

Aufgabe 2: Spiessgesellen

Teilnahme-ID: 56727

Bearbeiter dieser Aufgabe:
Carl Friedrich Mecking

18. April 2021

Inhaltsverzeichnis

1	Betrachtung des Problems	3
1.1	Lösungsansatz	3
2	Lösungsidee	3
2.1	Mengentheoretische Herleitung	3
2.2	Aufbau des Graphen	4
2.3	Zuweisung von Obstsorten und Schüsseln	5
2.3.1	Die Zuweisung	5
2.3.2	Die Reduzierung des Graphen	6
3	Umsetzung	7
3.1	Bereitstellung der Daten	7
3.2	Generierung der Adjazenzmatrix	7
3.3	Generierung der Match-List	8
3.4	Zuweisung der Schlüsselnummern zu den Obstsorten	8
3.4.1	Phase I: Zuweisung von unbeobachteten Sorten	8
3.4.2	Reduzierung um mehrere Knoten	9
3.4.3	Phase II: Informationssammlung	9
3.4.4	Phase III: Zuweisung von Knoten einmaliger Kantenkonstellation	10
3.4.5	Reduzierung um zwei Knoten	10
3.4.6	Phase IV: Zuweisung von Knoten mehrfach vorkommender Kantenkonstellation	11
3.5	Ausgabe	12
4	Beispiele	13
4.1	Dateiauswahl	13
4.2	Algorithmische Lösung	13
4.3	Ausgabe	14
5	Laufzeitverhalten	14
5.1	Phase I	15
5.2	Phase II	15
5.3	Phase III	15
5.4	Phase IV	16
5.5	Bestcase	16
5.6	Worstcase	17
6	Ergebnistabelle	17
7	Sourcecode	21
7.1	assignUnweighted	21
7.2	assignWeighted	22
7.3	removeSingle	25
7.4	removeMultiple	26

1 Betrachtung des Problems

Das Problem Spiessgesellen stellt ein bipartites Matchingproblem dar. Sei O die Menge der beobachteten Obstsorten auf den Spießen und S die Menge der zur Verfügung stehenden Schüsseln, so soll, sofern eine eindeutige Zuordnung möglich ist, die Anzahl n_S der Schüsseln ermittelt werden, in denen die gewünschte Teilmenge von Obstsorten $w \subseteq O$ zu finden ist. Dafür ist eine Anzahl n_B an Beobachtungen gegeben, welche immer eine Teilmenge von beobachteten Obstsorten $O_B \subseteq O$ und eine Teilmenge von Schüsseln $S_B \subseteq S$, umfassen.

1.1 Lösungsansatz

Um die Anzahl n_S der Zielschüsseln zu ermitteln, wird im Folgenden ein graphentheoretischer Ansatz vorgestellt, welcher anhand der Beobachtungen die Häufigkeit von Verbindungen zwischen gewählten Obstsorten und Schüsseln erkennt. Er nutzt diese, um Obstsorten eine eindeutige Schüssel zuzuweisen und durch die eindeutigen Zuweisungen, die Anzahl der möglichen Zuweisungen sukzessive verringert. Dadurch verringert sich die Laufzeit nach jeder Zuweisung. Ist keine eindeutige Zuordnung zu treffen, werden mehrere potenzielle Schüsseln einer Obstsorte zugewiesen, sodass bei der Ausgabe eine sinnvolle Aussage über die Anzahl der Zielschüsseln und deren Verhältnis zu den Wunschsorten getroffen werden kann.

2 Lösungsidee

Die Lösungsidee besteht aus zwei Hauptkomponenten. Im ersten Schritt wird ein bipartiter, gewichteter und ungerichteter Graph erstellt, welcher dazu verwendet wird, im nächsten Schritt den Schüsseln Obstsorten zuzuweisen. Nach jeder Zuweisung wird dann der Graph um die zugewiesenen Knoten verringert. Dies geschieht so lange, bis jede Obstsorte mindestens einer Schüssel zugeordnet wurde.

2.1 Mengentheoretische Herleitung

In diesem Abschnitt wird anhand eines einfachen Beispiels die mengentheoretische Grundlage der Lösungsidee gezeigt.

Mengen:

$O = \text{Obst}$, mit $o_i = \text{Sorten}$

$S = \text{Schüsseln}$, mit $s_i = \text{Schüssel}_{nr}$

Es sei:

$O = \{o_1, o_2, o_3, o_4, o_5\}$,

$S = \{s_1, s_2, s_3, s_4, s_5\}$,

$R = O \times S, G \subseteq R, G$ ist bipartit

$$R = \begin{pmatrix} (o_1, s_1), & \dots, & (o_1, s_5), \\ (o_2, s_1), & \dots, & (o_2, s_5), \\ \vdots & \ddots & \vdots \\ (o_5, s_1), & \dots, & (o_5, s_5) \end{pmatrix}$$

Beobachtungen: $B_i \subseteq R$

$B_1 = \{o_1, o_2, o_3\} \times \{s_1, s_2, s_3\}$

$$B_1 = \begin{pmatrix} (o_1, s_1), & (o_2, s_1), & (\mathbf{o_3, s_1}), \\ (o_1, s_2), & (o_2, s_2), & (o_3, s_2), \\ (o_1, s_3), & (o_2, s_3), & (o_3, s_3) \end{pmatrix}$$

$B_2 = \{o_3, o_4, o_5\} \times \{s_1, s_4, s_5\}$

$$B_2 = \begin{pmatrix} (\mathbf{o_3, s_1}), & (o_4, s_1), & (o_5, s_1) \\ (o_3, s_4), & (o_4, s_4), & (o_5, s_4) \\ (o_3, s_5), & (o_4, s_5), & (o_5, s_5) \end{pmatrix}$$

$$B_1 \cap B_2 = \{o_3, s_1\} \Rightarrow o_3 \longleftrightarrow s_1, o_3 \text{ bildet eineindeutig ab auf } s_1.$$

Allgemein gilt:

Relation $R = O \times S, I = \{1, \dots, n\}$, Beobachtung $B_i \subseteq R, i, j \in I, N_I \subseteq I$

$$|Q| = \left| \bigcap_{i \in N_I} B_i \right| = 0 \quad (1)$$

- Keine Zuordnung von o_i zu s_i möglich.

$$|Q| = \left| \bigcap_{i \in N_I} B_i \right| = 1 \quad (2)$$

- $Q = (o_i, s_j)$, damit ist eine Zuordnung von o_i nach s_j und umgekehrt eindeutig möglich.

$$|Q| = \left| \bigcap_{i \in N_I} B_i \right| = n_O \cdot n_S, \quad (3)$$

mit n_O Anzahl der Obstsorten in der Schnittmenge und n_S Anzahl der Schüsseln in der Schnittmenge.

- O_{N_I} wird definiert als Obstsorten in der Schnittmenge.
 S_{N_I} wird definiert als Schüsseln in der Schnittmenge.
 $Q = O_{N_I} \times S_{N_I}$, Q entspricht einem vollständigen bipartiten Teilgraphen.
 Eine eindeutige Zuordnung ist nicht möglich. Eine Zuordnung o_i, o_j zu s_i, s_j muss als Gruppe erfolgen.

2.2 Aufbau des Graphen

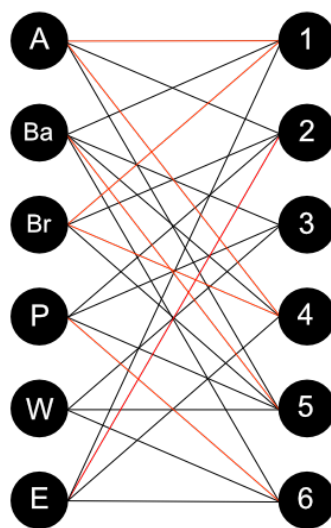


Abbildung 1: Vollständig generierter Graph

Die Abbildung 1 zeigt einen vollständig generierten Graphen auf Basis des Beispiels der Aufgabenstellung. Es ist zu erkennen, dass die Menge der beobachteten Obstsorten der Menge der beobachteten

Schüsseln als Knoten gegenübergestellt wird. Die Buchstaben entsprechen dabei den Anfangsbuchstaben der Obstsorten und die Zahlen den Nummern der Schüsseln.

Die verschiedenen Beobachtungen werden nun dazu verwendet, den leeren Graphen schrittweise mit gewichteten Kanten zu füllen. Jede Obstsorte $o \in O_B$ wird mit jeder Schüssel $s \in S_B$ einer Beobachtung verbunden. Tritt dabei eine Verbindung zum ersten Mal auf, erhält sie das Kantengewicht eins. Für jede weitere Verbindung zwischen einer Obstsorte und einer Schüssel wird nicht eine neue Kante eingefügt, sondern das Kantengewicht der bereits bestehenden Kante um eins erhöht.

Zu Darstellungszwecken ist in der Abbildung 1 kein Kantengewicht als Zahl angeführt, sondern als farbliche Markierung. Dabei entspricht eine schwarze Kante dem Gewicht eins und eine rote Kante dem Gewicht zwei. Mehr als zwei Verbindungen zwischen Obstsorte und Schüssel sind in diesem Beispiel nicht zu finden¹. Abbildung 2 zeigt wie der Graph, nach schrittweise Einfügen jeder Beobachtung, aufgebaut wird.

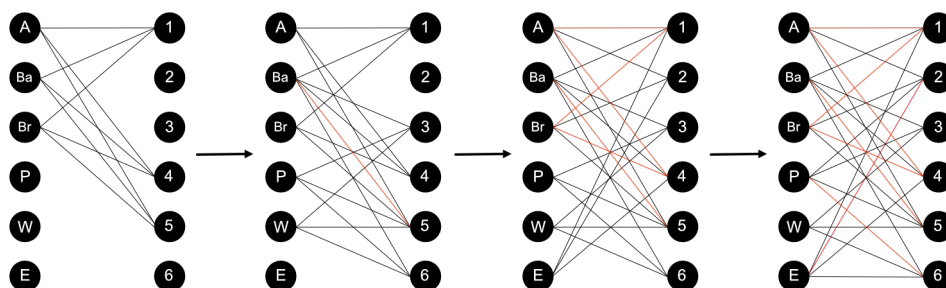


Abbildung 2: Aufbau des Graphen nach Beobachtungen

2.3 Zuweisung von Obstsorten und Schüsseln

Mithilfe des entstandenen Graphens kann nun erkannt werden, welche Verbindungen, zwischen Obstsorten und Schüsseln, am häufigsten auftreten. Das höchste Kantengewicht zwischen genau zwei Knoten führt bei Bildung der Schnittmenge über die dazu beitragenden Beobachtungen zu einer Mächtigkeit von eins der Schnittmenge, wonach eine eindeutige Zuordnung von Obstsorte zur Schüssel möglich ist. Das höchste Kantengewicht zwischen mehreren Knoten, lässt nur einen Rückschluss auf eine gruppierte Zuweisung zwischen Obstsorten und Schüsseln zu (siehe Abschnitt 2.1).

