Aufgabe 2: Spießgesellen

Teilnahme-Id: 55628

Bearbeiter dieser Aufgabe:
Michal Boron

April 2021

Inhaltsverzeichnis

1 Lösungsidee

1.1 Formulierung des Problems

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

Axiom 1. Jeder **Obstsorte** wird genau ein einzigartiger natürlicher Index zugewiesen.
Man schreibt: $o(x, i)$ — eine Obstsorte x besitzt einen Index i.

Definition 1 (Spießkombination)**.** Als eine **Spießkombination** $K = (F, Z)$ bezeichnen wir eine Veknüpfung von zwei Mengen $F \subseteq A$ und $Z = \{i \in B \mid \exists x \in F : o(x, i)\}$.

Gegeben sind m Spießkombinationen, wobei jede Spießkombination $K_i = (F_i, Z_i)$ aus einer Menge von Obstsorten $F_i \subseteq A$ und einer Menge von 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 eine ***Wunschliste*** $W \subseteq A$.

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

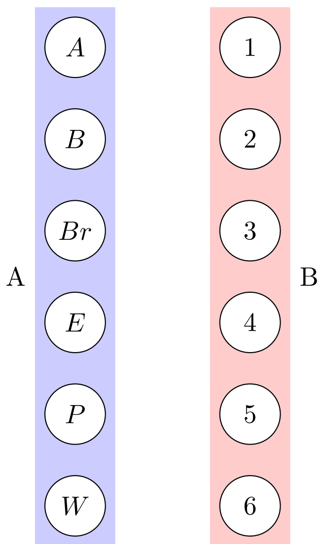
In den folgenden Überlegungen wird angenommen, dass das Axiom 1 für alle Obstsorten in der Eingabe gilt. Es ist aber möglich, dass die Spießkombinationen in einer Eingabe diesem Axiom nicht folgen, das heißt, es an einer Stelle einen Widerspruch gibt. Ein solcher Fall darf nach der Aufgabenstellung nicht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

1.2 Bipartiter Graph

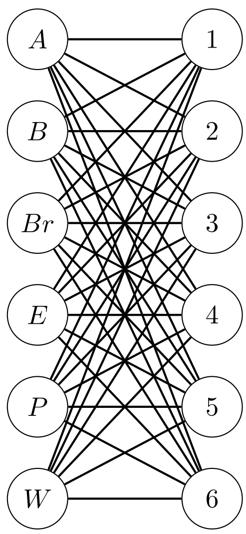
Man kann die beiden Mengen A und B zu Knoten eines bipartiten Graphen $G = (A \cup B = V, E)$ umwandeln. Die Menge der Kanten E wird im Folgenden festgelegt. Man stellt den Graphen als eine Adjazenzmatrix M der Größe $n \times n$ dar. Als M_i bezeichnet wird die Liste der Länge n, die die Beziehungen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

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

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



(a) Die entsprechenden Mengen des Graphen



(b) Der Graph am Anfang

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 3.0).

Am Anfang ist M dementsprechend voll mit Einsen. Bei der Erstellung der Adjazenzmatrix kann man den Vorteil nutzen, dass die Liste der Nachbarn eines Knotens $x \in A$ nur aus Nullen und Einsen besteht, indem man diese Liste als eine Bitmaske darstellt (mehr dazu in der Umsetzung).

Jede i–te Spießkombination $K_i = (F_i, Z_i)$ bringt Informationen über die Obstsorten in F_i. Man kann Folgendes festellen.

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

(i) $i \in Z$,

(ii) $i \notin B \setminus Z$.

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). □

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

Definition 2 (Zusammenhangskomponente)**.** Ein ungerichteter Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ heißt zusammenhängend, wenn es von jedem Knoten u zu jedem anderen Knoten v mindestens einen Pfad gibt. Ein maximaler zusammenhängender Teilgraph eines ungerichteten Graphen $\mathcal{G}$ heißt **Zusammenhangskomponente** $C = (V_c \subseteq \mathcal{V}, E_c \subseteq \mathcal{E})$ von $\mathcal{G}$.

