

# Aufgabe 2: Spießgesellen

Teilnahme-Id: 55628

Bearbeiter dieser Aufgabe:  
Michal Boron

April 2021

## Inhaltsverzeichnis

<b>1</b>	<b>Lösungsidee</b>	<b>2</b>
1.1	Formulierung des Problems . . . . .	2
1.2	Bipartiter Graph . . . . .	2
1.3	Logik . . . . .	3
1.4	Zusammenhangskomponenten . . . . .	4
1.5	Prüfung auf Korrektheit der Eingabe . . . . .	7
1.6	Laufzeit . . . . .	8
<b>2</b>	<b>Umsetzung</b>	<b>12</b>
2.1	Klasse Solver . . . . .	12
2.2	Klasse Graph . . . . .	13
<b>3</b>	<b>Beispiele</b>	<b>14</b>
3.1	Beispiel 0 (Aufgabenstellung — Teil a) . . . . .	14
3.2	Beispiel 1 (BWINF) . . . . .	15
3.3	Beispiel 2 (BWINF) . . . . .	15
3.4	Beispiel 3 (BWINF) . . . . .	15
3.5	Beispiel 4 (BWINF) . . . . .	15
3.6	Beispiel 5 (BWINF) . . . . .	16
3.7	Beispiel 6 (BWINF) . . . . .	16
3.8	Beispiel 7 (BWINF) . . . . .	16
3.9	Beispiel 8 . . . . .	16
3.10	Beispiel 9 . . . . .	17
3.11	Beispiel 10 . . . . .	17
3.12	Beispiel 11 . . . . .	17
3.13	Beispiel 12 . . . . .	18
3.14	Beispiel 13 . . . . .	18
3.15	Beispiel 14 . . . . .	19
3.16	Beispiel 15 . . . . .	19
<b>4</b>	<b>Quellcode</b>	<b>20</b>

# 1 Lösungsidee

## 1.1 Formulierung des Problems

**Axiom 1.** Jeder **Obstsorte** wird genau ein *einzigartiger natürlicher Index* zugewiesen.

Man schreibt:  $o(x, i)$  — eine Obstsorte  $x$  besitzt einen Index  $i$ .

Gegeben sind eine Menge von  $n$  Obstsorten  $A$  und eine Menge von  $n$  ganzen Zahlen  $B = \{1, 2, \dots, n\}$ , zu der die Indizes der Obstsorten aus  $A$  gehören.

**Definition 1** (Spießkombination). Als eine **Spießkombination**  $K = (F, Z)$  bezeichnet man eine Verknüpfung von zwei Mengen  $F \subseteq A$  und  $Z$ , wobei  $Z = \{i \in B \mid \forall x \in F : o(x, i)\}$ .

Gegeben sind auch  $m$  Spießkombinationen, wobei jede  $i$ -te Spießkombination aus einer Menge von Obstsorten  $F_i \subseteq A$  und einer Menge der Indizes  $Z_i \subseteq B$  besteht. Nach der Definition 1 besteht die Menge  $Z_i$  nur aus den in  $B$  enthaltenen Indizes, die zu den Obstsorten in  $F_i$  gehören, deshalb sind die beiden Mengen  $F_i$  und  $Z_i$  auch gleichmächtig.

Außerdem gegeben ist auch eine **Wunschliste**  $W \subseteq A$ .

Die Aufgabe ist ein Entscheidungsproblem. Es soll entschieden werden, ob die Menge der Indizes der in  $W$  enthaltenen Obstsorten  $W' \subseteq B$  anhand der  $m$  Spießkombinationen eindeutig bestimmt werden kann. Falls ja, soll sie auch ausgegeben werden.

In den folgenden Überlegungen wird angenommen, dass das Axiom 1 für alle Obstsorten in der Eingabe gilt. Es ist aber möglich, dass die Spießkombination in einer Eingabe diesem Axiom nicht folgen, das heißt, es an einer Stelle einen Widerspruch gibt. Laut der Aufgabenstellung ist ein solcher Fall nicht ausgeschlossen. Um diesen Fall zu verhindern, muss man die Korrektheit der Eingabe überprüfen. Mehr dazu folgt im Teil 1.5.

## 1.2 Bipartiter Graph

Man kann die beiden Mengen  $A$  und  $B$  zu Knoten eines bipartiten Graphen  $G = (A \cup B = V, E)$  umwandeln. Die Menge der Kanten  $E$  wird im Folgenden festgelegt. Man stellt den Graphen als eine Adjazenzmatrix  $M$  der Größe  $n \times n$  dar. Als  $M_i$  bezeichnet wird die Liste der Länge  $n$ , die die Beziehungen eines Knotens  $i \in A$  zu jedem Knoten  $j \in B$  als 1 (Kante) oder 0 (keine Kante) darstellt. Als  $M_{i,j}$  bezeichnet wird die  $j$ -te Stelle in der  $i$ -ten Liste der Matrix.

Nach Axiom 1 gehört jeder Obstsorte aus  $A$  genau ein Index aus  $B$ . Dennoch man kann am Anfang keiner Obstsorte einen Index zuweisen. Deshalb wird zunächst jeder Knoten aus  $A$  mit jedem Knoten aus  $B$  durch eine Kante verbunden:

$$E = A \times B = \{(x, y) \mid x \in A \text{ und } y \in B\}.$$

Am Anfang ist  $M$  dementsprechend voll mit 1-en. Bei der Erstellung der Adjazenzmatrix kann man den Vorteil nutzen, dass die jeweilige Liste von Nachbarn des jeden Knotens  $x \in A$  nur aus 0-en und 1-en besteht, indem diese Liste als eine Bitmaske dargestellt werden kann (mehr dazu in der Umsetzung).

Jede  $i$ -te Spießkombination bringt mit sich Informationen über die Obstsorten in  $F_i$ . Man kann Folgendes feststellen.

**Lemma 1.** Sei  $K = (F, Z)$  eine Spießkombination. Für jede Obstsorte  $o(x, i)$ , wobei  $x \in F$ , gilt:

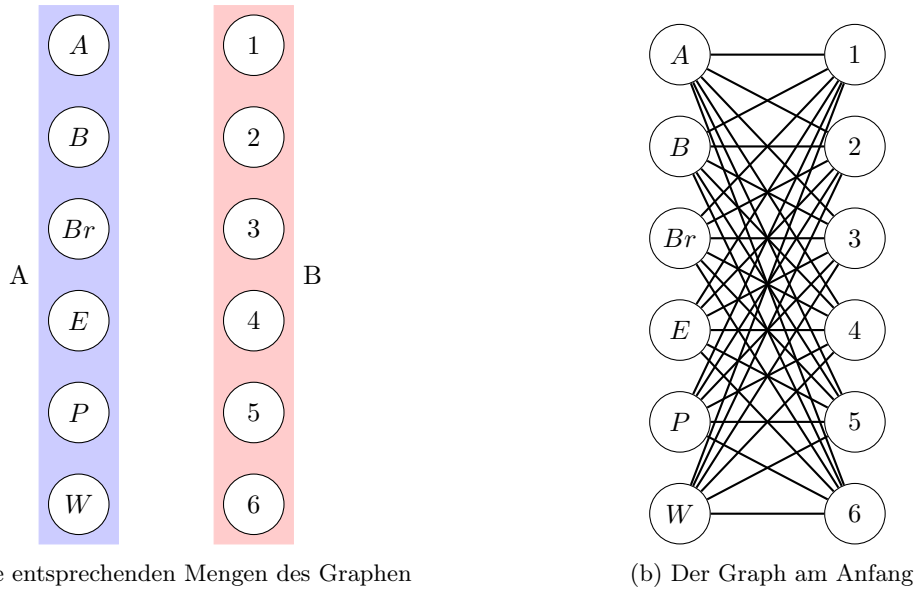
- (i)  $i \in Z$ ,
- (ii)  $i \notin B \setminus Z$ .

Deshalb darf man alle Kanten, die aus jedem Knoten  $x \in F$  zu jedem Knoten  $y \in B \setminus Z$ , sowie die aus jedem Knoten  $p \in Z$  zu jedem Knoten  $q \in A \setminus F$  führen, aus  $E$  entfernen.

*Beweis.* Nach Definition 1 gilt (i). Nach Axiom 1 besitzt jede Obstsorte einen einzigartigen Index  $i$ , deshalb kann  $i$  nicht gleichzeitig zu  $Z$  und  $B \setminus Z$  gehören (ii).

Die Folgerung gilt, da jede Kante zwischen zwei beliebigen Knoten  $x \in A$  und  $y \in B$  die Möglichkeit darstellt, dass  $x$  einen Index  $y$  besitzen kann. Wenn eine Teilmenge von  $A$  und  $B$  in Form einer Spießkombination ausgegliedert wird, schrumpft die Anzahl an möglichen Zuweisungen zwischen jedem  $x$  und jedem  $y$ . □

Abbildung 1: Beide Abbildungen stellen den Graphen für das Beispiel aus der Aufgabenstellung dar. Die Buchstaben stehen für die entsprechenden Obstsorten aus diesem Beispiel (s. auch 3.1).



Aus Lemma 1 ergibt sich direkt auch eine andere Beobachtung.

**Korollar 1.** Sei  $C = (L_c \cup R_c, E_c)$  eine Zusammenhangskomponente in  $G$ . Sei  $K = (F, Z)$  eine Spießkombination. Falls  $F \subseteq L_c$  gilt, dann gilt für jede Obstsorte  $o(x, i)$ , wobei  $x \in F$ :

- (i)  $i \in Z$ ,
- (ii)  $i \notin R_c \setminus Z$ .

Deshalb werden alle Kanten, die aus jedem Knoten  $x \in F$  zu jedem Knoten  $y \in R_c \setminus Z$ , sowie die aus jedem Knoten  $p \in Z$  zu jedem Knoten  $q \in L_c \setminus F$  führen, aus  $E$  entfernt.

### 1.3 Logik

Da Bitmasken für die Darstellung jeder Liste  $M_i$  ( $i \in A$ ) verwendet werden, kann die Laufzeit bei der Verarbeitung der jeweiligen Spießkombination optimiert werden (mehr dazu im Teil Laufzeit), weil man für die Operation des Entfernens Logikgatter verwenden kann.

Betrachten wir eine Spießkombination  $s = (F_s, Z_s)$ . Wir erstellen 3 Bitmasken  $bf, bn$  und  $br$  jeweils der Länge  $n$ . Die Bitmaske  $bf$  besteht aus  $n$  1-en. In der Maske  $bn$  stehen die 1-Bits an allen Stellen, die den Indizes in  $Z_s$  entsprechen. Die Bitmaske  $br$  wird auf folgende Weise definiert (mehr dazu in der Umsetzung):

$$br := \neg(bn) \wedge bf.$$

So können wir auf allen Listen  $M_i$ , wobei  $i \in F_s$ , die AND-Operation mit der Maske  $bn$  durchführen:

$$M_i := M_i \wedge bn.$$

Analog führen wir die AND-Operation mit der Maske  $br$  auf allen Listen  $M_j$ , wobei  $j \in A \setminus F_s$ , durch:

$$M_j := M_j \wedge br.$$

Abbildung 3: Beide Abbildungen stellen die Adjazenzmatrix für das Beispiel aus der Aufgabenstellung dar. Die Buchstaben in der ersten Spalte stehen für die entsprechenden Obstsorten und die Zahlen in der ersten Zeile stehen für die Indizes aus demselben Beispiel (s. auch 3.1). Auf der Abb. 4b stehen  $bn$  und  $br$  für die entsprechenden Bitmasken.