2.3.1 Die Zuweisung

Zunächst werden nur die Kanten betrachtet, welche das momentan maximale Gewicht tragen. In der Ausgangssituation des Beispiels der Aufgabenstellung ist das maximale Kantengewicht zwei. Die Abbildung 3 stellt den Graphen nur mit den Kanten maximaler Gewichtung dar. Diese Darstellung hebt solche Verbindungen hervor, welche tatsächlich ausschlaggebend für eine Zuweisung sind. Sollten von einigen Knoten weiterhin mehrere Kanten ausgehen, kann unter der Voraussetzung, dass diese Knoten exakt die gleichen Kanten besitzen, eine Zuweisung getroffen werden. Beispielsweise sind die Knoten A und Br mit exakt den gleichen Knoten verbunden, nämlich 1 und 4. In diesem Szenario bestünde also die Möglichkeit die Sorten A und Br den Schüsseln 1 und 4 zuzuweisen, wobei unklar bleiben würde, welche Frucht konkret in welcher Schüssel läge.

In vielen Fällen kann es jedoch vorkommen, dass von einem Sorten-Knoten mehrere Kanten ausgehen und diese Konstellation auch nur einmal auftritt. In diesem Fall wäre die Anzahl der ausgehenden Kanten, nicht gleich der Anzahl von Sorten-Knoten, welche genau diese Konstellation aufweisen. Somit wäre keine eindeutige Zuweisung an diesem Knoten möglich. Aus diesem Grund fokussiert sich der Algorithmus immer zuerst auf die Sorten-Knoten, welche nur eine einzige Kante besitzen. An diesen Stellen kann immer eine eindeutige Zuweisung stattfinden. Nach Reduzierung des Graphen wird die Möglichkeit einer Zuweisung erneut geprüft.

¹Aus Vereinfachungsgründen wurden in diesem Beispiel auf mehr als zwei Verbindungen verzichtet. Beispiele mit mehr als zwei Verbindungen entnehmen Sie den Testdateien 1 - 7.

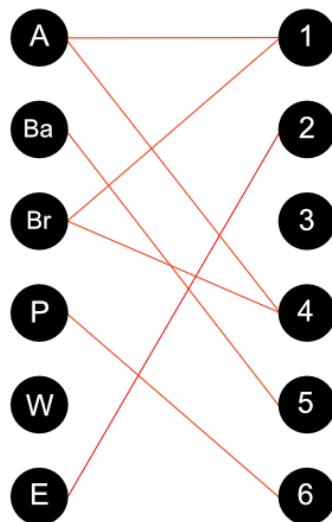


Abbildung 3: Graph mit Kanten maximaler Gewichtung

2.3.2 Die Reduzierung des Graphen

Nach jeder Zuweisung, muss der Graph entsprechend reduziert werden. Das Reduzieren des Graphen ermöglicht es, die oben genannte Problemsituation, einer einmaligen Kantenkonstellation eines Sorten-Knotens, zu lösen. Wenn zuerst die eindeutigen Sorten zugewiesen werden, welche nur durch eine Kante mit einer Schlüssel verbunden sind und der Graph infolgedessen um die entsprechenden Kanten reduziert wird, bleiben schlussendlich nur noch Knoten mit mehreren Kanten übrig. Es ist darauf hinzuweisen, dass die Anzahl der Kanten identisch mit der Anzahl von Sortenknoten gleicher Kantenkonstellation ist. Damit wurde eine Problemstelle soweit reduziert, sodass man nun wieder eine eindeutige Zuweisung treffen kann.

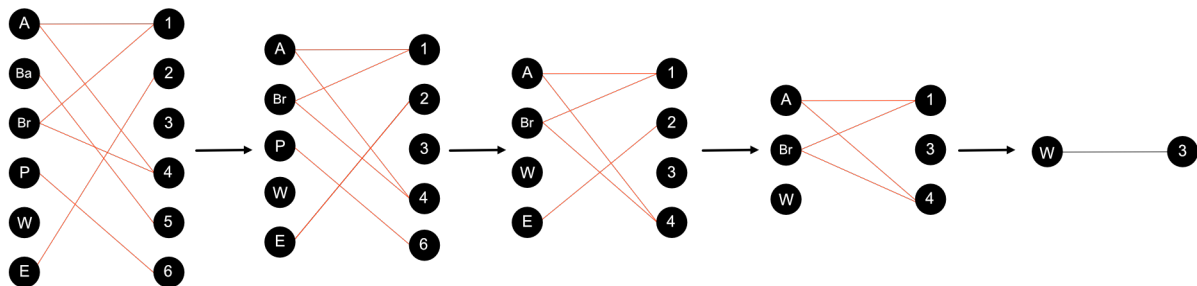


Abbildung 4: Sukzessive Reduzierung des Graphens

Die Abbildung 4 zeigt die sukzessive Reduzierung des Graphens. In der ersten Phase werden alle Kanten entfernt, die eine maximale Gewichtung haben und einen Sorten-Knoten eindeutig mit einem Schlüssel-Knoten verbinden. Im vierten Schritt bleiben nun nur noch Knoten übrig, von denen zwei Kanten ausgehen. Da die Anzahl ausgehender Kanten so groß ist, wie die Anzahl der Knoten, welche genau diese Kantenkonstellation aufweisen, kann nun auch an dieser Stelle eine Zuweisung stattfinden. Da diese Zuweisung auch schon vorher möglich war, ist dem Beispiel geschuldet und ist in den Testdateien größtenteils nicht der Fall. Im Weiteren werden alle Kanten der maximalen Gewichtung und Knoten, welche durch diese verbunden wurden, beseitigt. Nun wird ein neues Kantenmaximum ermittelt. Das neue maximale Kantengewicht beträgt eins. Die übriggebliebenen Knoten lassen sich nach zuvor erörterten Schema zuordnen.

3 Umsetzung

Mit Start des Programms öffnet sich ein Konsoleninterface, über welches man eine der angegebenen Testdateien auswählen kann. Der Pfad, der angegebenen Datei wird einer Daten-Klasse übergeben, welche die Datei ausliest und gegebene Informationen in einem Daten-Objekt speichert. Dieses Datenobjekt wird infolgedessen dazu verwendet eine Adjazenzmatrix zu generieren, die den Graphen darstellt und eine Liste zu initialisieren, welche für jede Obstsorte eine Liste von zugehörigen Schlüsselnummern speichert. Diese Liste wird im Folgenden als “Match-Liste” bezeichnet. Sowohl die Matrix als auch die Match-Liste werden nun mit weiteren Parametern einer rekursiven Methode übergeben. Mittels weiterer Methoden werden die zum Reduzieren der Matrix gefundenen Zuweisungen in der Match-Liste gespeichert und zurückgegeben, wenn alle Zuweisungen stattgefunden haben, bzw. alle Kanten aus der Matrix entfernt wurden. Im Weiteren wird die Match-Liste noch ausgewertet, sodass eine möglichst informative Aussage bezüglich des Ergebnisses getroffen werden kann. Die Implementierung dieser Methoden wird im Folgenden näher erläutert.

3.1 Bereitstellung der Daten

Wie bereits erwähnt, besteht bei der Implementierung zur Bereitstellung der Daten aus den Testdateien eine Klasse, welche die Informationen in einem Datenobjekt speichert. Jenes Datenobjekt speichert eine Obergrenze für die Anzahl an Obstsorten und Wunschsorten als String. Darüber hinaus enthält es zwei separate Array-Lists, welche die gesehenen Obstsorten und die Nummern der Schlüsselnummern ebenfalls als String speichern. Mit einem bestimmten Index erhält man also über beide Listen eine entsprechende Beobachtung, welche die Obstsorten und die Schlüsselnummern beinhaltet. Diese Informationen können stückweise aus den Testdateien ausgelesen werden.

Zudem wird eine weitere ArrayList generiert, welche alle Obstsorten als String enthält. Dies ist später bei der Generierung der Match-Liste und der Adjazenzmatrix von Bedeutung. Um die benannte Liste zu erstellen, werden zunächst alle Wunschsorten der Liste hinzugefügt. Die Liste der beobachteten Sorten wird durchlaufen und stückweise der Liste aller Obstsorten hinzugefügt. Um doppelte Einträge in der Liste aller Sorten zu vermeiden, wird für jede beobachtete Obstsorte die Liste aller Obstsorten durchlaufen. Ist die beobachtete Obstsorte bereits in der Liste aller Obstsorten enthalten, wird sie nicht mehr hinzugefügt. Dieser Schritt ist nötig, da Obstsorten auf verschiedenen Spießen und unter den Wunschsorten mehrfach auftreten können.

Ein Datenobjekt der Aufgabenstellung beinhaltet beispielsweise folgende Konstanten:

Obergrenze für die Anzahl der Obstsorten: 6

Wunschsorten: “Weintraube Brombeere Erdbeere”

Beobachtete Sorten: [“Apfel Banane Brombeere”, “Banane Pflaume Weintraube”, “Apfel Brombeere Erdbeere”, “Erdbeere Pflaume”]

Beobachtete Schlüsselnummern: [“1 4 5”, “3 5 6”, “1 2 4”, “2 6”]

Alle Sorten: [“Weintraube”, “Brombeere”, “Erdbeere”, “Apfel”, “Banane”, “Pflaume”]

In manchen Fällen ist die in der Datei angegebene Obergrenze für die Anzahl der Obstsorten größer, als die Größe der ArrayList aller Obstsorten. Das bedeutet, dass nicht alle Obstsorten auf den beobachteten Spießen oder unter den eigenen Wunschsorten zu finden sind. In diesem Fall wird der Liste aller Sorten noch eine weitere Sorte mit dem Namen “unknown-n” hinzugefügt, wobei n hier für die n-te unbekannte Sorte steht. Die erste unbekannte Sorte trägt also den Namen “unknown-1”.

3.2 Generierung der Adjazenzmatrix

Die Methode “genMatrix” nimmt ein zuvor beschriebenes Datenobjekt als Parameter entgegen und gibt ein zweidimensionales Integer-Array als Adjazenzmatrix zurück. Dazu wird zunächst eine mit nullen gefüllte Matrix initialisiert. Die Größe der Zeilen und Spalten entspricht dabei der vom Datenobjekt angegebenen Obergrenze für die Anzahl der Obstsorten. Diese Größe wird sowohl in der Spaltenzahl als auch in der Zeilenzahl um eins erhöht, da die Werte in der ersten Zeile und Spalte zur eindeutigen Identifizierung der Knoten dienen. Dies ist nötig, da die Knoten, im späteren Verlauf, durch Zuweisungen reduziert werden und sie dementsprechend nicht mehr eindeutig über ihre Indizes zu identifizieren sind. Des Weiteren wird jede Zeile und Spalte von null an nummeriert. Anschließend wird jede Beobachtung, das heißt die beiden Listen, welche die Schlüsselnummern und Sorten der Beobachtungen speichern, durchlaufen.

