungsweise Greedy-Best-Fit-Weight-Desc-Startheuristik (BF-WD)

Gruppe	avg Laufzeit (ms)		avg Iterationen		avg $w(R)$	
	FF-N	BF-WD	FF-N	BF-WD	FF-N	BF-WD
Normal	13.6	16.1	11.7	2.8	127.7	128.7
DenseWeights	12.4	16.4	10.1	2.5	74.6	74.7
Small	11.6	10.8	7.2	1.4	86.5	86.5
Large	25.4	27.6	16.2	7.9	187.2	189.3
Short	11.4	9.4	5.4	1.4	57.9	57.8
Long	33.9	38.9	19.6	5.4	271.2	271.1
Sparse	13.8	14.9	7.2	2.5	187.9	189.6
Dense	14.2	15.0	8.8	1.0	84.0	84.0
Huge	1 281.1	1 147.4	84.6	23.8	1 126.3	1 128.0

Für die besseren Startlösungen ist die Anzahl an nötigen Iterationen für Interchange-Fill deutlich kleiner. Auf die Laufzeiten wirkt sich das aber nicht erkennbar aus. Für die Problemgruppe Huge ist Interchange-Fill mit Greedy-Best-Fit-Weight-Desc zwar 133ms und damit ~10% schneller, aber dafür für die Problemgruppen Long und Large ~14%, beziehungsweise 9% langsamer. Es lässt sich also kein eindeutiger Zusammenhang zur Laufzeit herstellen.

Beide Interchange-Fill-Varianten liefern ähnlich gute Lösungen, die Variante mit Greedy-Best-Fit-Weight-Desc-Startheuristik aber für manche Problemgruppen ein wenig bessere. In Tabelle 4.13 findet sich abschließend noch der Vergleich mit den ursprünglichen Ergebnissen von Greedy-Best-Fit-Weight-Desc.

Tabelle 4.13.: Vergleich von Greedy-Best-Fit-Weight-Desc (BF-WD) und darauf aufbauendem Interchange-Fill (IF)

Problemgruppe	avg Laufzeit (ms)		avg $w(R)$		ΔLaufzeit	$\Delta w(R)$
	BF-WD	IF	BF-WD	IF	(%)	(%)
Normal	9.3	16.1	124.6	128.7	73.12%	3.29%
DenseWeights	9.5	16.4	72.1	74.7	72.63%	3.61%
Small	8.7	10.8	85.7	86.5	24.14%	0.93%
Large	10.8	27.6	176	189.3	155.55	7.56%
Short	7.6	9.4	56.9	57.8	23.68%	1.58%
Long	12.7	38.9	263.8	271.1	206.30%	2.77%
Sparse	8.5	14.9	186.4	189.6	75.29%	1.72%
Dense	10.5	15.0	84.0	84.0	42.86%	0.00%
Huge	33.8	1 147.4	1 086.1	1 128.0	3 294.67%	3.86%

5. Fazit

In dieser Arbeit wurde das MeetingsProblem vorgestellt und verschiedene Lösungsansätze dafür evaluiert. Um die Vielzahl an verschiedenen Variationen der getesteten Algorithmen und die dabei anfallenden Daten handhaben zu können, wurde ein einfaches Softwaregerüst in Java entwickelt. Mit diesem konnte die Ausführung und Auswertung der Algorithmen gut automatisiert werden.

In Kapitel 4.3 wurde zunächst untersucht, wie groß die Probleminstanzen werden können, damit sie noch optimal mittels ganzzahliger linearer Programmierung in vertretbarer Zeit zu lösen sind. Es wurden außerdem verschiedene Varianten für Gurobi ausprobiert und festgestellt, dass eigentlich schon deutlich früher gute Lösungen gefunden werden. Mit begrenzter Laufzeit lässt sich Gurobi daher auch noch auf etwas größeren Probleminstanzen als Heuristik ausführen. Aber auch dieser Ansatz stößt für die größten hier getesteten Instanzen an sein Limit.