Spießkombination:  $F = \{\text{Banane, Pflaume, Weintraube}\}$   
 $Z = \{3, 5, 6\}$

	6	5	4	3	2	1
$bn$	1	1	0	1	0	0
$br$	0	0	1	0	1	1

	6	5	4	3	2	1
$A$	0	1	1	0	0	1
$B$	0	1	1	0	0	1
$Br$	0	1	1	0	0	1
$E$	1	0	0	1	1	0
$P$	1	0	0	1	1	0
$W$	1	0	0	1	1	0

	6	5	4	3	2	1
$A$	0	0	1	0	0	1
$B$	0	1	0	0	0	0
$Br$	0	0	1	0	0	1
$E$	0	0	0	0	1	0
$P$	1	0	0	1	0	0
$W$	1	0	0	1	0	0

(a)  $M$  vor der neuen Spießkombination (b)  $M$  nach der Verarbeitung der beschriebenen Spießkombination.

Auf der obigen Abbildung werden **blau** und **rot** die entsprechenden Listen gekennzeichnet, auf denen die AND-Operation mit der entsprechenden Bitmaske durchgeführt wurde. **Rot** werden die Bits gekennzeichnet, die sich nach der Verarbeitung der Spießkombination veränderten.

Was die beschriebenen Operationen verursachen, wird anhand der folgenden Fallunterscheidung erläutert.

- Falls es sich um einen Knoten  $x \in F_s$  handelt, betrachten wir dazu die entsprechende Liste  $M_x$  und einen Knoten  $y \in B$ .
  - Falls der Knoten  $y$  zu  $Z_s$  gehört, aber an der Stelle  $M_{x,y}$  0 steht, bleibt es auch 0.
  - Falls der Knoten  $y$  zu  $Z_s$  gehört und an der Stelle  $M_{x,y}$  1 steht, bleibt es auch 1.
  - Falls der Knoten  $y$  zu  $Z_s$  nicht gehört und an der Stelle  $M_{x,y}$  0 steht, bleibt es auch 0.
  - Falls der Knoten  $y$  zu  $Z_s$  nicht gehört, aber an der Stelle  $M_{x,y}$  1 steht, wird die Stelle  $M_{x,y}$  zu 0.
- Falls es sich um einen Knoten  $x \in A \setminus F_s$  handelt, betrachten wir dazu die entsprechende Liste  $M_x$  und einen Knoten  $y \in B$ .
  - Falls der Knoten  $y$  zu  $Z_s$  nicht gehört, aber an der Stelle  $M_{x,y}$  0 steht, bleibt es auch 0.
  - Falls der Knoten  $y$  zu nicht  $Z_s$  gehört und an der Stelle  $M_{x,y}$  1 steht, bleibt es auch 1.
  - Falls der Knoten  $y$  zu  $Z_s$  gehört, aber an der Stelle  $M_{x,y}$  1, wird die Stelle  $M_{x,y}$  zu 0.
  - Falls der Knoten  $y$  zu  $Z_s$  gehört und an der Stelle  $M_{x,y}$  0 steht, bleibt es auch 0.

## 1.4 Zusammenhangskomponenten

Nach der Verarbeitung der allen  $m$  Spießkombinationen verfügen wir über den Graphen  $G$ , in dem viele Kanten in  $E$  entfernt wurden. Auf diese Weise können wir schon anfangen, die Indizes der Obstsorten aus  $W$  festzulegen. Definieren wir zunächst, was generell ein **Matching** ist.

**Definition 2** (Matching). Sei  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  ein ungerichteter Graph. Als ein **Matching** bezeichnen wir eine Teilmenge  $\mathcal{S} \subseteq \mathcal{E}$ , sodass für alle  $v \in \mathcal{V}$  gilt, dass höchstens eine Kante aus  $\mathcal{S}$  inzident zu  $v$  ist. Wir bezeichnen einen Knoten  $v \in \mathcal{V}$  als in  $\mathcal{S}$  **gematcht**, wenn eine Kante aus  $\mathcal{S}$  inzident zu  $v$  ist. [1, S. 732].

Zwischen verschiedenen Typen des Matchings unterscheidet man auch das **perfekte Matching**.

**Definition 3** (Perfektes Matching). Sei  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  ein ungerichteter Graph. Ein **perfektes Matching** ist so ein Matching, in dem alle Knoten aus  $\mathcal{V}$  gematcht sind. [1, S. 735, Übung]

Um die Aufgabe in der Form zu lösen, eignet sich gut der **Satz von Hall**, der als ein Ausgangspunkt der ganzen Matching-Theorie gilt. Um sich dieses Satzes zu bedienen, muss man noch den Begriff der **Nachbarschaft** einführen.

**Definition 4** (Nachbarschaft). Sei  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  ein ungerichteter Graph. Für alle  $X \subseteq \mathcal{V}$  definieren wir die **Nachbarschaft** von  $X$  als  $N(X) = \{y \in \mathcal{V} \mid \forall x \in X : (x, y) \in \mathcal{E}\}$ . [1, S. 735, Übung]

**Satz 1** (Satz von Hall). Sei  $\mathcal{G} = (\mathcal{L} \cup \mathcal{R}, \mathcal{E})$  ein bipartiter, ungerichteter Graph. Es existiert ein perfektes Matching genau dann, wenn es für alle Teilmengen  $\mathcal{K} \subseteq \mathcal{L}$  gilt:  $|\mathcal{K}| \leq |N(\mathcal{K})|$ . [1, S. 736, Übung]

*Beweis.* Auf den Beweis verzichte ich. Ein Beweis ist beispielsweise hier <sup>1</sup> zu finden.  $\square$

An dieser Stelle stellen wir Folgendes fest.

**Lemma 2.** Sei  $C = (V_C, E_C)$  eine beliebige Zusammenhangskomponente in  $G$ . Dann bildet  $C$  nach Verarbeitung jeder  $k$ -ten Spießkombination selbst einen vollständigen, bipartiten Graphen.

*Beweis.* Diese Aussage kann durch die vollständige Induktion für jedes  $k \in \mathbb{N}$  bewiesen werden.

**Induktionsanfang:** Die beiden Mengen  $A$  und  $B$  sind gleichmächtig und ganz am Anfang ist  $G$  vollständig. Sei die erste Spießkombination  $K_1 = (F_1, Z_1)$ , wobei  $F_1 \neq A$ . (Falls  $F_1 = A$ , dann gilt sofort die Aussage für  $k = 1$ .) Nach der Verarbeitung von  $K_1$  entstehen zwei Zusammenhangskomponenten:  $C_1 = (L_1 \cup R_1, E_1)$  und  $C_2 = (L_2 \cup R_2, E_2)$ ,  $C_1 \cup C_2 = G$ , wobei o.B.d.A.  $L_1 \cup R_1 = F \cup Z$ . Dann sind  $L_1$  und  $R_1$  nach Definition 1 auch gleichmächtig. Ebenfalls sind dann  $L_2$  und  $R_2$  gleichmächtig. Nach Lemma 1 gilt, dass alle Kanten zwischen  $C_1$  und  $C_2$  aus  $E$  entfernt werden, aber alle Kanten innerhalb von  $C_1$  und innerhalb von  $C_2$  beibehalten werden. Dies bedeutet, dass die Komponenten  $C_1$  und  $C_2$  selbst vollständige, bipartite Graphen sind.

Damit ist die Aussage für  $k = 1$  bewiesen und der Induktionsanfang erledigt.

**Induktionsschritt:** Es gelte die Aussage, also die **Induktionsannahme**, für ein beliebiges, aber festes  $k \in \mathbb{N}$ , d.h., es gelte, dass jede Zusammenhangskomponente in  $G$  nach Verarbeitung von  $k$  Spießkombinationen selbst einen vollständigen, bipartiten Graphen bildet.

Zu zeigen ist die Aussage für  $k + 1$ , also, dass jede Zusammenhangskomponente in  $G$  nach Verarbeitung von  $k + 1$  Spießkombinationen selbst einen vollständigen, bipartiten Graphen bildet.

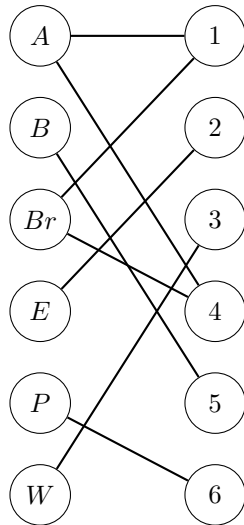
Sei  $K_i = (F_i, Z_i)$  die  $k + 1$ -te Spießkombination. Zu untersuchen ist die folgende Fallunterscheidung:

- (i) Sei  $D = (V_D, E_D)$  eine Zusammenhangskomponente in  $G$ . Sei  $F_i \cup Z_i = V_D$ . Da alle Knoten der Spießkombination sich mit allen Knoten von  $D$  decken, können keine Kanten nach Korollar 1 aus  $E$  entfernt werden, deshalb entsteht keine neue Zusammenhangskomponente, also ist jede Zusammenhangskomponente nach der Induktionsannahme ein vollständiger, bipartiter Graph.
- (ii) Sei  $D = (L_D \cup R_D = V_D, E_D)$  eine Zusammenhangskomponente in  $G$ . Sei  $F_i \cup Z_i \subsetneq V_D$ , also gehört  $F_i \cup Z_i$  nur zu einer Zusammenhangskomponente in  $G$ , aber deckt sich nicht mit allen Knoten.  $D$  ist laut Induktionsannahme selbst ein vollständiger, bipartiter Graph. Nach Korollar 1 werden alle Kanten zwischen allen  $x \in F_i$  und allen  $y \in R_D \setminus Z_i$ , sowie zwischen allen  $p \in Z_i$  und allen  $q \in L_D \setminus F_i$  entfernt. So entstehen zwei neue Zusammenhangskomponenten:  $C_1 = (L_1 \cup R_1, E_1)$  und  $C_2 = (L_2 \cup R_2, E_2)$ , o.B.d.A.  $L_1 \cup R_1 = F_i \cup Z_i$  und  $C_2 = V_D \setminus (F_i \cup Z_i)$ , die ebenfalls selbst vollständige, bipartite Graphen sind. Jede andere Zusammenhangskomponente in  $G$  ist nach der Induktionsannahme ein vollständiger, bipartiter Graph.
- (iii) Sei  $1 \leq t \leq n$  beliebig, aber fest. Seien  $C_1, C_2, \dots, C_t$  untereinander unterschiedliche Zusammenhangskomponenten in  $G$ . Gehöre  $F_i \cup Z_i$  zu mehreren Komponenten  $C_p, \dots, C_q$ . Dann gilt für jede Zusammenhangskomponente  $C_i$  entweder (i) oder (ii), abhängig davon, ob  $C_i$  vollständig zu  $F_i \cup Z_i$  gehört oder nur zum Teil. Das bedeutet, entweder entsteht keine neue Zusammenhangskomponente (i) oder  $C_i$  wird in zwei neue Zusammenhangskomponenten gespalten (ii).