Der Index einer Sorte in der Matrix entspricht dabei dem Index einer Sorte in der Liste aller Sorten, welche zuvor von dem Datenobjekt generiert wurde. In der jeweiligen Zeile wird dann an den Indizes der Schlüsselnummern der Wert um eins erhöht. Abbildung 5 zeigt die Generierung der Adjazenzmatrix nach dem oben benannten Ablauf.

0	1	2	3	4	5	6		0	1	2	3	4	5	6
1	0	0	0	0	0	0		1	0	0	1	0	1	1
2	0	0	0	0	0	0		2	2	1	0	2	1	0
3	0	0	0	0	0	0	→	3	2	1	0	2	1	0
4	0	0	0	0	0	0		4	1	0	1	1	2	1
5	0	0	0	0	0	0		5	0	1	1	0	1	2
6	0	0	0	0	0	0		6	1	2	0	1	0	1

Abbildung 5: Generierung der Adjazenzmatrix nach Beispiel der Aufgabenstellung

3.3 Generierung der Match-List

Die Generierung der Match-Liste erfolgt nach dem Initialisieren der Matrix in der Main-Methode. Die Match-Liste ist eine ArrayList, welche Objekete vom Typ "Match" speichert. Ein Objekt vom Typ Match speichert wiederum eine Sorte als String und außerdem eine weitere Integer-Liste, welche für die jeweilige Sorte die Schlüsselnummern speichert. Da eine Sorte in mehreren potenziellen Schüsseln liegen kann, darf ein Objekt vom Typ Match nicht nur eine Schlüsselnummer speichern. Zunächst wird also eine leere ArrayList vom Typ Match initialisiert. Anschließend wird für jede Sorte ein neues Match-Objekt der Match-Liste hinzugefügt, das eine leere Liste von Schlüsselnummern enthält. Die Lösungsmethode füllt die Listen der Schlüsselnummern der jeweiligen Matches in der Match-Liste

3.4 Zuweisung der Schlüsselnummern zu den Obstsorten

Der folgende Algorithmus ist in vier Phasen zu unterteilen, welche teilweise abwechselnd nacheinander aufgerufen werden und dazu dienen, Schüsseln den Obstsorten zuzuweisen und währenddessen die Matrix sukzessive zu reduzieren.

Die Methode "assignUnweighted" nimmt die zuvor generierte Matrix und die Match-Liste als Parameter entgegen und weist zuerst alle Sorten den Schüsseln zu, welche nicht beobachtet werden konnten und somit direkt zuzuordnen sind. Hier findet die Phase I statt. Nach Entfernung dieser Knoten aus der Matrix, wird die rekursive Methode "assignWeighted" aufgerufen, die in weitere drei Phasen zu unterteilen ist. Hier geht es darum die Knoten zuzuweisen, welche noch durch gewichtete Kanten miteinander verbunden sind. Die vier Phasen und die Methoden zur Reduzierung der Matrix werden im Folgenden näher erläutert.

3.4.1 Phase I: Zuweisung von unbeobachteten Sorten

In manchen Situationen kann es dazu kommen, dass unter den Beobachtungen nicht alle Obstsorten und Schlüsselnummern enthalten sind, welche durch die Obergrenze festgelegt wurden. Die Kanten zwischen Knoten dieser Sorten und Schüsseln sind mit null gewichtet und werden vorab entfernt. Dies hat zum Vorteil, dass die Matrix in den folgenden Phasen nicht unnötig oft durchlaufen wird mit reduzierender Wirkung auf die Laufzeit. Zwei Listen speichern zunächst die Indizes der Zeilen und Spalten, welche nur Kanten besitzen, die mit null gewichtet sind. Dazu wird die Matrix einmal durchlaufen, wobei zwei boolsche Variablen speichern, ob eine jeweilige Zeile oder Spalte durchgehend nur mit Kanten des Gewichtes null gefüllt sind. In diesem Fall wird der jeweilige Zeilen- oder Spaltenindex in der richtigen Liste eingetragen. Nun gibt es eine Liste von Spaltenindizes (Schüsselknoten) und eine Liste von Zeilenindizes (Sortenknoten), welche zugewiesen und im Anschluss entfernt werden müssen. Für die Zuweisung werden die Zeilenindizes durchlaufen. Die Zeilen-ID an einem Index, welche in der ersten Zeile zu finden ist, wird dazu verwendet in der Match-Liste ein Match-Objekt zu wählen. Folglich werden der Schlüsselnummerliste des entsprechenden Match-Objektes alle Spalten-IDs, an den Stellen der in der Liste gespeicherten

Spaltenindizes, hinzugefügt. Wenn die Match-Liste vollständig ergänzt wurde, findet die Reduzierung der Adjazenzmatrix statt.

3.4.2 Reduzierung um mehrere Knoten

Die Methode “removeMultiple” nimmt eine Adjazenzmatrix, eine Liste von Spaltenindizes und eine Liste von Zeilenindizes als Parameter entgegen und gibt eine Matrix wieder, welche um diese Spalten und Zeilen reduziert wurde. Dazu wird die ursprüngliche Matrix durchlaufen. Für jede Kante wird ebenfalls der Teil der Liste zu entfernender Zeilenindizes und Spaltenindizes durchlaufen, der bis jetzt noch nicht übersprungen wurde. An einem zu entfernenden Zeilen- oder Spaltenindex angekommen, wird die entsprechende Kante nicht in die neue Matrix kopiert. Zwei Variablen zählen dabei, wie viele Zeilen und Spalten bereits übersprungen wurden, damit die restlichen Kanten ordnungsgemäß kopiert werden können.

Nach der Zuweisung und Reduzierung der Matrix um unbeobachtete Knoten, wird mit der Methode “assignWeighted” fortgefahren, die sich über die folgenden drei Phasen erstreckt und abwechselnd, rekursiv ausführt. Da in dem Beispiel der Aufgabenstellung alle vorkommenden Sorten auch beobachtet werden, findet Phase I nicht statt. Von dem Reduzieren der Matrix um mehrere Knoten muss jedoch auch in Phase IV Gebrauch gemacht werden, weshalb der Prozess erneut in Abbildung 8 dargestellt wird.

3.4.3 Phase II: Informationssammlung

Zunächst werden alle Knoten zugewiesen, die nur durch eine einzige Kante maximalen Gewichts verbunden sind (vgl. dazu Abschnitt 2 der Lösungsidee). Dadurch können nämlich später auch Knoten zugewiesen werden, welche durch mehrere Kanten maximalen Gewichtes verbunden sind. Um jedoch zu ermitteln, ob bereits alle Knoten zugewiesen wurden, welche nur durch eine Kante maximalen Gewichtes verbunden sind, beziehungsweise solche überhaupt existieren, müssen noch einige Informationen über die Matrix gesammelt werden. Diese “Informationssammlung” geschieht zu Anfang eines jeden rekursiven Aufrufs und ist ausschlaggebend dafür, mit welcher der beiden weiteren Phasen in diesem rekursiven Aufruf fortgefahren werden muss. Dazu wird für jede Spalte der Adjazenzmatrix das Kantenmaximum ermittelt sowie eine modifizierte Spaltensumme berechnet. Jede Kante der Spalte wird dabei mit dem Zeilenindex multipliziert und dann erst aufsummiert. Zudem wird noch ein globales Maximum ermittelt, das heißt die am höchsten gewichtete Kante der gesamten Adjazenzmatrix. Dazu wird jede Zeile und Spalte der Matrix einmal durchlaufen.

Zuerst werden die Knoten in Betracht gezogen, welche durch Kanten mit dem momentan höchsten Gewicht verbunden sind. Unter diesen Knoten muss für Phase III und IV also noch ermittelt werden, ob es solche gibt, die eine einmalig vorkommende Kantenkonstellation haben oder nur solche, die mehrfach vorkommen. An dieser Stelle wird Gebrauch von der modifizierten Spaltensumme gemacht. Die Wahrscheinlichkeit für das Auffinden einer gleichen Kantenkonstellation von zwei Knoten mit dem Kriterium einer gleichen modifizierten Spaltensumme ist sehr hoch und auf jeden Fall höher, als nehme man die gewöhnliche Spaltensumme als Kriterium. Die Indizes der Knoten, welche eine einmalige Kantenkonstellation haben, werden für den weiteren Gebrauch in einer Liste gespeichert. Knoten mit mehrfach vorkommender Kantenkonstellation werden beim Durchlaufen der Matrix identifiziert und in einer Hash-Map gespeichert. Dazu wird das Array mit den Spaltenmaxima durchlaufen. An einem Spaltenmaximum angekommen, welches dem globalen Maximum entspricht, wird das Array von diesem Index aus ein weiteres Mal durchlaufen. Findet das Programm wieder eine Spalte mit einem globalen Maximum, so wird geprüft, ob die modifizierte Spaltensumme an den beiden Indizes ebenfalls gleich ist. In einer solchen Konstellation bestünde eine hohe Wahrscheinlichkeit, dass zwei Knoten mit gleicher Kantenkonstellation bestehen. Daraus folgt, dass der Index dieses Knotens nicht der Liste von Knotenindizes einmaliger Kantenkonstellationen hinzugefügt werden kann und in die HashMap überführt wird. Um sicher zu gehen, dass es sich hierbei um zwei Knoten handelt, die tatsächlich eine gleiche Kantenkonstellation haben und somit aus der Liste von Knoten einmaliger Kantenkonstellationen auszuschließen sind, werden die Spalten an den beiden Indizes noch einmal ganz durchlaufen. In diesem Rahmen wird die Gleichgewichtung aller Kanten sichergestellt. Schlussendlich ist die Liste nur mit Indizes von Knoten mit einmaliger Kantenkonstellation gefüllt. Die HashMap speichert dabei einen Index einer Spalte, welche eine mehrfach vorkommende Kantenkonstellation besitzt als Schlüssel und als Wert eine Liste, welche wiederum jeden weiteren Index einer Spalte speichert, die die gleiche Kantenkonstellation hat. So kann direkt ermittelt werden, welche Spalten eine gleiche Kantenkonstellation beinhalten.

Abbildung 6 zeigt die Spaltenmaxima und modifizierten Summen bezüglich des Beispiels der Aufgabenstellung. Die Spalten mit dem Index 1 und 4 weisen eine gleiche Kantenkonstellation auf. Die Spalten 2, 5 und 6 beinhalten ebenfalls alle eine Kante mit globalem Maximum, haben jedoch eine individuelle Kantenkonstellation. Ihre Indizes werden der zuvor genannten Listen hinzugefügt. Spalte 3 enthält keine Kante mit globalem Maximum und wird dementsprechend für den momentanen Rekursionsaufruf ignoriert.