Es wurden daher eine Reihe von Heuristiken ausprobiert. Für viele Probleme stellen einfache Greedy-Heuristiken einen ersten Ansatz dar, um mit besonders schweren Problemen umzugehen. Durch das Testen vieler verschiedener Sortierungskriterien und Zuordnungsmöglichkeiten für die Treffen, konnten Varianten ausgeschlossen werden, von denen man sich eventuell das bessere Ergebnis erwartet hätte. Es hat sich aber gezeigt, dass der Greedy-Algorithmus basierend auf der absteigenden Sortierung nach Gewicht mit Best-Fit Zeitslotauswahl die besten Ergebnisse liefert und ohne Probleme bis zu den größten hier getesteten Probleminstanzen skaliert. Die durchschnittlichen Lösungswerte weichen von den optimalen Werten maximal um 10% ab und haben für viele der hier betrachteten Instanzen sogar nur eine Abweichung von 2-5%.

Mit Max-k-Dis wurde ein weiterer Ansatz vorgestellt. Die Lösungswerte verbesserten sich damit im Vergleich zu Greedy etwas weiter, während die Laufzeiten verglichen zu Gurobi immer noch um ein Vielfaches kleiner ausfielen. Für größere Probleminstanzen hat sich das Lösen der Teilprobleme mittels Interchange-Fill als besonders gut herausgestellt. Die durchschnittlichen Lösungswerte waren für die kleineren Probleminstanzen ähnlich gut wie die optimalen Werte, und wichen für größere Probleme maximal um 5% ab. Für die größten getesteten Probleminstanzen konnten in durchschnittlich 1.7 Sekunden sogar bessere Lösungen gefunden werden, als Gurobi nach 20 Sekunden Laufzeit lieferte.

Zum Abschluss wurde mit Interchange-Fill eine einfache Verbesserungsheuristik vorgestellt, die mit einer Greedy-basierten Startlösung für die großen Probleminstanzen auch die maximalen Lösungswerte von Max-k-Dis überbieten konnte und gleichzeitig

kleinerer Laufzeiten aufwies. Im Vergleich zur Greedy-Heuristik konnten die Ergebnisse um durchschnittlich 1-7.5% verbessert werden.

Wenn Interchange-Fill eine Verbesserung gefunden hat aber nur wenige Zeitslots verändert wurden, dann werden in der nächsten Iteration beim Auffüllen Kombinationen von Treffen und Zeitslots ausprobiert, die davor schon überprüft wurden. Eventuell kann es Sinn machen, sich die veränderten Zeitslots der letzten Iteration im Algorithmus zu merken und damit den folgenden Durchlauf zu optimieren.

Interchange-Fill landet irgendwann in einem lokalen Maximum. Es gibt also auch hier noch Potential für Erweiterungen des Algorithmus.

Literaturverzeichnis

[CR09] Chakaravarthy, Venkatesan T. ; Roy, Sambuddha: Approximating Maximum Weight K-colorable Subgraphs in Chordal Graphs. In: *Inf. Process. Lett.* 109 (2009), März, Nr. 7, 365–368. http://dx.doi.org/10.1016/j.ipl.2008.12.007. – DOI 10.1016/j.ipl.2008.12.007. – ISSN 0020–0190

[CT03] In: Casey, Stephen ; Thompson, Jonathan: *GRASPing the Examination Scheduling Problem.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2003. – ISBN 978–3–540–45157–0, 232–244

[Dom15] Domschke, Wolfgang ; Drexl, Andreas (Hrsg.) ; Klein, Robert (Hrsg.) ; Scholl, Armin (Hrsg.): *Einführung in Operations Research.* http://dx.doi.org/10.1007/978-3-662-48216-2. Version: 9., überarb. u. verb. Aufl. 2015, 2015 (SpringerLink : Bücher)

[DPDK17] Désirée, Balthasar ; Prof. Dr. Karlheinz, Geißler: *Wir haben nicht zu viel Zeit, sondern zu viel zu tun.* http://www.lto.de/recht/job-karriere/j/zeitmanagement-stress-beruf-termindruck-arbeit/. Version: 2017

[DW75] De Werra, D.: On a particular conference scheduling problem. In: *INFOR* 13 (1975), Nr. 3, S. 308 – 315. – ISSN 03155986

[FGL+91] Feige, U. ; Goldwasser, S. ; Lovasz, L. ; Safra, S. ; Szegedy, M.: Approximating clique is almost NP-complete. In: *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, 1991, S. 2–12

[FLM+16] Fischetti, Matteo ; Lodi, Andrea ; Monaci, Michele ; Salvagnin, Domenico ; Tramontani, Andrea: Improving branch-and-cut performance by random sampling. In: *Mathematical Programming Computation* 8 (2016), Nr. 1, 113–132. http://dx.doi.org/10.1007/s12532-015-0096-0. – DOI 10.1007/s12532–015–0096–0. – ISSN 1867–2957

