Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Synthesealgorithmus zur effizienten Einteilung von Software-Teams

Synthesis Algorithm for Efficient Classification of Software Teams

Bachelorarbeit

im Studiengang Informatik

von

Nils Rieke

Prüfer: Prof. Dr. rer. nat. Kurt Schneider
Zweitprüfer: Dr. rer. nat. Jil Klünder
Betreuer: M. Sc. Lukas Nagel

Hannover, 02.10.2021

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 02.10.2021

Nils Rieke

Zusammenfassung

Personen, die auf Gruppen oder andere zu belegende Plätze aufgeteilt werden müssen, stellen meist einen erhöhten Aufwand dar, wenn es darum geht, sie gleichmäßig einzuteilen. Mit steigender Anzahl von Personen und zu belegenden Plätzen wird die Anzahl der Kombinationen von möglichen Zuteilungen sehr schnell größer. Daraus ergibt sich, dass ein Algorithmus diese Einteilung deutlich besser vornehmen und dabei schneller vorgehen kann als eine vergleichbare Arbeit, die ohne solche Hilfsmittel durchgeführt wird.

Im Rahmen des alljährlich stattfindenden Software-Projekts an der Leibniz Universität Hannover, in dem Studierendengruppen eine Software entwickeln, gab es in der Vergangenheit immer hohe Teilnehmerzahlen, welche auch in künftigen Durchläufen zu erwarten sind. Aufgrund dieser Menge ist mit dem erhöhten Aufwand für die Zuteilung von Studierenden auf Projektplätze auch das Risiko verbunden, keine optimalen Lösungen für dieses Problem zu finden. Als optimale Lösung wird eine Zuteilung verstanden, die so wenig Studierenden wie möglich ein Projekt zuordnet, welches diese nicht als einen der drei Wünsche angegeben haben.

Da dieses Problem nicht neu ist, gibt es bereits mehrere Verfahren, wie es zu lösen ist, ohne dabei jede einzelne Kombination zu testen und zu bewerten – was für kleine Größen noch funktionieren kann, ab einer entsprechenden Anzahl von Personen mit den aktuell zur Verfügung stehenden Rechenkapazitäten aber nicht mehr realisierbar ist. Diese Arbeit befasst sich insbesondere mit der Ungarischen Methode, einem Zuordnungsalgorithmus, um auf dessen Grundlage ein Werkzeug zu entwickeln, das eine optimale Zuteilung findet, bei der so viele Teilnehmerinnen und Teilnehmer wie möglich in eine gewünschte Gruppe eingeteilt werden.

Abstract

Synthesis Algorithm for Efficient Classification of Software Teams

Having to distribute persons to groups or vacant slots usually costs a lot of effort when trying to assign them evenly. The more persons and slots to be filled, the more possible combinations of assignments there are. This leads to the fact that an algorithm is able to do this task much better and faster than a human could do.

The annual course *Software-Projekt* ("software project") of the Leibniz Universität Hannover in which groups of students develop a software always had many participants in the past and is also expected to have high counts of students in future iterations. Due to this quantity, the increased effort for assigning students to projects is also associated with the risk of not finding optimal solutions to this problem. An optimal solution is considered to be an assignment that assigns as few students as possible to a project they have not chosen as one of their three wishes.

As this problem isn't new, there already are methods to solve such without trying and rating every single combination – which will work for smaller amounts of students, but won't be feasible with the currently available computing power as more students need to be distributed. This thesis focuses on the Hungarian Method, an assignment algorithm, and builds a tool based on said algorithm which is designed to find an optimal solution in which as many students as possible get one of their desired groups.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Problemstellung 2
1.2 Lösungsansatz 3
1.3 Struktur der Arbeit 4

2 Grundlagen **5**
2.1 Zuordnungsprobleme 5
2.2 Grundlagen der Graphentheorie 7
2.2.1 Matchings 8
2.2.2 Hopcroft-Karp-Algorithmus 10
2.2.3 Minimale Knotenüberdeckung 11
2.3 Lineare Optimierung 12
2.3.1 Dualität 13
2.4 Die Ungarische Methode 14
2.4.1 Das einfache Zuordnungsproblem 14
2.4.2 Das allgemeine Zuordnungsproblem 15
2.4.3 Anwendung der Ungarischen Methode 15

3 Verwandte Arbeiten **19**
3.1 Betrachtung als Graph 19
3.1.1 Finden eines Maximum-Matchings 20
3.1.2 Zuordnungsproblem mit hierarchischer Ordnung und Gruppen-Beschränkungen 20
3.2 Betrachtung als Matrix 21

4 Implementierung und Entwicklung **23**
4.1 Konzept und Planung 23
4.1.1 Kombinatorik 24
4.1.2 Idee 25
4.2 Aufbau des Programms 27
4.2.1 Teil I: Erschaffung von Lösungsräumen 30
4.2.2 Teil II: Fähigkeitsbasierte Teameinteilung 32
4.3 Korrektheit und Laufzeit 35

4.4 Graphische Benutzeroberfläche . . . 36
4.4.1 Konzept und Entwurf . . . 36
4.4.2 Beschreibung und Erklärung . . . 38
4.4.3 Fehlerbehandlung und Überprüfung der Eingaben . . . 40

5 Tests und Evaluierung 43

6 Fazit und Ausblick 47
6.1 Zusammenfassung . . . 47
6.2 Ausblick . . . 48

A Pseudoalgorithmen 51

B Grafiken und Bilder 55

Kapitel 1

Einleitung

Das Software-Projekt ist eine der zentralsten Veranstaltungen des Informatik-Bachelor-Studiums an der Leibniz Universität Hannover, da viele Studierende hier erstmals mit einer Erfahrung konfrontiert werden, die einem Projekt in einem echten Betrieb oder Unternehmen ähnelt. Sie „organisieren sich in kleinen Teams, legen Rollen und Verantwortlichkeiten fest und führen die nötigen Arbeiten aus“ [30, Abs. 1].

Die Studierenden, die im letzten Jahr ihres Studiums stehen, entwickeln dabei eine Software in Gruppen aus drei bis fünf Mitgliedern. Über den Zeitraum von 15 Wochen während des Wintersemesters müssen sie ihr Projekt und sich als Team selbst organisieren und zum Beispiel auch die Treffen mit dem Kunden oder der Kundin ihrer Software koordinieren. Dieser oder diese ist normalerweise ein Mitarbeiter oder eine Mitarbeiterin des Fachgebiets Software Engineering, damit der Aufwand für die Studierenden in einem angemessenen Rahmen gehalten wird, aber dennoch einen entsprechenden Lerneffekt mit sich bringt. Weiterhin hat jede Gruppe noch einen Team-Coach, der dem Team bei sozialen Problemen weiterhelfen kann, aber keine technischen Fragen beantwortet und ebenso nicht zwischen Kunde und Gruppe vermittelt [18].

Darüber hinaus können essentielle Eindrücke wie Teamarbeit, agile Softwareentwicklung und Kundenkontakt, aber vor allem praktische Erfahrungen vermittelt werden, welche in den meisten anderen, eher theoretischen Veranstaltungen des Studiums, oft nicht oder nur eingeschränkt praktisch erklärt werden können.

Für alle Beteiligten, also Studierende, Kunden sowie das Fachgebiet Software Engineering ist es von größtem Interesse, möglichst vielen Studierenden einen ihrer drei Wünsche zuzuteilen, um das gesamte Arbeitsklima in den Gruppen zu erleichtern und auch den Lerneffekt zu verstärken. Chuang et al. [6, S. 11f.] haben in einer Studie untersucht, wie sich Zufriedenheit auf den Lerneffekt auswirkt und haben hier einen positiven Zusammenhang entdeckt.

Des Weiteren ist auch die Reputation des Fachgebiets und auch der

Universität an sich ein Faktor: Trotz der Tatsache, dass die Kunden und Kundinnen normalerweise von der Fachgruppe kommen, ist es möglich, dass externe Personen, wie zum Beispiel die Polizei, diese Möglichkeit nutzen, mit einem Software-Team zusammenzuarbeiten. Das läuft dementsprechend reibungsloser ab, wenn die Studierenden sich auch für das Thema interessieren und Spaß an der Aufgabe haben.

Um eine entsprechend hohe Zufriedenheit zu erlangen, ist eine zufällige Verteilung von Studierenden auf Gruppenplätze keine Option, weswegen diese Problematik ein typisches, gewichtetes Zuordnungsproblem darstellt. Befindet sich in einem Durchlauf der Veranstaltung beispielsweise mindestens ein Projekt, das von besonders vielen Studierenden gewählt wird, ist die Aufgabe, eine Verteilung mit hoher Zufriedenheit zu erreichen, noch schwieriger zu lösen, als es bei einer gleichmäßigen Verteilung der Projektwünsche der Fall wäre.

Daher bietet es sich an, einen Algorithmus diese Arbeit übernehmen und ihn eine Lösung finden zu lassen, die im besten Fall keine unzufriedenen Studierenden zurücklässt. Einer dieser Algorithmen ist die Ungarische Methode, welche bei dieser Art von Problemen eine optimale Lösung finden kann, also eine, die so vielen Studierenden wie möglich eines ihrer drei Wünsche zuordnet.

1.1 Problemstellung

In den vergangenen Durchläufen der Veranstaltung gab es keinen Lösungsweg, der konstant verfolgt wurde, weswegen das Problem der Verteilung von Studierenden auf Projektplätze bisher auf verschiedene Arten gelöst wurde.

Die *erste Möglichkeit* ist es, die Studierenden sich über die Online-Plattform „Stud.IP“ eigenständig organisieren zu lassen. Das hat aber dazu geführt, dass sehr viele Studierende auf einmal versucht haben, sich in die begrenzten Gruppenplätze einzutragen und dementsprechend nicht jeder und jede eine Wunschgruppe erhalten hat.

Bei diesem Lösungsansatz birgt der geringere Aufwand für die Organisierenden auch mehr Risiko: Hier findet die Einteilung auf Seiten der Studierenden statt und hinterlässt möglicherweise eine allgemeine Unzufriedenheit, weil sich die Gruppen bei beliebten Projekten innerhalb weniger Sekunden füllen. Eine Studentin oder ein Student, die oder der nicht ihren oder seinen Erstwunsch bekommt, ist vielleicht verärgert, schafft es deswegen möglicherweise auch nicht, eine vorher überlegte Alternative zu wählen und muss sich in eine unerwünschte Gruppe einteilen.

Studierende können im Anschluss an die eigenständige Gruppeneinteilung die Chance wahrnehmen, mit Kommilitoninnen und Kommilitonen ihren Platz zu tauschen, wenn beide Beteiligte einverstanden sind. Das erhöht die gesamte Zufriedenheit der Gruppenwahlen ein wenig, bedeutet aber einen

organisatorischen Mehraufwand, um die Einteilung der Stud.IP-Gruppen abzuändern.

Die *zweite Möglichkeit* bedient sich einer Anmeldeseite, auf der die Studierenden Projektwünsche und Fähigkeiten wie Teamfähigkeit oder ihre Erfahrung mit bestimmten Programmiersprachen angeben. Diese Daten müssen anschließend durch die Veranstalter verarbeitet werden, was mit Durchsehen, Sortieren und Zuordnen nach Aussage von Organisierenden im Fall des Software-Projekts einige Stunden in Anspruch nehmen kann. Dieses Vorgehen stellt eine bessere Methode dar, eine anfängliche und zu erwartende Unzufriedenheit der Studierenden zu verhindern, da die Entscheidung über die Projektwahl erst später getroffen wird. Allerdings ist sie mit einem größeren Aufwand für die Veranstalter verbunden, weil hier ohne zusätzliche Unterstützung ein Zuordnungsproblem gelöst werden muss, das viele Personen verteilen muss – so waren es 2012 und 2013 insgesamt 165 Studierende, die am Software-Projekt teilgenommen haben [18].

Beide Ansätze haben allerdings das große Problem, dass sie unregelmäßig arbeiten und so eher eine optimale Lösung verpassen. In *Möglichkeit 1* kommen nur subjektive Wünsche der Studierenden zum Tragen, weswegen einige (entsprechend der Größe der Gruppen) die für sie beste Einteilung erhalten, andere aber unerwünschte Gruppen erhalten. Je nach Beliebtheit eines bestimmten Projekts gibt es dann mehr oder weniger Studierende, die keinen ihrer Wünsche bekommen. *Möglichkeit 2* ermöglicht eine Gesamtübersicht über die Wünsche und Daten, und kann theoretisch eine optimale Lösung finden. Bei größeren Teilnehmerzahlen ist dies jedoch eine sehr komplexe Aufgabe, wodurch auch hier eine solche Lösung leicht übersehen werden kann.

1.2 Lösungsansatz

Geht man im Software-Projekt nach den Ausführungen von Klünder et. al. [18] von hohen zweistelligen oder niedrigen dreistelligen Teilnehmerzahlen aus, ist es nicht möglich, alle Kombinationen der Verteilung einzeln zu überprüfen, aufgrund der sehr hohen Anzahl an Kombinationsmöglichkeiten.

Aus diesem Grund ist es notwendig, einen Algorithmus anzuwenden, der immer mindestens eine optimale Lösung findet. Diese Arbeit beschäftigt sich daher mit der Ungarischen Methode, ein Verfahren, das eine solche Lösung finden kann. Auf der Grundlage dieses Zuordnungsalgorithmus wird ein Werkzeug entwickelt, das sowohl die Ungarische Methode als auch eine Gruppen-Bewertungsfunktion verwendet, um zunächst Studierende anhand ihrer Wünsche auf Projekte zuzuteilen und anschließend die entstandenen Projektgruppen entsprechend der Fähigkeiten der Studierenden so anzuordnen, dass mehrere Gruppen eines Projekts möglichst ausgeglichen sind.

Diese Zuordnung findet als Anschluss an ein extern verwaltetes An-

meldewerkzeug statt, auf dem die Studierenden ihre Projektwünsche und Fähigkeiten angeben können. Daraus ergibt sich eine Datei, die dann die Grundlage für die kommenden Verarbeitungen in dieser Arbeit bildet.

Je nach Starteingabe findet die Ungarische Methode möglicherweise eine andere Verteilung der Studierenden, die für sich aber dennoch optimal ist. Um also weitere Verteilungen zu finden, gewichtet das im Rahmen dieser Arbeit entwickelte Tool die Projektwünsche der Studierenden mittels mehrerer Durchläufe zusätzlich noch unterschiedlich und ermöglicht dadurch eine breite Übersicht an möglichen, optimalen Lösungen.

Auf diese Art kann eine Lösung gefunden werden, die schneller berechnet wurde, als es bisher der Fall war.

1.3 Struktur der Arbeit

Der Rest dieser Arbeit ist folgendermaßen aufgebaut: Kapitel 2 befasst sich mit den nötigen Grundlagen der Graphentheorie, der Ungarischen Methode und der linearen Programmierung, mithilfe derer der Zuordnungsalgorithmus seinerzeit bewiesen wurde. Die Grundlagen der Graphentheorie bilden die Basis für einige Algorithmen des im Rahmen dieser Arbeit entwickelten Tools. Kapitel 3 listet verwandte Arbeiten auf, die sich mit dem Thema Zuordnung befassen und beschreibt Varianten, wie diese Probleme noch gelöst werden können. In Kapitel 4 wird in Abschnitt 4.1 zunächst eine Übersicht über die Grundidee und Planungen gegeben, mit deren Hilfe dann im weiteren Verlauf des Kapitels die Implementierung der Algorithmen sowie die graphische Benutzeroberfläche im Detail beschrieben und erläutert wird. Kapitel 5 stellt kleine Testläufe dar und wertet sie tabellarisch aus. Am Ende findet sich noch das Abschlussfazit.

Kapitel 2

Grundlagen

Zuordnungsprobleme beinhalten häufig viele weitere Strukturen, wie graphische Veranschaulichungen und Algorithmen oder Optimierungsverfahren wie die lineare Optimierung [19]. Wegen dieser Vielfältigkeit befasst sich dieses Kapitel mit ebendiesen Grundlagen und versucht das Wissen zu vermitteln, das viele Zuordnungsalgorithmen voraussetzen.

2.1 Zuordnungsprobleme

David W. Pentico [33] schreibt über Zuordnungsprobleme, dass sie schon lange und vor allem in vielen verschiedenen Ausprägungen und Spezialisierungen existieren. Er datiert den Ursprung dieser auf 1955 durch Kuhn [21] mit seiner Lösung für das klassische Zuordnungsproblem, auch wenn sie bereits 1952 in den Ausführungen von Votaw und Orden [36] genannt wurden. In den vergangenen Jahren wurden viele weitere Varianten dieses klassischen Zuordnungsproblems beschrieben und vorgestellt [33]. Durch diese vielen Anwendungsgebiete sind sie auch im Alltag einsetzbar, können also – wie im vorliegenden Problem dieser Arbeit – ebenfalls in Bildungseinrichtungen auftreten. Das liegt daran, dass Personen, wenn sie verschiedene Arbeiten zu erledigen haben, oftmals auf diese Arbeiten aufgeteilt werden müssen, unabhängig davon, ob es sich um Gruppenarbeiten handelt oder um Einzelarbeiten.

Im Folgenden zeigt ein Beispiel, wie ein solches Problem aussehen kann. Hierbei sollen drei Angestellte eines Unternehmens drei Aufgaben übernehmen, für die sie allerdings unterschiedlich geeignet sind. Jede Person kann nur eine Arbeit übernehmen. 0 bedeutet hier, dass die Person für diese Tätigkeit ungeeignet ist, 3 stellt eine hohe Eignung dar, mit der das Unternehmen den höchsten Gewinn erzielt. Aus Unternehmenssicht soll hier der Gewinn maximiert werden, was im besten Fall eine Summe beziehungsweise einen Gewinn von 3 Personen $\cdot$ 3 Geldeinheiten $=$ 9 Geldeinheiten bedeuten würde.

Person	Aufgabe I	Aufgabe II	Aufgabe III
A	0	1	3
B	3	2	1
C	3	2	0

Tabelle 2.1: Eignung von Personen für Aufgaben

Für einfache und/oder kleine Ausgangssituationen wie diese ist eine Lösung schnell zu finden. Eine dieser möglichen und optimalen Zuordnungen ist in Tabelle 2.1 unterstrichen und rot eingefärbt angegeben. Eine andere Möglichkeit wäre es, die Aufgaben von B und C zu tauschen.

Für die Übersichtlichkeit wurden die Zuordnungen in folgende Abbildung übertragen und diejenigen, die nicht in der oben präsentierten Lösung enthalten sind, entfernt.

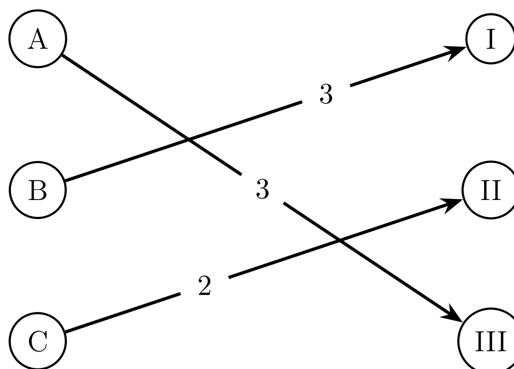

Abbildung 2.1: Optimale Zuordnung mit Eignungen für das obige Beispiel

Es fällt auf, dass diese Abbildung ein Graph ist, im Gegensatz zu der tabellarischen Form in Tabelle 2.1. Der Wechsel der Form ist beabsichtigt, denn diese graphische Interpretierung zeigt auf, dass Zuordnungsprobleme nicht an eine Darstellung gebunden sind und eine Form annehmen können, die zum Bearbeiten oder Verstehen besser geeignet ist [3].