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 darf man alle Kanten, die aus einem Knoten $x \in F$ zu einem Knoten $y \in R_c \setminus Z$, sowie die aus einem Knoten $p \in Z$ zu einem Knoten $q \in L_c \setminus F$ führen, aus E entfernen.

1.3 Logik

Da Bitmasken für die Darstellung der Listen M_i ($i \in A$) verwendet werden, kann die Laufzeit bei der Verarbeitung der jeweiligen Spießkombination optimiert werden (mehr dazu im Abschnitt 1.6), 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 Einsen. 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:

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

Spießkombination: F = {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

(a) M vor der neuen Spießkombination

	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

(b) M nach der Verarbeitung der beschriebenen Spießkombination.

Abbildung 2: 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 3.0). Auf der Abb. 2b stehen bn und br für die entsprechenden Bitmasken.

Auf der obigen Abbildung werden blau und rot diejenigen 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 verändern.

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

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

1.4 Zusammenhangskomponenten

Nach der Verarbeitung aller 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 3 (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 4 (Perfektes Matching)**.** Sei $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ ein ungerichteter Graph. Ein **perfektes Matching** ist 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 5 (Nachbarschaft)**.** Sei $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ ein ungerichteter Graph. Für alle $X \subseteq \mathcal{V}$ definieren wir die **Nachbarschaft** von X als $N(X) = \{y \in \mathcal{V} \mid \exists x \in X : (x, y) \in \mathcal{E}\}$.[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 für alle Teilmengen $\mathcal{K} \subseteq \mathcal{L}$ gilt: $|\mathcal{K}| \leqslant |N(\mathcal{K})|$.*[1, S. 736, Übung]

Beweis. Auf den Beweis [1] verzichten wir. □

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 vollständige Induktion über $k \in \mathbb{N}$ 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 Schlussfolgerung nach Lemma 1 werden alle Kanten zwischen allen $x \in F_1$ und allen $y \in B \setminus Z_1$, sowie zwischen allen $p \in Z_1$ und allen $q \in A \setminus F_1$ entfernt. Nach der Verarbeitung von K_1 entstehen so 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 $L_1 \cup R_1 = F_1 \cup Z_1$. Dann sind L_1 und R_1 nach Definition 1 auch gleichmächtig. Ebenfalls sind dann L_2 und R_2 gleichmächtig. Nach der gennanten Schlussfolgerung 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:

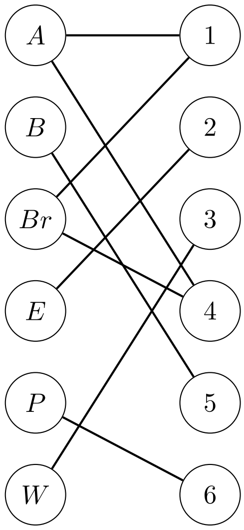
(i) Sei $D = (V_D, E_D)$ eine Zusammenhangskomponente in G. Sei $F_i \cup Z_i = V_D$. Da alle Knoten der Spießkombination sich mit allen Knoten von D decken, können keine Kanten nach der Folgerung aus Korollar 1 aus E entfernt werden, deshalb entsteht keine neue Zusammenhangskomponente, also ist jede 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 der Schlussfolgerung 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 $L_2 \cup R_2 = V_D \setminus (F_i \cup Z_i)$, die ebenfalls selbst vollständige, bipartite Graphen sind. Jede andere Zusammenhangskomponente in G ist nach der Induktionsannahme ein vollständiger, bipartiter Graph.

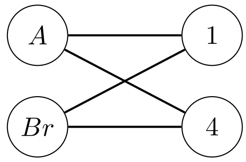
(iii) Sei $1 \leqslant t \leqslant n$ beliebig (n ist die Anzahl der Obstsorten), aber fest. Seien $C_1, C_2, ..., C_t$ paarweise verschiedene Zusammenhangskomponenten in G. Gehöre $F_i \cup Z_i$ zu mehreren Komponenten $C_p, ..., C_q$. Dann gilt für jede Zusammenhangskomponente C_i entweder (i) oder (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 $k \in \mathbb{N}$. □

[1] s. etwa: 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`



(a) Der Graph nach der Verarbeitung aller Spießkombinationen



(b) Die übrige Zusammenhangskomponente mit mehr als 2 Knoten

Abbildung 3: Abbgebildet ist das Beispiel aus der Aufgabenstellung nach der Verarbeitung von allen m Spießkombinationen.