0	1	2	3	4	5	6
1	0	0	1	0	1	1
2	2	1	0	2	1	0
3	2	1	0	2	1	0
4	1	0	1	1	2	1
5	0	1	1	0	1	2
6	1	2	0	1	0	1
Σ	20	22	15	20	19	21
max	2	2	1	2	2	2

Abbildung 6: Ermittlung von modifizierten Spaltensummen und -maxima

3.4.4 Phase III: Zuweisung von Knoten einmaliger Kantenkonstellation

Ist die Liste von Indizes der Knoten einmaliger Kantenkonstellationen noch nicht leer, so wird mit Phase III fortgesetzt. In dieser Phase bestehen noch gewichtete Kanten und außerdem Knoten, deren Kantenkonstellation einmalig ist. Unter weiteren, im Folgenden erläuterten Voraussetzungen, kann man diese Knoten eindeutig zuweisen und somit die Zuweisungsmöglichkeiten anderer Knoten reduzieren. Alleine die individuelle Kantenkonstellation ist nicht ausreichend für eine eindeutige Zuweisung, da der entsprechende Knoten nur eine einzige Kante mit globalem Maximum besitzen darf. Um eine zu entfernende Kante zu finden, werden zunächst die Indizes durchlaufen, welche in der Liste von Knoten einmaliger Kantenkonstellation gespeichert sind. Ist in den entsprechenden Spalten nur ein globales Maximum zu finden, wird an dieser Stelle ebenfalls die Zeile durchsucht. Wenn auch in dieser Zeile weiter kein globales Maximum zu finden ist, wurde eine Kante mit globalem Maximum gefunden, welche zwei Knoten eindeutig miteinander verbindet. Nun kann also tatsächlich eine eindeutige Zuweisung stattfinden.

Für eine Zuweisung werden nun die IDs von Knoten der ersten Zeile und Spalte verwendet. In der Match-Liste wird das Match-Objekt gewählt, dessen Index zur ID des jeweiligen Sorten-Knotens passt. In der Schlüsselnummerliste des gewählten Match-Objektes wird nun die Schlüsselnummer (Spalten-ID), hinzugefügt.

3.4.5 Reduzierung um zwei Knoten

Nachdem in Phase III eine eindeutige Zuweisung gefunden werden konnte, müssen nun in der Matrix zwei Knoten und alle ihre Kanten entfernt werden. Dafür nimmt die Methode "removeSingle" eine Matrix, einen Zeilenindex und einen Spaltenindex als Parameter entgegen und gibt eine reduzierte Matrix zurück. Dabei wird die ursprüngliche Adjazenzmatrix kopiert, wobei die Spalte und Zeile der betreffenden Indizes nicht kopiert werden. Sie funktioniert also genauso, wie die Methode "removeMultiple", nur mit dem alleinigen Unterschied, dass nur eine Zeile und Spalte entfernt werden kann. Zunächst werden also alle Zeilen und Spalten der Matrix durchlaufen. Trifft das Programm auf eine Zeile oder Spalte, welche einen auszulassenden Index hat, wird an dieser Stelle nicht kopiert. Zwei Variablen speichern dabei jeweils, ob bereits eine Zeile oder Spalte übersprungen wurde, damit danach korrekt weiter kopiert werden kann. Zum Schluss wird die reduzierte Adjazenzmatrix zurückgegeben. Zur Übersicht wird nach jeder Zuweisung einmal die Matrix auf die Konsole geschrieben. Dazu durchläuft die Methode "printMatrix" eine ihr übergebene Matrix und gibt sie übersichtlich auf die Konsole aus. So kann man den Zuweisungsprozess schrittweise mitverfolgen.

Nach der Zuweisung und Reduzierung der Matrix startet der nächste Rekursionsaufruf. Diesmal wird der solve-Methode eine reduzierte Matrix und eine erweiterte Match-Liste übergeben. Das Programm ermittelt die Spaltenmaxima und modifizierte Summen dementsprechend neu. Abbildung 7 zeigt den Zuweisungsprozess nach den ersten drei Rekursionsaufrufen. Da in den ersten drei Rekursionsaufrufen immer Kanten mit globalem Maximum und Knoten mit individuellen Kantenkonstellationen vorhanden sind, führt das Programm drei Mal am Stück Phase III aus.

0	1	2	3	4	5	6
1	0	0	1	0	1	1
2	2	1	0	2	1	0
3	2	1	0	2	1	0
4	1	0	1	1	2	1
5	0	1	1	0	1	2
6	1	2	0	1	0	1

0	1	3	4	5	6
1	0	1	0	1	1
2	2	0	2	1	0
3	2	0	2	1	0
4	1	1	1	2	1
5	0	1	0	1	2

0	1	3	4	6
1	0	1	0	1
2	2	0	2	0
3	2	0	2	0
5	0	1	0	2

0	1	3	4
1	0	1	0
2	2	0	2
3	2	0	2

Abbildung 7: Zuweisung von Knoten und Reduzierung der Matrix in Phase III

3.4.6 Phase IV: Zuweisung von Knoten mehrfach vorkommender Kantenkonstellation

Inzwischen wurden alle Knoten, welche mit nur einer Kante maximaler Gewichtung verbunden sind, zugewiesen und aus der Adjazenzmatrix entfernt. In der letzten Matrix aus Abbildung 7 kann man erkennen, dass es unter den Knoten, mit Kanten eines globalen Maximums, nur noch solche gibt, welche eine exakt gleiche Kantenkonstellation haben. Unter der Voraussetzung, dass die Anzahl von Kanten maximaler Gewichtung an diesen Knoten genauso groß ist wie die Anzahl der Knoten, die die exakt gleiche Kantenkonstellation aufweisen, kann man eine eindeutige Zuweisung treffen.

Dazu wird zunächst die HashMap durchlaufen, welche für jede Spalte mit mehrfach vorkommender Kantenkonstellation eine Liste von Indizes speichert, deren Spalten diesselbe Kantenkonstellation haben. Eine der Spalten mit mehrfach vorkommender Kantenkonstellation wird zunächst durchlaufen, um festzustellen, wie viele Kanten maximaler Gewichtung in dieser Kantenkonstellation auftauchen. Im gleichen Zuge werden die Zeilenindizes dieser Spaltenmaxima in einer separaten Liste gespeichert. Die HashMap speichert für jeden Spaltenindex einer Spalte, welche eine Kante maximalen Gewichtes enthält und eine mehrfach vorkommende Kantenkonstellation hat, eine Liste von weiteren Spaltenindizes, welche ebenfalls die gleiche Kantenkonstellation haben. Daher besteht die Gesamtheit der Indizes von Spalten gleicher Kantenkonstellation aus dem Schlüsselindex selbst und der dazugehörigen Liste. In der Liste alleine würde dementsprechend noch ein Index fehlen. Deswegen wird der Schlüsselwert selbst noch der dazugehörigen Indexliste hinzugefügt. Ist diese Liste so groß, wie die Anzahl von Kanten maximaler Gewichtung in dieser Konstellation, so ist die Anzahl der Sortenknoten, welche eine gleiche Kantenkonstellation aufweisen so groß, wie die Anzahl der Schlüsselknoten, welche durch diese Konstellation verbunden sind. Mit der Erfüllung diesen Kriteriums kann eine Zuweisung stattfinden.

Die Zuweisung der Knoten und deren Entfernung aus der Matrix findet dann wieder gemäß des Ablaufes in Phase I statt. Da hier mehrere Knoten zugewiesen wurden, findet die Methode "removeMultiple" Anwendung. Ihr werden die momentane Matrix und die beiden Listen mit zu entfernenden Zeilen- und Spaltenindizes übergeben. Nach der Reduzierung um die zugewiesenen Knoten beginnt der nächste Rekursionsaufruf mit der reduzierten Adjazenzmatrix und der ergänzten Match-Liste. Sofern auch alle Knoten mit mehrfach auftauchender Kantenkonstellation zugewiesen und beseitigt wurden, wird ein globales Maximum ermittelt, welches geringer ist als das Vorherige. Abbildung 8 zeigt, wie der Graph nach erstmaliger Aktivität der Phase IV verändert wird. Übrig bleiben zwei Knoten, welche durch eine Kante mit dem Gewicht eins verbunden sind (vgl. Abschnitt 2.2.2). Aufgrund der begrenzten Knotenanzahl in dem Beispiel der Aufgabenstellung kommt es nur einmal zur Phase IV. Grundsätzlich sollen nach Zuweisung der unbeobachteten Sorten in Phase I Informationen über die aktuelle Matrix in Phase II gesammelt werden und darauf basierend die Ausführung der Phase III oder Phase IV angestoßen werden. Zuerst wird so lange Phase III ausgeführt, bis nur noch Knoten mit mehrfach auftauchender Kantenkonstellation vorhanden sind. Dann findet Phase IV solange statt, bis auch diese Knoten zugewiesen wurden und ein neues Kantenmaximum zu ermitteln ist. Mehrere Knoten mit individueller Kantenkonstellation können

nicht auf einen Schlag zugewiesen und entfernt werden, da die Zuweisung einzelner Knoten vielfach durch die eindeutige Zuordnung anderer Knoten bedingt wird. Viele Fälle, die der Algorithmus abdeckt werden aus dem Beispiel der Aufgabenstellung nicht ersichtlich, weshalb sich der folgende Abschnitt 3 mit einigen umfangreicheren Beispielen beschäftigt.

0	1	3	4		
1	0	1	0	0	3
2	2	0	2	1	1
3	2	0	2		

Abbildung 8: Zuweisung von Knoten und Reduzierung der Matrix in Phase IV

3.5 Ausgabe

Nachdem die zuvor vorgestellten Methoden die Objekte der Match-Liste ergänzt haben und also jeder Sorte eine Menge von Schlüsseln zugewiesen werden konnte, müssen die Ergebnisse noch auf die Konsole ausgegeben werden. Bei jeder Zuweisungen und Reduzierung wurde schon die, zum jeweiligen Zeitpunkt reduzierte, Matrix in für den Graphen repräsentativer Weise auf die Konsole ausgegeben. Nun wird noch übersichtlich dargestellt welche Sorten in welchen Schlüsseln zu finden sind und in wie vielen potenziellen Schlüsseln die Wunschsorten liegen können.

Um darzustellen, welche Sorten in welchen Schlüsseln zu finden sind, wird die Match-Liste einmal durchlaufen. Dabei wird der Sortenname des jeweiligen Match-Objektes ausgegeben. Direkt danach wird die Liste der Schlüsselnummern des jeweiligen Match-Objektes durchlaufen und jede Schlüsselnummer neben der betreffenden Sorte ausgegeben.

Die Anzahl der Schlüsseln, in denen die Wunschsorten liegen, sind zu ermitteln, indem die Schlüsselnummerlisten von den jeweiligen Wunschsorten durchlaufen werden. Die Zielschlüsselnummern werden in einer Liste gespeichert. Für jede Schlüsselnummer einer Wunschsorte wird die Liste von Zielschlüsselnummern durchlaufen, um auszuschließen, dass Schlüsselnummern doppelt auftauchen. Anschließend werden die Wunschsorten, die Zielschlüsseln und die Anzahl der Zielschlüsseln ausgegeben.