Wird die Aufgabenstellung allerdings größer und beinhaltet mehr Personen oder Aufgaben, ist eine Einteilung ohne zusätzliche Hilfsmittel eine sehr komplexe Aufgabe, die irgendwann unlösbar wird. Algorithmen können daher dann nur auf effiziente Weise eine Lösung finden, wenn sie nicht jede Kombination überprüfen, sondern die (optimale) Lösung auf eine andere Berechnungsweise ermitteln.

An dem Beispiel in Tabelle 2.1 lässt sich außerdem erkennen, dass Zuordnungsprobleme oft auch Optimierungsprobleme sind. Nicht selten wird versucht, Gewinne zu maximieren, Verluste zu minimieren oder Eignungen/Wünsche zu berücksichtigen, die eine Zuteilung komplexer machen.

Der Vollständigkeit halber seien an dieser Stelle auch noch solche Zu-

ordnungsprobleme erwähnt, die ohne Gewichtung der möglichen Paarungen auskommen und nur mittels der Frage „Ist diese Paarung erlaubt?“ eine (optimale) Lösung suchen. Ob eine Paarung erlaubt ist, ist dann nur eine Frage technischer Natur („Ist es praktisch möglich, dass Person A Aufgabe B übernimmt?“) und involviert keinen optimierenden Faktor. Diese Probleme sind oftmals leichter zu lösen, da der Optimierungsaspekt hier wegfällt und eine gefundene Lösung, die alle Personen einteilt, immer auch eine optimale ist – bei gewichteten Paarungen gäbe es möglicherweise eine bessere Kombination mit höherer oder niedrigerer Gesamtsumme.

Zuordnungsprobleme können auf verschiedene Weisen dargestellt und behandelt werden [19, 3]. Eine von ihnen ist eine Betrachtung als Matrix, deren Zeilen und Spalten die zu verbindenden Paare bilden. Die Einträge dieser Matrix entsprechen entweder der Priorisierung (bei gewichteten Zuordnungsproblemen) oder der Gültigkeit dieser Paarung (bei ungewichteten Zuordnungsproblemen). Eine andere Betrachtungsweise ist die oben angeführte Betrachtung als Graph, bei dem die beiden Knotengruppen die Paare und die Kanten die Zuteilungen bilden.

2.2 Grundlagen der Graphentheorie

Wie zuvor erwähnt ist es möglich, Zuordnungsprobleme als (bipartiten) Graph darzustellen. Im Folgenden finden sich einige Definitionen, die insbesondere in der späteren Implementierung Anwendung finden, die sich teilweise auf die graphische Betrachtungsweise der Probleme stützt.

Definition 1 ([9]) *Ein Graph $G = (V, E)$ besteht aus einer Menge aus Elementen,* Knoten *genannt, und einer Menge aus Elementen,* Kanten *genannt, sodass jede Kante genau zwei Knoten* besucht*, diese werden die* Endpunkte *der Kante genannt. Eine Kante* verbindet *die Endpunkte.*

Ein bipartiter, gewichteter Graph ist ein spezieller Graph, für den einige Besonderheiten gelten, die in der folgenden Definition genannt werden.

Definition 2 ([35]) *Ein bipartiter, gewichteter Graph $G = (X, Y, E)$ besteht aus zwei Partitionen von Knoten [...], bei dem eine Kante nur Knoten aus beiden Gruppen verbindet (also $E \subseteq X \times Y$). Ein Gewicht $w(e)$ ist jeder Kante aus E zugeordnet.*

Die Abbildung 2.2 zeigt ein Beispiel für einen bipartiten Graphen, dessen linke Knoten den einen Teil der Paarungen (z. B. Studierende) und dessen rechte Knoten den anderen Teil (z. B. Projektplätze) darstellen. Eine Kante bedeutet hier eine erlaubte Zuordnung, welche ein bestimmtes Gewicht hat. Es ist also in diesem Beispiel zulässig, *L1 R1* oder *R2* zuzuordnen, *L3* darf aber nicht *R3* zugeordnet werden.

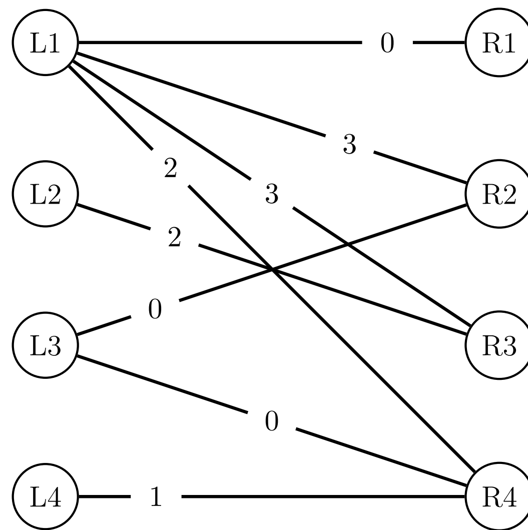

Abbildung 2.2: Bipartiter Graph mit gewichteten Kanten

2.2.1 Matchings

Zuordnungen befassen sich, wenn man sie graphisch betrachtet, mit den Kanten eines Graphen, da diese die erlaubten Einteilungen darstellen. Aus diesem Grund werden nun Matchings definiert, die die Eigenschaft haben, je zwei Knoten durch exakt eine Kante zu verbinden.

Definition 3 ([9]) *Ein Matching M in G ist eine Teilmenge der Kanten aus G, sodass keine zwei Kanten denselben Knoten besuchen.*

Matchings existieren in jedem Graph mit mindestens einer Kante. Weil Zuordnungen allerdings immer eine Teilgruppe auf eine andere Teilgruppe abbilden, die jeweils untereinander keine Verbindungen haben, wird sich im Folgenden nur noch auf bipartite Graphen beschränkt.

Um Matchings im Zusammenhang mit einer optimalen Zuordnung besser verstehen zu können, müssen weitere Definitionen geklärt werden, welche die Begriffe *perfektes Matching*, *alternierender* und *augmentierender Weg* sowie das *Maximum-Matching* einführen.

Definition 4 ([7]) *Ein Matching M in $G = (V, E)$ ist eine Teilmenge der Kanten und* perfekt *genau dann, wenn jeder Knoten aus V von genau einer Kante aus $M \subseteq E$ besucht wird.*

Definition 5 ([20]) *Sei M ein Matching. Ein Weg $P = (v_0, e_l, v_1, ..., e_k, v_k)$ mit $k \geq 0$ und $v_i \in V(G)$, $e_i \in E(G)$, $0 \leq i \leq k$ in G heißt* alternierender *Weg, wenn die Kanten in P abwechselnd in und nicht in M liegen. Falls beide Endpunkte von P von keiner Kante aus M besucht werden, so nennen wir P einen* augmentierenden *Weg.*

Satz 6 ([1]) *Ein Matching M ist ein* Maximum-Matching *genau dann, wenn es in G keinen augmentierenden Weg gibt.*

Zuordnungen (oder die Algorithmen, die sie lösen, wenn sie mit der graphischen Sichtweise entwickelt wurden) streben ein *perfektes Matching* an, wenn die Anzahl linker und rechter Knoten gleich ist, und streben ein *Maximum-Matching* an, wenn die Anzahl nicht gleich ist – das hängt damit zusammen, dass ein perfektes Matching per Definition im zweiten Fall unmöglich zu finden ist, da immer mindestens ein Knoten übrig bleibt. Beide Matchings finden die größtmögliche Anzahl an Kanten in einem Graphen, dementsprechend also die Zuteilung mit den meisten Paarungen. Sollte eine Aufgabenstellung explizit auf eine Zuteilung ausgelegt sein, die nicht alle Personen oder Aufgaben einteilt, ist ein anderes Verfahren zum Finden dieser notwendig.

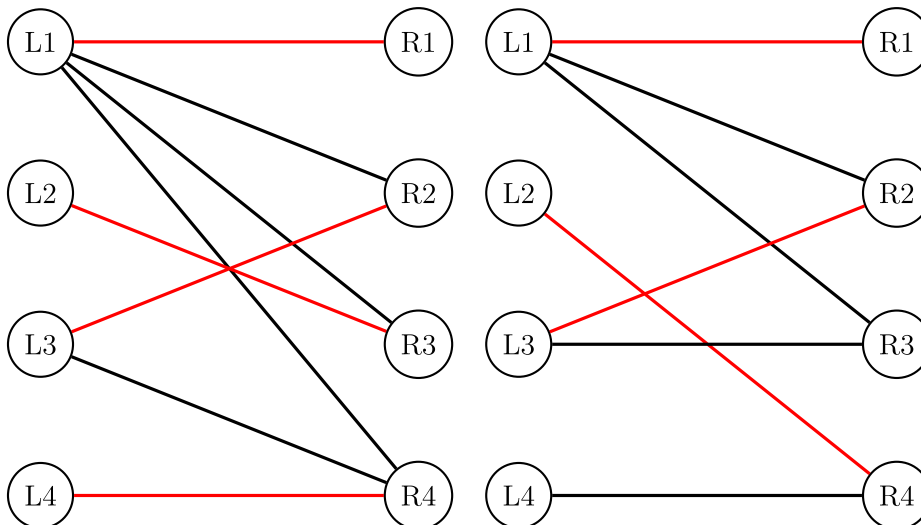

Abbildung 2.3: Perfektes Matching Abbildung 2.4: Maximum-Matching

Abbildungen 2.3 und 2.4 zeigen jeweils ein Matching, bei dem die im Matching enthaltenen Kanten rot eingefärbt dargestellt werden. Abbildung 2.3 beschreibt ein perfektes Matching, also eines, bei dem jeder Knoten von einer Matching-Kante besucht wird. In Abbildung 2.4 ist ein Maximum-Matching zu sehen. Diese Matchings, die auch *größtmögliche Matchings* genannt werden, besitzen die höchste Kardinalität (= das Matching hat die meisten Kanten) aller Matchings. Anders gesagt gibt es kein Matching in dem Graphen, das mehr Matching-Kanten enthält.

Während dies noch kleine Beispiele sind, bei denen ein perfektes oder Maximum-Matching leicht bestimmt werden kann, ist das bei größeren Problemstellungen nicht länger möglich. Es gibt aber Algorithmen, die in der Lage sind, Matchings in großen Graphen zu finden. Ein Beispiel hierfür

ist der Hopcroft-Karp-Algorithmus. Da in dem im Rahmen dieser Arbeit entwickelten Programm als einer der zentralsten Schritte ein Maximum-Matching sowie eine minimale Knotenüberdeckung gefunden werden muss, sind im Folgenden dieser Algorithmus und die Knotenüberdeckung eines Graphen erklärt, welche im Programm Anwendung finden.

2.2.2 Hopcroft-Karp-Algorithmus

Der Hopcroft-Karp-Algorithmus berechnet in einem bipartiten Graphen ein Maximum-Matching. Der Algorithmus wurde 1973 von John Edward Hopcroft und Richard Manning Karp [15] entwickelt und besitzt eine Zeitkomplexität von $O((|E| + |V|)\sqrt{|V|})$.

Der Algorithmus nutzt das Prinzip der augmentierenden Pfade, um ein Maximum-Matching zu finden. Wird ein augmentierender Pfad gefunden, können die in ihm enthaltenen Matching-Kanten aus dem Matching entfernt, und nicht enthaltene diesem hinzugefügt werden. Hierdurch erhält man ein Matching mit einer zusätzlichen Kante, es wurde in seiner Kardinalität also um eins erhöht. In jedem Iterationsschritt des Algorithmus werden zusätzlich noch weitere augmentierende Pfade gesucht, um Zeit einzusparen. Hopcroft und Karp [15] erwähnen dies auch, denn sie sagen über vorherige Methoden aus, dass diese $O(mn)$ Schritte benötigen (m ist die Anzahl der Kanten, n die Anzahl der Knoten). Der Hopcroft-Karp-Algorithmus benötigt nur $O((m + n)\sqrt{n})$ Schritte [15, S. 225].

Auf diese Weise werden nach und nach längere augmentierende Pfade gefunden, bis kein längerer mehr existiert. An diesem Punkt ist nach Satz 6 ein größtmögliches Matching gefunden worden.

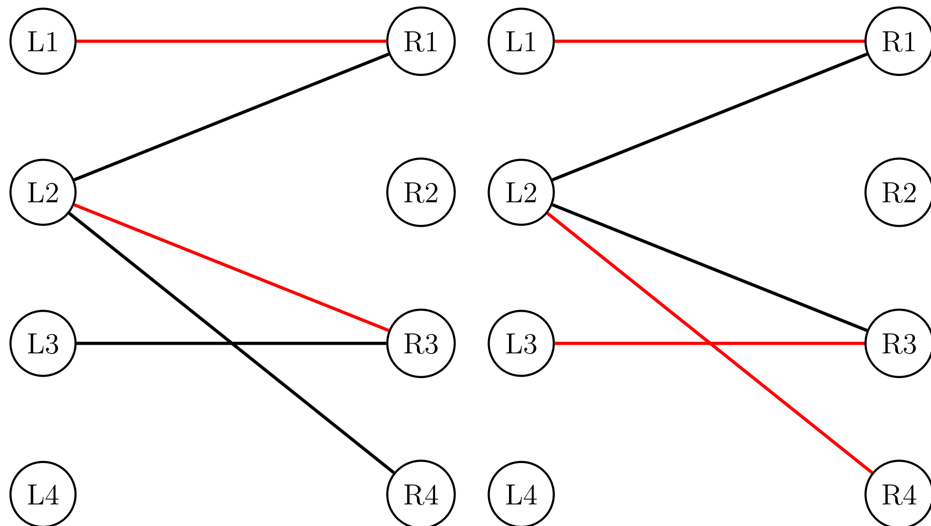

Abbildung 2.5: Iteration 1 Abbildung 2.6: Iteration 2

In den Abbildungen 2.5 und 2.6 sind die zwei Iterationen gezeigt, die der Algorithmus durchläuft, um ein Maximum-Matching in diesem Graphen zu finden. Zunächst werden alle augmentierenden Pfade der kürzesten Länge (hier der Länge 1) gesucht und die jeweilige Kante in das Matching M aufgenommen. Nach Prüfung auf längere augmentierende Pfade kann ein solcher mit der Länge 3 gefunden werden ($L3 \rightarrow R3 \rightarrow L2 \rightarrow R4$). Der Algorithmus ist daher noch nicht beendet und startet Iteration 2. Die in M enthaltenen Kanten dieses Pfades werden aus M entfernt und die nicht enthaltenen werden hinzugefügt. Daraus ergibt sich ein um eins größeres Matching, das auch das finale Matching ist, da kein anderer Pfad der Länge 3 oder höher gefunden werden kann. M ist in diesem Beispiel also $\{(L1, R1), (L2, R4), (L3, R3)\}$.

2.2.3 Minimale Knotenüberdeckung

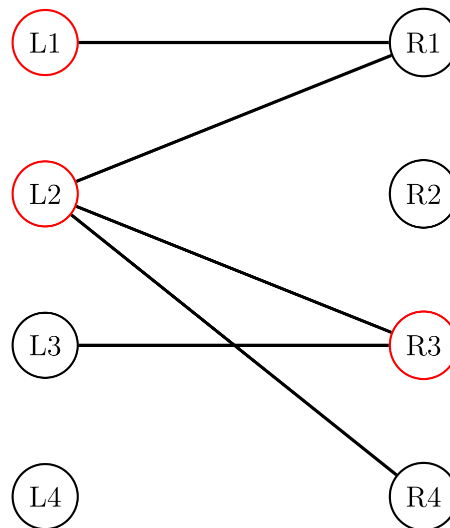

Abbildung 2.7: Minimale Knotenüberdeckung

Die minimale Knotenüberdeckung (*englisch*: minimum vertex cover) ist eine spezielle Form der Knotenüberdeckung in einem Graphen. Das Finden einer solchen ist ein NP-schweres Problem, damit also nicht in Polynomialzeit lösbar [4]. Zunächst sind erneut die Definitionen zu klären:

Definition 7 ([4]) *Eine Knotenmenge $X \subseteq V$ eines Graphen $G = (V, E)$ ist eine* Knotenüberdeckung *genau dann, wenn jede Kante aus E wenigstens einen Knoten aus X besucht.*

Definition 8 ([16]) *Das minimale Knotenüberdeckungsproblem eines ungerichteten Graphen $G = (V, E)$, bei dem V die Knotenmenge und E die Kantenmenge beschreibt, besteht darin, die kleinste Teilmenge $V' \subseteq V$ zu finden, sodass $\forall \langle i, j \rangle \in E$ entweder $i \in V'$ oder $j \in V'$ (oder beide) gilt. V' ist dann eine Knotenüberdeckung.*

Zusätzlich sei noch der Satz von König genannt, der nach dem ungarischen Mathematiker Dénes Kőnig benannt ist:

Satz 9 ([2] (Seite 74, Theorem 5.3)) *In einem bipartiten Graph ist die Anzahl der Kanten eines Maximum-Matchings gleich der Anzahl der Knoten einer minimalen Knotenüberdeckung.*

In Abbildung 2.7 wird der Zusammenhang mit einem Maximum-Matching (siehe Abbildung 2.6) sichtbar, denn jeder Knoten der minimalen Knotenüberdeckung ist einer der beiden Endknoten einer Matching-Kante.

2.3 Lineare Optimierung

Die lineare Optimierung, die auch als lineare Programmierung bezeichnet wird, ist eine Möglichkeit zum Lösen einer Problemstellung, die eine klare Haupt- und mehrere Nebenbedingungen formuliert, bei der ein Optimum gefunden werden soll. In diesem Zusammenhang existiert auch der Begriff LP für Lineares Programm, welches die Bezeichnung für ein solches Problem ist. Im Folgenden ist ein Beispiel gegeben, das die Grundidee der linearen Optimierung verdeutlicht. Es ist entnommen aus dem Buch „Mathematische Methoden für Ökonomen“ von Mosler et al. [24].

Ein Betrieb kann zwei Produkte $P1$ und $P2$ aus den Zutaten $Z1$, $Z2$ und $Z3$ herstellen. Pro verkauftem Produkt erhält das Unternehmen 50 ($P1$) respektive 33 ($P2$) Geldeinheiten. Für die Herstellung eines der beiden Produkte werden jeweils unterschiedliche Mengen an Zutaten benötigt, die in Tabelle 2.2 aufgelistet werden. Außerdem stehen dem Betrieb nur eine begrenzte Anzahl der Zutaten zur Verfügung.