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: $|K| \leqslant |N(K)|$.

Sei K eine beliebige Teilmenge von L_c mit der Mächtigkeit $|K| \leqslant |L_c|$. Da C selbst ein vollständiger, bipartiter Graph ist, besitzt jeder Knoten in C die Kardinalität von $|L_c| = |R_c|$. So gilt: $N(K) = |K| \cdot |L_c|$ und es gilt: $|K| \leqslant |N(K)|$. □

Nach der Verarbeitung aller Spießkombinationen entsteht ein Graph mit vielen Zusammenhangskomponenten (s. Abb. 3). 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: $L_c \subseteq W$, dann gilt: $R_c \subseteq W'$.*

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 damit ein perfektes Matching 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: $R_c \subseteq W'$. □

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. □

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 $C_1 = (L_1 \cup R_1, E_1), ..., C_k = (L_k \cup R_k, E_k)$ 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 zur Wunschliste 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, dessen 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 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.

Indentifizieren wir zuerst die Probleme, die auftreten können. In den folgenden Überlegungen nehmen wir an, dass jede Spießkombination $K = (F_i, Z_i)$ 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 verschiedene Indizes oder zwei Obstsorten denselben 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. Damit $|F_1| = |Z_1|$ gilt, muss gelten: $\exists o(z, k) : z \notin F_1 \land k \in Z_1$. Dann muss zwar kein Widerpsurch 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 $y \notin F_1 \land y \in F_2 \land x \notin F_2 \land i = j$, 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) vor.

Um zu prüfen, ob die Eingabe Axiom 1 widerspricht, muss man deshalb nur untersuchen, ob die Kardinalität jedes Knotens mit der Kardinalität eines seiner 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 verwendeteten Obstsorten und Indizes. In diesem Fall muss man die nicht genutzten Obstsorten und Indizes beim Einesen entsprechend markieren und sie beim Prüfen auf Korrektheit der Eingabe überspringen. Mehr dazu in der Umsetzung.

Sehen Sie dazu die folgenden Beispiele: Beispiel 8, Beispiel 9, Beispiel 14.

1.6 Laufzeit

- n — die Anzahl der Obstsorten
- m — die Anzahl der Spießkombinationen
- w — die Anzahl der Wünsche (also $|W|$), im worst–case $w = n$, weil $w \leqslant n$

Da wir Bitmasken in unserem Programm verwenden, ist noch eine Konstante einzuführen: β, die für die β–Bit–Architektur eines Rechners[2] 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 $O(\frac{|k|}{\beta})$ ausgefüht, wobei $|k|$ die Länge eines `bitset` ist. Eine Operation wird nicht auf einem einzlenen Bit ausgeführt, sondern es handelt sich um eine Operation auf einem integrierten Schaltkreis, deshalb stellt β 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. Diese Konstante wird ebenfalls beim Hashing verwendet (s. Umsetzung).