Damit der Wunschspieß tatsächlich durch Abgehen der Zielschlüsseln zusammengestellt werden kann, muss die Anzahl der Zielschlüsseln mit der Anzahl der Wunschsorten übereinstimmen. Dies kann nicht zwingend vorausgesetzt werden. Sind beispielsweise die Sorten Apfel und Banane in den Schlüsseln 1 und 2 zu finden und auf den Wunschspieß sollen nur Äpfel, so gibt es keine Garantie, dass der Wunschspieß korrekt zusammengestellt wird. Sollen jedoch sowohl Bananen als auch Äpfel auf den Wunschspieß, so ist die Anzahl der Sorten so groß wie die Anzahl der Zielschlüsseln. In diesem Fall würde eine Garantie bestehen. Um demnach eine sinnvolle Aussage über die Lösbarkeit des Problems zu treffen, wird noch die Anzahl der Wunschsorten mit der Anzahl der Schlüsseln verglichen. Je nach dem ob die Werte gleich groß sind oder nicht wird eine Meldung auf die Konsole ausgegeben.

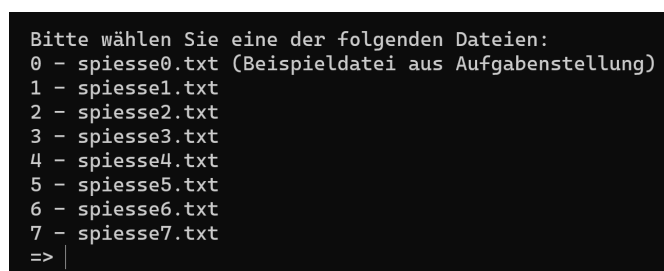
Ist die Anzahl von Wunschsorten nicht so groß wie die Anzahl von Zielschlüsseln, werden noch die Sorten ermittelt, welche zu dem Spieß hinzugefügt werden müssten, damit eine Garantie für dessen richtige Zusammenstellung bestünde. Dafür wird zunächst die Match-Liste durchlaufen. Für jedes Match-Objekt einer Wunschsorte werden erneut die Match-Objekte durchlaufen, deren Sorten nicht zu den Wunschsorten gehören. Dass die Match-Objekte der Wunschsorten in der Match-Liste vorne stehen, erleichtert diese Suche. Ist die Schlüsselnummerliste eines Match-Objektes einer Wunschsorte gleich der Schlüsselnummerliste eines Match-Objektes, welches nicht zu einer Wunschsorte gehört, so muss diese Sorte noch zu den Wunschsorten hinzugefügt werden. Entsprechend folgt eine Ausgabe.

4 Beispiele

In diesem Abschnitt wird die Lösung verschiedener Testfälle mit dem zuvor vorgestellten Algorithmus präsentiert. Da in dem vorherigen Beispiel der Aufgabenstellung viele Problemsituationen noch nicht aufgetaucht sind, sollen diese hier näher behandelt werden. Zunächst wird die erste Testdatei als Beispiel gewählt. Hier werden alle zuvor beschriebenen Phasen abgedeckt.

4.1 Dateiauswahl

Das Programm, das mit dem beschriebenen Algorithmus die vorgegebenen Testdateien löst, ist ein Konsolenprogramm, welches in einer Shell ausgeführt wird. Beim Programmstart öffnet sich ein Konsoleninterface, in dem man eine Ziffer zwischen eins und sieben repräsentativ für die Testdateien wählen kann. Abbildung 9 zeigt die Eingabeaufforderung bei Start des Programmes.



```
Bitte wählen Sie eine der folgenden Dateien:
0 - spiesse0.txt (Beispieldatei aus Aufgabenstellung)
1 - spiesse1.txt
2 - spiesse2.txt
3 - spiesse3.txt
4 - spiesse4.txt
5 - spiesse5.txt
6 - spiesse6.txt
7 - spiesse7.txt
=> |
```

Abbildung 9: Wahl einer Testdatei

4.2 Algorithmische Lösung

Sobald eine Testdatei erfolgreich gewählt wurde, gibt das Programm eine Reihe von Matrizen auf die Konsole aus. Abbildung 10 zeigt die sukzessive Reduzierung der Matrix durch Zuweisung von Knoten, mit den Daten der ersten Testdatei. Angefangen mit der Ausgangsmatrix, kann man erkennen, wie danach Phase I und dann abwechselnd Phase III und IV angewandt werden. Die Informationssammlung in Phase II wird nicht ausgegeben. Die Matrizen werden immer nach Anwendung einer Phase ausgegeben. Beispielsweise gibt es in Zeile drei und Spalte sieben der Ausgangsmatrix noch Sorten und Schüsseln, welche in den Beobachtungen nicht aufgetaucht sind. Hier wird, wie in der Umsetzung beschrieben, zuerst Phase I angewandt.

Nach der Zuweisung und Reduzierung der Matrix um diese Knoten bestehen nur noch Kanten mit einem Gewicht größer als null. Momentan beträgt das höchste Kantengewicht drei, nämlich in den Spalten zwei und vier. Da eine gleiche Kantenkonstellation an diesen Knoten besteht und es weiter keine Knoten mit Kanten maximalen Gewichtes gibt, wird Phase IV angewandt. Dementsprechend werden die Schüsseln zwei und vier den Sorten mit der ID zwei und vier zugeordnet. Im Anschluss werden die zugewiesenen Knoten entfernt.

Zunächst findet wieder Phase II statt, wobei ermittelt wird, dass das größte Kantengewicht zwei beträgt. In Spalte sechs, acht, neun und zehn gibt es Kanten maximaler Gewichtung. Sie alle haben eine individuelle Kantenkonstellation. Daher wird nun vier Mal am Stück Phase III ausgeführt.

Nach der Zuweisung und Entfernung dieser Knoten wird durch Phase II ein neues maximales Kantengewicht ermittelt. In der siebten Matrix der Abbildung 10 kann man erkennen, dass das höchste Kantengewicht bei eins liegt. In den Spalten der ID eins und drei existiert ein solches Maximum und außerdem eine individuelle Kantenkonstellation. Daher wird wieder zwei Mal Phase III angewandt. Die Matrix 9 zeigt, dass durch die Durchführung der Phasen in der richtigen Reihenfolge eine Zuweisung und Reduzierung so weit stattfinden konnte, dass es keine Knoten und Kanten mehr in der Adjazenzmatrix gibt.

0	Ausgangsmatrix: 0 1 2 3 4 5 6 7 8 9 10 1 1 1 0 1 0 0 0 1 1 0 2 1 3 0 3 2 1 0 2 2 1 3 0 0 0 0 0 0 0 0 0 0 4 1 3 0 3 2 1 0 2 2 1 5 0 2 0 2 2 1 0 1 1 1 6 0 0 1 0 0 1 0 0 0 1 7 0 1 1 1 1 1 0 1 0 2 8 0 1 1 1 1 2 0 0 1 1 9 1 2 0 2 1 0 0 2 1 1 10 1 2 0 2 1 1 0 1 2 0	3	Anwendung Phase III: 0 1 3 6 8 9 10 1 1 0 0 1 1 0 6 0 1 1 0 0 1 7 0 1 1 1 0 2 8 0 1 2 0 1 1 9 1 0 0 2 1 1 10 1 0 1 1 2 0	6	Anwendung Phase III: 0 1 3 10 1 1 0 0 6 0 1 1 7 0 1 2
1	Anwendung Phase I: 0 1 2 3 4 5 6 8 9 10 1 1 1 0 1 0 0 1 1 0 2 1 3 0 3 2 1 2 2 1 4 1 3 0 3 2 1 2 2 1 5 0 2 0 2 2 1 1 1 1 6 0 0 1 0 0 1 0 0 1 7 0 1 1 1 1 1 1 0 2 8 0 1 1 1 1 2 0 1 1 9 1 2 0 2 1 0 2 1 1 10 1 2 0 2 1 1 1 2 0	4	Anwendung Phase III: 0 1 3 8 9 10 1 1 0 1 1 0 6 0 1 0 0 1 7 0 1 1 0 2 9 1 0 2 1 1 10 1 0 1 2 0	7	Anwendung Phase III: 0 1 3 1 1 0 6 0 1
2	Anwendung Phase IV: 0 1 3 5 6 8 9 10 1 1 0 0 0 1 1 0 5 0 0 2 1 1 1 1 6 0 1 0 1 0 0 1 7 0 1 1 1 1 0 2 8 0 1 1 2 0 1 1 9 1 0 1 0 2 1 1 10 1 0 1 1 1 2 0	5	Anwendung Phase III: 0 1 3 9 10 1 1 0 1 0 6 0 1 0 1 7 0 1 0 2 10 1 0 2 0	8	Anwendung Phase III: 0 3 6 1
				9	Anwendung Phase III: 0

Abbildung 10: Sukzessive Reduzierung der Adjazenzmatrix durch Zuweisung von Knoten

4.3 Ausgabe

Nach der Zuweisung der Knoten und Reduzierung der Adjazenzmatrix folgt die Ausgabe. In Abbildung 11 kann man erkennen, wie für jede Sorte eine Liste an Schlüsselnummern ausgegeben wird, in denen die Sorten zu finden sind. Danach folgen zwei Aussagen über die Möglichkeit, den Wunschspieß zusammenzustellen. Zuerst wird ausgegeben, in wie vielen Schüsseln die Wunschsorten zu finden sind. Infolgedessen wird ausgegeben, ob der Wunschspieß durch Auswahl der Wunschsorten schon eindeutig zusammengestellt werden kann. Dies ist in Abbildung 11 der Fall. Ist dies jedoch nicht möglich, wird ausgegeben, welche Sorten zusätzlich auf den Wunschspieß müssten, damit dieser mit Garantie richtig zusammengestellt werden kann. Abbildung 12 zeigt die Ergebnisausgabe für die Testdatei 7. Hier müssten zusätzlich zwei Sorten mehr auf den Wunschspieß, um dessen richtige Zusammenstellung zu garantieren.

```
Die jeweiligen Obstsorten können in folgenden Schüsseln liegen:
Clementine: 1
Erdbeere: 2 4
Grapefruit: 7
Himbeere: 2 4
Johannisbeere: 5
Banane: 3
Feige: 10
Ingwer: 6
Apfel: 8
Dattel: 9

Demnach können die Wunschsorten Clementine, Erdbeere, Grapefruit, Himbeere und Johannisbeere in insgesamt 5 Schüsseln zu finden sein.
Der Wunschspieß kann durch Abgehen der Schüsseln eindeutig zusammengestellt werden.
```

Abbildung 11: Ergebnisausgabe für die Daten der ersten Testdatei

5 Laufzeitverhalten

In diesem Abschnitt findet eine Laufzeitanalyse zu dem zuvor vorgestellten Algorithmus statt. Dabei wird ausschließlich der Hauptalgorithmus, also die Methoden “assignUnweighted” und “assignWeighted”,

```

Die jeweiligen Obstsorten können in folgenden Schüsseln liegen:
Apfel: 3 10 20 26
Clementine: 24
Dattel: 6 16 17
Grapefruit: 3 10 20 26
Mango: 6 16 17
Sauerkirsche: 8 14
Tamarinde: 5 23
Ugli: 18 25
Vogelbeere: 6 16 17
Xenia: 3 10 20 26
Yuzu: 8 14
Zitrone: 5 23
Erdbeere: 15
Feige: 2 11 13 22
Himbeere: 2 11 13 22
Ingwer: 4
Kiwi: 1
Litschi: 3 10 20 26
Nektarine: 7
Orange: 2 11 13 22
Quitte: 2 11 13 22
Banane: 18 25
Pflaume: 9 21
Weintraube: 9 21
Johannisbeere: 12 19
Rosine: 12 19

Demnach können die Wunschsorten Apfel, Clementine, Dattel, Grapefruit, Mango, Sauerkirsche, Tamarinde, Ugli, Vogelbeere, Xenia, Yuzu und Zitrone in insgesamt 14 Schüsseln zu finden sein.
Durch Hinzufügen der Sorten Litschi und Banane zu dem Wunschspieß, besteht eine Garantie für dessen richtige Zusammenstellung.