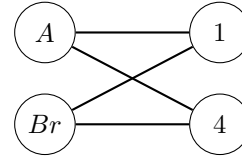
Da alle möglichen Fälle untersucht wurden, ist der Induktionsschritt vollzogen und die Behauptung gilt für jedes  $k \in \mathbb{N}$ .  $\square$

<sup>1</sup>Anup Rao. Lecture 6 Hall's Theorem. October 17, 2011. University of Washington. [Zugang 21.01.2021]  
<https://homes.cs.washington.edu/~anuprao/pubs/CSE599sExtremal/lecture6.pdf>

Abbildung 5: Abgebildet ist das Beispiel aus der Aufgabenstellung nach der Verarbeitung der allen  $m$  Spießkombinationen.



(a) Der Graph nach der Verarbeitung der allen Spießkombinationen



(b) Die übrige Zusammenhangskomponente mit mehr als 2 Knoten

**Lemma 3.** Sei  $C = (L_c \cup R_c, E_c)$  eine beliebige Zusammenhangskomponente in  $G$ . Dann existiert immer ein perfektes Matching zu  $C$ .

*Beweis.* Nach Lemma 2 ist jede Zusammenhangskomponente in  $G$  ein vollständiger, bipartiter Graph. Nach Satz von Hall existiert ein perfektes Matching, wenn für alle Teilmengen  $K \subseteq L_c$  gilt:  $|K| \leq |N(K)|$ . Die obere Behauptung kann für beliebig große Mächtigkeiten  $|K| = k \in \mathbb{N}$  durch die vollständige Induktion bewiesen werden.

**Induktionsanfang:** Für  $k = 1$  hat der einzelne Knoten  $x \in K \subseteq L_c$  die Kardinalität  $\Delta(x) = 1$ . Deshalb gilt:  $|K| = 1 \leq |N(x)| = 1$ . Damit stimmt die Behauptung für  $k = 1$  und der Induktionsanfang ist erledigt.

**Induktionsschritt:** Es gelte die Aussage für ein beliebiges  $k \in \mathbb{N}$ , also für eine Teilmenge  $K \subseteq L_c$ , die aus  $k$  Knoten besteht und in der jeder Knoten  $x \in K$  die Kardinalität  $\Delta(x) = k$  hat.

Es gelte also:  $|K| \leq |N(K)|$ .

Zu zeigen ist die Aussage für  $k + 1$ , also für eine Teilmenge  $K' \subseteq L_c$  der Mächtigkeit  $|K'| = k + 1$ :

$$|K'| \leq |N(K')|.$$

Wir verifizieren:

Jeder Knoten in  $C$  hat den Grad  $k + 1$ , also:  $|K'| = k + 1 \leq |N(K')| = (k + 1)^2 = k^2 + 2k + 1$ . Folglich stimmt die Behauptung für  $k + 1$ .

Der Induktionsschritt ist damit vollzogen und es wurde bewiesen, dass die Behauptung für beliebige Mächtigkeit von  $K$  gilt. Dadurch wurde auch bewiesen, dass es in einer Zusammenhangskomponente in  $G$  immer ein perfektes Matching gibt.  $\square$

Nach der Verarbeitung der allen Spießkombinationen entsteht ein Graph mit vielen Zusammenhangskomponenten (s. Abb. 5). An dieser Stelle muss man noch die Wunschliste  $W$  untersuchen, um die entsprechende Menge  $W'$  zu bestimmen. Dazu muss man die folgenden zwei Beobachtungen betrachten.

**Lemma 4.** Sei  $C = (L_c \cup R_c, E_c)$  eine Zusammenhangskomponente in  $G$ . Wenn gilt:  $\forall x \in L_c : x \in W$ , dann werden alle  $y \in R_c$  in  $W'$  hinzugefügt.

*Beweis.* Nach Axiom 1 besitzt jede Obstsorte genau einen einzigartigen Index. Die Zusammenhangskomponente  $C$  beschreibt nach Lemmata 2 und 3, dass jede Obstsorte  $p \in L_c$  jeden Index  $q \in R_c$  haben kann, weil  $C$  ein vollständiger, bipartiter Graph ist und ein perfektes Matching stets existiert.

Dadurch, dass  $\forall x \in L_c : x \in W$  gilt, ist ohne Bedeutung, welchen Index die jeweilige Obstsorte besitzt, da die Lösung des Problems eine Menge  $W'$  mit den Indizes der Obstsorten aus  $W$  sein soll. Dadurch, dass  $L_c \subseteq W$  gilt, gilt auch:  $R_c \subseteq W'$ .  $\square$

**Lemma 5.** Sei  $C = (L_c \cup R_c, E_c)$  eine Zusammenhangskomponente in  $G$ . Wenn gilt:  $\exists x \in L_c : x \notin W$  und  $\exists y \in L_c : y \in W$ , dann kann die Menge  $W'$  nicht eindeutig bestimmt werden.

*Beweis.* Nach Axiom 1 besitzt jede Obstsorte genau einen einzigartigen Index. Die Zusammenhangskomponente  $C$  beschreibt nach Lemmata 2 und 3, dass jede Obstsorte  $p \in L_c$  jeden Index  $q \in R_c$  haben kann, weil  $C$  ein vollständiger, bipartiter Graph ist und ein perfektes Matching stets existiert.

Angenommen,  $\exists r \in L_c : r \notin W$ . Dann ist es unmöglich, festzustellen, welcher Index aus  $R_c$  der Obstsorte  $r$  gehört. Also ist es auch unmöglich, festzustellen, welche Indizes in  $W'$  hinzugefügt werden sollen. Deshalb ist es unmöglich (unabhängig von allen anderen Zusammenhangskomponenten des Graphen  $G$ ), eine eindeutige Menge der Indizes der gewünschten Obstsorten festzulegen. Dadurch gibt es keine eindeutige Lösung zu diesem Problem für diese Eingabe.  $\square$

Direkt aus Lemma 4 ergibt sich das folgende Korollar. Man bedient sich dessen und des Lemmas 5, um das ganze Problem zu lösen, also: Ob die Menge  $W'$  eindeutig bestimmt werden kann.

**Korollar 2.** Seien  $C_1 = (L_1 \cup R_1, E_1), \dots, C_k = (L_k \cup R_k, E_k)$  alle Zusammenhangskomponenten in  $G$ , für jede  $i$ -te von denen gilt:  $\exists x \in L_i : x \in W$ . Falls für jede  $i$ -te von diesen Komponenten gilt:  $L_i \subseteq W$ , dann kann  $W'$  eindeutig und vollständig bestimmt werden.

Man stellt fest, dass man die Menge  $W$  untersuchen kann und wenn ein  $x \in W$  in  $G$  die Kardinalität  $\Delta(x) = 1$  besitzt, kann der einzelne Nachbar von  $x$  in  $W'$  hinzugefügt werden (Lemma 4). Im sonstigen Fall, also wenn  $\Delta(x) > 1$ , muss die ganze Zusammenhangskomponente  $C_x = (L_x \cup R_c, E_x)$ , zu der  $x$  gehört, untersucht werden, ob gilt:  $\forall p \in L_x : p \in W$  (Lemmata 4 und 5).

Man erstellt eine Liste  $\bar{W}$  der Länge  $n$ , in der die Zugehörigkeit einer Obstsorte zu  $W$  durch 1 oder 0 gekennzeichnet wird (s. Umsetzung). Außerdem erstellt wird eine Liste  $\bar{R}$  der Länge  $n$ , in der jede gewünschte Obstsorte  $x$  als 1 gekennzeichnet wird, falls der Knoten  $x$  in  $G$  bereits besucht wurde (s. Umsetzung).

Wenn man einen Knoten  $x \in W$  untersucht, deren Kardinalität  $\Delta(x) > 1$  ist, kann man die Liste der Nachbarknoten  $n(x)$  von  $x$  aufrufen. Da eine Zusammenhangskomponente selbst vollständig ist (Lemma 2), kann man die Liste der Nachbarknoten  $n(y)$  eines beliebigen Nachbarn  $y$  von  $x$  ( $y \in n(x)$ ) aufrufen. So kann man jeden Knoten  $z \in n(y)$  untersuchen, ob bei jedem  $z$  eine 1 in  $\bar{W}$  steht. Falls ja, wird  $z$  auch in  $\bar{R}$  markiert, sodass man denselben Vorgang bei einem anderen Knoten in dieser Komponente nicht wiederholen muss. Falls alle  $z$  zu  $W$  gehören, wird die ganze Liste  $n(x)$  in  $W'$  hinzugefügt. Sonst werden alle Knoten dieser Komponente gespeichert, insbesondere diese Obstsorten, die zu  $W$  nicht gehören.

Man wiederholt diesen Vorgang, bis alle gewünschten Obstsorten mit 1 in  $\bar{R}$  markiert werden.

Ausgegeben wird entweder die vollständige Menge  $W'$  oder eine Meldung über die jeweilige Zusammenhangskomponente, zu der Obstsorten gehören, die nicht gewünscht waren. Diese werden auch in der Ausgabe aufgezählt.

## 1.5 Prüfung auf Korrektheit der Eingabe

Am Ende des Teils 1.1 wurde bemerkt, dass die Korrektheit und Vollständigkeit der Lösung davon abhängt, ob alle Obstsorten in einer Eingabe Axiom 1 folgen.

Identifizieren wir zuerst die Probleme, die auftreten können. In den folgenden Überlegungen nehmen wir an, dass jede Spießkombination so gebildet wird, dass gilt:  $|F_i| = |Z_i|$ . (Falls man dies nicht angenommen hätte, wäre eine Eingabe schon an dieser Stelle falsch, da eine Obstsorte zwei Indizes oder zwei Obstsorten einen Index haben müssten. Außerdem ist dieser Fehler leicht herauszufinden, indem man beim Einlesen prüft, ob die beiden Mengen gleichmächtig sind.) Im Allgemeinen kommt es zu einem Widerspruch, wenn die Eingabe dem Axiom 1 nicht folgt. Das heißt, es können die folgenden Möglichkeiten auftreten:

(P1) In einer Eingabe existieren zwei Obstsorten:  $o(x, i)$  und  $o(y, i)$ , wobei  $x \neq y$ ,

(P2) In einer Eingabe existieren zwei Obstsorten:  $o(x, i)$  und  $o(x, j)$ , wobei  $i \neq j$ .

Untersuchen wir die Situation, in der die folgenden zwei Obstsorten existieren:  $o(x, i)$  und  $o(y, j)$ . Nehmen wir an dieser Stelle an, dass  $i = j$ . Betrachten wir dazu zwei Spießkombinationen:  $K_1 = (F_1, Z_1)$  und  $K_2 = (F_2, Z_2)$ . Es gelte:  $x \in F_1$  und entsprechend  $i \in Z_1$ .