[GO17a] Gurobi Optimization, Inc.: *Gurobi Optimizer Reference Manual.* http://www.gurobi.com. Version: 2017

[GO17b] Gurobi Optimization, Inc.: *MIP Logging.* http://www.gurobi.com/documentation/7.0/refman/mip_logging.html. Version: 2017

[GO17c] Gurobi Optimization, Inc.: *Mixed-Integer Programming (MIP) - A Primer on the Basics.* http://www.gurobi.com/resources/getting-started/mip-basics. Version: 2017

[Kar72] Karp, Richard M. ; Miller, Raymond E. (Hrsg.) ; Thatcher, James W. (Hrsg.) ; Bohlinger, Jean D. (Hrsg.): *Reducibility among Combinatorial Problems.* Boston, MA : Springer US, 1972. – 85–103 S. http://dx.doi.org/10.1007/978-1-4684-2001-2_9. http://dx.doi.org/10.1007/978-1-4684-2001-2_9. – ISBN 978–1–4684–2001–2

[Knu74] Knuth, D. E.: Postscript About NP-hard Problems. In: *SIGACT News* 6 (1974), April, Nr. 2, 15–16. http://dx.doi.org/10.1145/1008304.1008305. – DOI 10.1145/1008304.1008305. – ISSN 0163–5700

[KRSW11] In: Katz, Bastian ; Rutter, Ignaz ; Strasser, Ben ; Wagner, Dorothea: *Speed Dating*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–20662–7, 292–303

[MHH06] Malkawi, Mohammad ; Hassan, Mohammad A. ; Hassan, Osama A.: *A New Exam Scheduling Algorithm Using Graph Coloring*. 2006

[Pel09] In: Pelillo, Marcello: *Heuristics for maximum clique and independent set Heuristics for Maximum Clique and Independent Set*. Boston, MA : Springer US, 2009. – ISBN 978–0–387–74759–0, 1508–1520

[TFGA15] Tim, Lindholm ; Frank, Yellin ; Gila, Bracha ; Alex, Buckley: *The Java Virtual Machine Specification Java SE 8 Edition*. http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html. Version: 2015

[WC10] Wikimedia Commons, Kilom691 Pgdx: *A hypergraph with 7 vertices and 4 edges*. https://commons.wikimedia.org/wiki/File:Hypergraph-wikipedia.svg. Version: 2010. – Accessed 31.3.2016

Anhang

A. Ergänzende Auswertungsergebnisse

Tabelle A.1.: Optimale durchschnittliche Lösungswerte von Gurobi

Gruppe	avg $w(R)$
Normal	129.2
DenseWeights	76.1
Small	86.5
Large	193.1
Short	58.0
Long	277.8
Sparse	192.4
Dense	84.0
Huge	-

Tabelle A.2.: Durchschnittliche Lösungswerte von Gurobi für die Problemgruppe Huge in Abhängigkeit von der Laufzeit

Laufzeit	avg $w(R)$
1s	845.1
2s	845.1
3s	845.1
4s	845.1
5s	845.1
6s	864.0
7s	969.8
8s	1 040.7
9s	1 070.6
10s	1 084.7
11s	1 091.6
12s	1 097.2
13s	1 098.6
14s	1 098.7
15s	1 101.3
16s	1 101.1
17s	1 101.6
18s	1 102.1
19s	1 102.1
20s	1 102.1

Tabelle A.3.: Durchschnittliche Laufzeiten und Lösungswerte der verschiedenen Greedy-Algorithmen für die Problemgruppe Huge