```

Abbildung 12: Ergebnisausgabe für die Daten der letzte Testdatei

betrachtet, welche für die Zuweisung der Knoten und die Reduzierung der Matrix zuständig sind. Alle anderen Methoden, zum Einlesen oder Ausgeben der Daten, werden in die Laufzeitbetrachtung nicht miteinbezogen. Zuerst wird grundlegend die Laufzeit jeder Komponente analysiert. Darauf folgend findet eine Betrachtung im Bestcase und Worstcase unter Beachtung der Rekursion statt.

5.1 Phase I

Bei der Suche nach unbeobachteten Sorten und Schüsseln, wird die Matrix einmal vollkommen durchlaufen. Sei n die Anzahl der Sorten, verläuft diese Suche in $O(n^2)$. Nachdem die benannten Knoten ermittelt werden konnten, müssen diese zugewiesen und entfernt werden. Sei n_{ub} die Anzahl der unbeobachteten Sorten, verläuft das Zuweisen von Schüsseln und Sorten in $O(n_{ub}^2)$. Für die Reduzierung wird danach erneut der Teil der Adjazenzmatrix durchlaufen und kopiert, der nicht entfernt werden soll. Hier ergibt sich also eine Laufzeit von $O(n + (n - n_{ub})^2)$. Zusammenfassend ergibt sich also für Phase I eine Laufzeit von:

$$O(n^2 + n_{ub}^2 + n + (n - n_{ub})^2) \leq O(3n^2)$$

5.2 Phase II

In Phase II werden sowohl die Spaltenmaxima als auch die modifizierten Spaltensummen ermittelt. Dazu wird die reduzierte Adjazenzmatrix wieder einmal ganz durchlaufen. Die Laufzeit liegt hier also bei $O(n^2)$, wobei n hier die Dimension der reduzierten Adjazenzmatrix darstellt. Berücksichtigt man dabei, dass Phase I schon durchlaufen wurde, entspricht dieses n der Dimension der Ausgangsmatrix n abzüglich der Anzahl der unbeobachteten Sorten n_{ub} . Dies wird in den Abschnitten 5.5 und 5.6 im Rahmen der Zusammenführung aller Teilschritte berücksichtigt. Nachdem Spaltenmaxima und modifizierte Summen ermittelt wurden, muss anhand dieser Informationen nun entschieden werden, ob mit Phase III oder IV fortgefahren wird.

Dazu werden zunächst alle Spalten durchlaufen. Für jede Spalte, welche eine Kante maximalen Gewichtes enthält, werden wieder alle Spalten durchlaufen. Wenn bei zwei Spalten, welche mindestens eine Kante maximalen Gewichtes enthalten, zudem die Spaltensumme gleich ist, wird nochmal jede einzelne Zeile durchlaufen, um zu prüfen, dass die Spalten tatsächlich identisch sind. Sei n die Anzahl der Schüsseln, bzw. der Spalten, n_{max} die Anzahl der Spalten, welche eine Kante maximalen Gewichtes enthalten und $n_{max/sum}$ die Anzahl von Spalten, welche eine Kante maximalen Gewichtes enthalten und deren modifizierte Spaltensumme identisch ist, ergibt sich hier folgendes Laufzeitverhalten:

$$O(n^2) \leq O(n - n_{max} + (n_{max} - n_{max/sum}) \cdot n + n_{max/sum}^2 \cdot n) \leq O(n^3)$$

5.3 Phase III

Wenn noch Knoten mit Kanten maximalen Gewichtes und individueller Kantenkonstellation bestehen, findet Phase III statt. Hier wird zunächst jede Spalte durchlaufen, welche eine solche individuelle Kan-

tenkonstellation besitzt. Es muss geprüft werden, ob in der jeweiligen Spalte nur eine Kante maximalen Gewichtes vorkommt. Ist dies der Fall, wird die entsprechende Zeile nochmal auf das gleiche Kriterium geprüft. Wenn ein Knotenpaar ermittelt werden konnte, welches durch eine Kante maximalen Gewichtes eindeutig verbunden wurde, erfolgt eine Zuweisung, Reduzierung und folglich der nächste Rekursionsaufruf. Je nach dem, wie viele Spalten mit einmaliger Kantenkonstellation, in der Liste von Spalten einmaliger Kantenkonstellationen und maximalen Gewichtes, vor einer Spalte auftauchen, welche zusätzlich auch noch oben genannte Kriterien erfüllt, ist die Laufzeit unterschiedlich. Im Worstcase verläuft die Suche in $O(n_{eK} \cdot 2n)$, wobei n_{eK} die Anzahl von Knoten mit Kanten maximalen Gewichtes und einmaliger Kantenkonstellation ist.

Nachdem ein entsprechendes Knotenpaar, welches durch eine einzige Kante maximalen Gewichtes verbunden ist, gefunden werden konnte, muss dieses Knotenpaar zugewiesen werden und die Adjazenzmatrix muss wieder, vor dem nächsten Rekursionsaufruf, reduziert werden. Da es sich bei der Zuweisung diesmal nur um zwei Knoten handelt und lediglich dem jeweiligen Match-Objekt eine Schlüsselnummer hinzugefügt werden muss, verläuft das Matching hier in $O(2)$. Das Reduzieren der Matrix mit der Methode "removeSingle" verläuft in $O(n + (n - 1)^2)$, da bis auf zwei Knoten die restliche Adjazenzmatrix kopiert wird.

Zusammenfassend handelt es sich in Phase III also um eine Laufzeit von:

$$O(n_{eK} \cdot 3n + 2 + (n - 1)^2) \leq O(n^2)$$

5.4 Phase IV

Wenn nach Phase II nur noch Knoten mit mehrfach vorkommender Kantenkonstellation und Kanten maximaler Gewichtung auftauchen, findet Phase IV statt. In Phase IV wird die HashMap so lange durchlaufen, bis auf eine Kantenkonstellation getroffen wird, in der die Anzahl von Kanten maximaler Gewichtung so groß ist wie die Anzahl von Knoten, welche diese Konstellation aufweisen. Daher ist es hier wieder von dem konkreten Problem abhängig, wie schnell Phase IV abläuft. Im Worstcase muss jedoch jeder Eintrag der HashMap einmal durchlaufen werden, bis ein bereits genanntes Kriterium erfüllt wird. Sei n_{mK} die Anzahl der Spalten (Schlüsselknoten), mit mehrfach vorkommender Kantenkonstellation, verläuft diese Suche also in $O(n_{mK} \cdot n)$. Bei der darauffolgenden Zuordnung werden wieder alle ermittelten Schlüsselknoten den entsprechenden Sortenknoten zugewiesen. Hierbei ergibt sich eine Laufzeit von $O(n_{mK}^2)$. Für die Reduzierung werden alle Zeilen und Spalten der Matrix durchlaufen und kopiert, sofern sie nicht zu den zugewiesenen, zu entfernenden Zeilen und Spalten zählen. Ähnlich, wie bei Phase III beträgt für die Reduzierung die Laufzeit $O(n + (n - n_{mK})^2)$. Für Phase IV kann man also im Worstcase von folgender Laufzeit ausgehen:

$$O(n_{mK} \cdot n + n_{mK}^2 + n + (n - n_{mK})^2) \leq O(2n^2)$$

5.5 Bestcase

Zuvor wurden die einzelnen Phasen separat und nicht unter Berücksichtigung der Rekursion betrachtet. In diesem Abschnitt soll das Laufzeitverhalten des gesamten Algorithmus in einem Bestcase-Szenario analysiert werden. Je mehr Sorten durch die vorgegebene Obergrenze bestimmt werden und je weniger Beobachtungen gegeben sind, desto mehr Knoten bestehen, welche nicht durch Kanten verbunden sind und deshalb vorab aus der Adjazenzmatrix entfernt werden können. Dies wirkt sich natürlich eher negativ auf das Ergebnis aus. Würde es keine einzige Beobachtung geben, würde der Algorithmus mit der ersten Phase das Problem in $O(3n^2)$ lösen, da für das Ermitteln der kantenlosen Knoten, deren Zuweisung und Reduzierung jeweils ein Durchlaufen der gesamten Matrix nötig ist. Geht man davon aus, dass mindestens eine Beobachtung vorhanden sein muss, ist die Laufzeit besonders gut, wenn diese Knoten eine einmalige Kantenkonstellation besitzen. Dadurch kann die Matrix bei jeder Anwendung der Phase III reduziert werden und muss nicht mehrmals in der vollen Länge durchlaufen werden. Mit Beachtung der zuvor benannten Variablen ergibt sich hier folgende Laufzeit:

$$O(n^2 + n_{ub}^2 + n + (n - n_{ub})^2) + \sum_{i=0}^{n-n_{ub}} ((n - n_{ub} - i)^2 + (n - n_{ub} - n_{max} - i + (n_{max} - n_{max/sum}) \cdot (n - n_{ub} - i) +$$

$$n_{\max/\text{sum}}^2 \cdot (n - n_{\text{ub}} - i) + n_{\text{eK}} \cdot 3(n - n_{\text{ub}} - i) + 2 + (n - n_{\text{ub}} - i - 1)^2))$$

Je nach dem, wie viele Sorten beobachtet wurden, verschlechtert sich die Laufzeit. In Beispielen, in denen man viele Knoten vorab zuweisen kann und ein Großteil der Knoten, welche durch Kanten verbunden sind, einmalige Kantenkonstellationen haben, lässt sich das oben beschriebene Laufzeitverhalten auf $O(n^2)$ abrunden. Größtenteils wird die Adjazenzmatrix mehrere Male durchlaufen, wobei immer eine Laufzeit von $O(n^2)$ entsteht. Nach jedem Durchlauf findet eine Zuweisung und Reduzierung statt. Die Laufzeit dieser Methoden ist in nahezu allen Fällen geringer, als das einfache Durchlaufen der Matrix. In manchen Spezialfällen kann es jedoch dazu kommen, dass das Durchlaufen der Adjazenzmatrix besonders aufwendig ist. So ein Fall wird im folgenden Worstcase-Abschnitt betrachtet.