- (F1) Falls  $y \in F_1$  und  $i = j$ , dann ist  $i$  bereits in  $Z_1$ . Sodass  $|F_1| = |Z_1|$  gilt, muss gelten:  $\exists o(z, k) : z \notin F_1 \wedge k \in Z_1$ . Dann muss zwar kein Widerspruch erfolgen, aber wir haben der Obstsorte einen Index zugewiesen, also kann an dieser Stelle die Beziehung zwischen  $z$  und  $k$  gar nicht festgestellt werden. Falls alle anderen Spießkombinationen widerspruchsfrei sind, wird  $z$  ein Index  $\ell \in B \setminus F_1$  zugewiesen.
- (F2) Falls  $y \notin F_1 \wedge y \in F_2 \wedge x \notin F_2 \wedge i = j$ , dann ist  $i$  bereits in  $Z_1$ . Dann muss für  $i$  auch gelten:  $i \in Z_2$ . Am Anfang ist der bipartite Graph  $G$  vollständig. Nach der Verarbeitung der Spießkombination  $K_1$  werden alle Kanten zwischen allen  $p \in Z_1$  und allen  $q \in A \setminus F_1$ , sowie alle Kanten zwischen allen  $p \in F_1$  und allen  $q \in B \setminus Z_1$  entfernt, darunter auch die Kante zwischen  $y$  und  $i$ . Nach der Verarbeitung von  $K_2$  wird auch die Kante zwischen  $x$  und  $i$  entfernt, da  $x \notin F_2$ . Der Knoten  $x$  hat dann eine Kardinalität um 1 kleiner als der Rest der Knoten auf dieser Komponente. Insbesondere: Wenn die Mengen  $F_1$  und  $F_2$  jeweils eine Mächtigkeit von 2 haben, hat  $x$  dann den Grad 0.

Bei der Untersuchung der Situation für (P2) geht man durch eine analoge Fallunterscheidung wie in (F1) und (F2).

Deshalb, um zu prüfen, ob die Eingabe Axiom 1 widerspricht, muss man nur untersuchen, ob die Kardinalität des jeden Knotens mit der Kardinalität eines der seinen Nachbarn nicht übereinstimmt. Dazu muss man beachten, dass die Zahl  $n$  in einigen Beispieldateien größer ist als die Anzahl der in Spießkombinationen und in der Wunschliste verwendeten Obstsorten und Indizes. In diesem Fall muss man die nicht genutzten Obstsorten und Indizes beim Einlesen entsprechend markieren und sie beim Prüfen auf Korrektheit der Eingabe überspringen. Mehr dazu in der Umsetzung.

## 1.6 Laufzeit

$n$  — die Anzahl der Obstsorten

$m$  — die Anzahl der Spießkombinationen

$w$  — die Anzahl der Wünsche (also  $|W|$ ), im worst-case  $w = n$

Da wir Bitmasken in unserem Programm verwenden, ist noch eine Konstante einzuführen:  $\beta$ , die für die  $\beta$ -Bit-Architektur eines Rechners<sup>2</sup> steht, auf dem das Programm ausgeführt wird. D.h., bei der 64-Bit-Architektur beträgt  $\beta = 64$ . Die bitweisen Operationen in C++ auf `bitset` werden in der Laufzeit von  $O(\frac{|k|}{\beta})$  ausgeführt, wobei  $|k|$  die Länge eines `bitset` ist. Eine Operation wird nicht auf einem einzelnen Bit ausgeführt, sondern es handelt sich um eine Operation auf einem integrierten Schaltkreis, deshalb stellt  $\beta$  die Größe des Datenwortes des Rechners dar. Außerdem muss die Länge eines `bitset` konstant sein, d.h., man muss schon im Programm eine feste Länge für alle Eingabegrößen eingeben. Diese feste Länge nennen wir  $N$  und setzen  $N = 26$ , da so viele Obstsorten das größte Beispiel auf der BWINF-Webseite umfasst.

- Einlesen:  $O(n(\frac{N}{\beta} + m \log n) + w(\log n + \log w))$  (worst-case)
  - Erstellung der Adjazenzmatrix  $M$ :  $O(n \cdot \frac{N}{\beta})$
  - Erstellung der Liste `used` (s. Umsetzung):  $O(n)$
  - Einlesen der Menge  $W$ :  $O(w \log w)$ <sup>3</sup>  
Implementierung von `set` in C++ als Rot-schwarz-Bäume<sup>4</sup>.
  - Einlesen der Spießkombinationen:  $O(m \cdot n \log n)$ <sup>3</sup> (worst-case)  
Im schlimmsten Fall enthält jede Spießkombination alle Obstsorten. Die logarithmische Laufzeit ist durch Einfügen in eine Menge verursacht.
  - Zuweisung der internen Indizes den Obstsorten (s. Umsetzung):  $O(n \log n)$ <sup>3</sup>  
Implementierung von `map` in C++ als Rot-schwarz-Bäume<sup>5</sup>

<sup>2</sup>[https://en.wikipedia.org/wiki/Word\\_\(computer\\_architecture\)](https://en.wikipedia.org/wiki/Word_(computer_architecture))

<sup>3</sup>Praktisch sollte man zu dieser Laufzeit noch den Aufwand vom Vergleichen von zwei Wörtern in der Menge hinzurechnen, aber die Längen der Wörter sind so klein, dass ich dies als eine vernachlässigbare Konstante betrachte.

<sup>4</sup><https://en.cppreference.com/w/cpp/container/set>

<sup>5</sup><https://en.cppreference.com/w/cpp/container/map>



- Umwandlung der gewünschten Obstsorten von Strings zu Integers:  $O(w(\log n + \log w))$   
Das Suchen in einer `map` hat logarithmische Laufzeit bezüglich der Anzahl der Obstsorten:  $O(\log n)^3$ . Das Einfügen in eine `set` hat logarithmische Laufzeit bezüglich der Anzahl der Wünsche:  $O(\log w)^3$ . Also gilt für die gesamte Laufzeit:  $O(w(\log n + \log w))$ .
- Umwandlung der Obstsorten in allen Spießkombinationen von Strings zu Integers:  $O(m \cdot n \log n)$   
In jeder Spießkombination können sich im worst-case alle  $n$  Obstsorten befinden. Das Suchen in einer `map` hat logarithmische Laufzeit bezüglich der Anzahl der Obstsorten:  $O(\log n)^3$ . Das Einfügen in eine `set` hat ebenfalls logarithmische Laufzeit bezüglich der Anzahl der Obstsorten:  $O(\log n)^3$ . Deshalb beträgt die Laufzeit für alle Obstsorten in einer Spießkombination höchstens:  $O(n \log n)$ .
- Die gesamte Laufzeit für diesen Teil (worst-case):  $O(n \cdot \frac{N}{\beta}) + O(n) + O(w \log w) + O(n \log n) + O(m \cdot n \log n) + O(w(\log n + \log w)) + O(m \cdot n \log n) = O(n \cdot \frac{N}{\beta} + n + w \log w + n \log n + m \cdot n \log n + w(\log n + \log w) + m \cdot n \log n) = O(n \cdot \frac{N}{\beta} + m \cdot n \log n + w(\log n + \log w)) = O(n(\frac{N}{\beta} + m \log n) + w(\log n + \log w))$   
Man darf nicht vergessen, dass die Laufzeit optimiert werden könnte, wenn die Eingabe nicht aus Wörtern bestünde, sondern aus Zahlen. Man könnte beispielsweise die Wörter früher in Zahlen umwandeln lassen.
- Verarbeitung der Spießkombinationen:  $O(n \cdot \frac{N}{\beta}(m + n))$  (worst-case)
  - Verarbeitung einer Spießkombination  $K = (F, Z)$ :  $O(n \cdot \frac{N}{\beta})$   
Man geht davon aus, dass eine Spießkombination im worst-case alle Obstsorten enthält.
    - \* Erstellung der Bitmaske  $bf$ :  $O(n)$
    - \* Erstellung der Bitmaske  $bn$ :  $O(|F|)$ , worst-case:  $O(n)$   
Die Operation hat eine lineare Laufzeit bezüglich der Anzahl der Elementen in einer Spießkombination. Eine Spießkombination kann im worst-case alle  $n$  Obstsorten beinhalten.
    - \* Erstellung der Bitmaske  $br$ :  $O(\frac{N}{\beta})$
    - \* Entfernen der Kanten:  $O(n \cdot \frac{N}{\beta})$   
Für jede Liste  $M_i$  wird geprüft, ob  $i$  sich in  $F$  befindet. Diese Operation kann in  $O(1)$  ausgeführt werden, indem wir durch die Menge  $F$  gleichzeitig iterieren, wie durch die Matrix  $M$  (s. Umsetzung). An jeder Liste  $M_i$  wird genau eine bitweise Operation durchgeführt.
    - \* Die gesamte Laufzeit für eine Spießkombination beträgt (worst-case):  
 $O(n) + O(n) + O(\frac{N}{\beta}) + O(n \cdot \frac{N}{\beta}) = O(n + n + \frac{N}{\beta} + n \cdot \frac{N}{\beta}) = O(n \cdot \frac{N}{\beta})$
  - Verarbeitung der allen Spießkombinationen entsprechend:  $O(m \cdot n \cdot \frac{N}{\beta})$
  - Kopieren der Adjazenzmatrix in `Graph G` (s. Umsetzung): average-case:  $O(n \cdot \delta \cdot \frac{N}{\beta})$ , worst-case:  $O(n^2 \cdot \frac{N}{\beta})$   
Für jede Liste  $M_i$  werden alle 1-en in  $G$  als Kanten vom Knoten  $i$  eingefügt. Dazu bediene ich mich der eingebauten Funktion `_Find_next()`, die jeweils das nächste 1-Bit in einem `bitset` findet. Jedoch ihre Laufzeit ist mir nicht bekannt. Ich gehe davon aus, dass dieser Vorgang ebenfalls in der Zeit von  $O(\frac{N}{\beta})$  abzuschließen ist. (Falls die Laufzeit viel schlechter wäre, könnte man die hier<sup>6</sup> beschriebene Idee anwenden, die mit einem logarithmischen Aufwand bezüglich der Länge der Bitmaske läuft.)  
Deshalb erfolgt die Iteration über eine Liste  $M_i$  in  $O(\delta \cdot \frac{N}{\beta})$ , wobei  $\delta$  die Anzahl der 1-en ist. Im schlimmsten Fall, wenn alle Spießkombinationen aus  $n$  Obstsorten bestehen, gilt:  $\delta = n$ , also gilt es:  $O(n \cdot \frac{N}{\beta})$ . Dennoch im allgemeinen Fall gilt:  $\delta \ll n$ . Den Vorgang muss man für alle  $n$  Obstsorten ausführen.  
Man könnte auch denken, dass das Kopieren unnötig ist. Falls man den Graphen nicht in eine Adjazenzliste-Form kopiert, muss man sowieso an einer Stelle die entsprechenden 1-en aus der Adjazenzmatrix ablesen.
  - Die gesamte Laufzeit für diesen Teil beträgt (worst-case):  
 $O(m \cdot n \cdot \frac{N}{\beta}) + O(n^2 \cdot \frac{N}{\beta}) = O(m \cdot n \cdot \frac{N}{\beta} + n^2 \cdot \frac{N}{\beta}) = O(n \cdot \frac{N}{\beta}(m + n))$

<sup>6</sup><https://stackoverflow.com/questions/58795338/find-next-array-index-with-true-value-c>