	Z1	Z2	Z3	Umsatz
P1	5	3	3	50
P2	10	3	1	33
maximale Verfügbarkeit	100	36	30	

Tabelle 2.2: Verbrauch und Vorrat der Zutaten

Das Problem formuliert sich nun wie folgt, wobei x_1 und x_2 die Mengen der Produkte $P1$ und $P2$ seien:

Maximiere (max)	$50x_1 + 33x_2$
Nebenbedingungen (NB)	$5x_1 + 10x_2 \leq 100,$
	$3x_1 + 3x_2 \leq 36,$
	$3x_1 + x_2 \leq 30,$
	$x_1, x_2 \geq 0$

Abbildung 2.8: Problem der linearen Optimierung

2.3.1 Dualität

„Zu jedem LP gehört ein zweites LP, das als das duale LP des ursprünglichen LP bezeichnet wird“ [24, S. 379]. Weiterhin ist ein sogenanntes kanonisches Problem ein Problem, bei dem alle Nebenbedingungen Ungleichungen der $\leq$ -Art (Maximierungsprobleme) oder der $\geq$ -Art (Minimierungsprobleme) sind und darüber hinaus alle Variablen positiv sind [24]. Beispielsweise besitzt ein Maximierungsproblem kanonischer Form ein Minimierungsproblem kanonischer Form, welches das duale LP des ersten Problems, dem primalen LP, ist.

max	cx
NB	$Ax \leq b,$
	$x \geq 0$

Abbildung 2.9: Primales LP

min	yb
NB	$yA \geq c,$
	$y \geq 0$

Abbildung 2.10: Duales LP

Abbildungen 2.9 und 2.10 zeigen die beiden zueinander gehörenden Linearen Programme, wobei c der Zielvektor für die Zielfunktion des primalen LPs, b der Vektor für die rechte Seite der Nebenbedingungen (NBs) und A die Matrix für die einzelnen Summanden der linken Seite der NBs ist. Die unterschiedlichen Reihenfolgen der Faktoren innerhalb der beiden LPs begründen sich durch die fehlende Kommutativität bei Matrizenmultiplikationen.

2.4 Die Ungarische Methode

Die Ungarische Methode wurde von Harold William Kuhn im Jahr 1955 entwickelt und beruht auf der Arbeit von D. König [22] und E. Egerváry [11], zwei ungarischen Mathematikern. Sie wurde 1957 von James Munkres [27] untersucht, was zu der Erkenntnis geführt hat, dass sie in polynomieller Zeit läuft, weswegen sie auch unter dem Namen Kuhn-Munkres-Algorithmus bekannt ist. Bei dieser Methode wird eine Matrix erstellt, die mithilfe der Quellen (also der zu verteilenden Entitäten) und der Ziel-Entitäten generiert wird, über die ein Algorithmus daraufhin die optimale Lösung von Quellen zugeteilt auf Ziele findet. Munkres beschreibt den Algorithmus in seiner Arbeit als angemessen effizient und weiterhin als Möglichkeit, eine optimale Zuteilung zu finden [27, S. 32].

Die Ungarische Methode besitzt nach Verbesserungen durch Edmonds, Karp [10] und Tomizawa [34] eine Laufzeit von $O(n^3)$ [26]. Zuvor hatte der Algorithmus noch eine Zeitkomplexität von $O(n^4)$ [26].

Kuhn führt in seinen Erkenntnissen zwei Ausprägungen des Zuordnungsproblems aus: das einfache und das allgemeine Zuordnungsproblem. Ersteres ist eine Herangehensweise an das Problem, die noch keine Präferenzen berücksichtigt, weswegen dieses Problem an dieser Stelle nur kurz beschrieben sei. Erst zweiteres berücksichtigt Gewichtungen an den Zuordnungen.

2.4.1 Das einfache Zuordnungsproblem

Das einfache Zuordnungsproblem, das überwiegend auf den Ergebnissen von König [22] basiert, geht der Frage nach, wie man n Personen auf n Jobs verteilt, bei denen die Personen aber nur für bestimmte Jobs qualifiziert sind.

Mathematisch betrachtet nutzt man diese Voraussetzung, dessen Veranschaulichung an Kuhn [21] angelehnt ist:

$$\text{Person} \begin{cases} 1 \\ 2 \\ 3 \\ 4 \end{cases} \text{ist qualifiziert für Job} \begin{cases} 1,\ 2 \text{ und } 3 \\ 3 \text{ und } 4 \\ 4 \\ 4 \end{cases}$$

Aus dieser Gegebenheit wird nun folgende Matrix erstellt, bei der die Zeilen die Personen und die Spalten die Jobs darstellen. Die Matrixeinträge geben jeweils an, ob eine Person für einen Job geeignet (1) oder nicht geeignet ist (0):

$$Q = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ein Algorithmus, der das einfache Zuordnungsproblem lösen soll, würde jetzt die größtmögliche Anzahl Einsen markieren, derart, dass in keiner Zeile und Spalte zwei Einsen markiert sind. Das geschieht nach Kuhns Idee durch

einen sogenannten Transfer. Eine vollständige Lösung, also eine, bei der keine weitere Zuteilung mehr gefunden werden kann, permutiert ihre Zuteilungen und kann auf diese Weise Lücken erschaffen, sodass weitere Zuteilungen möglich sind [21].

2.4.2 Das allgemeine Zuordnungsproblem

Das allgemeine Zuordnungsproblem führt diesen Gedanken nun weiter und baut in weiten Teilen auf den Erkenntnissen von Egerváry [11] auf. Hierbei werden nicht mehr nur die Qualifizierungen selbst untersucht, sondern es wird auch versucht, die „Bewertung“ einer solchen Zuteilung zu maximieren. Diese Art der Zuordnung entspricht in vielerlei Hinsicht dem Problem, welchem sich auch diese Arbeit widmet.

Mathematisch gesprochen wird also wieder eine Zuteilung aus n Personen $(i = 1, \dots, n)$ gesucht, die auf n Jobs $(j = 1, \dots, n)$ aufgeteilt werden, diesmal allerdings unter Berücksichtigung einer Bewertungsmatrix $R = (r_{ij})$ mit $r_{ij} \geq 0$ für alle i, j. Eine Zuteilung wird definiert als die Wahl eines Jobs j_i für eine Person i, sodass kein Job zwei verschiedenen Personen zugeteilt wird. Die Zuteilung kann also als Permutation der Zahlen $1, 2, \dots, n$ verstanden werden:

$$\begin{pmatrix} 1 & 2 & \dots & n \\ j_1 & j_2 & \dots & j_n \end{pmatrix}$$

Aufgabe des allgemeinen Zuordnungsproblems ist es nun, die folgende Frage zu lösen: Für welche dieser Zuteilungen ist die Summe $r_{1j_1} + r_{2j_2} + \cdots + r_{nj_n}$ der Bewertungen maximal?

2.4.3 Anwendung der Ungarischen Methode

Die Ausführung des Algorithmus gliedert sich in vier große Schritte [13]. Mit einem Beispiel sind diese hier aufgeführt. Die Schritte orientieren sich in Erklärung und Notation teilweise an den Ausführungen von Munkres [27]. Der Pseudocode findet sich unter Algorithmus 1 in Anhang A. Der Eintrag an der Stelle $[i, j]$ der Matrix entspricht der Gewichtung der Zuordnung von i zu j.

Die Einträge im folgenden Beispiel sind an die Werte angelehnt, die auch das im Rahmen dieser Arbeit verwendete Tool bei der Verwendung der Ungarischen Methode benutzt. Sie stellen daher die Prioritäten der Studierenden bei der Projektwahl dar. Um jedem Studenten und jeder Studentin einen seiner oder ihrer Wünsche zuteilen zu können, wird hier minimiert: 1, 2 oder 3 stehen für den Erst-, Zweit- oder Drittwunsch der Studierenden, 6 ist die Priorität eines unerwünschten Projekts, um sicherzustellen, dass dieses nur in der optimalen Lösung vorkommt, wenn keine andere Möglichkeit existiert.

Schritt 0: Erstelle eine quadratische $(n \times n)$-Matrix aus der Problemstellung. Zeilen entsprechen hier Studierenden, Spalten sind die verfügbaren Projektplätze. Ist die Anzahl von Studierenden und Projektplätzen nicht identisch, fülle die Matrix mit Platzhalterzeilen oder -spalten auf, um sie quadratisch zu machen. Diese Platzhalter sollten mit mindestens dem höchsten Wert in der gesamten Matrix gefüllt werden, um sicherzustellen, dass diese nur ausgewählt werden, wenn keine andere Möglichkeit besteht.

$$\begin{bmatrix} 6 & 3 & 3 & 2 \\ 6 & 2 & 3 & 1 \\ 1 & 2 & 3 & 6 \\ 2 & 2 & 2 & 3 \end{bmatrix}$$

Abbildung 2.11: Erstellung einer (4×4)-Matrix mit Prioritäten

Schritt 1: Ermittle für jede Zeile jeweils das kleinste Element und subtrahiere es von allen Elementen dieser Zeile. Ermittle anschließend aus der entstandenen Matrix für jede Spalte jeweils das kleinste Element und subtrahiere es von allen Elementen dieser Spalte.

$$\begin{bmatrix} 4 & 1 & 1 & 0 \\ 5 & 1 & 2 & 0 \\ 0 & 1 & 2 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Abbildung 2.12: Matrix nach Subtraktion der Zeilen- und Spaltenminima

Schritt 2: Finde eine minimale Anzahl n_1 an vertikalen oder horizontalen Linien, die alle Nullen in der Matrix überdecken (streichen). Diese Streichung ist nur optischer Natur und entfernt oder verändert die Werte in der Matrix nicht.

$$\begin{bmatrix} 4 & 1 & 1 & 0 \\ 5 & 1 & 2 & 0 \\ 0 & 1 & 2 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Abbildung 2.13: Matrix mit $n_1 = 3 < n = 4$ gestrichenen Zeilen und Spalten

Schritt 3:

- Wenn $n_1 = n$, wurde eine optimale Lösung (= geringste Gesamtsumme) gefunden. Entferne die optischen Streichungen wieder und suche nun eine Verteilung der Nullen derart, dass in keiner Zeile und Spalte zwei Nullen ausgewählt werden (*unabhängige Nullen*) und beende danach den Algorithmus. Die optimale Zuteilung steht dann an den Stellen der unabhängigen Nullen.

- Wenn $n_1 < n$, ermittle das kleinste Element h_1 in der Matrix, das noch nicht gestrichen wurde (es gilt $h_1 > 0$). Addiere dieses zu allen Elementen, die zweimal, also sowohl vertikal als auch horizontal, gestrichen wurden und subtrahiere es von allen ungestrichenen Elementen der Matrix. Entferne die optischen Streichungen, sodass wieder alle Einträge der Matrix ungestrichen auftreten.

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 0 & 1 & 2 & 6 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Abbildung 2.14: Veränderte Matrix mit kleinstem ungestrichenen Element $h_1 = 1$

Schritt 4: Wiederhole Schritt 2 und 3 unter Verwendung der veränderten Matrix. Da in jeder Iteration die Summe der Elemente der Matrix um $n(n - n_k)h_k$ reduziert wird, wird der Algorithmus zwangsläufig nach einer endlichen Anzahl von Schritten ein Ende finden [27].

$$\begin{bmatrix} \hline 3 & 0 & 0 & 0 \\ \hline 4 & 0 & 1 & 0 \\ \hline 0 & 1 & 2 & 6 \\ \hline 0 & 0 & 0 & 2 \end{bmatrix}$$

Abbildung 2.15: Matrix mit $n_1 = 4 = n = 4$ gestrichenen Zeilen und Spalten

Es wurde eine optimale Lösung gefunden, nun suche nach einer Verteilung von unabhängigen Nullen. Sie ist optimal, weil die Gesamtsumme der Gewichte minimal ist. Eine dieser Lösungen ist folgende Zuteilung (Elemente über rote Boxen markiert):

$$\begin{bmatrix} 6 & 3 & \boxed{3} & 2 \\ 6 & 2 & 3 & \boxed{1} \\ \boxed{1} & 2 & 3 & 6 \\ 2 & \boxed{2} & 2 & 3 \end{bmatrix}$$

Abbildung 2.16: Eine optimale Lösung, gezeigt ist die Ausgangsmatrix

Das schließt den Ablauf des Algorithmus ab. Lediglich verbleibt noch die Art und Weise, wie die minimale Anzahl an Linien und die zur Lösung gehörenden Nullen zu finden sind. Dies gehört nicht mehr zum eigentlichen Algorithmus und ist Teil der Implementierung, die später beschrieben wird.

Kapitel 3

Verwandte Arbeiten

Zuordnungsprobleme und auch Lösungsverfahren für diese existieren mittlerweile schon lange, daher haben sich im Laufe der Zeit verschiedene Varianten zum Lösen der unterschiedlichen Zuordnungsprobleme angesammelt [33]. Es gibt verschiedene Ausprägungen der Zuordnungsprobleme, einige von ihnen sind speziell zugeschnitten auf einen bestimmten Anwendungsfall, was daran liegt, dass zahlreiche Problemstellungen eigene Besonderheiten haben, die eine Abänderung eines allgemeineren Verfahrens wie zum Beispiel der Ungarischen Methode verlangen.

Neben diesen Besonderheiten der Ausgangsprobleme wurde die Ungarische Methode im Laufe der Zeit schon mehrfach angepasst und/oder verbessert (beispielsweise durch Edmonds und Karp [10]), weswegen sich dieses Kapitel auf verwandte Arbeiten bezieht, die ein Zuordnungsproblem lösen, nicht aber unbedingt mit der Ungarischen Methode [26].

Die in diesem Kapitel aufgeführten Verfahren gehen das Problem der Zuordnung von verschiedenen Betrachtungsweisen aus an: Die zwei häufigsten sind entweder eine Betrachtung des Problems als gewichteten bipartiten Graphen, in dem ein Maximum-Matching oder ein perfektes Matching gesucht wird, oder eine Betrachtung als Matrix, deren Zeilen und Spalten die zuzuordnenden Paare, und die Matrixeinträge die Gewichtungen darstellen.

Es ist zu betonen, dass beide Betrachtungsweisen nichts an der eigentlichen Problemstellung ändern. Die verschiedenen Betrachtungsweisen dienen jeweils nur dem besseren Verständnis eines Algorithmus und können jederzeit anders dargestellt werden.

3.1 Betrachtung als Graph

Zunächst sei mit der Betrachtungsweise aus Sicht eines gewichteten, bipartiten Graphen begonnen.

3.1.1 Finden eines Maximum-Matchings

Zvi Galil [14], ein israelisch-amerikanischer Informatiker und Mathematiker, beschreibt in seinem Artikel aus 1986 eine Vorgehensweise, um ein Maximum-Matching in einem gewichteten bipartiten Graphen zu finden. Dazu formuliert er das Problem als ein Problem linearer Optimierung.

Mit jedem Schritt des Algorithmus wird jetzt versucht, den augmentierenden Weg durch den Graph zu erweitern, sodass das Gewicht des Pfads maximal wird, während das duale Problem beachtet wird.

Es wird deutlich, dass ein Schritt des Algorithmus, den augmentierenden Weg zu erweitern, grundsätzlich dem Algorithmus zur Findung des kürzesten Pfads von Edsger W. Dijkstra [8] entspricht. Die Quelle als Start für den Dijkstra-Algorithmus ist ein hypothetischer neuer Knoten, der mit allen nicht verbundenen Knoten einer Partition verbunden ist und zu jedem die Länge 0 hat.

Auch wenn Galil [14] nicht explizit eine Zuordnung von Personen als primäres Ziel anstrebt, sondern diese nur als ein Beispiel erwähnt, wäre dies dennoch eine der Möglichkeiten, ein Zuordnungsproblem zu lösen.

3.1.2 Zuordnungsproblem mit hierarchischer Ordnung und Gruppen-Beschränkungen

Ismail H. Toroslu und Yilmaz Arslanoglu [35] untersuchen ein noch verschärfteres Zuordnungsproblem. In ihrem Bericht über das „assignment problem with hierarchical-ordering and set constraints“, zu Deutsch das „Zuordnungsproblem mit hierarchischer Ordnung und Gruppen-Beschränkungen“, kurz AHSC, beschreiben sie ein Zuordnungsproblem, das nicht nur Gewichtungen oder Präferenzen an den Paarungen hat, sondern darüber hinaus noch ein hierarchisches System, wie es zum Beispiel beim Militär der Fall ist, und spezielle Einschränkungen für Gruppierungen innerhalb der Partitionen des bipartiten Graphs berücksichtigt.

Auf die Realität übertragen soll das AHSC sicherstellen, dass beispielsweise kein Soldat niederen Rangs auf ein Tätigkeitsfeld zugeteilt wird, das bereits ein Soldat höheren Rangs beinhaltet. Gleichermaßen veranschaulicht die Gruppen-Beschränkung eine Zuordnung, bei der Familien oder Teams gemeinsam auf ähnliche Tätigkeiten verteilt werden müssen und nicht getrennt werden können.

Aufgrund dieser Menge an Einschränkungen betonen sie, dass es quasi nicht möglich ist, alle Beschränkungen zu beachten bei gleichzeitiger Maximierung der Gewichtungen an den Zuordnungen. Daher formulieren sie die drei Bedingungen, die so gut wie möglich erreicht werden sollen, als eindeutige Optimierungsziele:

- Anzahl Verstöße gegen die Hierarchievorgaben minimieren
- Anzahl Verstöße gegen die Gruppierungsvorgaben minimieren
- Summe der Gewichtungen der Kanten (des bipartiten Graphen) maximieren

Sie führen AHSC als NP-schweres Problem an (es kann also nicht in Polynomialzeit gelöst werden) und erklären, dass ähnliche Problemstellungen mit mehreren Zielen durch sogenannte Evolutionäre Algorithmen lösbar sind. Evolutionäre Algorithmen sind Algorithmen, die sich im Grundgedanken an der Evolution von Lebewesen orientieren und sich über Generationen durch Prinzipien wie Mutation oder Selektion weiterentwickeln [12].

Zur Lösung des Problems benennen sie drei Lösungsansätze, die sich überwiegend in ihrer Zielfunktion des Algorithmus unterscheiden. Die am Ende erfolgreichste Methode – also diejenige, bei der die verschiedenen Ziele im besten Maße erreicht wurden – ist die Anhäufungsmethode (en. „aggregation method"), bei der sie alle drei Ziele in einen einzelnen Wert zusammengefasst haben. Daran anschließend fügen sie der am häufigsten verbreitete Variante dieser Methode, eine gewichtete Summe zu bilden, normalisierte Koeffizienten hinzu, damit sie sich nicht zu sehr an einem der drei Ziele orientiert.

Im Gegensatz zur Ungarischen Methode, ihrer Erweiterung durch Munkres [27] und der Anwendung in dieser Arbeit ist dieses Zuordnungsproblem ein sehr spezielles und eingeschränktes, während erstere mit den anzugebenden Prioritäten nur an eine einzige Einschränkung gebunden ist. Daher haben beide Lösungsverfahren Aufgabenstellungen, für die sich nicht geeignet sind.

Edmonds und Karp [10] machen das klassische Zuordnungsproblem im Rahmen ihrer Untersuchungen über das „minimum-cost flow problem" in Bezug auf dessen Laufzeit effizienter. Während die Ungarische Methode noch eine Laufzeit von $O(n^4)$ besitzt, läuft der neu entwickelte Algorithmus von Edmonds und Karp [10] nur noch in $O(n^3)$. Durch je einen Knoten auf den beiden Seiten des bipartiten Graphen wird ein Netzwerk simuliert, auf dem der neue Algorithmus dann eine Zuteilung finden kann.

Tomizawa [34] kann ebenfalls eine Laufzeit von $O(n^3)$ erzielen, während er das „transportation problem" bearbeitet. Als Grundidee wird hier der Algorithmus zum Finden des kürzesten Pfades von Dijkstra [8] verwendet, bei dem die Pfade entsprechend des vorliegenden Zuordnungsproblems angepasst werden.

3.2 Betrachtung als Matrix

Giorgio Carpaneto und Paolo Toth [5] untersuchen das „min-sum linear assignment problem" (AP). Dieses zeichnet sich dadurch aus, dass hier nicht

die Gewichtungen einzelner Paarungen maximiert, sondern die Kosten ebendieser minimiert werden sollen. Hier wird erneut lineare Programmierung zur Lösung des Problems verwendet, wie auch Galil [14] sie in seiner Arbeit benutzt hat.