5.6 Worstcase

Wenn besonders viele Knoten eine gleiche Kantenkonstellation aufweisen, dann muss in Phase II bei der Informationssammlung häufig geprüft werden, ob zwei Spalten der Adjazenzmatrix ausschließlich Kanten gleicher Gewichtung besitzen. Diese Prüfung ist besonders aufwendig. Somit ergibt sich für Phase II, wie zuvor beschrieben, folgende Laufzeit:

$$O(n - n_{\max} + (n_{\max} - n_{\max/\text{sum}}) \cdot n + n_{\max/\text{sum}}^2 \cdot n)$$

Im absoluten Worstcase-Szenario gibt es nur Knoten mit gleicher Kantenkonstellation. In diesem Fall ist $n = n_{\max} = n_{\max/\text{sum}}$. Dadurch fällt der vordere Teil des oberen Terms weg. Der übrige Teil lässt sich in dieser Betrachtung von $n_{\max/\text{sum}}^2 \cdot n$ zu n^3 umformulieren. Je mehr Knoten also eine gleiche Kantenkonstellation besitzen, desto schlechter ist die Laufzeit, zumindest für große n . Im Worstcase ergibt sich also eine Laufzeit von $O(n^3)$.

Abschließend kann man sagen, dass der Algorithmus einen Großteil der Problemkonstellationen in $O(n^2)$ löst, was der Darstellung des Graphen als Adjazenzmatrix geschuldet ist und diese zum Finden von verschiedenen Kantenkonstellationen mehrfach durchlaufen werden muss. Je mehr Knoten jedoch mit gleicher Kantenkonstellation bestehen, desto größer wird hier die Laufzeit, wobei als Obergrenze $O(n^3)$ gilt.

6 Ergebnistabelle

Testdatei	Zuweisung der Sorten und Schüsseln	Aussage über den Wunschspieß
spiesse0.txt	Weintraube: 3 Brombeere: 4 1 Apfel: 4 1 Banane: 5 Pflaume: 6 Erdbeere: 2	Demnach können die Wunschsorten Weintraube, Brombeere und Apfel in insgesamt 3 Schüsseln zu finden sein. Der Wunschspieß kann durch Abgehen der Schüsseln eindeutig zusammengestellt werden.
spiesse1.txt	Clementine: 1 Erdbeere: 4 2 Grapefruit: 7 Himbeere: 4 2 Johannisbeere: 5 Banane: 3 Feige: 10 Ingwer: 6 Apfel: 8 Dattel: 9	Demnach können die Wunschsorten Clementine, Erdbeere, Grapefruit, Himbeere und Johannisbeere in insgesamt 5 Schüsseln zu finden sein. Der Wunschspieß kann durch Abgehen der Schüsseln eindeutig zusammengestellt werden.

spiesse2.txt	Apfel: 1 Banane: 10 11 5 Clementine: 10 11 5 Himbeere: 10 11 5 Kiwi: 6 Litschi: 7 Dattel: 9 2 Erdbeere: 8 Feige: 9 2 Johannisbeere: 4 Ingwer: 12 Grapefruit: 3	Demnach können die Wunschsorten Apfel, Banane, Clementine, Himbeere, Kiwi und Litschi in insgesamt 6 Schüsseln zu finden sein. Der Wunschspieß kann durch Abgehen der Schüsseln eindeutig zusammengestellt werden.
spiesse3.txt	Clementine: 5 Erdbeere: 8 Feige: 10 7 Himbeere: 1 Ingwer: 10 7 Kiwi: 12 Litschi: 11 2 Grapefruit: 11 2 Johannisbeere: 9 Nektarine: 4 Orange: 3 Dattel: 13 Apfel: 14 6 Banane: 14 6 unknown-0: 15	Demnach können die Wunschsorten Clementine, Erdbeere, Feige, Himbeere, Ingwer, Kiwi und Litschi in insgesamt 8 Schüsseln zu finden sein. Durch Hinzufügen der Sorte Grapefruit zu dem Wunschspieß, besteht eine Garantie für dessen richtige Zusammenstellung.
spiesse4.txt	Apfel: 9 Feige: 13 Grapefruit: 8 Ingwer: 6 Kiwi: 2 Nektarine: 7 Orange: 14 Pflaume: 12 Clementine: 15 Erdbeere: 16 Himbeere: 11 Mango: 1 Banane: 17 Quitte: 4 Litschi: 3 Dattel: 10 Johannisbeere: 5	Demnach können die Wunschsorten Apfel, Feige, Grapefruit, Ingwer, Kiwi, Nektarine, Orange und Pflaume in insgesamt 8 Schüsseln zu finden sein. Der Wunschspieß kann durch Abgehen der Schüsseln eindeutig zusammengestellt werden.

spiesse5.txt	Apfel: 4 19 1 Banane: 9 3 Clementine: 20 Dattel: 6 Grapefruit: 4 19 1 Himbeere: 5 Mango: 4 19 1 Nektarine: 14 Orange: 16 2 Pflaume: 10 Quitte: 9 3 Sauerkirsche: 16 2 Tamarinde: 12 Johannisbeere: 13 Kiwi: 15 7 Litschi: 15 7 Rosine: 17 Ingwer: 11 Erdbeere: 8 unknown-0: 18	Demnach können die Wunschsorten Apfel, Banane, Clementine, Dattel, Grapefruit, Himbeere, Mango, Nektarine, Orange, Pflaume, Quitte, Sauerkirsche und Tamarinde in insgesamt 13 Schüsseln zu finden sein. Der Wunschspieß kann durch Abgehen der Schüsseln eindeutig zusammengestellt werden.
spiesse6.txt	Clementine: 7 Erdbeere: 10 Himbeere: 18 Orange: 20 Quitte: 4 Rosine: 15 11 Ugli: 15 11 Vogelbeere: 6 Apfel: 3 Banane: 8 Feige: 22 Ingwer: 14 Johannisbeere: 12 Kiwi: 23 Litschi: 13 Nektarine: 2 Weintraube: 21 Dattel: 5 Pflaume: 9 Tamarinde: 19 Grapefruit: 17 Mango: 1 Sauerkirsche: 16	Demnach können die Wunschsorten Clementine, Erdbeere, Himbeere, Orange, Quitte, Rosine, Ugli und Vogelbeere in insgesamt 8 Schüsseln zu finden sein. Der Wunschspieß kann durch Abgehen der Schüsseln eindeutig zusammengestellt werden.

spiesse7.txt	<p> Apfel: 10 20 26 3 Clementine: 24 Dattel: 6 16 17 Grapefruit: 10 20 26 Mango: 6 16 17 Sauerkirsche: 14 8 Tamarinde: 23 5 Ugli: 25 18 Vogelbeere: 6 16 17 Xenia: 10 20 26 3 Yuzu: 14 8 Zitrone: 23 5 Erdbeere: 15 Feige: 11 13 22 2 Himbeere: 11 13 22 2 Ingwer: 4 Kiwi: 1 Litschi: 10 20 26 3 Nektarine: 7 Orange: 11 13 22 2 Quitte: 11 13 22 2 Banane: 25 18 Pflaume: 21 9 Weintraube: 21 9 Johannisbeere: 19 12 Rosine: 19 12 </p>	<p>Demnach können die Wunschsorten Apfel, Clementine, Dattel, Grapefruit, Mango, Sauerkirsche, Tamarinde, Ugli, Vogelbeere, Xenia, Yuzu und Zitrone in insgesamt 14 Schüsseln zu finden sein.</p> <p>Durch Hinzufügen der Sorten Litschi und Banane zu dem Wunschspieß, besteht eine Garantie für dessen richtige Zusammenstellung.</p>
--------------	---	---

7 Sourcecode

In diesem Abschnitt sind sowohl die Methoden “assignUnweighted” als auch “assignWeighted” aufgeführt, welche hauptsächlich zur algorithmischen Lösung dienen. Zudem wird die Implementierung der Methoden “removeSingle” und “removeMultiple” gezeigt, welche notwendig sind, um die Matrix zu reduzieren.

7.1 assignUnweighted

```

1  /**
   * Weist Knoten mit ungewichteten Kanten zu und entfernt sie.
   *
   * @param matrix      Die Adjazenzmatrix
   * @param matches     Die Liste der gefundenen Zuweisungen
   * @return            Vollstaendige Liste der Zuweisungen
   */
private static ArrayList<Match> assignUnweighted(int[][] matrix,
9  ArrayList<Match> matches) {
    // Liste der zu entfernenden Spaltenindizes
11   ArrayList<Integer> colIndices = new ArrayList<>();

    // Liste der zu entfernenden Zeilenindizes
13   ArrayList<Integer> rowIndices = new ArrayList<>();

    // Durchlaeuft die Matrix und ermittelt Zeilen und Spalten,
17   // die nur mit null gewichtet sind.
    for (int i = 1; i < matrix.length; i++) {
19       boolean isColEmpty = true;
        boolean isRowEmpty = true;
21       for (int j = 1; j < matrix[i].length; j++) {
            if (matrix[i][j] != 0) {
23                 isRowEmpty = false;
            }

            if (matrix[j][i] != 0) {
25                 isColEmpty = false;
            }
27         }

        if (isColEmpty) {
31             colIndices.add(i);
        }
33     }

        if (isRowEmpty) {
35             rowIndices.add(i);
        }
37     }
39 }

    // Weist die Schuesselnummerknoten den Sortenknoten zu, wenn es Knoten ohne Kanten gab.
41   if (colIndices.size() > 0) {
        for (int i = 0; i < rowIndices.size(); i++) {
43             for (int j = 0; j < colIndices.size(); j++) {
                matches.get(matrix[rowIndices.get(i)][0]-1)
45                 .addBowNum(matrix[0][colIndices.get(j)]);
            }
47         }

        // Reduziert die Matrix um die zugewiesenen Knoten.
49         matrix = removeMultiple(matrix, rowIndices, colIndices);
        printMatrix(matrix, "Anwendung Phase I:");
51     }
53 }

    // Startet den ersten Rekursionsaufruf der Hauptloesungsmethode.
55   return assignWeighted(matrix, matches);
}