- Prüfung der Korrektheit der Eingabe:  $O(n)$ 
  - Für jeden Knoten im Graphen wird geprüft, ob er in `used` markiert ist:  $O(1)$  und ob seine Kardinalität mit der eines seinen Nachbarknoten übereinstimmt:  $O(1)$ . Der Zugriff auf einen der Nachbarknoten erfolgt auch in  $O(1)$ .  
Deshalb für alle Knoten im bipartiten Graphen gilt:  $O(n + n) = O(n)$
- Prüfung der Existenz einer Lösung:  $O(n \log n)$  (worst-case)
  - Erstellung von  $\bar{W}$ :  $O(n)$
  - Erstellung von  $\bar{R}$ :  $O(n)$
  - Prüfung der Wunschliste:  $O(w \log w)$   
Es wird geprüft, ob die Kardinalität des jeden Knotens 1 beträgt
    - \* Prüfung auf Kardinalität  $\Delta(x) = 1$ :  $O(1)$
    - \* Zugriff auf die Liste der Nachbarknoten in  $G$ :  $O(1)$
    - \* ggf. Einfügen in  $W'$ :  $O(\log w)$
    - \* ggf. Einfügen in `multi` (s. Umsetzung):  $O(\log w)$  (worst-case)  
Im schlimmsten Fall hat keiner der Knoten in der Wunschliste die Kardinalität von 1.
    - \* ggf. Markierung in  $\bar{W}$ :  $O(1)$
  - Iteration durch `multi`:  $O(n \log w)$  (worst-case)  
Die folgenden Operation werden nur dann ausgeführt, wenn die Komponente, zu der der iterierte Knoten gehört, noch nicht bearbeitet wurde, also ob dieser Knoten in  $\bar{R}$  markiert wurde. Wir können feststellen, dass diese Bedingung im schlimmsten Fall nur  $\frac{w}{2}$ -mal erfüllt werden kann. Alle Komponenten mit genau 1 Knoten aus  $A$  wurden bereits behandelt und in diesem Fall müssten alle Komponenten aus genau 2 Knoten aus  $A$  bestehen.  $\frac{w}{2}$  ist somit die maximale Anzahl an Zusammenhangskomponenten in  $G$ , die mehr als einen Knoten aus  $A$  besitzen.
    - \* Prüfung auf Markierung in  $\bar{R}$ :  $O(1)$
    - \* ggf. Zugriff auf die Liste der Nachbarn eines iterierten Knotens  $x$  ( $n(x)$ ):  $O(n)$   
Im schlimmsten Fall muss man auf alle Knoten aus  $B$  zugreifen.
    - \* ggf. Zugriff auf die Liste der Nachbarn eines Nachbarn  $y$  eines iterierten Knotens ( $n(y)$ ):  $O(n)$   
Im schlimmsten Fall muss man auf alle Knoten aus  $A$  zugreifen.
    - \* ggf. Iteration durch  $n(y)$ :  $O(n)$  (worst-case)  
In dieser Schleife wird geprüft, ob der iterierte Index  $i$  in  $\bar{W}$  markiert ist:  $O(1)$ , und dann wird  $i$  in  $\bar{R}$  markiert:  $O(1)$ . Wenn es einen Knoten gibt, der nicht gewünscht ist, aber sich auf der Komponente befindet, wird dies mit einer boolschen Variable `prob` markiert:  $O(1)$ . Im worst-case kann die Schleife  $n$ -mal iteriert werden, wenn es nur eine Zusammenhangskomponente in  $G$  gibt. Jedoch wird die äußere Schleife nur einmal iteriert, da alle gewünschten Obstsorten auf der Komponente als besucht in  $\bar{R}$  markiert werden.
    - \* ggf. Kopieren der Knoten aus dieser Komponente zu `problems` (s. Umsetzung):  $O(n)$  (worst-case)
    - \* ggf. Einfügen der Knoten aus dieser Komponente zu  $W'$ :  $O(n \log w)$  (worst-case)  
Das Einfügen in eine Menge hat eine logarithmische Laufzeit bezüglich  $w$ .
    - \* Um die gesamte Laufzeit für diesen Teil zu bestimmen, müssen wir bemerken, dass diese Laufzeit von der Anzahl der Knoten der Menge  $A$  auf allen Zusammenhangskomponenten abhängt, auf deren sich mind. eine gewünschte Obstsorte befindet. Durch die Markierung in  $\bar{R}$  wird jeder Knoten auf jeder von diesen Zusammenhangskomponenten nur einmal behandelt. Damit ergibt sich im worst-case die Laufzeit von  $O(n \log w)$ , indem die gewünschten Obstsorten auf allen Zusammenhangskomponenten in  $G$  verteilt sind.
  - Ausgabe für eine Eingabe, für die  $W'$  nicht eindeutig bestimmt werden kann:  $O(n \log n)$   
Es wird durch alle Komponenten iteriert, die mind. eine ungewünschte Obstsorte enthalten, und alle Knoten werden mit den Namen der Obstsorten aufgezählt. Deshalb muss jedes Mal die Suchfunktion in der `map` mit den Zuweisungen der internen Indizes der Obstsorten und den Obstsorten aufgerufen werden:  $O(\log n)^3$ . Im schlimmsten Fall muss in der Ausgabe durch alle Knoten in  $A$  iteriert werden, falls es nur eine Zusammenhangskomponente in  $G$  gibt.

- Die gesamte Laufzeit für diesen Teil beträgt (worst-case, nach oben geschätzt:  $n > w$ ):  $O(n) + O(n) + O(w \log w) + O(n \log w) + O(n \log n) = O(n + n + w \log w + n \log w + n \log n) \subset O(n \log n)$

Fassen wir die Laufzeit im worst-case zusammen. Also ein worst-case kann so ein Fall gelten, in dem alle  $m$  Spießkombinationen und die Wunschliste aus allen  $n$  Obstsorten bestehen.

- Einlesen:  $O(n(\frac{N}{\beta} + m \log n) + w(\log n + \log w))$
- Verarbeitung der Spießkombinationen:  $O(n \cdot \frac{N}{\beta}(m + n))$
- Prüfung der Korrektheit der Eingabe:  $O(n)$
- Prüfung der Existenz einer Lösung:  $O(n \log n)$

$$\begin{aligned} & O(n(\frac{N}{\beta} + m \log n) + w(\log n + \log w)) + O(n \cdot \frac{N}{\beta}(m + n)) + O(n) + O(n \log n) = \\ &= O(n(\frac{N}{\beta} + m \log n) + w(\log n + \log w) + n \cdot \frac{N}{\beta}(m + n) + n + n \log n) = \\ &= O(n(m \log n + m \frac{N}{\beta} + n \frac{N}{\beta}) + w(\log n + \log w) + n \log n) \end{aligned}$$

Nach der Abschätzung  $w \leq n$  ergibt sich:

$$O(n(m \log n + m \frac{N}{\beta} + n \frac{N}{\beta}) + n \log n) = O(n(m \log n + m \frac{N}{\beta} + n \frac{N}{\beta}))$$

Wir können bemerken, dass fast alle modernen Rechner auf einer mind. 32-Bit-Architektur basieren, das heißt, wir können  $\beta = 32$  setzen. Für die maximale Anzahl an Obstsorten, die in den BWINF-Beispielen auftreten, gilt dann:  $\frac{N}{\beta} = \frac{26}{32} < 1$ . In diesem Fall ist dieser Bruch ein vernachlässigter Faktor. Es ergibt sich im schlimmsten Fall:

$$O(n(m \log n + m + n))$$

## Literatur

- [1] T.H. Cormen u.a. *Introduction To Algorithms. Third edition.* Introduction to Algorithms. MIT Press, 2009. ISBN: 9780262533058.

## 2 Umsetzung

### 2.1 Klasse Solver

Die Matrix  $M$  wird als ein **vector** von **bitset** dargestellt. Dazu muss man erwähnen, dass ein **bitset** in C++ eine feste Länge besitzen muss. Dazu wurde die maximale Größe von  $n$  eingegeben, also 26. Das müsste ggf. im Program selbst umgestellt werden, falls man eine größere Datei einlesen möchte.

Die feste Länge ist auch der Grund dafür, dass die Bitmaske  $br$  im Teil 1.3 auf folgende Weise definiert wird:

$$br := \neg(bn) \wedge bf.$$

Im Fall, wenn man mit einer Bitmaske einer festen Größe operiert, würde eine einfache Negation der Bitmaske  $bn$  nicht hinreichend.

Die Obstsorten werden als Strings eingelesen, aber in der Methode `readFile()` wird jeder Obstsorte ein interner Index<sup>7</sup> zugewiesen und in weiteren Operationen im Programm werden die Obstsorten als einfache Integers behandelt. Es werden dazu zwei Maps festgelegt: `fruit2ID` und `ID2Fruit`, in denen die Obstsorten und die entsprechenden internen Indizes gespeichert sind.

Jede Spießkombination wird als `pair<set<int>, set<int>` dargestellt, also als ein Tupel entsprechend aus der Menge der internen Indizes der Obstsorten und der Menge der Indizes der Obstsorten (aus der Textdatei). Alle Spießkombinationen werden in einem **vector** gespeichert, der `infos` heißt.

In `wishes` werden als eine **set** von Integers die internen Indizes der gewünschten Obstsorten, also der Elemente der Menge  $W$ , gespeichert. Analog werden in `result`, also ebenfalls in eine **set** von Integers, die Indizes der gewünschten Obstsorten, also die Elemente der Menge  $W'$ , hinzugefügt.

Der **vector**, der `used` heißt, wird verwendet, um die benutzten internen Indizes der Obstsorten, wie auch die Indizes der Obstsorten zu markieren, die im Graphen verwendet werden, da es in einzigen Textdateien ein größeres  $n$  gibt als die Anzahl der in den Spießkombinationen und der Wunschliste verwendeten Obstsorten und Indizes. Diese Markierung zeigt sich bei der Prüfung auf korrekte Eingabe hilfreich.

Die Methode `analyzeInfo()` nimmt als Parameter eine Spießkombination. In der Methode werden die drei Bitmasken  $bn, br, bf$  erstellt. Um die Laufzeit zu optimieren, wird durch die Menge der internen Indizes der Obstsorten dieser Spießkombination `fruits` gleichzeitig mit den Bitmasken aus der Matrix iteriert. Da die Menge `fruits` vorsortiert ist, müssen wir nicht bei jeder Obstsorte in der Matrix prüfen, ob sie sich in `fruits` befindet, um die Entscheidung zu treffen, welche der beiden Bitmasken anzuwenden. Nachdem alle  $m$  Spießkombinationen verarbeitet wurden, werden in der Methode `analyzeAllInfos()` alle übrigen 1-Beziehungen in der Adjazenzmatrix als Kanten in den Graphen  $G$  der Klasse `Graph` kopiert. Die jeweiligen 1-en in `matrix` werden mithilfe der eingebauten Funktion `_Find_next()` gefunden.