Das AP wird auf folgende Weise definiert: Gegeben sei eine quadratische Kosten-Matrix (a) der Größe n, dann finde eine Permutation f_i mit $i = 1, \ldots, n$ der Zahlen $1, \ldots, n$, die folgende Summe minimiert:

$$z = \sum_{i=1}^{n} a_{if_i}$$

Da die Primal-Dual-Algorithmen eine Laufzeitkomplexität von nur $O(n^3)$ haben [5], werden diese Algorithmen den weniger effizienten Primal-Algorithmen vorgezogen. Probleme linearer Programmierung können in ein primales und ein duales Problem aufgeteilt werden, ein Primal-Dual-Algorithmus versucht genau für diese aufgeteilte Problemstellung eine Lösung zu finden.

Im Folgenden untersuchen Carpaneto und Toth [5] dann zwei Vorgehensweisen bei der linearen Programmierung, sowohl die Findung eines Primal-Dual-Algorithmus für vollständige als auch für dünnbesetzte Kostenmatrizen, also solche, die sehr viele Nullen als Werte beinhalten.

Die hier gewonnenen Erkenntnisse werden für den neu zu entwickelnden Algorithmus verwendet, da der aktuell genutzte für vollständige Matrizen langsamer als für dünnbesetzte Matrizen läuft. Carpaneto und Toth [5] machen sich die Tatsache zu Nutze, dass dünnbesetzte Matrizen schneller bearbeitet werden und reduzieren deshalb vollständige Matrizen auf solche mit vielen Nullen.

Wie Pentico [33] bereits herausgestellt hat, gibt es sehr viele verschiedene Anwendungsfälle für Zuordnungsprobleme und auch Variationen dieser. Die hier aufgeführten verwandten Arbeiten beschäftigen sich zum Beispiel mit speziellen Problemen, also solchen, die eine sehr besondere Aufgabenstellung haben. Auch sind Arbeiten aufgeführt, die sich mit verwandten Themenbereichen wie dem Finden eines Pfades mit minimalen Kosten in einem Netzwerk beschäftigen, bei denen die Themenbereiche entsprechend auf Zuordnungsprobleme zugeschnitten werden. Diese Arbeit bearbeitet hingegen das klassische Zuordnungsproblem, wie es von Kuhn [21] 1955 formuliert wurde und löst es auch mithilfe der von ihm entwickelten Ungarischen Methode.

Kapitel 4

Implementierung und Entwicklung

Das im Rahmen dieser Arbeit entwickelte Tool wurde vollständig in der Programmiersprache C# mit dem .NET-Framework 4.7.2[1] geschrieben und nutzt die Extensible Application Markup Language[2] [23] – kurz XAML – für die Erstellung der Benutzeroberfläche. Die Entscheidung für diese Sprache hat zwei Gründe: Zum einen ist das im Rahmen dieser Arbeit entwickelte Tool damit ein guter Anschluss an das bereits vorhandene Anmeldewerkzeug, welches ebenfalls in dieser Sprache geschrieben ist. Zum anderen ermöglicht die Sprache Zugriff auf die Statistikwerkzeuge des .NET-Frameworks, die die Ergebnisse der Zuteilungen visualisieren.

4.1 Konzept und Planung

Bevor der Aufbau des Programms in diesem Abschnitt erläutert wird, sei noch begründet, warum eine Lösung für das Problem dieser Arbeit unter keinen Umständen manuell durch Ausprobieren aller Möglichkeiten zu finden ist. Bei den zu erwartenden Teilnehmerzahlen ist dieses Ausprobieren ein zu großer Aufwand, sodass es unmöglich ist, alle Kombinationen auszutesten. Ein Algorithmus, der entweder näherungsweise oder exakt eine Lösung findet, ist daher unerlässlich.

Die Entscheidung für die Anwendung der ursprünglichen Variante der Ungarischen Methode von Kuhn [21] liegt sowohl in ihrer Schlichtheit begründet – sie besteht aus lediglich vier Schritten, die wiederholt werden – als auch darin, dass sie universell einsetzbar ist und bei Bedarf später erweitert werden kann. Erweiterungen oder Verbesserungen, wie die von Toroslu und Arslanoglu [35], haben weitere Nachteile, wie zum Beispiel, dass nicht jede Nebenbedingung optimiert werden kann. Wie später noch

[1] https://support.microsoft.com/de-de/topic/microsoft-net-framework-4-7-2-...
[2] https://docs.microsoft.com/de-de/dotnet/desktop/wpf/xaml/...

ausgeführt wird, hat die zweite Phase des im Rahmen dieser Arbeit entwickelten Tools eine schlechtere Laufzeit als die erste, daher ist der Unterschied zwischen $O(n^4)$ und $O(n^3)$ zu vernachlässigen.

Der zweite Teil der Einteilung, der sich mit der Gruppenverteilung innerhalb eines Projekts beschäftigt, ist eine spezielle Methode eines Nutzers[3] von der Plattform „StackOverflow", die nur für Probleme funktioniert, bei denen die Gruppenmitglieder Fähigkeiten angeben, die sie sortierbar machen. Das macht sie sehr gut auf das vorliegende Problem anwendbar.

4.1.1 Kombinatorik

Es wird zur Vereinfachung der folgenden Rechnungen eine realitätsnahe Verteilung von $n = 150$ Studierenden zu Gruppen von je $k = 6$ Mitgliedern angenommen, was also eine Gruppenanzahl von $\frac{150}{6} = r = 25$ Gruppen ergibt. Würde man jeden Gruppenplatz getrennt betrachten, wären dies genau $n!$, also $150! \approx 5 \cdot 10^{270}$ Möglichkeiten, die Studierenden unabhängig von der Gruppenzugehörigkeit einzeln auf die verfügbaren Plätze zu verteilen.

Der aktuell schnellste Supercomputer der Welt, der „Fugaku", der 442 Petaflops, also $442 \cdot 10^{15}$ beziehungsweise 442 Billiarden Gleitkommaoperationen pro Sekunde, durchführen kann [31], bräuchte grob 10^{255} Sekunden oder in etwa 10^{247} Jahre, um alle Kombinationen einzeln zu überprüfen – das ist nicht praktikabel.

Da hier allerdings Gruppen gebildet werden sollen, muss auch diese Möglichkeit über Gruppeneinteilungen überprüft und ausgeschlossen werden. Die Berechnung der Möglichkeiten hier gestaltet sich etwas anders, welche den Binomialkoeffizienten verwendet.

Der *Binomialkoeffizient* ist die Berechnung von Kombinationen ohne Zurücklegen [24]. Dieser berechnet sich auf die folgende Art

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

und beschreibt, wie viele Möglichkeiten es gibt, k Elemente aus n Elementen auszuwählen.

Mit dieser Definition lässt sich nun die Anzahl der möglichen Kombinationen berechnen, die tatsächlich nötig wären, wenn die Gruppenzugehörigkeiten mit berücksichtigt werden:

- Die ersten sechs Studierende haben noch freie Auswahl und dementsprechend gibt es $\binom{n}{k} = \binom{150}{6}$ Möglichkeiten, sechs Studierende zu verteilen.
- Den nächsten sechs Studierenden bleiben nicht mehr alle Plätze, es gibt nur noch $\binom{150-6}{6}$ Möglichkeiten.

[3] „m69 'snarky and unwelcoming'": `https://stackoverflow.com/a/34572278/10189237`

- Das setzt sich entsprechend fort. Am Ende muss das Gesamtergebnis noch durch $r! = 25!$ geteilt werden, da die 6er-Gruppen bei dieser Berechnung in anderer Reihenfolge doppelt auftreten. Als Beispiel existieren die zwei Kombinationen $(a)(b)\ldots(y)$ und $(b)(a)\ldots(y)$ mit a, b und y als Repräsentation dreier 6er-Gruppen und beschreiben die gleiche Kombination, da es keine Rolle spielt, an welcher Position der Permutation eine Gruppe auftaucht.

Daraus ergibt sich die folgende Rechnung, um alle Kombinationen C zu erhalten:

$$
\begin{aligned}
C =& \underbrace{\binom{n}{k}\binom{n-k}{k}\binom{n-2k}{k}\cdots\binom{2k}{k}\binom{k}{k}}_{r\text{-mal}}\frac{1}{r!} \\
=& \frac{n!}{k!\cdot(n-k)!}\cdot\frac{(n-k)!}{k!\cdot(n-2k)!}\cdot\frac{(n-2k)!}{k!\cdot(n-3k)!}\cdots\frac{2k!}{k!\cdot\underbrace{(2k-k)!}_{k!}}\cdot 1\cdot\frac{1}{r!} \\
=& \underbrace{\frac{n!}{k!}\cdot\frac{1}{k!}\cdot\frac{1}{k!}\cdots\frac{1}{k!\cdot k!}}_{(r-1)\text{-mal}}\cdot 1\cdot\frac{1}{r!} \\
=& \frac{n!}{(k!)^r\cdot r!}
\end{aligned}
$$

Mit eingesetzten Beispielwerten ergibt das Folgendes:

$$
\begin{aligned}
C =& \frac{n!}{(k!)^r\cdot r!} = \frac{150!}{(6!)^{25}\cdot 25!} \\
=& 13.581.111.704.879.300\cdot 10^{150}\approx 13\cdot 10^{165}
\end{aligned}
$$

Auch wenn die Anzahl der Kombinationen auf diese Weise von mehr als $5\cdot 10^{270}$ auf etwa $13\cdot 10^{165}$ verringert werden konnte, ist dies immer noch deutlich zu viel, um in einen realistischen Rahmen zu passen. Der Supercomputer „Fugaku“ [31] würde 10^{142} Jahre zum Berechnen aller Kombinationen benötigen – zwar weniger als 10^{247} Jahre, aber immer noch nicht praktisch umsetzbar.

4.1.2 Idee

Die zentrale Idee hinter dem Programm ist eine Zweiteilung der Zuordnung, die nacheinander abläuft. Dahinter verbirgt sich eine Kombination aus der Ungarischen Methode [21] und einer individuellen Lösung zum Angleichen von mehreren Gruppen.

Zum weiteren Verständnis ist anzumerken, dass der Begriff *Projekt* als Bezeichnung für ein gesamtes Projekt mit mehreren Einzelgruppen genutzt

wird, während eine *Gruppe* eine von eventuell mehreren Teilgruppen eines Projekts ist. So kann zum Beispiel die Rede vom Projekt „PDFZensor“ sein, das von drei Teams – den Gruppen – zu je sechs Studierenden bearbeitet wird, die üblicherweise denselben Kunden oder dieselbe Kundin und denselben Team-Coach haben. Auch werden die Begriffe *Phase* und *Teil* variierend genutzt, im Fall des inhaltlichen Bezugs auf das Programm dieser Arbeit meinen diese Begriffe beide die zentralen Teile des Programms (s. Abbildung 4.1).

Grundsätzlich sind genau zwei Faktoren für die Einteilung der Studierenden elementar: die drei Projektwünsche der Studierenden (das Programm benachrichtigt den Benutzer, wenn diese Voraussetzung bei mindestens einem oder einer Studierenden nicht zutrifft) und die von ihnen angegebenen Fähigkeiten, programmiertechnischer und zwischenmenschlicher Natur. Der Fokus liegt dabei auf dem ersten Teil, also dem Versuch der Erfüllung eines Projektwunsches, da dieser eindeutig und unveränderbar bestimmt, zu welchem Projekt ein Student oder eine Studentin zugeteilt wird. Der zweite Teil zielt anschließend darauf ab, die entstandenen Zuordnungen so anzugleichen, dass jede Gruppe in etwa gleich gute Gruppenmitglieder hat – „gleich gut“ meint hier, dass für alle Studierende jeweils der Durchschnitt ihrer Fähigkeitswerte berechnet wird und anhand dieser Durchschnitte versucht wird, die Gruppen so ausgeglichen wie möglich einzuteilen.

Wird dieser erste Teil also beispielsweise Studentin A zum Projekt B zuteilen, das insgesamt die drei Gruppen $B1$, $B2$ und $B3$ beinhaltet, steht zu diesem Zeitpunkt noch nicht fest, zu welcher der drei Gruppen sie zugeteilt wird, aber sie wird in jedem Fall eine dieser drei erhalten. Daher nennen wir die Menge von möglichen Gruppen einen „Lösungsraum“.

Der Grund hinter der Entscheidung für die Zweiteilung ist simpel: Während die Ungarische Methode (oder ein anderer Algorithmus, der Zuordnungen lösen kann) gut für Situationen funktioniert, in denen ein einzelner Faktor die Grundlage der Gewichtung bildet (in diesem Fall sind das die Projektwünsche), würden sehr verzerrte Ergebnisse herauskommen, machte man die Zuteilung von mehreren Faktoren, also den angegebenen Fähigkeiten der Studierenden, abhängig. Das ist sehr ähnlich zu dem beschriebenen Problem von Toroslu und Arslanoglu [35] in Abschnitt 3.1.2, bei dem sie genau diese Schwierigkeit explizit erwähnen.

Die Ungarische Methode allein ist daher keine Option. Es bleibt noch die gegenteilige Frage offen, ob der zweite Teil des Programms, also die gruppenbezogene Einteilung anhand der Fähigkeiten, für sich genommen als einziger Hauptteil in Frage käme. Auch das ist auszuschließen, da hier das gleiche Problem vorliegt: zu viele Einschränkungen verkomplizieren die Zuordnung. Hinzu kommt, dass der zweite Teil bedeutend zeitintensiver ausfällt, da er im Gegensatz zum $O(n^4)$-Algorithmus iterativ mögliche Kombinationen der Gruppenkonstellationen durchtestet, um hier eine gute Verteilung zu finden.

Die effizientere Ungarische Methode zweimal hintereinander anzuwenden, anstelle des hier gewählten Ansatzes, wäre ebenso mit Schwierigkeiten verbunden gewesen: Um eine ausgeglichene Gruppe zu erstellen, wird wiederholt ein Durchschnitt berechnet, was in Dezimalzahlen resultiert, die der Algorithmus verarbeiten müsste. Weil dieser allerdings nur für ganze Zahlen konzipiert wurde [21, S. 90], ist eine andere zweite Methode unumgänglich.

4.2 Aufbau des Programms

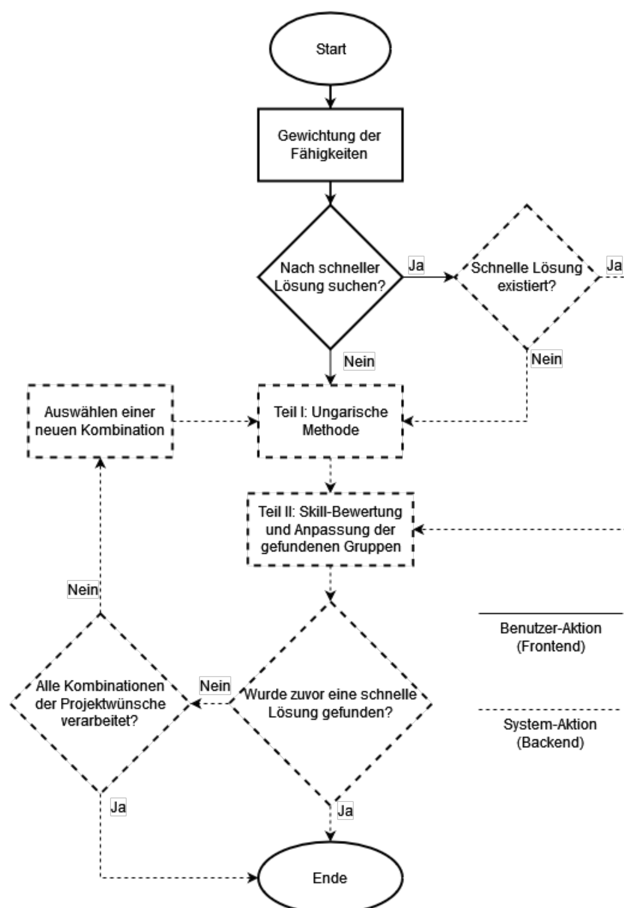

Abbildung 4.1: Programmaufbau mit den zwei zentralen Teilschritten

Das Programm startet über eine Entgegennahme der XML-Datei aus dem Anmeldewerkzeug. Im Folgenden wird der restliche Ablauf des Programms erklärt, wobei die beiden zentralen Teilschritte des Programms, die in Abbildung 4.1 zu sehen sind, nur erwähnt und erst später in separaten Abschnitten erklärt werden, da beide sehr umfangreich in ihrer Beschreibung

sind.

Die Entgegennahme von Fähigkeitspriorisierungen durch den Benutzer oder die Benutzerin ist eine der wichtigsten Einstellmöglichkeiten, die sich nach Auswählen der Eingabedatei eröffnen. Hier geht es darum, dass die Fähigkeiten, für welche die Gruppen in Teil 2 (s. Abbildung 4.1) angeglichen werden sollen, anpassbar gemacht werden. Gibt es zum Beispiel ein Projekt, bei dem der Kunde oder die Kundin wünscht, dass sein oder ihr Programm in der Programmiersprache C++ geschrieben wird, kann die Fähigkeit „Programmiererfahrung mit C++“ in ihrer Gewichtung angehoben werden, sodass zwei Studierende eher in zwei getrennte Gruppen als in eine gemeinsame sortiert werden (das Ziel ist nach wie vor eine ausgeglichene Verteilung). An der grundsätzlichen Verteilung der Studierenden auf dieses eine Projekt ändert dies aber nichts – denn das geschieht in Teil 1, welcher lediglich die Projektwünsche berücksichtigt.

Neben diesen Einstellmöglichkeiten könnte noch eine weitere Option sein, Wünsche der Studierenden nach bestimmten Gruppenpartnerinnen und Gruppenpartnern oder Tutorinnen und Tutoren berücksichtigen zu können. Eine Realisierung dieser Möglichkeit könnte sich vorteilhaft auf die Zufriedenheit einer Teilnehmerin oder eines Teilnehmers auswirken, was dementsprechend auch positive Auswirkungen auf den Lernerfolg hätte [6, S. 11f.]. Weil dies allerdings eine Anpassung des Anmeldewerkzeugs für das Software-Projekt verlangt und darüber hinaus die Zuteilung um eine weitere Einschränkung erweitern würde, wurde dieser Ansatz hier nicht weiter verfolgt.

Als Nächstes ist es möglich, einzustellen, ob eine schnelle Lösung – „basic assignment“ in der graphischen Benutzeroberfläche genannt – gefunden werden soll. Diese Einstellung beeinflusst das Verhalten der ersten Phase und wie in Abbildung 4.1 zu sehen ist, umgeht sie diese potenziell. Hierbei überprüft das Programm in einem Extraschritt mithilfe des Hopcroft-Karp-Algorithmus [15] über ein Maximum-Matching, ob eine Zuordnung möglich ist, bei der jeder Student und jede Studentin einen seiner oder ihrer drei Projektwünsche erhält. Diese Überprüfung findet statt, bevor die Prioritäten der Studierenden durch Schritt 2 und 3 der Ungarischen Methode verändert werden (siehe dafür das Studierenden-Beispiel der Ungarischen Methode in Abschnitt 2.4.3); sie wird also direkt auf die Ausgangssituation angewendet.