- Einlesen: average–case: $O(w \log w + n(m \log n + \frac{N}{\beta}))$; worst–case: $O(n(\log n(m + n) + w + \frac{N}{\beta}))$
  Wie es in der Umsetzung vermerkt wird, ist die Struktur der Beispiele aus der BWINF–Webseite sehr spezifisch. In allen Beispielen (bis auf das Beispiel aus der Aufgabenstellung) beginnen alle Obstsorten mit paarweise verschiedenen Buchstaben. In der folgenden Betrachtung bezeichnen wir so einen Fall als den average–case.
  - Erstellung der Liste `used` (s. Umsetzung): $O(n)$
  - Erstellung der Hashtabelle (s. Umsetzung): $O(N)$
  - Einlesen der Menge W als Strings: $O(w)$[3]
  - Kopieren von W zu `all_fruits` (s. Umsetzung): $O(w \log w)$[3]
    Die linear–logarithmische Laufzeit ist durch das Einfügen in eine Menge verursacht. Implementierug von `set` in C++ als Rot-schwarz-Bäume[4].
  - Einlesen jeder Spießkombination $K = (F_i, Z_i)$: $O(m \cdot n \log n)$
    Im schlimmsten Fall gilt: $|F_i| = |Z_i| = n$.
    * Die Menge Z_i wird eingelesen, indem man die Indizes in ein `set` einfügt: $O(n \log n)$.
    * Jeder Index $i \in Z_i$ wird in `used` markiert: $O(1)$
    * Die Elemente der Menge F_i werden als Strings eingelesen und in einer Liste gespeichert: $O(n)$
    * Die vorangegangene Liste wird zu `all_fruits` kopiert: $O(n \log n)$[3]
  - Die Liste `all_fruits` wird iteriert und die iterierten Strings werden gehasht (s. Umsetzung): $O(n)$ (average–case), $O(n^2)$ (worst–case)
    Wir verwenden *open address hashing* mit *linear probing*. Im schlimmsten Fall müssen wir $(n-1)$–mal zusätzlich iterieren, um an eine freie Stelle für einen String zu gelangen (s. auch Beispiel 16). So ergibt sich im schlimmsten Fall für das Suchen und Einfügen in einer Hashtabelle: $O(n)$. In der Mehrheit der Fällen müssen wir jedoch gar nicht iterieren, da ein gesuchtes Element an der Stelle steht, die durch die Hashfunktion ermittelt wurde (s. Umsetzung). So ergibt sich im average–case: $O(1)$. Nur im Beispiel 16 ist mit der Laufzeit von $O(n)$ zu rechnen.
    In der Menge `all_fruits` befinden sich maximal alle n Obstsorten.

[2] https://en.wikipedia.org/wiki/Word_(computer_architecture)
[3] 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 zu vernachlässigende Konstante betrachte.
[4] https://en.cppreference.com/w/cpp/container/set

- Es wird durch die Hashtabelle iteriert, um die internen Indizes den Obstsorten zuzuordnen und die Liste `ID2Fruit` zu erstellen (s. Umsetzung): $O(N)$
  Während dessen werden auch die internen Indizes in `used` in $O(1)$ markiert.
- Die Strings in W werden zu den zugehörigen internen Indizes umgewandelt und in eine Menge eingefügt: $O(w \log w)$ (average–case), $O(w \cdot n + w \log w) = O(w \cdot n)$ (worst–case)
  Dazu wird die beschriebene Hashfunktion verwendet.
- Die Strings in der Liste F_i der jeweiligen Spießkombination werden zu den zugehörigen internen Indizes umgewandelt und F_i wird zu einer Menge umgewandelt: $O(n \log n)$ (average–case), $O(n^2 \log n)$ (worst–case)
  Im schlimmsten Fall gehören zu F_i alle n Obstsorten. Im average–case beginnen alle Obstsorten mit unterschiedlichen Buchstaben und so läuft die Hashfunktion in $O(1)$.
- Erstellung der Adjazenzmatrix M: $O(n \cdot \frac{N}{\beta})$
- Die gesamte Laufzeit für diesen Teil beträgt für den worst–case: $O(n) + O(N) + O(w) + O(w \log w) + O(m \cdot n \log n) + O(n^2) + O(N) + O(w \cdot n) + O(n^2 \log n) + O(n \cdot \frac{N}{\beta}) = O(n + N + w + w \log w + m \cdot n \log n + n^2 + N + w \cdot n + n^2 \log n + n \cdot \frac{N}{\beta}) = O(n(\log n(m+n) + w + \frac{N}{\beta}))$
- Die gesamte Laufzeit für diesen Teil beträgt für den average–case: $O(n) + O(N) + O(w) + O(w \log w) + O(m \cdot n \log n) + O(n) + O(N) + O(w \log w) + O(n \log n) + O(n \cdot \frac{N}{\beta}) = O(n + N + w + w \log w + m \cdot n \log n + n + N + w \log w + n \log n + n \cdot \frac{N}{\beta}) = O(w \log w + n(m \log n + \frac{N}{\beta}))$