Die Methode `checkCoherence()` prüft, ob die Eingabe dem Axiom 1 folgt. Dennoch es gibt Beispiele, in denen  $n$  größer ist als die Anzahl der verwendeten Elementen in den Spießkombinationen und der Wunschliste. Bei der Zuweisung der internen Indizes jeder Obstsorte werden die verwendeten Obstsorten und Indizes in `used` markiert. Diesen Vorteil können wir nutzen, um die Prüfung der Kardinalität des jeden Knotens in  $G$  zu überprüfen: Es wird geprüft, ob zwei Nachbarn dieselbe Kardinalität besitzen. Wir nehmen dazu die Methode `getFirstNeighbor()` der Klasse `Graph` und vergleichen die Kardinalitäten für jeden Knoten und seinen ersten (beliebigen) Nachbarn. Alle Knoten, die in `used` nicht markiert wurden, werden übersprungen. Falls bei mindestens einem Knoten die Kardinalitäten nicht übereinstimmen, wird `false` ausgegeben und die Eingabe im gegebenen Beispiel ist fehlerhaft. Sonst wird `true` ausgegeben.

Nachdem die Korrektheit der Eingabe geprüft wurde, kann festgestellt werden, ob  $W'$  eindeutig bestimmt werden kann. Dazu dient die Methode `checkResult()`.

Es werden ein **vector** `todo` der Länge  $n$ , ein **vector** `ready` der Länge  $n$  und ein **set** `multip` erstellt. `todo` ist die Liste  $\bar{W}$ . `ready` ist die Liste  $\bar{R}$ .

Es wird zunächst über die Menge der Wünsche `wishes` iteriert. Falls ein Knoten  $x$  (der interne Index einer Obstsorte) in  $G$  die Kardinalität 1 besitzt, wird sein einzelner Nachbar in `result` hinzugefügt. Im

---

<sup>7</sup>Nummerierung ab 0.

sonstigen Fall wird **x** in **mult** hinzugefügt und die Stelle **x** in **todo** wird mit 1 markiert.

Dann wird geprüft, ob die Menge **mult** überhaupt irgendwelche Elemente enthält. Falls nicht, gibt die ganze Funktion an dieser Stelle **true** zurück.

Somit wird eine boolsche Variable **solv** erstellt, die für die Existenz einer Lösung steht. Am Anfang nimmt sie **true** als Wert. Dazu wird auch eine Liste von Mengen **problems** erstellt, die dazu da ist, die Obstsorten einer Zusammenhangskomponente, die eine nicht gewünschte Obstsorte enthält, zu speichern.

Danach wird über die Menge **mult** iteriert. Für jedes Element **x** aus dieser Menge wird eine boolsche Variable **prob** erstellt, die anzeigt, ob mindestens eine Obstsorte zu der Komponente gehört, die nicht gewünscht ist. Am Anfang hat sie den Wert **false**.

Es wird zunächst geprüft, ob an der Stelle **x** in **ready** eine 0 steht, d.h., ob der Knoten zu einer Zusammenhangskomponente gehört, die noch nicht bearbeitet wurde. Falls ja, dann werden zwei Listen **setB** und **setA** erstellt, die der Liste der Nachbarn von **x** und der Liste der Nachbarn von einem Nachbarn von **x** entsprechen. Es wird durch die Menge **setA** iteriert.

Falls an der Stelle eines internen Index einer Obstsorte in **todo** keine 1 steht, wird **solv = false** und **prob = true** gesetzt. Dies bedeutet, es gibt eine nicht gewünschte Obstsorte in der Komponente.

Sonst wird die Stelle des internen Index dieser Obstsorte in **ready** mit 1 markiert.

Danach wird geprüft, ob **prob == true**. Falls **prob == false** gilt, wird die Menge **setB** in **result** hinzugefügt. Sonst, wird die Menge **setA** in **problems** hinzugefügt, um sie danach als Nachricht für den Nutzer vorzustellen, dass die Menge  $W'$  aufgrund von bestimmten Obstsorten nicht eindeutig festgelegt werden kann.

Am Ende, nach der Schleife über **mult**, wird geprüft, ob es eine Lösung zur Aufgabe für eine Eingabe gibt: Es wird geprüft, ob **solv == true**. Falls ja, dann wird **true** zurückgegeben. Sonst wird eine Meldung ausgegeben, die alle Zusammenhangskomponenten beinhaltet, die eine nicht gewünschte Obstsorte enthalten, und die nicht gewünschten Obstsorten werden ebenfalls angezeigt.

## 2.2 Klasse Graph

Diese Klasse ist grundsätzlich aus Übersichtlichkeits- sowie aus Vereinfachungsgründen entstanden. Theoretisch könnte man die enthaltenen Methoden in der Klasse **Solver** speichern.

Der Graph ist als eine Adjazenzliste aus **vector** von **vector** von Integers gespeichert. Der Konstruktor nimmt zwei Parameter: die Größen der beiden Partitionen im bipartiten Graphen, obwohl sie in der Aufgabe gleich groß sind.

Zu den verfügbaren Methoden zählen: **addEdge()**, die eine ungerichtete Kante zwischen zwei Knoten einfügt; **getFirstNeighbor()**, die den ersten Nachbarn eines Knotens zurückgibt; **deg()**, die die Kardinalität eines Knotens zurückgibt; **getNeighbors()**, die den **vector**, also die Adjazenzliste eines Knotens zurückgibt. Außerdem gibt es ein paar Methoden, die zum Debugging dienen.

### 3 Beispiele

#### 3.1 Beispiel 0 (Aufgabenstellung — Teil a)

Textdatei: `spiesse0.txt`

Apfel, Brombeere, Weintraube

1, 3, 4

Auf Papier kann man das Beispiel auf folgende Weise lösen. Am Anfang weiß man nichts über die Obstsorten in  $A$  und die Indizes in  $B$ . Wir zeichnen deshalb einen vollständigen, bipartiten Graphen  $G$  (Abb. 7a). Jede Kante steht für eine mögliche Zuweisung eines Index einer Obstsorte.

Wir analysieren die 1. Spießkombination:  $F_1 = \{\text{Apfel, Banane, Brombeere}\}, Z_1 = \{1, 4, 5\}$ .

Wir stellen fest, dass Apfel, Banane und Brombeere jeweils keine Indizes 2, 3, 5 haben können, weil jeder dieser Obstsorten ein Index aus der Menge  $Z_1$  zugewiesen wird. Gleichzeitig merken wir, dass Erdbeere, Pflaume und Weintraube jeweils keinen der Indizes 1, 4, 5 besitzen können, weil diese ausschließlich den Obstsorten aus der Menge  $F_1$  zugewiesen werden. Somit stellen wir fest, dass die Anzahl der möglichen Zuweisungen schrumpft. Deshalb dürfen wir alle Kanten in  $G$  zwischen allen  $x \in F_1$  und allen  $y \in B \setminus Z_1$ , sowie zwischen allen  $p \in A \setminus F_1$  und allen  $q \in Z_1$  entfernen (Abb. 7b).

Wir analysieren die 2. Spießkombination:  $F_2 = \{\text{Banane, Pflaume, Weintraube}\}, Z_2 = \{3, 5, 6\}$ .

Wir stellen fest, dass Banane, Pflaume, Weintraube jeweils keinen der Indizes 1, 2, 4 haben können, weil jeder dieser Obstsorten ein Index aus der Menge  $Z_2$  zugewiesen wird. Entfernen wir alle Kanten in  $G$  zwischen allen  $x \in F_2$  und allen  $y \in B \setminus Z_2$ . Unter anderem wurden die Kanten (Banane, 1) und (Banane, 4) entfernt. Wir stellen fest, dass die Kardinlität des Knotens „Banane“ 1 beträgt — er ist nur mit dem Knoten 5 verbunden. Ebenfalls ist der Knoten 5 nur mit dem Knoten „Banane“ verbunden. Dies bedeutet, dass es nur eine einzige Möglichkeit gibt, diesen Knoten mit einem Index zu verbinden. Somit wurde der Index von Banane gefunden. Wir kennen schon eine Obstsorte:  $o(\text{Banane}, 5)$ .

Wir stellen auch fest, dass Apfel, Banane und Erdbeere jeweils keinen der Indizes 3, 5, 6 haben können, da sie nur den Obstsorten aus  $F_2$  zugewiesen werden dürfen. Insbesondere wissen wir schon, dass 5 mit Banane verbunden wird. Deshalb dürfen wir alle Kanten in  $G$  zwischen allen  $p \in A \setminus F_2$  und allen  $q \in Z_2$  entfernen. Wir bemerken, dass die Kardinlität des Knotens „Erdbeere“ nun 1 beträgt, der nur mit dem Knoten 2 verbunden ist. Ebenfalls ist der Knoten 2 mit keinem anderen verbunden. So legen wir fest:  $o(\text{Erdbeere}, 2)$ . Auf der Abbildung 7c wird der Graph  $G$  nach der Verarbeitung der 2. Spießkombination dargestellt. Die fetten Kanten zeigen an, dass die zwei Endknoten bereits verbunden sind, also, dass diesen Obstsorten ihre Indizes zugewiesen wurden.

Wir analysieren die 3. Spießkombination:  $F_3 = \{\text{Apfel, Brombeere, Erdbeere}\}, Z_3 = \{1, 2, 4\}$ .

An dieser Stelle wurde die Zuweisung für Erdbeere bereits gefunden. Die Knoten „Apfel“ und „Brombeere“ besitzen keine Kanten mehr als die Kanten, die sie jeweils mit 1 und 4 verbinden. Ebenfalls wurde der Index 5 Banane zugewiesen. Die Knoten „Pflaume“ und „Weintraube“ sind jeweils nur mit 3 und 6 verbunden. So können wir keine der übrigen Kanten zwischen irgendwelchen zwei Knoten in  $G$  entfernen.

Wir analysieren die 4. Spießkombination:  $F_4 = \{\text{Erdbeere, Pflaume}\}, Z_4 = \{2, 6\}$ .

In der Menge  $F_4$  tritt Erdbeere auf, deren Index bereits gefunden wurde. Somit wissen wir, dass der Index von Pflaume 6 sein muss. So entdecken wir eine neue Zuweisung:  $o(\text{Pflaume}, 6)$ . Wir können deshalb alle übrigen Kanten zwischen Pflaume und allen anderen Knoten entfernen. So bleibt der Knoten „Weintraube“ mit nur einer Kante übrig. Der einzelne Nachbar von diesem Knoten ist 3. So entdecken wir wieder eine neue Zuweisung:  $o(\text{Weintraube}, 3)$ .

Wir stellen auch fest, dass keine Kanten mehr entfernt werden können. Es bleibt immer noch ein Paar von Indizes und ein Paar von Obstsorten ohne eindeutige Zuweisung.

An dieser Stelle schauen wir die Wunschliste an:  $\{\text{Apfel, Brombeere, Weintraube}\}$ .

Der Index von Weintraube ist erfolgreich gefunden, aber die Indizes der übrigen Obstsorten nicht. Allerdings soll die Lösung der Aufgabe eine Menge an Indizes der gewünschten Obstsorten sein — es müssen keine konkreten Zuweisungen ausgegeben werden. Dies wurde erfolgreich gefunden, da die Indizes 1 und 4 nur Apfel oder Brombeere gehören können, weil keine anderen Kanten aus den Knoten 1 und 4 füh-

ren. Auf der Abbildung 7d wurden alle gefundenen Zuweisungen durch fette Kanten dargestellt und alle Wünsche mit ihren Indizes wurden entsprechend grün und blau markiert.