Um die Anzahl der Studierenden und Projektplätze anzugleichen, werden noch Platzhalter-Knoten eingefügt, die valide Zuordnungen darstellen, falls sich die Studierenden nicht restlos auf die Gruppen aufteilen lassen, also Gruppen mit unterschiedlichen Gruppengrößen entstehen. Ist einer dieser Knoten in der finalen Zuordnung enthalten, wird für speziell diese Zuordnungen der Hopcroft-Karp-Algorithmus [15] wiederholt, entsprechend mit den verbliebenen Projektplätzen. Die Ergebnisse beider Durchläufe werden verbunden und folgendermaßen ausgewertet:

- Wird ein Maximum-Matching mit einer Kardinalität gleich der Gesamtanzahl der Studierenden gefunden, es wurde also allen Studierenden einer der drei Projektwünsche zugeteilt, springt das Programm direkt in Phase II.

- Wurde kein Maximum-Matching gefunden oder ist dieses Matching in seiner Kardinalität kleiner als die Anzahl der Studierenden, gibt es keine Lösung ohne „Nicht-Wünsche". Das Programm fährt dann mit Phase I des Hauptalgorithmus fort und verhält sich, als wäre nicht nach einer schnellen Lösung gesucht worden.

Sind die beiden Phasen des Hauptalgorithmus abgearbeitet worden, überprüft das Programm, ob bereits alle Kombinationen der Prioritäten der Wünsche untersucht wurden, was folgendermaßen abläuft: Wird Teil I zum ersten Mal aufgerufen, wird der Erstwunsch mit der wichtigsten, der Zweitwunsch mit der zweitwichtigsten und der Drittwunsch mit der drittwichtigsten Priorität behandelt ($1 > 2 > 3$). Auf diese Kombination wird die Ungarische Methode [21] und der zweite Teil des Programms angewendet. Die daraus erhaltene Zuteilung wird abgespeichert. In den folgenden fünf Iterationen werden nun die anderen Kombinationen der Prioritäten, wie beispielsweise $3 > 1 > 2$, abgearbeitet. Falls zuvor eine schnelle Lösung gefunden wurde, wird auch diese Schleife übersprungen und das Programm beendet sich nach dem ersten Erreichen von Teil II.

Ist der Hauptalgorithmus an diesem Punkt noch nicht beendet, hat also keine schnelle Lösung gefunden, wird er mehrere Zuteilungen beziehungsweise Lösungen finden. Es muss daher eine *beste* Lösung definiert werden, die am Ende die erste sein wird, die angezeigt wird. Das ist anhand der vier Punkte unten zu erklären, welche in absteigender Wichtigkeit aufgeführt sind und wie Sortierungskriterien zu verstehen sind. Es wird immer zunächst der oberste Punkt beachtet und erst bei mehreren Zuteilungen, die diesen in gleichem Maße erfüllen, wird der darunterliegende Punkt betrachtet und diesem entsprechend absteigend sortiert. Eine Zuordnung, die zum Beispiel den obersten Punkt komplett erfüllt, ist *besser* als eine, die ihn nur teilweise erfüllt:

- Geringste Anzahl an unerwünschten Zuteilungen

- Höchste Anzahl an Erstwünschen

- Höchste Anzahl an Zweitwünschen

- Höchste Anzahl an Drittwünschen

Die Ungarische Methode [21] arbeitet durch Rechenoperationen an Zeilen und Spalten. Das hat zur Folge, dass eine neue Kombination der Prioritäten (was prinzipiell nur eine Permutation der Zeileneinträge 1, 2

und 3 ist) eine andere finale Zuteilung bedeuten kann, da die Projekte sich unterschiedlich schnell füllen. Um hier eine größere Lösungsmenge anbieten zu können, werden alle diese Kombinationen der Prioritäten und ihre Zuteilungen berechnet. Wurden alle sechs Kombinationen und ihre entsprechenden Zuordnungen errechnet, werden sie am Ende gemeinsam mit weiteren Statistiken in der graphischen Benutzeroberfläche angezeigt, wobei die beste der sechs Lösungen vorausgewählt ist.

4.2.1 Teil I: Erschaffung von Lösungsräumen

Die erste von zwei Kernphasen des Programms ist die Anwendung der Ungarischen Methode, siehe Abschnitt 2.4.3 für die Abfolge der Schritte und Algorithmus 1 in Anhang A für den Pseudocode.

Aufbau der Matrix und Idee

Durch die in der XML-Datei enthaltenen Daten aus der Anmeldeseite für das Software-Projekt ist es leicht möglich, die Anzahl der Projektplätze zu bestimmen, die neben den Studierenden die andere Hälfte der Zuteilungen bilden. Um noch die bestmögliche Auswahl zu ermöglichen, werden in **Schritt 0** die Projektplätze mit Platzhaltern aufgefüllt, die wie Joker agieren und mit der unwichtigsten Priorität eine valide Zuteilung darstellen, sofern die Anzahl der Studenten sich nicht restlos auf die Projekte aufteilen lässt. Ohne diese Platzhalter wäre von Anfang an festgesetzt, welche Gruppe wie viele Plätze frei hat; sie ermöglichen also weitere Flexibilität in der Zuteilung auf der Suche nach der Lösung mit den wenigsten unerwünschten Paarungen. Diese Platzhalter werden in einem zweiten Schritt später in echte Projektplätze umgewandelt.

Ein Rechenbeispiel sei gegeben: 155 Studierende sollen auf insgesamt 20 Gruppen zugeteilt werden. Es gibt dementsprechend Gruppen der Größe $\lfloor \frac{155}{20} \rfloor = 7$ und $7 + 1 = 8$, da die Aufteilung der Studierenden nicht ohne Rest aufgeht. Weil $155 \equiv 15 \pmod{20}$, gibt es 15 Gruppen der Größe 8 und $20 - 15 = 5$ Gruppen der Größe 7. Das Programm füllt an diesem Punkt die zu erstellende Matrix allerdings nicht mit allen verfügbaren Projektplätzen, sondern nimmt nur $20 \cdot 7 = 140$, während die restlichen 15 durch Platzhalter gefüllt werden. Da zu Anfang nicht bekannt ist, welche Gruppe 7 und welche Gruppe 8 Studierende erhalten soll, ist auf diese Weise sichergestellt, dass keine Gruppe unnötigerweise 8 Plätze enthält, und andersherum.

Die zentrale Idee hier ist, den Grundgedanken der Ungarischen Methode [21] etwas abzuwandeln: Während diese explizit eine Person auf eine Aufgabe zuteilt, wird in dem im Rahmen dieser Arbeit entwickelten Programm rechnerisch zwar mit den einzigartigen Projektplätzen gearbeitet, inhaltlich werden alle Plätze eines Projekts aber als eine Gesamtheit angesehen. Es spielt daher keine Rolle, welchen Platz ein Student oder eine

Studentin in einem Projekt bekommt, da das Projekt selbst mit all seinen Gruppen der wichtige Faktor ist.

Weil **Schritt 1**, also das Finden und Subtrahieren von Spalten- und Zeilenminima, implementationstechnisch keine Besonderheiten aufweist, wird hier direkt zu **Schritt 2** übergegangen.

Ermitteln der optimalen Lösung

Um die minimalen Zeilen und Spalten in der Matrix zu streichen, wird ein bipartiter Graph erstellt, der die Studierenden und Projekte als je eine Seite der Knoten beinhaltet, und bei dem für jede 0 in der Matrix eine Kante zwischen den entsprechenden Knoten gezogen wird.

Genau hier findet der Satz von König [22] (s. Satz 9) Anwendung, denn dieses Ziel ist über eine minimale Knotenüberdeckung erreichbar: wenn jede Kante im Graphen einen zu streichenden Eintrag aus der Matrix darstellt, reicht es, ein Maximum-Matching zu finden, um die Anzahl der Streichungen zu erhalten, da die Knoten Zeilen und Spalten der Matrix darstellen (|Matching| = |Knotenüberdeckung| = |Streichungen|).

Das Suchen nach dem Matching anstelle einer minmalen Knotenüberdeckung hat den Vorteil, dass hierdurch bereits eine Zuordnung gefunden wurde und somit unnötige Rechenarbeit gespart wird. Aus dem Matching wird abschließend dann die minimale Knotenüberdeckung ermittelt. Die notwendigen Pseudoalgorithmen sind in Anhang A zu finden.

Ist die in **Schritt 3** beschriebene Gleichung $n_1 = n$ wahr, wird die Zuordnung aus dem bereits berechneten Matching entnommen. Ist sie unwahr, wird der Rest dieses und des **4. Schritts** angewendet.

Zuteilung der restlichen Studierenden

Als Abschluss des Algorithmus werden noch die eventuell vorhandenen Platzhalter aus Schritt 0 gegen echte Projektplätze getauscht. Dazu wird die Ungarische Methode erneut angewendet, diesmal allerdings auf die Studierenden beschränkt, die einen Platzhalter im ersten Durchlauf zugewiesen bekommen haben.

Da es jetzt aufgrund der Modulo-Operation in Schritt 0 zwangsläufig maximal $(r - 1)$ Studierende gibt, wobei r für die Anzahl der Gruppen steht, ist sichergestellt, dass es maximal bei diesen insgesamt zwei Durchläufen der Ungarischen Methode bleiben wird. Das liegt daran, dass es jetzt mehr Gruppenplätze als Studierende gibt und in diesem Durchlauf jeder Student und jede Studentin eines der Projekte zugeteilt bekommt. Um die Matrix quadratisch zu machen, werden hier Platzhalter für die Studierenden verwendet, die am Ende allerdings nicht in die finale Zuordnung mit einbezogen werden.

4.2.2 Teil II: Fähigkeitsbasierte Teameinteilung

Mit dem Ergebnis aus der ersten der zwei Kernphasen des Programms schließt sich unmittelbar auch der zweite Teil an. Wie bereits im Abschnitt 4.1.1 beschrieben wurde, ist eine manuelle und stückweise Überprüfung aller Gruppen nicht möglich. Aus diesem Grund implementiert dieser zweite Teil eine Lösung, die nicht jede Studierendenkombination einzeln überprüft, sondern gruppenweise vorgeht. Siehe dafür den Pseudocode unter Algorithmus 4 in Anhang A.

Idee

Diese Idee stammt von einem Nutzer[4] der Plattform „StackOverflow". Sie zielt darauf ab, in kurzer Zeit ein Ergebnis zu erreichen, das ähnlich dem einer Überprüfung aller Kombinationen ist. Die Implementierung des im Rahmen dieser Arbeit entwickelten Programms orientiert sich in großen Teilen an diesem Ansatz, wurde in ihrer Bewertungsfunktion aber angepasst und so umgeschrieben, dass sie nicht nur für sechs Gruppen und 36 Studierende funktioniert, wie es in der Idee des Nutzers beschrieben steht.

Um Vergleiche einzusparen, werden bei diesem Ansatz die Studierenden zunächst in Gruppen eingeteilt, die anschließend über eine Gruppenbewertungsfunktion eingestuft werden. Wie viele Vergleiche eingespart werden, sei im Folgenden gezeigt: Bei Gruppen zu je 6 Studierenden gibt es

$$\frac{n!}{(k!)^r \cdot r!} = \frac{12!}{(6!)^2 \cdot 2!} = 462$$

Möglichkeiten, um zwei Gruppen zu füllen. Da jede Gruppe einzeln nach jeder neuen Zusammensetzung neu bewertet werden muss, sind das $462 \cdot 2 = 924$ Aufrufe der Bewertungsfunktion. Gibt es wie im Beispiel in Abschnitt 4.1.1 150 Studierende und 25 Gruppen, wären dies also $\binom{25}{2} = 300$ mögliche Paarungen dieser Gruppen und $300 \cdot 924 = 277.200$ Aufrufe der Bewertungsfunktion insgesamt. Das ist nur ein Bruchteil der $13 \cdot 10^{165}$ Aufrufe, die nötig wären, würde jede Kombination einzeln untersucht werden.

Für die Bewertung einer Gruppe wurde als Ansatz der *Durchschnitt* der Fähigkeiten der in ihr enthaltenen Mitglieder gewählt. Das stellt sicher, dass – im Gegensatz zu dem Median oder einer anderen Metrik – „Ausreißer" berücksichtigt werden: Eine Gruppe, bei der ein Mitglied einen vergleichsweise hohen Wert hat, wird auch in ihrer Gruppenbewertung einen erhöhten Durchschnittswert besitzen. Dies ist auch der Punkt, an dem die Fähigkeitseinstellungen des Nutzers oder der Nutzerin aus dem ersten Absatz in Abschnitt 4.2 zum Tragen kommen; die Fähigkeiten können hier in ihrem Gewicht angepasst werden und so die Gruppenbewertung beeinflussen.

[4] „m69 'snarky and unwelcoming'": https://stackoverflow.com/a/34572278/10189237

Auftrennung der Projekte

Ausgehend von der Zuordnung aus Phase I in Abschnitt 4.2.1 werden die Studierenden zunächst entsprechend ihrer initial zugeteilten Gruppen aufgeteilt. Falls ein Projekt nur eine Gruppe stellt, wird dieses Projekt übersprungen, weil keine Anpassung möglich ist. Ansonsten wird die Optimierungsfunktion für alle Gruppen eines Projekts so lange wiederholt aufgerufen, bis der Durchschnittswert der Gruppen sich nicht mehr verbessert. Das wird dann für jedes Projekt mit seinen Gruppen wiederholt.

Optimierungsfunktion

Die Optimierungsfunktion funktioniert so, dass zuerst alle Kombinationen der Paarungen berechnet werden. Mit anderen Worten: Welche Kombinationen berechnet der Binomialkoeffizient $\binom{3}{2}$, wenn ein Projekt beispielsweise aus drei Gruppen bestünde? Anschließend werden jeweils die beiden Gruppen dieser Kombinationen in einer weiteren Methode miteinander verglichen und bewertet. Hier wird die Idee oben implementiert: Mitglieder aus zwei Gruppen werden in allen möglichen Kombinationen in diese eingeteilt und jedes Mal durch die Gruppenbewertungsfunktion einzeln bewertet.

Die *beste* Gruppenkonstellation ist definiert als die Kombination von zwei Gruppen, deren jeweilige Einzelbewertung am wenigsten voneinander abweicht; die beiden Gruppen entsprechend also sehr ausgewogen sind. Die Einzelbewertung einer Gruppe ergibt sich wie bereits erwähnt aus dem Durchschnitt ihrer Mitglieder und wird umso höher, je mehr Mitglieder mit hohen Werten sie beinhaltet. Der Grund für die Ausgewogenheit der Gruppen ist ähnlich zu dem ganz zu Anfang beschrieben Grund, warum Studierende möglichst einen ihrer drei Projektwünsche zugeteilt bekommen sollten (Außenwirkung, Lerneffekt, Motivation): Thomas E. Muller [25] hat in einer Studie herausgefunden, dass die Mitglieder in ausgewogenen Gruppen zufriedener sind, sich mehr herausgefordert fühlen und die notwendige Arbeit auch gleichmäßiger aufteilen.

Weil dieser Idee ein verhältnismäßig komplexes Gerüst zugrunde liegt, sei dies noch einmal mit einem sehr kleinen Beispiel erklärt: Ein fiktives Software-Projekt besteht aus den vier Projekten A, B, C und D. In diesem Beispiel sollen die drei Gruppen $A1$, $A2$ und $A3$ für Projekt A optimiert werden. Sie bestehen jeweils aus vier Studierenden: 1 bis 12. Die Grafiken sind an die Erklärung des Nutzers von „StackOverflow“ angelehnt.

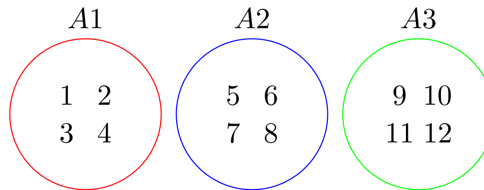

Abbildung 4.2: Studierende der drei Gruppen $A1$, $A2$ und $A3$

Aus den drei Gruppen in Abbildung 4.2 werden nun die Kombinationen bestimmt.

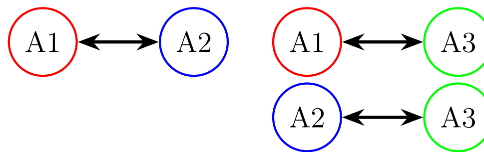

Abbildung 4.3: Kombinationsmöglichkeiten bei drei Gruppen

Für die erste der drei Kombinationen ($A1 \leftrightarrow A2$) in Abbildung 4.3 werden jetzt die Permutationen für 7 anstelle von 8 Mitgliedern berechnet. Das hat den Grund, dass bei acht Studierenden in zwei Gruppen sich die Kombinationen doppeln: So ist $(1)(2)\dots(8)$ ebenso enthalten wie $(8)(7)\dots(1)$. Da jeweils vier eine Gruppe bilden, würde hier also doppelt gerechnet werden. Dadurch, dass nur die Permutationen für 7 statt 8 Studierende berechnet werden, während (1) die ganze Zeit in einer Gruppe gehalten wird, wird Zeit gespart.

Wurden alle $\binom{7}{4} = 35$ Kombinationen bewertet, gibt es eine Kombination, die das beste Verhältnis der zwei Gruppen beschreibt. Als Beispiel sei die Verteilung in Abbildung 4.4 die beste Kombination, bei der $A1$ einen Gruppenwert von 2,3 und $A2$ einen solchen von 2,2 hat, sie in diesem Beispiel in der Differenz also ein sehr ausgewogenes Verhältnis zueinander haben.

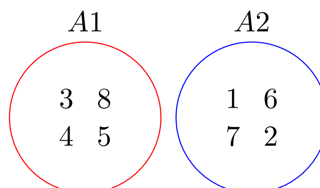

Abbildung 4.4: Bestes Verhältnis der Gruppen $A1$ und $A2$

Dies wird dann auch für die restlichen Kombinationen wiederholt, hier also $A1 \leftrightarrow A3$ und $A2 \leftrightarrow A3$. Es gibt zu diesem Zeitpunkt eine Sammlung von allen Zweier-Kombinationen der Gruppen innerhalb eines Projekts, gespeichert mit den jeweiligen bestmöglichen Verhältnissen zueinander (Bsp.:

$A1 \leftrightarrow A2$: 0,1; $A1 \leftrightarrow A3$: 0,5; $A2 \leftrightarrow A3$: 0,4). Aus diesen Zweier-Kombinationen wird jetzt eine Summe der Verhältnisse zusammen mit allen anderen übrigen Gruppenkombinationen dieses Projekts errechnet. Die beste Gesamtkombination bzw. Summe ist erneut die mit dem geringsten Wert, da dies eine hohe Ausgewogenheit bedeutet.

In diesem Beispiel würde also das Verhältnis von $A1 \leftrightarrow A2$ mit dem Gruppendurchschnittswert von $A3$ summiert werden, da $A3$ die einzig verbleibende Gruppenkombination ist. Danach würde dies mit $A1 \leftrightarrow A3$ & $A2$ und $A2 \leftrightarrow A3$ & $A1$ wiederholt werden, wodurch man drei Summen erhält.

Die geringste dieser Summen ist das Ergebnis einer Iteration. Mit den veränderten Gruppen wird die Optimierungsfunktion dann erneut aufgerufen, um das Ergebnis weiter zu verbessern, nun ausgehend von einer anderen Verteilung. Wird hier keine Verbesserung mehr erzielt, fährt der Algorithmus mit Projekt B fort und wiederholt den gesamten Ablauf. Am Ende werden die neu sortierten und anders befüllten Gruppen als neue Zuteilung gespeichert, zurückgegeben und im Programm über verschiedene Statistiken angezeigt.