Algorithmus	Laufzeit (ms)			$w(R)$		
	min	max	avg	min	max	avg
Best-Fit-Weight-Desc	30	52	33.8	1 022	1 172	1 086.1
First-Fit-Weight-Desc	25	27	25.9	1 014	1 146	1 073.8
Best-Fit-Wtcr-Desc	38	53	41	996	1 135	1 057.3
First-Fit-Wtcr-Desc	33	35	33.9	986	1 122	1 051.6
Next-Fit-Weight-Desc	35	37	36.2	976	1 120	1 043.7
Best-Fit-Wtmr-Desc	31	54	38	986	1 110	1 042.1
Best-Fit-Wtmr2-Desc	33	35	33.6	966	1 104	1 036.9
First-Fit-Wtmr-Desc	28	29	28.6	980	1 104	1 032.8
Worst-Fit-Weight-Desc	40	52	43	964	1 107	1 026.2
First-Fit-Wtmr2-Desc	28	30	29.4	957	1 093	1 023.6
Next-Fit-Wtcr-Desc	28	35	31	960	1 085	1 019.5
Best-Fit-Participations-Asc	34	42	36.6	949	1 111	1 018
Next-Fit-Participations-Asc	40	43	41.3	956	1 115	1 010.2
First-Fit-Participations-Asc	30	33	31.7	936	1 115	1 005.3
Worst-Fit-Participations-Asc	44	57	46.1	935	1 093	1 002.8
Worst-Fit-Wtcr-Desc	47	50	48.5	956	1 051	1 000.2
Next-Fit-Wtmr-Desc	38	40	38.7	920	1 040	978.7
Next-Fit-Wtmr2-Desc	38	43	40.1	924	1 037	971.7
Worst-Fit-Wtmr-Desc	42	56	46.2	906	1 015	964.6
Worst-Fit-Wtmr2-Desc	43	56	46	863	1 022	949.5
First-Fit-Members-Asc	25	28	26.7	831	994	909.1
Best-Fit-Members-Asc	30	64	35.4	830	992	906.2
Best-Fit-Participations-Desc	34	40	36.4	810	987	893.3
Next-Fit-Participations-Desc	41	45	42.4	823	966	891.9
First-Fit-Conflicts-Asc	32	35	33.2	815	967	890.6
Best-Fit-Conflicts-Asc	36	55	39.8	816	979	889.6
First-Fit-Participations-Desc	32	34	32.7	790	965	884.4
Worst-Fit-Participations-Desc	45	47	46	799	963	870.4
Next-Fit-Members-Asc	36	39	37.1	804	938	866.4
Worst-Fit-Members-Asc	40	52	44	785	957	860.1
Best-Fit-None	32	45	41.6	788	933	855.4
Next-Fit-None	34	43	39.7	752	934	850.7
First-Fit-None	24	26	24.7	748	924	845.1
Worst-Fit-Conflicts-Asc	46	49	47.4	759	915	841.6
Next-Fit-Conflicts-Asc	42	46	43	766	893	839.7
Worst-Fit-None	40	48	41.6	763	921	836.7
Best-Fit-Members-Desc	31	39	34.3	739	929	809.7
First-Fit-Members-Desc	27	29	27.5	729	925	806.3
Next-Fit-Members-Desc	37	40	37.7	712	920	805.9
Worst-Fit-Members-Desc	41	65	47.6	706	893	800.5
Best-Fit-Conflicts-Desc	37	47	39.6	704	848	760.1
Next-Fit-Conflicts-Desc	28	48	43.9	690	847	758.3
First-Fit-Conflicts-Desc	32	35	33.4	699	851	755.3
Worst-Fit-Conflicts-Desc	46	49	47.9	683	845	754.3
Worst-Fit-Wtcr-Asc	49	51	49.4	571	719	646

Next-Fit-Wtcr-Asc	28	100	41.7	598	718	638.1
Best-Fit-Wtcr-Asc	39	59	43.3	526	727	611.8
First-Fit-Wtcr-Asc	34	35	34.3	541	712	607.5

Tabelle A.4.: Vergleich der besten Interchange-Fill Lösungen mit Gurobi, siehe Kapitel 4.6 für eine Beschreibung der Abkürzungen

Gruppe	Algorithmus für Startlösung	Laufzeit (ms)		$w(R)$	
		avg	opt (%)	avg	opt (%)
Normal	SA-8-Gurobi	68.4	99.13%	129.2	100.00%
DenseWeight	FI-8-Gurobi	92.9	134.25%	76.0	99.87%
Small	SA-NF-Weight-Desc	7.2	51.80%	86.5	100.00%
Large	SA-8-Gurobi	272.8	43.56%	192.2	99.53%
Short	SA-WF-None	7.9	66.95%	58.0	100.00%
Long	SA-1-IF	72.2	0.54%	274.4	98.78%
Sparse	SA-8-Gurobi	43.9	151.9%	191.9	99.74%
Dense	SA-FF-Conflicts-Asc	8.5	57.05%	84.0	100.00%
Huge	SA-BF-Members-Desc	12 264.3	61.04%	1 136.4	103.11%