```

7.2 assignWeighted

```

/**
 * Rekursive Methode zur Zuweisung von Sorten und Schuesseln und
 * Reduzierung der Adjazenzmatrix
 *
 * @param matrix      Die momentane Adjazenzmatrix
 * @param matches     Die Liste der gefundenen Zuweisungen
 * @return            Die Liste aller Zuweisungen
 */
private static ArrayList<Match> assignWeighted (int[][] matrix,
ArrayList<Match> matches) {
    // Summen der Spalten
    int[] colSums = new int[matrix.length-1];

    // Maxima der Spalten
    int[] colMaxs = new int[matrix.length-1];

    // hoechstes Kantengewicht der gesamten Adjazenzmatrix
    int globalMax = -1;

    // Durchlaeuft die Spalten und ermittelt Spaltensumme, -maxima und das hoechste
    // Kantengewicht der gesamten Matrix.
    for (int i = 1; i < matrix.length; i++) {
        // Momentane(s) Spaltensumme / Spaltenmaximum
        int colMax = -1;
        int colSum = 0;

        // Durchlaeuft die Reihen und ermittelt momentane(s) Spaltensumme /-maximum
        for (int j = 1; j < matrix[i].length; j++) {
            if (matrix[j][i] > globalMax) {
                globalMax = matrix[j][i];
            }
            if (matrix[j][i] > colMax) {
                colMax = matrix[j][i];
            }

            // Spaltensumme wird mit Zeilenindex multipliziert, um eindeutige
            // Zuweisungen eher finden zu koennen.
            colSum += matrix[j][i] * j;
        }

        colMaxs[i-1] = colMax;
        colSums[i-1] = colSum;
    }

    // Liste von Spaltenindizes, bei denen in den Spalten ein globales Maximum
    // existiert und die Kantenkonstellation nur einmal vorkommt.
    ArrayList<Integer> singleIndexes = new ArrayList<>();
    HashMap<Integer, ArrayList<Integer>> multipleIndexes = new HashMap<>();

    // Durchlaeuft alle Spaltenmaxima.
    for (int i = 0; i < colMaxs.length; i++) {

        // Sucht nach Spalten, die ein globales Maximum besitzen und deren
        // Kantenkonstellation nur einmal vorkommt.
        if (colMaxs[i] == globalMax) {
            boolean isSingle = true;
            for (int j = 0; j < colMaxs.length; j++) {
                if (i != j && colMaxs[j] == globalMax && colSums[i] == colSums[j]) {

                    boolean isEqual = true;
                    for (int k = 1; k < matrix.length; k++) {
                        if (matrix[k][i+1] != matrix[k][j+1]) {
                            isEqual = false;
                            break;
                        }
                    }

                    isSingle = !isEqual;
                }
            }

            if (isEqual) {
                if (multipleIndexes.containsKey(i+1)) {

```

```

72         multipleIndexes.get(i+1).add(j+1);
73     } else {
74         multipleIndexes.put(i+1, new ArrayList<Integer>());
75         multipleIndexes.get(i+1).add(j+1);
76     }
77 }
78
79 }
80 }
81
82     if (isSingle) {
83         singleIndexes.add(i+1);
84     }
85 }
86 }
87
88 // Knoten mit einmaliger Kantenkonstellation und globalem Maximum existieren noch
89 // => diese Knoten zuerst zuordnen
90 if (!singleIndexes.isEmpty()) {
91
92     // Durchläuft die Spalten mit einmaligen Kantenkonstellationen und globalem Maximum.
93     for (int i = 0; i < singleIndexes.size(); i++) {
94
95         // Anzahl der Maxima in der momentanen Spalte
96         int maxCount = 0;
97
98         // Index der Zeile in der ein Maximum vorkommt | findet nur Verwendung,
99         // wenn in der Spalte ausschliesslich ein Maximum existiert
100        int maxRowIndex = 0;
101
102        // Ermittelt die Anzahl der Maxima in den entsprechenden Spalten und
103        // den Zeilenindex, falls nur ein Maximum existiert.
104        for (int j = 1; j < matrix.length; j++) {
105            if (matrix[j][singleIndexes.get(i)] == globalMax) {
106                maxCount++;
107                maxRowIndex = j;
108            }
109        }
110
111        // Nur ein Maximum existiert in der momentanen Spalte.
112        if (maxCount == 1) {
113
114            // Ermittelt, ob dieses Maximum auch in seiner Zeile nur einmal vorkommt.
115            boolean isSingleInRow = true;
116            for (int j = 1; j < matrix[i].length; j++) {
117                if (singleIndexes.get(i) != j && matrix[maxRowIndex][j] == globalMax) {
118                    isSingleInRow = false;
119                    break;
120                }
121            }
122
123            // Ermittelte Kante kommt sowohl in seiner Spalte als auch in
124            // seiner Zeile nur einmal vor
125            // => Zuweisung der Schuesselnummer zu der entsprechenden Frucht
126            // => Entfernen der beiden Knoten und deren Kanten
127            // => Naechster Rekursionsaufruf
128            if (isSingleInRow) {
129                matches.get(matrix[maxRowIndex][0]-1)
130                    .addBowlNum(matrix[0][singleIndexes.get(i)]);
131                matrix = removeSingle(matrix, maxRowIndex, singleIndexes.get(i));
132                printMatrix(matrix, "Anwendung Phase III:");
133                return assignWeighted(matrix, matches);
134            }
135        }
136    }
137
138 // Es existiert kein Knoten mit globalem Maximum und einmaliger
139 // Kantenkonstellation mehr.
140 } else {
141
142     // Durchläuft die Spalten mit gleicher Kantenkonstellation.
143     for (int key : multipleIndexes.keySet()) {

```

```

146 // Anzahl der Maxima in der momentanen Spalte
147 int maxCount = 0;
148
149 // Indizes von zugehoerigen Zeilen gleicher Kantenkonstellation
150 ArrayList<Integer> equalMaxRows = new ArrayList<>();
151
152 // Durchlaeuft momentane Spalte mit gleicher Kantenkonstellation.
153 for (int i = 1; i < matrix.length; i++) {
154
155     // Erhoeht die Anzahl der Maxima von der momentanen Spalte und fuegt die
156     // entsprechenden Zeilen der
157     // Liste hinzu.
158     if (matrix[i][key] == globalMax) {
159         maxCount++;
160         equalMaxRows.add(i);
161     }
162 }
163
164 // Ist die Anzahl der Maxima von Knoten gleicher Konstellation so gross,
165 // wie die Anzahl von Knoten, welche diese Konstellation von Kanten besitzen?
166 // => Zuweisung, Reduzierung und naechster Rekursionsaufruf
167 if (maxCount == multipleIndexes.get(key).size()+1) {
168     // Vervollstaendigt die Liste von Spaltenindizes der Spalten
169     // gleicher Kantenkonstellation
170     multipleIndexes.get(key).add(key);
171
172     // Durchlaeuft die Zeilen, also Sorten, welche zugewiesen werden muessen.
173     for (int i = 0; i < equalMaxRows.size(); i++) {
174
175         // Durchlaeuft die Anzahl der Spalten, also Schuesseln,
176         // welche zugewiesen werden muessen.
177         for (int j = 0; j < multipleIndexes.get(key).size(); j++) {
178
179             // Weist die momentane Schuessel der momentanen Sorte zu.
180             matches.get(matrix[equalMaxRows.get(i)][0]-1)
181                 .addBowlNum(matrix[0][multipleIndexes.get(key).get(j)]);
182         }
183     }
184
185     // Reduzierung der Matrix um die zugewiesenen Knoten
186     matrix = removeMultiple(matrix, equalMaxRows, multipleIndexes.get(key));
187
188     // Ausgabe der neuen Matrix
189     printMatrix(matrix, "Anwendung Phase IV:");
190
191     // Naechster Rekursionsaufruf
192     return assignWeighted(matrix, matches);
193 }
194 }
195 return matches;
196 }

```


7.3 removeSingle

```
/**
 * Kopiert eine Adjazenzmatrix ohne die angegebene Zeile und Spalte.
 *
 * @param matrix Die Adjazenzmatrix
 * @param row Die zu entfernende Zeile
 * @param col Die zu entfernende Spalte
 * @return Adjazenzmatrix ohne die entsprechende Zeile und Spalte
 */
private static int[][] removeSingle(int[][] matrix, int row, int col) {
    // Anzahl der uebersprungenen Zeilen | hier entweder 0 oder 1
    int rowSkipped = 0;
    // die neue Adjazenzmatrix
    int[][] newMatrix = new int[matrix.length-1][matrix.length-1];

    // Kopiert alle Zeilen ohne die angegebene Reihe
    for (int i = 0; i < matrix.length; i++) {
        if (i != row) {
            // Anzahl der uebersprungenen Spalten | hier entweder 0 oder 1
            int colSkipped = 0;

            // Kopiert alle Spalten ohne die angegebene Reihe
            for (int j = 0; j < matrix[i].length; j++) {
                if (j != col) {
                    newMatrix[i-rowSkipped][j-colSkipped] = matrix[i][j];
                } else {
                    colSkipped++;
                }
            }
            } else {
                rowSkipped++;
            }
        }

    return newMatrix;
}
```

7.4 removeMultiple

```

1  /**
2   * Kopiert eine Adjazenzmatrix ohne die angegebenen Zeilen und Spalten.
3   *
4   * @param matrix      Die Adjazenzmatrix
5   * @param rows        Liste der zu entfernenden Zeilen
6   * @param cols        Liste der zu entfernenden Spalten
7   * @return           Adjazenzmatrix ohne die entsprechenden Zeilen und Spalten
8   */
9  private static int[][] removeMultiple(int[][] matrix, ArrayList<Integer> rows,
10                                     ArrayList<Integer> cols) {
11      int[][] newMatrix = new int[matrix.length-rows.size()][matrix.length-cols.size()];
12
13      // Anzahl der uebersprungenen Zeilen
14      int skippedRows = 0;
15
16      // Kopiert die Zeilen, welche nicht entfernt werden sollen.
17      for (int i = 0; i < matrix.length; i++) {
18
19          // Testet, ob sich die momentane Zeile unter den zu entfernenden Zeilen befindet.
20          boolean isNotRow = true;
21          for (int k = 0; k < rows.size(); k++) {
22              if (i == rows.get(k)) {
23                  isNotRow = false;
24                  break;
25              }
26          }
27
28          // Kopiert entsprechende Kanten Zeile, wenn die Zeile nicht entfernt werden soll.
29          if (isNotRow) {
30              // Anzahl der uebersprungenen Spalten
31              int skippedCols = 0;
32
33              // Testet, ob sich die momentane Spalte unter den zu entfernenden Spalten befindet.
34              for (int j = 0; j < matrix[i].length; j++) {
35                  boolean isNotCol = true;
36                  for (int k = 0; k < cols.size(); k++) {
37                      if (j == cols.get(k)) {
38                          isNotCol = false;
39                      }
40                  }
41
42                  // Kopiert die Kante, wenn die Spalte nicht entfernt werden soll.
43                  if (isNotCol) {
44                      newMatrix[i-skippedRows][j-skippedCols] = matrix[i][j];
45                  } else {
46                      // Ansonsten wird die Spalte uebersprungen.
47                      skippedCols++;
48                  }
49              }
50          } else {
51              // Ansonsten wird die Zeile uebersprungen.
52              skippedRows++;
53          }
54      }
55
56      return newMatrix;
57  }

```