4.3 Korrektheit und Laufzeit

Um die Laufzeit des Programms zu bestimmen, muss jede Komponente betrachtet werden. Daraus ergibt sich, dass das Auswerten der Eingabedatei einen zeitlichen Aufwand von $O(n)$ misst. Weil die „Worst Case Time Complexity“ betrachtet wird, kann der Weg über eine schnelle Lösung ignoriert werden. Die Ungarische Methode läuft in einer Komplexität von $O(n^4)$, da sie hier in ihrer Grundform implementiert wurde.

Für den zweiten Hauptteil muss der Pseudocode (s. Algorithmus 4) betrachtet werden: Im schlechtesten Fall gibt es n Projekte, entsprechend ist die erste Schleife bereits mit einer Laufzeit von $O(n)$ verbunden. Die While-Schleife hat eine unbestimmte Laufzeit und läuft solange, bis keine Verbesserung mehr gefunden wurde. Entsprechend hat sie eine Laufzeit von $c \cdot n$, was in $O(n)$ liegt. Die innere For-Schleife läuft in $O(\frac{1}{2} \cdot n!) = O(n!)$, da die Studierenden hier auf *zwei* Gruppen aufgeteilt werden, aber das Finden aller Kombination die Fakultät bedingt. Das Ermitteln der optimalen Studierendenverteilung innerhalb von zwei Gruppen hat durch das Austesten der Kombination ebenfalls eine schlechte Laufzeit: $\frac{1}{2} \cdot n!$.

Alles in allem läuft das Programm im schlechtesten Fall also mit einer zeitlichen Komplexität von $O(n + n^4 + n^2 \cdot (n!)^2) = O(n^2 \cdot (n!)^2)$. Allerdings ist diese schlechte Laufzeit bei der Anwendung nur eingeschränkt spürbar, da die Fakultäten sich durch die kleinen Gruppengrößen in sehr kleinem Rahmen bewegen (bei 25 Gruppen der Größe 6 werden zum Beispiel nur $\binom{25}{2} = 300$ Gruppenkombinationen beziehungsweise $\binom{11}{6} = 462$ Studierendenkombinationen berechnet). Dieser Ansatz beinhaltet noch weitere Optimierungsmög-

lichkeiten, die während dieser Arbeit allerdings nicht implementiert werden konnten.

Die Ungarische Methode wird immer zu einem Ende kommen und eine optimale Lösung finden [27]. Der zweite Teil des Programms ist eine Annäherung an die optimale Verteilung von Studierenden, die sich stetig verbessert, dadurch, dass in jeder Iteration die Bewertung der Gruppen neu berechnet wird. Das Programm wird dementsprechend immer eine Lösung finden, die mindestens in der Zuteilung auf ein Projekt (Teil I des Hauptalgorithmus) optimal ist.

4.4 Graphische Benutzeroberfläche

Da der größte Teil der Berechnungen im Hintergrund und nicht sichtbar für den Benutzer oder die Benutzerin stattfindet, ist das Interface relativ schlicht gehalten.

4.4.1 Konzept und Entwurf

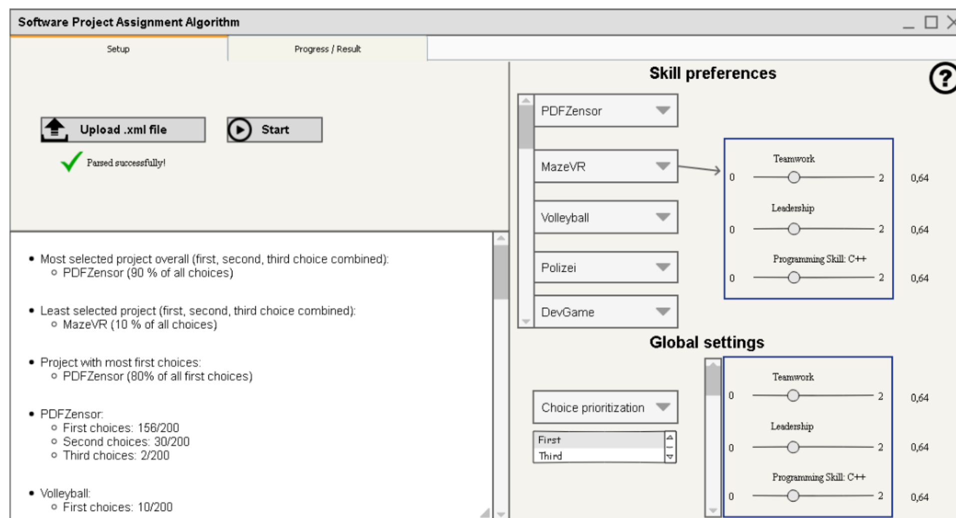

Abbildung 4.5: Setup-Tab des finalen Design-Mockups, größer im Anhang (s. Abbildung B.1)

Zu Beginn wurde das im Rahmen dieser Arbeit entwickelte Tool als Konsolenanwendung realisiert, das nur grobe Fähigkeitspriorisierungen entgegen nehmen konnte und neben der finalen Zuordnung keine Statistiken berechnete. Weil die alleinige Anwendung des in Abschnitt 4.2 beschriebenen Ablaufs keine gute Auskunft über die tatsächlichen Endergebnisse bietet, wurde daraufhin eine Oberfläche entwickelt, die Werte anzeigen kann, die für die weitere Benutzung sinnvoll sind.

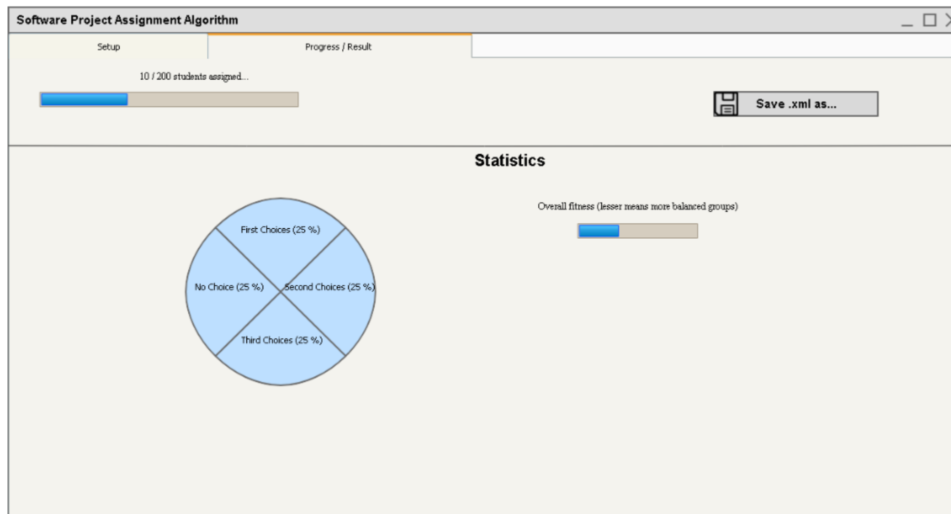

Abbildung 4.6: Ergebnis-Tab des finalen Design-Mockups, größer im Anhang (s. Abbildung B.2)

Der Fokus der Oberfläche liegt dementsprechend primär auf der Anzeige von sowohl Eingabe- als auch Ausgabestatistiken, daher wurde diesen Elementen entsprechend viel Platz eingeräumt. Das ist im finalen Mockup zu erkennen (s. Abbildungen 4.5 und 4.6): Während im Reiter „Setup“ der verfügbare Platz noch überwiegend für die Einstellmöglichkeiten genutzt wird, ist dennoch ein großer Teil der Fläche bereits für Statistiken wie „Anzahl der Studierenden“ oder „Meist gewähltes Projekt“ verwendet worden. Im zweiten Reiter „Progress/Result“ ist klar erkennbar, dass den Statistiken der größte Teil der verfügbaren Fläche zur Verfügung gestellt wird.

Die Statistiken wurden jeweils nach Relevanz und Nützlichkeit bezüglich der aktuellen Phase des Programms ausgewählt. Während die Eingabe-Statistiken ihren Fokus vor allem auf Statistiken haben, die einen Überblick über die Studierenden und ihre Wahlen geben sollen, sollen die Ausgabestatistiken aufzeigen, was der Algorithmus ergeben hat und auf was der Benutzer oder die Benutzerin der Anwendung sich verlassen kann, wenn mit den Zahlen und Zuteilungen weitergearbeitet wird.

Aus diesem Mockup hat sich dann die graphische Benutzeroberfläche entwickelt, die im folgenden Abschnitt beschrieben wird. Die in der Oberfläche verwendeten Icons und Symbole sind entweder Teil des WPFToolkits[5] oder entnommen von einer Webseite[6], die lizenzfreie Icon-Bilder zum Download zur Verfügung stellt.

[5] `https://www.nuget.org/packages/WPFToolkit/3.5.50211.1`
[6] `https://uxwing.com/`

4.4.2 Beschreibung und Erklärung

Das System aus zwei Reitern hat sich auch in der finalen Anwendung durchgesetzt und entsprechend gibt es einen Reiter zum Eingeben der notwendigen Daten und einen, der die Ergebnisse anzeigt und das Ergebnis speichern lässt.

Außerdem ist die Oberfläche größtenteils in englischer Sprache geschrieben, um ein breiteres Publikum ansprechen zu können und auch bei eventuellen Erweiterungsarbeiten eine Grundlage zu besitzen. Eine Ausnahme bilden (System-)Fehlermeldungen, die an das Betriebssystem gekoppelt sind und daher deutschsprachige Elemente beinhalten können.

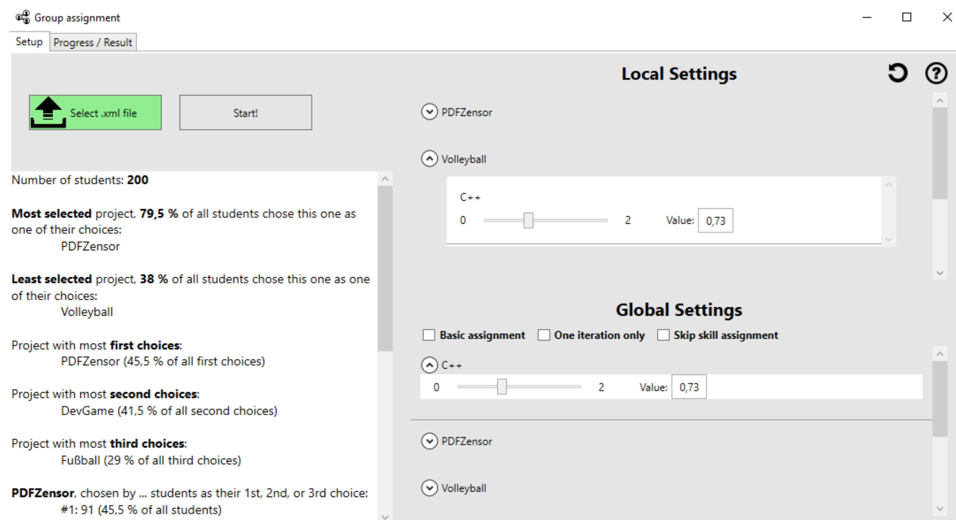

Abbildung 4.7: Setup-Tab der graphischen Oberfläche, größer im Anhang (s. Abbildung B.3)

In Abbildung 4.7 ist der Setup-Tab zu sehen, der angezeigt wird, wenn das Programm gestartet wird. Hier abgebildet ist der Zustand, nachdem die XML-Datei bereits ausgewählt und von der Anwendung statistisch ausgewertet wurde. Zur Beschreibung des ersten Reiters sei der sichtbare Bereich in drei Teile geteilt.

Links oben befinden sich der Auswahl-Knopf zum Einlesen der XML-Datei und der Start-Button zum Ausführen der Einteilung. Der „Select .xml file“-Button ist standardmäßig grau und verfärbt sich grün, wenn eine Datei erfolgreich ausgewählt wurde, um hier dem Nutzer oder der Nutzerin Feedback über den Auswahlprozess zu geben. Beim Drücken des Start-Knopfes werden außerdem die meisten Elemente in diesem Tab deaktiviert, um ein Verfälschen der Werte während der Einteilung zu verhindern.

Der linke untere Bereich zeigt verschiedene Statistiken über den Inhalt der eingelesenen Datei an. So werden hier zum Beispiel die Anzahl der

Studierenden, aber auch Daten wie das Projekt mit den meisten Erststimmen oder die Anzahl der angegebenen Fähigkeiten der Studierenden ausgegeben. Diese Daten dienen einerseits zur Übersicht und Kontrollmöglichkeit der Eingabedatei, andererseits auch als Orientierung zum Einstellen der Fähigkeiten auf der rechten Seite.

Die gesamte rechte Seite dient zur Gewichtung der Fähigkeiten der Studierenden. Diese können hier durch einen Regler, der multiplikativ wirkt, verstärkt oder abgeschwächt werden. Es gibt lokale und globale Regler, wobei sich im oberen Bereich die lokalen Regler befinden, die Einstellungen für ein einzelnes Projekt vornehmen können. Die globalen Regler darunter beeinflussen nur die lokalen Regler und haben keine eigene Funktion auf den Programmablauf gesehen. Die Checkbox „Basic assignment“ entscheidet über das Suchen nach einer schnellen Lösung (s. Abschnitt 4.2).

Die anderen beiden Checkboxen „One iteration only“ und „Skip skill assignment“ sind beides Maßnahmen zur Verkürzung der Laufzeit des Algorithmus. Wie im Kapitel zur Laufzeit beschrieben ist, hat der Algorithmus eine schlechte Laufzeit, die sich bemerkbar machen kann, wenn die Gruppengrößen ansteigen. Hiermit kann dem entgegen gewirkt werden, denn die erste der beiden genannten Checkboxen sorgt dafür, dass nur eine von sechs Iterationen des Hauptalgorithmus berechnet wird. Die zweite Checkbox verkürzt die Laufzeit noch weiter, da sie dafür sorgt, dass nur der erste und effizientere Teil des Hauptalgorithmus ausgeführt wird.

Abschließend sind rechts oben zwei Icons zum Zurücksetzen aller Eingaben und zum Anzeigen eines Hilfe-Pop-ups zu sehen.

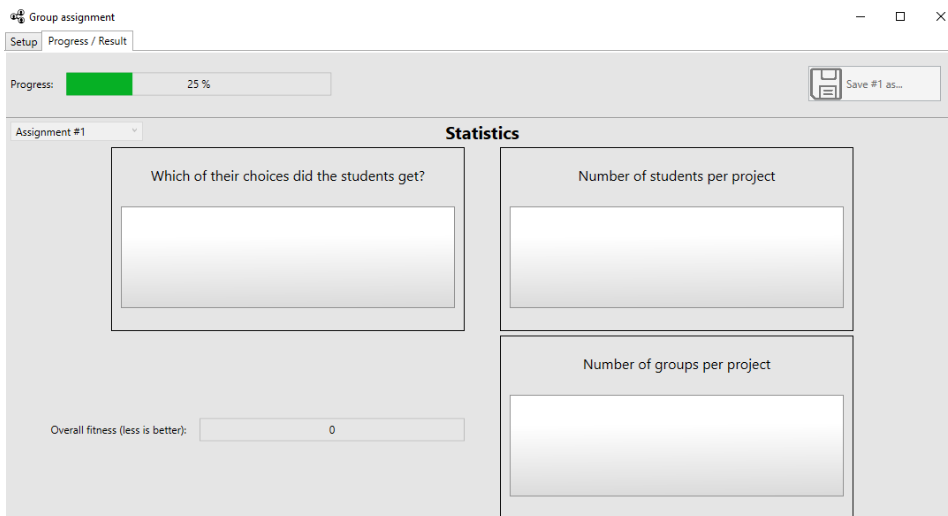

Abbildung 4.8: Fortschrittsanzeige der graphischen Oberfläche, größer im Anhang (s. Abbildung B.4)

Nach dem Start der Einteilung wechselt die Oberfläche auf den zweiten Reiter und zeigt hier einen Fortschrittsbalken an (s. Abbildung 4.8). Ist die Einteilung abgeschlossen, füllen sich die noch leeren Statistik-Container, die im Folgenden beschrieben und in Abbildung 4.9 gezeigt werden.

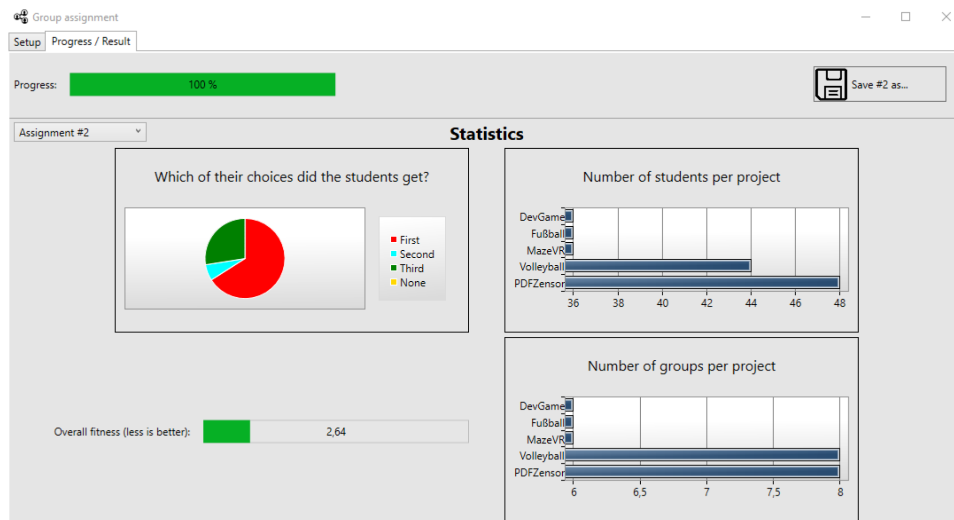

Abbildung 4.9: Ergebnis nach erfolgter Zuteilung, größer im Anhang (s. Abbildung B.5)

Sofern keine schnelle Lösung gefunden wurde und alle sechs Iterationen des Programms abgearbeitet wurden, ist das Ausklappmenü links oben aktiviert und ermöglicht es, die sechs Zuteilungen miteinander zu vergleichen, die das Ergebnis der sechs Durchläufe der Kernphasen des Programms sind. Die vier Graphen zeigen jeweils die Auswertung zu einer bestimmten Zuteilung an: Links oben wird dargestellt, wie viele Studierende welchen ihrer Wünsche bekommen haben, links unten ist die Gesamtbewertung der Gruppen berechnet (0 bedeutet vollständig ausgeglichene Gruppen), und rechts wird jeweils die Anzahl der Gruppen beziehungsweise Studierenden angezeigt.

Rechts oben befindet sich der Knopf zum Speichern der ausgewählten Zuteilung. Einige Elemente haben darüber hinaus noch kleine Tooltipps, die eingeblendet werden, wenn mit der Maus über ein Element gefahren wird.

4.4.3 Fehlerbehandlung und Überprüfung der Eingaben

Weil während der Einteilung fehlerhafte Eingabedaten zu unbestimmtem Verhalten der Anwendung führen können, wird die Eingabedatei beim Einlesen untersucht und macht den Nutzer oder die Nutzerin auf Fehler aufmerksam.