• 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 aller Spießkombinationen entsprechend: $O(m \cdot n \cdot \frac{N}{\beta})$
  - Kopieren der Adjazenzmatrix in `Graph G` (s. Umsetzung): average–case: $O(n \cdot \delta(i) \cdot \frac{N}{\beta})$, worst–case: $O(n^2 \cdot \frac{N}{\beta})$
    Für jede Liste M_i werden alle Einsen 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[5] beschriebene Idee anwenden, die mit einem logarithmischen Aufwand bezüglich der Länge der Bitmaske läuft.)
    Deshalb erfolgt die Iteration über eine Liste M_i in $O(\delta(i) \cdot \frac{N}{\beta})$, wobei $\delta(i)$ die Anzahl der Einsen in M_i ist. Im schlimmsten Fall, wenn alle Spießkombinationen aus n Obstsorten bestehen oder es keine Spießkombination gibt, gilt: $\delta(i) = n$, also gilt: $O(n \cdot \frac{N}{\beta})$. Im allgemeinen Fall gilt: $\delta(i) \ll n$. 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 Einsen aus der Adjazenzmatrix ablesen.
  - Die gesamte Laufzeit für diesen Teil beträgt (worst–case):
    $O(m \cdot n \cdot \frac{N}{\beta}) + O(n^2 \cdot \frac{N}{\beta})) = O(m \cdot n \cdot \frac{N}{\beta} + n^2 \cdot \frac{N}{\beta})) = O(n \cdot \frac{N}{\beta}(m+n))$

[5] 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 gilt für alle Knoten im bipartiten Graphen: $O(n + n) = O(n)$
- Prüfung der Existenz einer Lösung: $O((n + w) \log w)$ (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 jedes 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 `multip` (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 `multip`: $O(n \log w)$ (worst–case)
    Die folgenden Operationen werden nur dann ausgeführt, wenn die Komponente, zu der der iterierte Knoten gehört, noch nicht bearbeitet wurde, also wenn dieser Knoten in $\bar{R}$ nicht 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 $n(y)$ eines Nachbarn y eines iterierten Knotens: $O(n)$
      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 von w Elementen hat eine linear–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 denen sich mind. eine gewünschte Obstsorte befindet. Durch die Markierung in $\bar{R}$ wird jeder Knoten von diesen Zusammenhangskomponenten nur einmal behandelt. Damit ergibt sich im worst–case die Laufzeit von $O(n \log w)$, falls die gewünschten Obstsorten auf allen Zusammenhangskomponenten in G verteilt sind.
  - Ausgabe für eine Eingabe, für die W' nicht eindeutig bestimmt werden kann: $O(n)$
    Es wird durch alle Komponenten iteriert, die mind. eine ungewünschte Obstsorte enthalten, und alle Knoten werden mit den Namen der Obstsorten aufgezählt. 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): $O(n)+O(n)+O(w \log w)+O(n \log w) + O(n) = O(n + n + w \log w + n \log w + n) = O((n + w) \log w)$

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

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

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

Nach der Abschätzung $n = w$ ergibt sich im worst–case:

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

Fassen wir nun die Laufzeit im average–case zusammen. Als ein average–case gilt in diesem Fall so ein Fall, in dem alle Obstsorten mit paarweise möglichst verschiedenen Buchstaben anfangen.