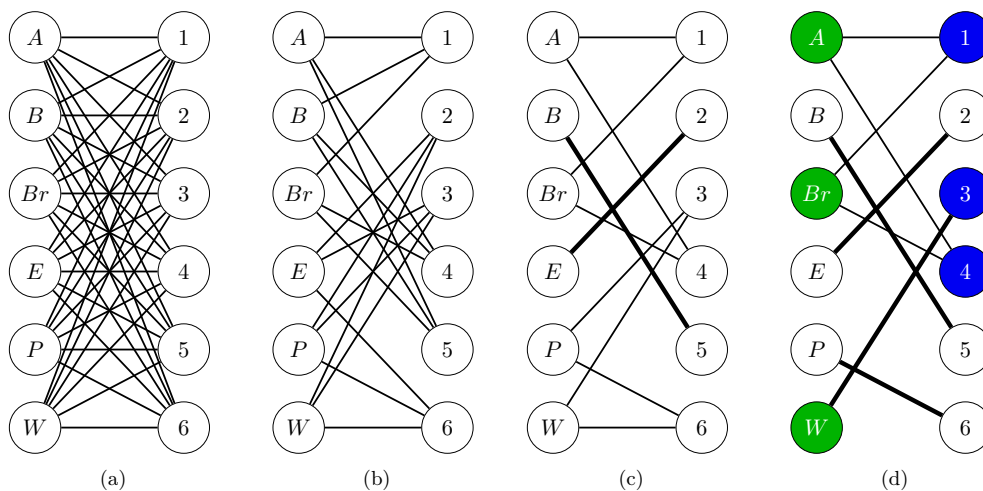


Abbildung 7

### 3.2 Beispiel 1 (BWINF)

Textdatei: spiesse1.txt

Wünsche: Clementine, Erdbeere, Grapefruit, Himbeere, Johannisbeere

1, 2, 4, 5, 7

### 3.3 Beispiel 2 (BWINF)

Textdatei: spiesse2.txt

Wünsche: Apfel, Banane, Clementine, Himbeere, Kiwi, Litschi

1, 5, 6, 7, 10, 11

### 3.4 Beispiel 3 (BWINF)

Textdatei: spiesse3.txt

Wünsche: Clementine, Erdbeere, Feige, Himbeere, Ingwer, Kiwi, Litschi

Dieses Beispiel ist unlösbar.

Für die folgenden Obstsorten konnte keine eindeutige Zuweisung gefunden werden.

Komponente: Grapefruit Litschi

--> Nicht auf der Wunschliste: Grapefruit

### 3.5 Beispiel 4 (BWINF)

Textdatei: spiesse4.txt

Wünsche: Apfel, Feige, Grapefruit, Ingwer, Kiwi, Nektarine, Orange, Pflaume

2, 6, 7, 8, 9, 12, 13, 14

### 3.6 Beispiel 5 (BWINF)

Textdatei: spiesse5.txt

Wünsche: Apfel, Banane, Clementine, Dattel, Grapefruit, Himbeere, Mango, Nektarine, Orange, Pflaume, Quitte, Sauerkirsche, Tamarinde

1, 2, 3, 4, 5, 6, 9, 10, 12, 14, 16, 19, 20

### 3.7 Beispiel 6 (BWINF)

Textdatei: spiesse6.txt

Wünsche: Clementine, Erdbeere, Himbeere, Orange, Quitte, Rosine, Ugli, Vogelbeere

4, 6, 7, 10, 11, 15, 18, 20

### 3.8 Beispiel 7 (BWINF)

Textdatei: spiesse7.txt

Wünsche: Apfel, Clementine, Dattel, Grapefruit, Mango, Sauerkirsche, Tamarinde, Ugli, Vogelbeere, Xenia, Yuzu, Zitrone

Dieses Beispiel ist unlösbar.

Für die folgenden Obstsorten konnte keine eindeutige Zuweisung gefunden werden.

Komponente: Apfel Grapefruit Litschi Xenia

--> Nicht auf der Wunschliste: Litschi

Komponente: Banane Ugli

--> Nicht auf der Wunschliste: Banane

### 3.9 Beispiel 8

Textdatei: spiesse8.txt

Besonderheit: Ein Beispiel für den Fall  $(P1) \rightarrow (F2)$ , s. Teil 1.5.

5

Apfel Erdbeere Banane

2

2 3

Banane Clementine

2 4

Dattel Erdbeere

Wünsche: Apfel, Erdbeere, Banane

Error: Es gibt Fehler in der Eingabedatei.

Die erste Spießkombination legt fest, dass nur Banane und Clementine die Indizes 2 und 3 besitzen dürfen. Allerdings sollen die Indizes 2 und 4 nach der zweiten Spießkombination den Obstsorten Dattel und Erdbeere gehören. Der Index 2 darf keinen zwei unterschiedlichen Obstsorten gehören. Deshalb kommt es zu einem Widerspruch — das Axiom 1 ist verletzt.



### 3.10 Beispiel 9

Textdatei: `spiesse9.txt`

Besonderheit: Ein Beispiel für den Fall  $(P2) \rightarrow (F2)$ , s. Teil 1.5.

5

Apfel Erdbeere Banane

2

2 3

Banane Clementine

4 5

Banane Dattel

Wünsche: Apfel, Erdbeere, Banane

Error: Es gibt Fehler in der Eingabedatei.

Die erste Spießkombination legt fest, dass nur Banane und Clementine die Indizes 2 und 3 besitzen dürfen. Allerdings sollen die Indizes 4 und 5 nach der zweiten Spießkombination den Obstsorten Banane und Dattel gehören. Die Obstsorte Banane darf keine zwei unterschiedliche Indizes besitzen. Deshalb kommt es zu einem Widerspruch — das Axiom 1 ist verletzt.

### 3.11 Beispiel 10

Textdatei: `spiesse10.txt`

Besonderheit: ein worst-case, in dem alle  $m = 28$  Spießkombinationen alle  $n = 26$  Obstsorten beinhalten. Die Wunschliste beinhaltet alle  $n$  Obstsorten.

Wünsche: Apfel, Banane, Clementine, Dattel, Erdbeere, Feige, Grapefruit, Himbeere, Ingwer, Johannisbeere, Kiwi, Litschi, Mango, Nektarine, Orange, Pflaume, Quitte, Rosine, Sauerkirsche, Tamarinde, Ugli, Vogelbeere, Weintraube, Xenia, Yuzu, Zitrone

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26

### 3.12 Beispiel 11

Textdatei: `spiesse11.txt`

Besonderheit: ein worst-case, in dem alle  $m = 28$  Spießkombinationen alle  $n = 26$  Obstsorten beinhalten. Die Wunschliste beinhaltet 24 Obstsorten.

Wünsche: Apfel, Banane, Clementine, Dattel, Feige, Grapefruit, Himbeere, Ingwer, Johannisbeere, Kiwi, Litschi, Mango, Nektarine, Orange, Pflaume, Rosine, Sauerkirsche, Tamarinde, Ugli, Vogelbeere, Weintraube, Xenia, Yuzu, Zitrone

Dieses Beispiel ist unlösbar.

Für die folgenden Obstsorten konnte keine eindeutige Zuweisung gefunden werden.

Komponente: Apfel Banane Clementine Dattel Erdbeere Feige Grapefruit Himbeere Ingwer  
Johannisbeere Kiwi Litschi Mango Nektarine Orange Pflaume Quitte Rosine Sauerkirsche  
Tamarinde Ugli Vogelbeere Weintraube Xenia Yuzu Zitrone

--> Nicht auf der Wunschliste: Erdbeere Quitte

### 3.13 Beispiel 12

Textdatei: spiesse12.txt

Besonderheit: Es entstehen  $\frac{n}{2}$  Zusammenhangskomponenten in  $G$ . Keiner Obstsorte kann ein Index eindeutig zugeordnet werden.

```
8
Feige Dattel Clementine Grapefruit
4
5 1 3 2 8 6
Dattel Banane Feige Erdbeere Clementine Himbeere
1 5 8 6
Himbeere Dattel Feige Banane
3 2 7 4
Apfel Clementine Erdbeere Grapefruit
8 7 1 4
Banane Apfel Grapefruit Dattel
```

Wünsche: Clementine, Dattel, Feige, Grapefruit

Dieses Beispiel ist unlösbar.

Für die folgenden Obstsorten konnte keine eindeutige Zuweisung gefunden werden.

Komponente: Clementine Erdbeere

--> Nicht auf der Wunschliste: Erdbeere

Komponente: Banane Dattel

--> Nicht auf der Wunschliste: Banane

Komponente: Feige Himbeere

--> Nicht auf der Wunschliste: Himbeere

Komponente: Apfel Grapefruit

--> Nicht auf der Wunschliste: Apfel

### 3.14 Beispiel 13

Textdatei: spiesse13.txt

Besonderheit: Es gibt  $m = n$  Spießkombinationen. Die Mächtigkeiten der Mengen dieser Spießkombinationen entsprechen:  $n, n-1, \dots, 1$ . Die Spießkombinationen werden aufeinander aufgebaut.

```
8
Apfel Himbeere Grapefruit Banane
8
5 1 3 2 4 8 6 7
Dattel Grapefruit Banane Feige Erdbeere Clementine Apfel Himbeere
1 5 4 6 8 2 7
Himbeere Dattel Feige Banane Apfel Grapefruit Clementine
4 8 2 5 1 7
Apfel Dattel Clementine Himbeere Banane Grapefruit
8 7 2 4 1
Clementine Banane Apfel Grapefruit Dattel
1 4 2 7
Grapefruit Clementine Dattel Apfel
7 2 1
Dattel Grapefruit Clementine
2 7
Clementine Grapefruit
2
Clementine
```

Wünsche: Apfel, Himbeere, Grapefruit, Banane

4, 5, 7, 8

### 3.15 Beispiel 14

Textdatei: `spiesse14.txt`

Besonderheit: Ein Beispiel für den Fall  $(P1) \rightarrow (F2)$ , s. Teil 1.5.

```
6
Apfel Clementine Feige
2
1 4 5
Apfel Banane Dattel
4 5 6
Dattel Erdbeere Feige
```

Wünsche:

Die erste Spießkombination legt fest, dass die Obstsorten Apfel, Banane, Dattel jeweils einen der Indizes  $\{1, 4, 5\}$  haben können. Dennoch die zweite Spießkombination legt im Widerspruch zur ersten fest, dass die Indizes  $\{4, 5, 6\}$  jeweils einer der Obstsorten Dattel, Erdbeere, Feige gehören können. In diesem Fall decken sich 4 und Dattel in beiden Spießkombinationen, aber 5 kann keiner Obstsorte zugeordnet werden.