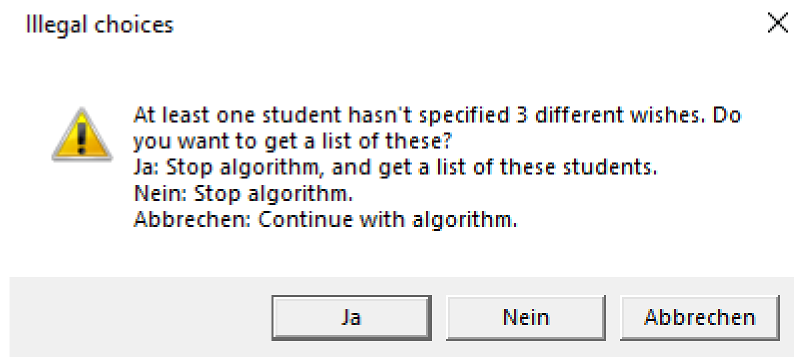

Abbildung 4.10: Unerlaubte Projektwahl

Einer dieser Fehler ist eine unerlaubte Projektwahl der Studierenden (s. Abbildung 4.10). In diesem Fall kann das im Rahmen dieser Arbeit entwickelte Programm eine Liste erstellen, welche die betroffenen Studierenden auflistet, sodass diese manuell korrigiert werden können, wenn es gewünscht ist.

Kapitel 5

Tests und Evaluierung

Um das im Rahmen dieser Arbeit entwickelte Programm testen zu können, wurden über ein Skript zufällige Eingaben als XML-Datei generiert. Hierbei wurde über eine Variable, die die Extremität aller Projektwahlen insgesamt beeinflusst, darauf geachtet, dass die Projekte mit unterschiedlicher Häufigkeit auftreten. Eine Extremität von 1 bedeutet, dass alle Studierenden die gleichen Projekte in ihren drei Wünschen gewählt haben, 0 heißt, dass jede Wahl komplett zufällig generiert wird.

Insgesamt wurden zwei Testläufe mit mehreren Verteilungen durchgeführt. Die ersten zehn Verteilungen beziehen sich auf 200 Studierende, die auf ungefähr 32 Gruppen aufgeteilt werden müssen, um eine realitätsnahe Quote von etwa sechs Studierenden je Gruppe zu erhalten. Die Verteilungen des zweiten Durchlaufs beziehen sich auf deutlich weniger Studierende, bei dem die im Ergebnis zugeteilten Gruppen aber vorausgesagt werden können, um so die Korrektheit des Algorithmus zu überprüfen, ob dieser also die erwartete Anzahl Studierende in ein unerwünschtes Projekt einteilt.

Durchlauf	Meistgewähltes Projekt in %	Zuordnungsdauer in Sek.	Nicht-Wünsche	
			Anzahl	Anteil in %
1	41,5	44	0	0
2	55	51	0	0
3	63,5	49	0	0
4	69,5	50	0	0
5	69,5	51	0	0
6	73,5	53	0	0
7	79,5	53	0	0
8	83,5	50	48	24
9	85,5	47	72	36
10	100	55	38	19

Tabelle 5.1: Überblick über Verteilung von 200 Studierenden

Untersucht wurde im ersten Testlauf das Gesamtverhalten des Hauptalgorithmus, was also beide Hauptteile (Ungarische Methode [21] und Gruppenzuweisung) und die sechs Iterationen davon einschließt. Die Fähigkeiten der Studierenden wurden dabei jedes Mal zufällig generiert, unabhängig von der Extremitätsvariable. Alle Studierenden besaßen daher je fünf Einträge für programmiertechnische und zwischenmenschliche Fähigkeiten, jeweils mit einem zufällig generierten Wert zwischen 0 und 3, was die beiden Grenzen des Anmeldewerkzeugs sind. Die Ergebnisse sind in Tabelle 5.1 zu finden.

Eine Zeile in Tabelle 5.1 ist folgendermaßen zu lesen: Das „meistgewählte Projekt“ ist das Projekt, das am häufigsten als einer der drei Projektwünsche auftaucht; in Durchlauf 1 hatten zum Beispiel 41,5 % aller Studierenden das meistgewählte Projekt ausgewählt. Die Zuordnungsdauer beschreibt die Dauer vom Drücken des „Start“-Knopfes bis zum Ausgeben des Ergebnisses. Die „Nicht-Wünsche“ sind die Zuteilungen, bei denen ein Student oder eine Studentin keinen seiner oder ihrer drei Wünsche erfüllt bekommen hat.

Durchlauf	Anzahl Projektwünsche[1]	Gruppengröße	Nicht-Wünsche[2]	
			SOLL-Wert	IST-Wert
1	3	3	0	0
2	2	3	1	1
3	1	3	2	2
4	0	3	3	3
5	2	6	4	4
6	1	6	5	5
7	0	6	6	6

Tabelle 5.2: Überprüfung auf erwartete Ergebnisse

Um im zweiten Durchlauf die Ergebnisse voraussagen zu können, ist es notwendig, dass alle Gruppen, in welche die Studierenden eingeteilt werden, die gleiche Größe haben müssen. Andernfalls ist es zum Testen nicht mehr nachvollziehbar, welche Gruppe am Ende welche Größe haben wird, weil dies mit der Platzhalter-Auffüllung zusammenhängt, die durch erneute Anwendung der Ungarischen Methode [21] gelöst wird. Eine Vorhersage der Ergebnisse würde eine manuelle Lösung dieses Algorithmus verlangen, was auch bei kleinen Eingabegrößen nicht praktikabel ist.

Ist an diesem Punkt also sichergestellt, dass alle Gruppen die gleiche Größe haben, können die Projektwünsche in der dafür manuell erstellten XML-Datei entsprechend angepasst werden: Sollen Gruppen der Größe 3 entstehen, es sich aber nur 2 Studierende insgesamt ein Projekt wünschen, wird klar, dass hier exakt ein Student oder eine Studentin ein unerwünschtes Projekt erhalten wird, da es keine Gruppen geben kann, die weniger als 3

[1] Jeweils auf das unbeliebteste Projekt bezogen.
[2] Absolute Werte

Mitglieder haben. Die anderen Studierende beziehungsweise Projektwünsche wurden anschließend gleichmäßig in der XML-Datei aufgeteilt.

Die Fähigkeiten der Studierenden wurden nicht in die Eingabedatei aufgenommen, da diese das am Ende zugeteilte Projekt nicht beeinflussen. Die gewonnenen Erkenntnisse sind in Tabelle 5.2 aufgelistet.

Tabelle 5.2 ist genau wie Tabelle 5.1 zu lesen. Die veränderte Spalte „Anzahl Projektwünsche“ beschreibt, wie oft ein Projekt gewünscht wurde. Sollen zum Beispiel Gruppen der Größe 6 gefüllt werden, bedeutet eine 2, dass sich nur 2 Studierende das unbeliebteste Projekt gewünscht haben. Die Gruppengröße ist die Größe aller Gruppen in dieser Verteilung. Die Zuordnungsdauer bei allen Verteilungen des zweiten Testlaufs betrug in etwa eine Sekunde, begründet durch die geringe Größe und die fehlenden Fähigkeitsangaben in der Eingabedatei.

In beiden Tabellen ist zu erkennen, dass das Programm in relativ kurzer Zeit eine Lösung findet, die nur wenig Studierende unzufrieden zurücklässt. Die leichte Abweichung in den späteren Durchläufen in Tabelle 5.1 ist durch die zufällige Generierung der Eingabedaten zu erklären. Diese Ergebnisse sind darüber hinaus auch in diesem Rahmen zu erwarten, da es aufgrund der begrenzten Gruppengrößen dazu kommen kann, dass zum Beispiel 25 Personen ein Projekt A als ihren Erstwunsch angegeben haben, während dieses aber nur Platz für zehn Studierende bereit hält. Dies kann auch bei Zweit- und Drittwünschen auftreten.

Durchlauf	Gruppengröße	Zuordnungsdauer in Min.
1	5	< 1
2	6	< 1
3	7	1,5
4	8	4
5	10	23
6	12	ca. 360

Tabelle 5.3: Zuordnungsdauer bei verschiedenen Gruppengrößen

Abschließend ist noch eine Auswertung von Laufzeiten für größere Gruppen in Abbildung 5.3 zu sehen. Die Anzahl der Studienanfänger an der Leibniz Universität Hannover im Bereich Informatik ist in den letzten Jahren ungefähr auf dem selben Niveau geblieben [32] und das Software-Projekt hatte im Wintersemester 2016/2017 eine Teilnehmerzahl von 97 [17]. Es ist also davon auszugehen, dass die Teilnehmerzahlen des Software-Projekts sich weiter in diesem Rahmen bewegen werden. Getestet wurde hier mit 180 Studierenden, was als obere Grenze für das Software-Projekt verstanden werden kann.

Aufgrund der Verwendung des Binomialkoeffizienten im zweiten Teil des Algorithmus, der direkt mit der Fakultät einhergeht, verlängert sich die Zu-

ordnungszeit bei kleinen Steigerungen der Gruppengrößen sehr schnell. Hier liegt Optimierungspotenzial, besonders, um die Fakultät in der Berechnung entweder klein zu halten oder gänzlich zu entfernen.

Die Gruppengrößen 5 bis 12 stellen realistische Größen dar, wobei alle Größen bis einschließlich 10 noch in einem vertretbaren Rahmen liegen. Größen, die darüber hinausgehen, sollten entweder durch eine der Einstellmöglichkeiten in der Benutzeroberfläche in ihrer Rechenzeit abgekürzt werden oder nur berechnet werden, wenn das Ergebnis nicht unmittelbar vorliegen muss.

Kapitel 6

Fazit und Ausblick

In diesem Kapitel werden die während dieser Arbeit gefundenen Erkenntnisse zusammenfassend aufgeführt. Weiterhin werden einige Anmerkungen gegeben, wie das Zuordnungswerkzeug weiter verbessert und optimiert werden kann.

6.1 Zusammenfassung

Im Rahmen dieser Arbeit sollte für das Fachgebiet Software Engineering der Leibniz Universität Hannover ein Werkzeug entwickelt werden, das eine Zuteilung von Studierenden auf Software-Teams (des Software-Projekts in diesem Anwendungsfall) anhand von zuvor getroffenen Projektwünschen und Fähigkeitsangaben finden soll, die so viele dieser Wünsche erfüllt wie möglich. Diese Eingabedaten entstammen einem Anmeldewerkzeug, das die Anmeldungen der Studierenden zum Software-Projekt entgegennimmt und die dort getätigten Eingaben in einer XML-Datei abspeichert. Diese Datei bildet den Startpunkt für das im Rahmen dieser Arbeit entwickelte Tool.

Um eine Zuteilung zu finden, wurde zunächst die Ungarische Methode angewandt, ein Zuordnungsalgorithmus, der anhand von Prioritäten Studierende auf Gruppenplätze zuteilen kann. Die erste Realisierung der Methode erfolgte als Konsolenanwendung, die in einem späteren Schritt zu einer Anwendung mit einer graphischen Benutzeroberfläche umgeschrieben wurde. Zum Erhalten der Lösung – das eigentliche Zurückgeben der finalen Zuteilung ist nicht mehr Teil der Ungarischen Methode – wurden graphentheoretische Algorithmen, wie zum Beispiel der Hopcroft-Karp-Algorithmus, verwendet, nachdem das vorliegende Zuordnungsproblem als Graph interpretiert wurde.

Die Zuteilung als Ergebnis der Ungarischen Methode und der anschließenden Algorithmen ist in dieser Arbeit ebenfalls auch der Ausgangspunkt für eine verfeinerte Verteilung von Studierenden innerhalb von Projektgruppen. Insgesamt ermöglicht dieses Programm einerseits eine Zuteilung über

Prioritäten bei Projektwünschen, andererseits auch eine darauf folgende Ausgeglichenheit der Projektgruppen, welche die Zufriedenheit innerhalb einer Gruppe und damit den Lerneffekt positiv beeinflusst.

Mit dem Zusatz von Statistiken, die sowohl bei der Eingabe als auch bei der Ausgabe der Daten zum besseren Verständnis über die Zuteilung angezeigt werden, findet dieses Programm also eine Lösung, die so wenig Studierende wie möglich mit einem zugeteilten Projekt zurücklässt, das nicht zuvor gewünscht wurde.

Unter realitätsnahen Bedingungen findet die Anwendung überwiegend in verhältnismäßig kurzer Zeit eine Lösung, je nach Studierendenzahl und Gruppengröße wenige Sekunden bis zu mehreren Minuten – das war bei den bisher verfolgten Lösungsansätzen ein größerer zeitlicher Aufwand. Weitere Tests ergaben, dass die Zuteilung genau die erwarteten Ergebnisse errechnet, damit also korrekt funktioniert und eine optimale Lösung liefert. Diese Validierung erfolgte über eine manuell erstellte XML-Datei, in der die final zugeteilten Projekte mit geringem Aufwand direkt ermittelt werden konnten, um so ein Kontrollergebnis liefern zu können.

6.2 Ausblick

Auch wenn das Programm bei kleinen Gruppengrößen eine Zuteilung in kurzer Zeit findet, gibt es dennoch einige Teile, die in künftigen Arbeiten behandelt werden können. So ist zum Beispiel das Thema Optimierung ein wichtiger Punkt. Aktuell läuft der Algorithmus nur mit einer sehr schlechten Laufzeit, die für Personenanzahlen und Gruppengrößen in einer größeren Dimension zu einem Problem werden kann. Um dies zu verwirklichen, sollte die Implementierung des zweiten Hauptteils angepasst und mit zukünftigen, effizienteren Ansätzen kombiniert werden.

Weitere Erweiterungsmöglichkeiten sind eine graphische Überarbeitung der Oberfläche des Programms, um diese ansprechender erscheinen zu lassen oder weitere Einstellmöglichkeiten einzufügen, wie zum Beispiel die zuvor genannten Wunschmöglichkeiten nach Gruppenpartnerinnen und Gruppenpartnern oder Tutorinnen und Tutoren. Auch könnte der Fortschrittsbalken um eine Anzeige der zu erwartenden verbleibenden Zeit ergänzt werden.

Neben der Zuteilung am Ende gäbe es weiterhin noch die Möglichkeit zur Erweiterung, Tendenzen zwischen Studierenden und Projektwünschen ausgeben zu lassen, die einen Zusammenhang zwischen der oder dem Studierenden selbst und ihrer oder seiner Wünsche herstellen. Als Beispiel könnte eine solche Tendenz ergeben, dass ein Projekt, in welchem eine Web-Anwendung entwickelt werden soll, eher von Studierenden gewählt wird, die gute JavaScript-Kenntnisse angegeben haben. Auf diese Weise könnte für zukünftige Durchläufe schon eine grobe Prognose erstellt werden, wie die Wünsche ausfallen werden.

Unter der Prämisse von angemessen großen Problemstellungen oder einem gefundenen, effizienteren Verfahren für die Zuordnung ist dieses Programm leicht auch für andere Gebiete als das Software-Projekt einsetzbar, bei denen Personen auf Gruppen verteilt werden müssen. Darüber hinaus kann dieses Programm Personen anhand ihrer persönlichen Fähigkeiten noch ausgeglichener einteilen und so eine Zuteilung erstellen, die sowohl Gruppenwünsche berücksichtigt als auch Gruppen bildet, die im Vergleich zueinander ausgeglichen sind.

Dies macht es zu einem guten Hilfsmittel bei einzuteilenden Personenmengen, die ohne maschinelle Unterstützung zu komplex erscheinen und bei denen die Personen möglicherweise selbst noch spezielle Fähigkeiten besitzen.

Anhang A

Pseudoalgorithmen

Algorithmus 1: Ungarische Methode

```
Input: Studierende S, Projekte P
Output: Zuteilung Z
matrix[][] ← fillMatrix(S, P)
makeMatrixSquare(matrix)
forall Zeilen i in matrix do
    s ← getMinElement(matrix, i)
    forall Element e in i do
        matrix[e][] ← matrix[e][] - s
    end
end
forall Spalten j in matrix do
    s ← getMinElement(matrix, j)
    forall Element e in j do
        matrix[][e] ← matrix[][e] - s
    end
end
while true do
    deletions ← getMinimumDeletions()
    if deletions = S.count() then return findUniqueZeros()
    h ← getSmallestUndeleted()
    forall Elemente e in matrix do
        if e.deletions = 2 then matrix[e] + h
        if e.deletions = 0 then matrix[e] - h
    end
    resetAllDeletions()
end
```

Algorithmus 2: Minimale Knotenüberdeckung ermitteln [29]

```
Input: Matching M, Studierende S, Projekte P
Output: Knotenüberdeckung K

Function FindMinimumVertexCover(M, S, P) : K
    u ← S \ M.getAllStudents())          // Nicht im Matching
     enthaltene Knoten
    v ← ∅                    // Besuchte Knoten

    forall Knoten w in u do
        v ← v ∪ FindAlternatingVertices(M, w, v)
    end

    v ← u ∪ v
    return (S \ v) ∪ (P ∩ v)
end
Function FindAlternatingVertices(M, w, v) : v
    if w ∈ v then return ∅

    v.add(w)
    u ← w.getIncidentEdges() \ M          // Nicht im Matching
     enthaltene Kanten

    forall Kanten e in u do
        x ← e.getProject()          // Besuchter Knoten
        y ← ∅
        forall Kanten m in M do    // Kante im Matching suchen
            if m.getProject() = x then y ← m
        end

        if y ≠ ∅ then                  // Kante ist im Matching
            v.add(x)
            x ← y.getStudent()
            v ← v ∪ FindAlternatingVertices(M, x, v)
        end
    end
    return v
end
```

Algorithmus 3: Hopcroft-Karp-Algorithmus [28]

```
Input: Erste Knotenpartition U, Zweite Knotenpartition V, Knoten
       NIL
Output: Matchinggröße M
Function BFS()
    forall Knoten u in U do
        if PairU[u] = NIL then
            Dist[u] ← 0
            Enqueue(Q, u)
        else
            Dist[u] ← ∞
        end
    end
    Dist[NIL] ← ∞
    while Empty(Q) = false do
        u ← Dequeue(Q)
        if Dist[u] < Dist[NIL] then
            forall Knoten v in Adj[u] do
                if Dist[PairV[v]] = ∞ then
                    Dist[PairV[v]] ← Dist[u] + 1
                    Enqueue(Q, PairV[v])
                end
            end
        end
    end
    return Dist[NIL] ≠ ∞
end
Function DFS()
    if u ≠ NIL then
        forall Knoten v in Adj[u] do
            if Dist[PairV[v]] = Dist[u] + 1 then
                if DFS(PairV[v]) = true then
                    PairV[v] ← u
                    PairU[u] ← v
                    return true
                end
            end
        end
        Dist[u] ← ∞
        return false
    end
    return true
end
```

```
Function Hopcroft-Karp
    forall Knoten u in U do
        PairU[u] ← NIL
    end
    forall Knoten v in V do
        PairV[v] ← NIL
    end
    M ← 0
    while BFS() = true do
        forall Knoten u in U do
            if PairU[u] = NIL then
                if DFS(u) = true then
                    M ← M + 1
                end
            end
        end
    end
    return M
end
```

Algorithmus 4: Fähigkeitsbasierte Teameinteilung

```
Input: Zuteilung Z
Output: Zuteilung Z

P[] ← Z.getAllProjects()
forall Projekte p in P do
    if p.countGroups() = 1 then continue
    G[] ← p.getGroups()
    oldScore ← ∞
    while true do
        optimals[] ← ∅
        forall 2er-Kombinationen k in G do
            optimals.add(bestStudentDistributionWithin2Groups(k))
        end
        c[] ← G.getGroupCombinations()
        newScore ← getBestGroupCombination(c, optimals)
        if newScore ≥ oldScore then break
        oldScore ← newScore
    end
end
Z = ApplyNewGroupsToAssignment()
return Z
```