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

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

Nach der Abschätzung $n = w$ ergibt sich im average–case:

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

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 vernachlässiger Faktor. Es ergibt sich im average–case:

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

Literatur

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

2 Umsetzung

2.1 Klasse `Hashing`

Alle auf der BWINF–Webseite verfügbaren Beispiele beinhalten Obstsorten, die mit paarweise verschiedenen Buchstaben anfangen. (Die Ausnahme ist das Beispiel aus der Aufgabenstellung, in dem zwei Obstsorten mit demselben Buchstaben beginnen.) Man kann diese Tatsache bei der Wahl einer Datenstruktur für die Namen der Obstsorten nutzen. Im Programm operiert man zwar nicht auf Strings, sondern auf internen Indizes, die Integers sind. Allerdings muss man die Verbindungen zwischen dem Namen der Obstsorte und ihrer internen Index speichern. Die Anwendung vom Hashing in der Lösung zu dieser Aufgabe für die gegebenen Beispiele erweist sich besonders vorteilhaft im Vergleich zur Verwendung von einfachen Maps, indem man auf die Elemente in der Hashtabelle in $O(1)$ zugreifen kann (bis auf Beispiel 16).

Dazu verwenden wir *open address hashing* mit *linear probing*. Diese Art von Hashing löst eine Kollision auf, indem eine neue Stelle für ein Element durch Probieren gefunden wird.
Am Anfang erstellen wir eine Hashtabelle der Größe 26 (im Programm ist dies eine Konstante `MAXN`, s. auch `bitset` unten), weil diese Zahl der Größe des lateinischen Alphabets entpricht und gleichzeitg die maximale Größe von n darstellt. Diese Tabelle besteht aus Tupeln von `string` und `int`. Am Anfang wird diese Tabelle mit Tupeln aus einem leeren String und dem Wert -1 gefüllt.

Wir kontruieren dazu eine Hashfunktion, die als Parameter einen String nimmt. Diese Funktion ermittelt die Stelle in der Hashtabelle für den gegebenen String. Sei ℓ eine Zahl zwischen 0–25, die dem ersten Buchstaben im String entspricht (0 – A, 25 – Z). So wird mithilfe der Methode `hash()` die Stelle p für einen gegebenen String auf folgende Weise ermittelt:

$$p \equiv \ell \pmod{\texttt{MAXN}}.$$

Wenn an dieser Stelle in der Hashtabelle ein leerer String steht, wird der gegebene String mithilfe der Methode `saveHash()` dort gespeichert. Falls es zu einer Kollision kommt, wird die Stelle p auf folgende Weise inkrementiert:

$$p :\equiv (p+1) \pmod{\texttt{MAXN}}.$$

p wird solange inkrementiert, bis eine Stelle gefunden wird, an der sich ein leerer String befindet. Es ist gesichert, dass jeder String in der Eingabe in der Hashtabelle gespeichert wird, da die Größe der Tabelle dem größten n entspricht. Falls man ein größeres Beispiel lösen möchte, soll man die Konstante `MAXN` im Programm ändern.

Mithilfe der Methode `getHash()` wird die Stelle eines Strings in der Hashtabelle zurückgegeben, falls dieser in der Tabelle enthalten ist. Sonst wird der Wert -1 zurückgegeben. Diese Funktion geht analog vor, wie die Funktion `saveHash()`. Man ermittelt zunächst die Stelle in der Hashtabelle mithilfe der Hashfunktion, iteriert von dort und prüft, ob das iterierte Element dem gegebenen String entspricht. Wenn der Iterator auf eine Stelle stößt, die einen leeren String enthält, wird sofort -1 zurückgegeben, da der String sich in der Hashtabelle nicht befinden kann, weil dieser String nach dem oben beschriebenen Prinzip an der ersten leeren Stelle gespeichert werden sollte.

Die Methode `readFile()` liest die Daten aus der Textdatei ein. Nach dem Einlesen von n und m, also der Anzahl der Obstsorten und der Anzahl der Spießkombinationen, werden die Wünsche eingelesen. Die Menge W wird zunächst als ein `vector` von Strings mit dem Namen `wishes_words` gespeichert. Außerdem werden die Strings zu einem anderen `set` von Strings `all_fruits` kopiert. In dieser Liste werden alle Obstsorten gespeichert, die in der Eingabe auftreten. Danach folgt das Einlesen der Spießkombinationen. Jede Menge Z_i wird als `set` von Integers gespeichert. Gleichzeitig werden auch die Indizes, die in Z_i auftreten, in der Liste `used` markiert.