### 3.16 Beispiel 15

Textdatei: `spiesse15.txt`

Besonderheit: Die obere Schranke an Obstsorten ist sehr groß im Vergleich zu der Anzahl der in den Spießkombinationen und in der Wunschliste erwähnten Obstsorten. Dieses Beispiel betont, dass mein Programm problemlos so eine große obere Schranke behnadeln kann. Außerdem zeigt sich hier der Vorteil der Verwendung von Bitmasken im Vergleich zu einer „Standard“-Adjazenzmatrix.

```
26
Apfel Dattel Feige
3
1 2 4
Apfel Banane Dattel
2 3 6
Dattel Erdbeere Feige
1 6
Banane Erdbeere
```

Wünsche:

## 4 Quellcode

```

1 //diese Methode liest Informationen aus der Datei ein
void Solver::readFile(string path) {
3     fstream file;
    file.open(path, ios::in);
5     if (file.is_open()) {

6         //Anzahl der Obstsorten
        file >> n;

7         //Erstellung der Adjazenzmatrix
        for (int i = 0; i < n + 1; i++) {
11             bitset<MAXN> b = bitset<MAXN>().set();
13             matrix.pb(b);
        }

15         //Erstellung der Liste der im Programm verwendeten internen Indizes
17         used = vector<int>(n + n);

19         string data;
        getline(file, data);
21         getline(file, data);
        istringstream iss(data);

23         //die gewünschten Obstsorten werden erstmal als Strings gespeichert
25         vector<string> wishes_words{istream_iterator<string>{iss}, istream_iterator<string>{}};

27         //die Menge der allen Obstsorten als Strings
        set<string> all_fruits(wishes_words.begin(), wishes_words.end());

29         //Anzahl der Spiesskombinationen
31         file >> m;
        getline(file, data);

33         //Spiesskombinationen werden erstmal als ein Menge von Zahlen und
35         // eine Menge von Strings gespeichert
        vector<pair<set<int>, vector<string>>> tempInfos;

37         for (int i = 0; i < m; i++) {
39             getline(file, data);
            istringstream iss(data);
41             vector<string> words{istream_iterator<string>{iss}, istream_iterator<string>{}};

43             //die Menge der Indizes der jeweiligen Spiesskombination wird
            // erstellt
45             set<int> currNum;
            for (auto x: words) {
47                 currNum.insert(stoi(x));

49                 //der interne Index eines Index einer Obstsorte wird
                // als verwendet markiert
51                 used[stoi(x)+n-1] = true;
            }

53             getline(file, data);
            istringstream issss(data);

55             //die Menge der Obstsorten der jeweiligen Spiesskombination wird
            // erstmal als Strings erstellt
57             vector<string> currFruits{istream_iterator<string>{iss}, istream_iterator<string>{}};

59             //alle Obstsorten werden zu einer gemeinsamen Menge hinzugefügt
            for (auto x: currFruits)
61                 all_fruits.insert(x);

63             //falls die beiden Menge einer Spiesskombination nicht gleichmächtig sind
            if (currNum.size() != currFruits.size()) {
65                 cerr << "Error: Es gibt Fehler in der Eingabedatei.\n";
67                 exit(0);
69             }

71             //eine Spiesskombination als ein Menge von Zahlen und

```

```

73     // eine Menge von Strings wird gespeichert
    tempInfos.pb({currNum, currFruits});
75 }
77 //Zuweisung der internen Indizes 0..(n-1) jeder Obstsorte
    int it = 0;
79     for (auto x: all_fruits) {
        ID2Fruit[it] = x;
        fruit2ID[x] = it;
81
        //der interne Index einer Obstsorte wird als verwendet merkiert
83         used[it] = true;
        it++;
85     }
87     file.close();
89     //die gewuenschten Obstsorten werden als Indizes gespeichert
    for (auto x: wishes_words)
91         wishes.insert(fruit2ID.find(x)->second);
93     //die Spiesskombinationen als Mengen der Indizes der Obstsorten
    // einer Spiesskombination und Mengen der internen Indizes der Obstsorten
95     for (auto x: tempInfos) {
        set<int> currFruits;
97         for (auto y: x.second)
            currFruits.insert(fruit2ID.find(y)->second);
99         infos.pb({x.first, currFruits});
    }
101 }
103 //falls die Textdatei nicht geoeffnet werden kann
    else {
105         cerr << "Error: File could not be opened. Abort.\n";
        exit(0);
    }
107 }

109 //diese Methode bearbeitet die Informationen aus einer Spiesskombination
void Solver::analyzeInfo(pair<set<int>, set<int>> info) {
111     //fruits -- die Menge der Obstsorten der Spiesskombination
    //nums -- die Menge der Indizes der Obstsorten der Spiesskombination
113     set<int> nums = info.first, fruits = info.second;

115     bitset<MAXN> mask_nums = 0, mask_full = 0, mask_rev = 0;

117     //eine Bitmaske mit 1-en auf allen n Stellen
    for (int i = 0; i < n; i++) mask_full ^= (1 << i);
119
    //die Bitmaske fuer die Obstsorten aus der Menge
121     for (auto x: nums) mask_nums ^= (1 << (x-1));

123     //die Bitmaske fuer die Obstsorten ausserhalb der Menge
    mask_rev = ~(mask_nums) & mask_full;
125
    //es wird ueber die Menge der Obstsorten der Spiesskombination
127     // iteriert
    auto it = fruits.begin();
129
    //es wird ueber die Obstsorten in der Adjazenzmatrix iteriert
131     for (int i = 0; i < n; i++) {
        //falls eine Obstsorte zur Menge der Obstsorten der
133         // Spiesskombination gehoert
        if (it != fruits.end() && *it == i) {
135             matrix[i] &= mask_nums;
            it++;
137         }
        //falls eine Obstsorte ausserhalb der Menge der Obstsorten
139         // der Spiesskombination liegt
        else
141             matrix[i] &= mask_rev;
    }
143 }

```

```

145 //diese Methode bearbeitet alle Informationen aus allen Spiesskombinationen
void Solver::analyzeAllInfos(){
147 //alle Spiesskombinationen werden bearbeitet
    for (auto x: infos)
149         analyzeInfo(x);

151 //alle noch uebrigen Kanten werden in den Graphen hinzugefuegt
    for (auto x: ID2Fruit) {
153         for (int i = matrix[x.first]._Find_next(-1); i < MAXN; i = matrix[x.first]._Find_next(i))
            G.addEdge(x.first, i);
155     }
}

157 //diese Methode prueft, ob die eingegeben Informationen ueber den Graphen
159 // keinen Widerspruch ergeben
bool Solver::checkCoherence(){
161 //falls zwei im Programm verwendete Knoten auf einer Komponente
// unterschiedliche Kardinalitaeten haben, ist die Eingabe falsch
163 for (int i = 0; i < int(used.size()); i++)
    if (used[i]){
165         //falls ein Knoten die Kardinalitaet von 0 besitzt
        if (G.deg(i) == 0)
167             return false;

169         //wir nehmen den ersten Nachbarn von i
        int neigh = G.getFirstNeighbor(i);
171
        //wir vergleichen die Kardinalitaeten
173         if (G.deg(neigh) != G.deg(i))
            return false;
175     }

177 return true;
}

179 //diese Methode prueft, ob eine eindeutige Loesung exisitiert
181 bool Solver::checkResult(){
//die Menge der Obstsorten (Knoten), die einen Grad von
183 // mehr als 1 haben
    set<int> multip;

185
//die Liste der Knoten, deren Komponente geprueft werden muss
187 vector<int> ready(n);

189 //die Liste der Knoten der gewuenschten Obstsorten
    vector<int> todo(n);
191
//es wird durch die Menge der Wuensche iteriert
193 for (auto x: wishes) {
//falls der Grad des Knotens der gewuenschten Obstsorte
// gleich 1 ist
195     if (G.deg(x) == 1) {
        int neigh = G.getFirstNeighbor(x);
197
        //der einzige Nachabr wird in die Ergebnismenge hinzugefuegt
        result.insert(neigh - n + 1);
201     }
//falls der Grad des Knotens der gewuenschten Obstsorte
203 // mehr als 1 betraegt
    else {
205         //der gewuenschte Knoten wird in die Menge
        // fuer die Untersuchung der Komponente hinzugefuegt
207         multip.insert(x);

209         //die Obstsorte wird in die To-do-Liste
        // fuer die Untersuchung der Komponente hinzugefuegt
211         todo[x] = 1;
    }
213 }

215 //falls alle gewuenschten Obstsorten einen Grad von 1 haben
    if (multip.size() == 0)
217         return true;

```

```

219 //diese Variable zeigt an, ob eine Loesung fuer die Eingabe existiert
    bool solv = true;
221
    //eine Liste mit Mengen der Obstsorten der Komponenten,
    // zu deren Knoten gehoeren, die nicht auf der Wunschliste stehen
    vector<set<int>> problems;
225
    //es wird durch die Menge der Knoten der gewuenschten Obstsorten iteriert,
    // deren Grad groesser als 1 betraegt
    for (auto x: multip) {
229 //diese Variable zeigt an, ob mind. ein Knoten zur Komponente
        // gehoert, aber nicht gewuenscht ist
        bool prob = false;
231
        //falls die Komponente eines Knotens noch nicht geprueft wurde
        if (!ready[x]) {
235 //die Menge der Knoten der Indizes der Obstsorten der jeweiliger Komponente
            vector<int> setB = G.getNeighbors(x);
237 //die Menge der Knoten der Obstsorten der jeweiliger Komponente
            vector<int> setA = G.getNeighbors(setB[0]);
239
            //es wird durch die Knoten der Obstsorten dieser Komponente
            // iteriert
            for (auto y: setA) {
243 //falls die Obstsorte x nicht gewuenscht ist,
                // gibt es keine eindeutige Loesung fuer die Eingabe
                if (!todo[y]) {
245 solv = false;
                    prob = true;
247 }
                //falls die Obstsorte x gewuenscht ist,
                // wird sie als geprueft markiert
                else
251 ready[y] = 1;
            }
253
            //falls mind. ein Knoten zur Komponente gehoert, aber nicht
            // gewuenscht ist
            if (prob) {
255 //die Menge der Obstsorten der Komponente enthaelt
                // mind. einen Knoten einer Obstsorte, die nicht gewuenscht ist
                set<int> currSet;
261 for (auto y: setA)
                    currSet.insert(y);
                problems.pb(currSet);
263 }
            //falls alle Knoten, die zur Komponente gehoeren, gewuenscht sind
            else
265 //alle Indizes der Obstsorte der Komponente werden
                // in die Menge der Indizes der gewuenschten Obstsorten hinzugefuegt
                for (auto y: setB)
269 result.insert(y - n + 1);
        }
271 }
273
    //falls eine Loesung fuer die Eingabe existiert
    if (solv) {
275 return true;
    }
277
    //falls es keine eindeutige Loesung fuer die Eingabe gibt
    else {
279 cout << "Fuer die folgenden Obstsorten konnte keine eindeutige Zuweisung gefunden werden.\n";
        for (auto x: problems) {
281 cout << "Komponente: ";
            //Aufzaehlen der Obstsorten, die zur Komponente gehoeren
            for (auto y: x)
285 cout << ID2Fruit.find(y)->second << " ";
        }
287
        //Aufzaehlen der Obstsorten, die zur Komponente gehoeren,
        // aber nicht gewuenscht sind
        cout << "\n\t--> Nicht auf der Wunschliste: ";
289 for (auto y: x)

```

```
291         if (!todo[y])
            cout << ID2Fruit.find(y)->second << "␣";
293     cout << "\n";
    }
295     return false;
    }
297 }
```

./tex/spiesse.m