Anhang B

Grafiken und Bilder

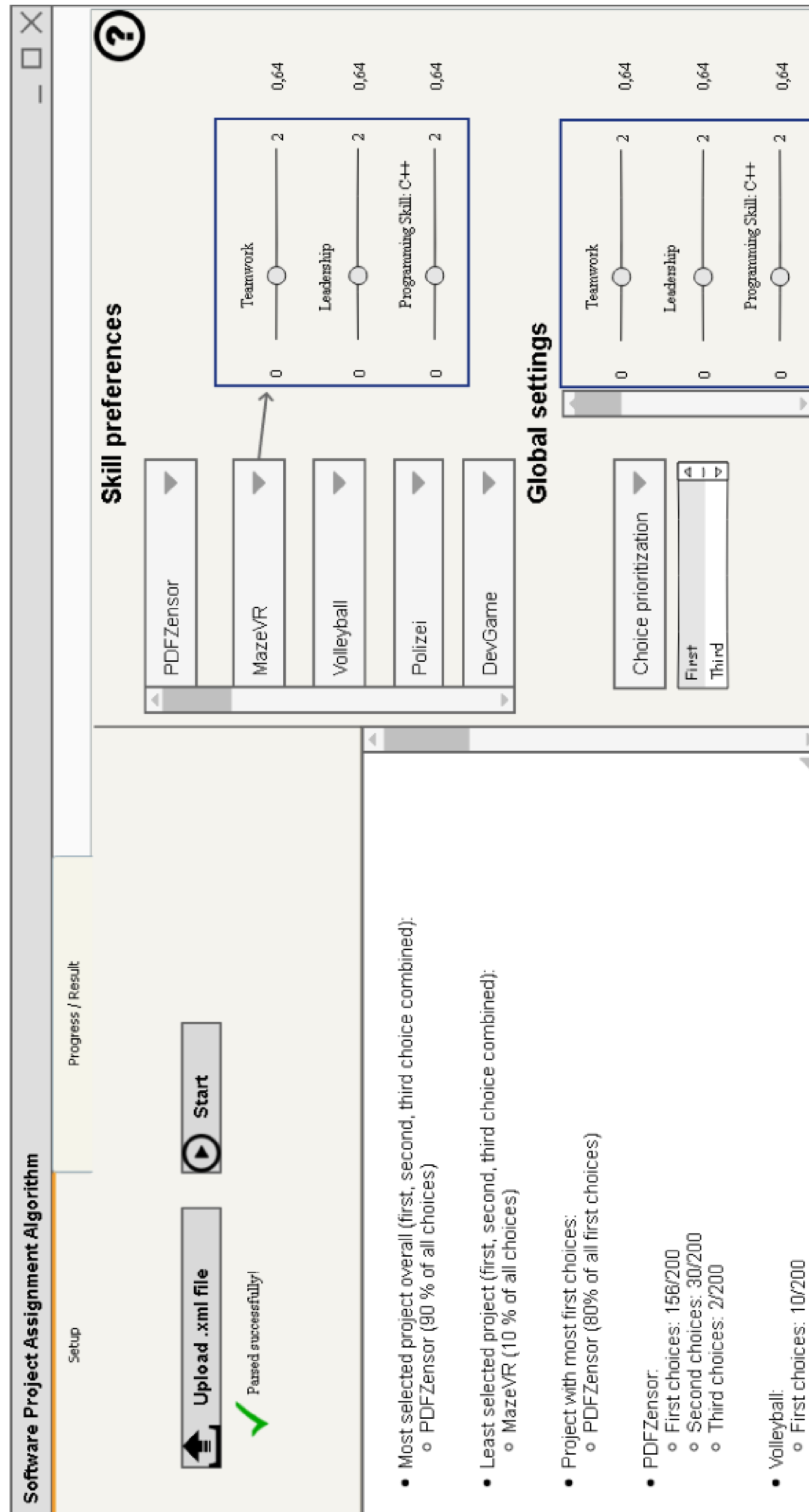

Abbildung B.1: Setup-Tab des finalen Design-Mockups

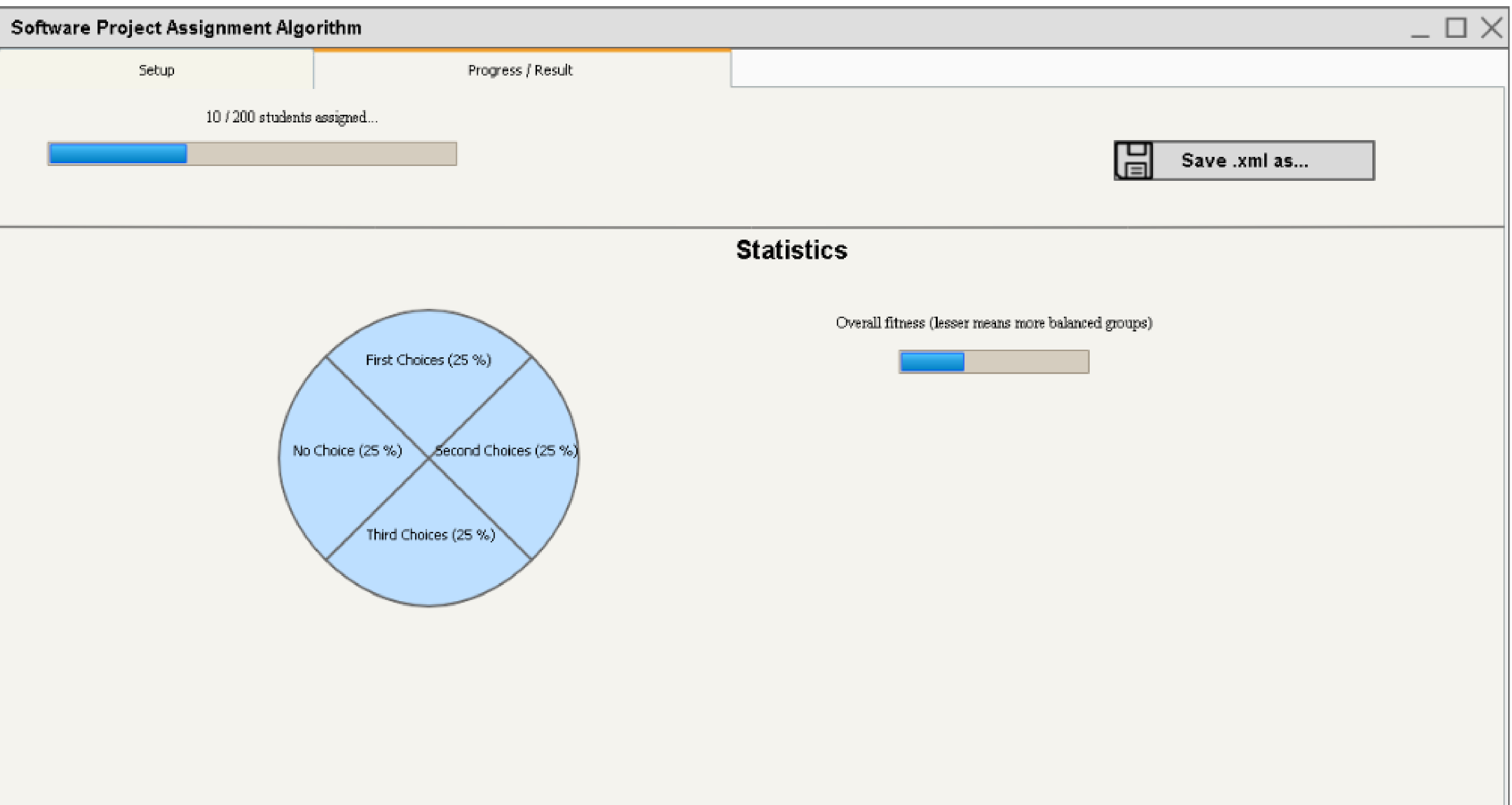

Abbildung B.2: Ergebnis-Tab des finalen Design-Mockups

Abbildung B.3: Setup-Tab der graphischen Oberfläche

Abbildung B.4: Fortschrittsanzeige der graphischen Oberfläche

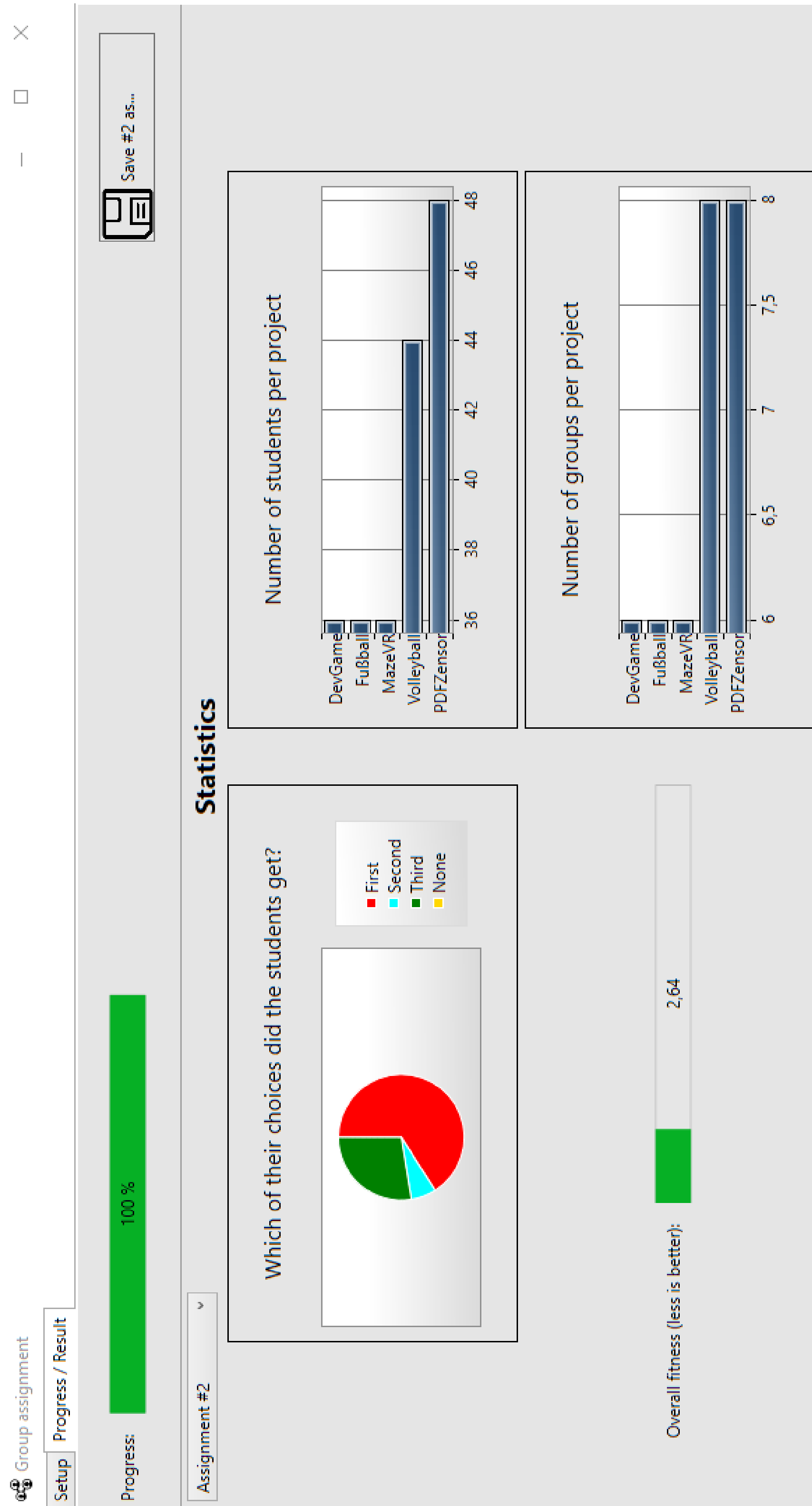

Abbildung B.5: Ergebnis nach erfolgter Zuteilung

Abbildungsverzeichnis

2.1 Optimale Zuordnung mit Eignungen für das obige Beispiel . . 6
2.2 Bipartiter Graph mit gewichteten Kanten 8
2.3 Perfektes Matching 9
2.4 Maximum-Matching 9
2.5 Iteration 1 10
2.6 Iteration 2 10
2.7 Minimale Knotenüberdeckung 11
2.8 Problem der linearen Optimierung 13
2.9 Primales LP 13
2.10 Duales LP 13
2.11 Erstellung einer (4×4)-Matrix mit Prioritäten 16
2.12 Matrix nach Subtraktion der Zeilen- und Spaltenminima . . . 16
2.13 Matrix mit $n_1 = 3 < n = 4$ gestrichenen Zeilen und Spalten . 16
2.14 Veränderte Matrix mit kleinstem ungestrichenen Element $h_1 = 1$ 17
2.15 Matrix mit $n_1 = 4 = n = 4$ gestrichenen Zeilen und Spalten . 17
2.16 Eine optimale Lösung, gezeigt ist die Ausgangsmatrix 18

4.1 Programmaufbau mit den zwei zentralen Teilschritten 27
4.2 Studierende der drei Gruppen $A1$, $A2$ und $A3$ 34
4.3 Kombinationsmöglichkeiten bei drei Gruppen 34
4.4 Bestes Verhältnis der Gruppen $A1$ und $A2$ 34
4.5 Setup-Tab des finalen Design-Mockups, größer im Anhang (s. Abbildung B.1) 36
4.6 Ergebnis-Tab des finalen Design-Mockups, größer im Anhang (s. Abbildung B.2) 37
4.7 Setup-Tab der graphischen Oberfläche, größer im Anhang (s. Abbildung B.3) 38
4.8 Fortschrittsanzeige der graphischen Oberfläche, größer im Anhang (s. Abbildung B.4) 39
4.9 Ergebnis nach erfolgter Zuteilung, größer im Anhang (s. Abbildung B.5) 40
4.10 Unerlaubte Projektwahl 41

B.1 Setup-Tab des finalen Design-Mockups 56
B.2 Ergebnis-Tab des finalen Design-Mockups 57
B.3 Setup-Tab der graphischen Oberfläche 58
B.4 Fortschrittsanzeige der graphischen Oberfläche 59
B.5 Ergebnis nach erfolgter Zuteilung 60

Tabellenverzeichnis

2.1 Eignung von Personen für Aufgaben 6
2.2 Verbrauch und Vorrat der Zutaten 12

5.1 Überblick über Verteilung von 200 Studierenden 43
5.2 Überprüfung auf erwartete Ergebnisse 44
5.3 Zuordnungsdauer bei verschiedenen Gruppengrößen 45

Literaturverzeichnis

[1] C. Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences*, 43(9):842–844, 1957.

[2] J. A. Bondy, U. S. R. Murty, et al. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[3] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics, 2012.

[4] S. Cai, K. Su, C. Luo, and A. Sattar. Numvc: An efficient local search algorithm for minimum vertex cover. *Journal of Artificial Intelligence Research*, 46:687–716, 2013.

[5] G. Carpaneto and P. Toth. Primal-dual algrorithms for the assignment problem. *Discrete Applied Mathematics*, 18(2):137–153, 1987.

[6] C.-H. Chuang, Y.-N. Chen, L.-W. Tsai, C.-C. Lee, and H.-C. Tsai. Improving learning performance with happiness by interactive scenarios. *The Scientific World Journal*, 2014:12, 2014.

[7] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS Journal on Computing*, 11(2):138–148, 1999.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, Dec. 1959.

[9] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[10] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, Apr. 1972.

[11] J. Egerváry. Matrixok kombinatorius tulajdonságairól. *Matematikai és Fizikai Lapok*, 38:16–28, 1931.

[12] A. E. Eiben and J. E. Smith. What is an evolutionary algorithm? In *Introduction to Evolutionary Computing*, pages 25–48, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[13] M. M. Flood. The traveling-salesman problem. *Operations Research*, 4(1):61–75, 1956.

[14] Z. Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18(1):23–38, Mar. 1986.

[15] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[16] S. Khuri and T. Bäck. An evolutionary heuristic for the minimum vertex cover problem. *Genetic Algorithms within the Framework of Evolutionary Computation–Proc. of the KI-94 Workshop*, pages 86–90, 1994.

[17] J. Klünder, O. Karras, F. Kortum, M. Casselt, and K. Schneider. Different views on project success when communication is not the same. In *International Conference on Product-Focused Software Process Improvement*, PROFES 2017, pages 497–507. Springer, 2017.

[18] J. Klünder, O. Karras, F. Kortum, and K. Schneider. Forecasting communication behavior in student software projects. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2016, New York, NY, USA, 2016. Association for Computing Machinery.

[19] P. A. Krokhmal and P. M. Pardalos. Random assignment problems. *European Journal of Operational Research*, 194(1):1–17, 2009.

[20] S. Krumke and H. Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. Lehrbuch : Informatik. Teubner, 2005.

[21] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[22] D. König and H. Sachs. *Theorie der endlichen und unendlichen Graphen, Teubner-Archiv zur Mathematik*, volume 6. Teubner;, Leipzig, 1986.

[23] L. A. MacVittie. *XAML in a Nutshell : [a desktop quick reference]*. O'Reilly, 2006.

[24] K. Mosler, R. Dyckerhoff, and C. Scheicher. *Mathematische Methoden für Ökonomen*. Springer Gabler, Berlin, Heidelberg, 2018.

[25] T. E. Muller. Assigning students to groups for class projects: An exploratory test of two methods. *Decision Sciences*, 20(3):623–634, 1989.

[26] E. Munapo. Development of an accelerating hungarian method for assignment problems. *Eastern-European Journal of Enterprise Technologies*, 4(4 (106)):6–13, Aug. 2020.

[27] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.

[28] o. V. Hopcroft-Karp algorithm. `https://en.wikipedia.org/wiki/Special:PermaLink/1037485999#Pseudocode`. letzer Zugriff: 19.09.21.

[29] o. V. Kőnig's theorem (graph theory). `https://en.wikipedia.org/wiki/Special:PermaLink/1042450324#Constructive_proof`. Algorithmus aus Beweis abgeleitet. letzer Zugriff: 19.09.21.

[30] o. V. Softwareprojekt 2020/2021. `https://www.pi.uni-hannover.de/de/se/lehre/swp/`. letzer Zugriff: 06.09.21.

[31] o. V. Top500. June 2021. `https://www.top500.org/lists/top500/2021/06/`. letzer Zugriff: 16.09.21.

[32] o. V. Zahlenspiegel 2020 der leibniz universität hannover. `https://www.uni-hannover.de/fileadmin/luh/content/planung_controlling/statistik/zahlenspiegel/zahlenspiegel_2020.pdf`. letzer Zugriff: 02.10.2021.

[33] D. W. Pentico. Assignment problems: A golden anniversary survey. *European Journal of Operational Research*, 176(2):774–793, 2007.

[34] N. Tomizawa. On some techniques useful for solution of transportation network problems. *Networks*, 1(2):173–194, 1971.

[35] I. H. Toroslu and Y. Arslanoglu. Genetic algorithm for the personnel assignment problem with multiple objectives. *Information Sciences*, 177(3):787–803, 2007.

[36] D. F. Votaw and A. Orden. The personnel assignment problem. *Symposium on Linear Inequalities and Programming*, pages 155–163, 1952.