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 einigen Textdateien ein größeres n gibt als die Anzahl der in den Spießkombinationen und der Wunschliste verwendeten Obstsorten und Indizes. Diese Markierung erweist sich bei der Prüfung auf korrekte Eingabe als hilfreich.

Danach wird die Menge F_i als ein `vector` von Strings gespeichert und alle Obstsorten, die in der Eingabe auftreten, werden ebenfalls zu `all_fruits` kopiert. Die Menge Z_i und der `vector` F_i werden temporär als Tupel in den `vector tempInfos` hinzugefügt.

Danach erfolgt das Hashing. `all_fruits` wird iteriert und jedes Mal wird die Funktion `getHash` aufgerufen, um zu prüfen, ob der iterierte String sich bereits in der Hashtabelle befindet. Wenn nicht, wird er mithilfe der Funktion `saveHash()` in dieser Tabelle gespeichert.
Im Weiteren werden die Obstsorten nicht als String behandelt, sondern wird jeder Obstsorte ein interner

Index zugewiesen. Theoretisch könnte man einfach die Stellen der Obstsorten in der Hashtabelle nehmen. Da in vielen Fällen gilt: $n < 26$, können wir, anstatt diese Stellen zu nehmen, die unabhängig von n zwischen 0 und 26 liegen, die internen Indizes der Obstsorten zu Zahlen zwischen 0 und $n-1$ komprimieren. Wir erstellen eine neue Variable `it` und setzen sie zu 0. Wir iterieren durch die Hashtabelle. Wenn an einer Stelle sich ein nichtleerer String befindet, wird dort der Wert des Integers `it` im Tupel gemeinsam mit diesem String gespeichert (die Hashtabelle besteht aus Tupeln von Strings und Integers). Folglich wird `it` inkrementiert. Gleichzeitig wird der an dieser Stelle stehende String in einen `vector ID2Fruit` hinzugefügt und der interne Index dieser Obstsorte wird in `used` markiert. Man beachte, dass die i–te Stelle in `ID2Fruit` dem i–ten internen Index einer Obstsorte entspricht.

Am Ende wird die Liste `wishes_words` iteriert und an jedem iterierten String wird die Methode `getIndex` aufgerufen, die den internen Index dieses Strings in der Hashtabelle zurückgibt. Auf eine analoge Weise werden dann die Strings in `tempInfos` zu ihren entsprechenden internen Indizes umgewandelt.

2.2 Klasse `Solver`

Diese Klasse vererbt die Klasse `Hashing`.

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 (s. oben die Größe der Hashtabelle).
Die feste Länge ist auch der Grund dafür, dass die Bitmaske br im Abschnitt 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 ausreichen. Wenn $n < 26$, dann enthält eine Bitmaske bei einer Negation $26-n$ zusätzliche Einsen. Aus Gründen der Übersichtlichkeit und der möglichen Weiterentwicklung des Programms möchte man sie zu Nullen umwandeln.

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 ist.
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 Einsen in `matrix` werden mithilfe der eingebauten Funktion `_Find_next()` gefunden.

Die Methode `checkCoherence()` prüft, ob die Eingabe dem Axiom 1 folgt. Dennoch gibt es 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 Kardinalität jedes 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. Ansonsten 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`, also die Menge W', hinzugefügt. Im sonstigen Fall wird `x` in `multip` hinzugefügt und die Stelle `x` in `todo` wird mit 1 markiert.
Dann wird geprüft, ob die Menge `multip` überhaupt irgendwelche Elemente enthält. Falls nicht, gibt die ganze Funktion an dieser Stelle `true` zurück.
Sonst 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 `multip` 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 `multip`, wird geprüft, ob es eine Lösung zur Aufgabe für eine Eingabe gibt: Es wird gepüft, 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.

2.3 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

3.0 Beispiel 0 (Aufgabenstellung — Teil a)

Textdatei: `spiesse0.txt`

Wünsche $\boxed{\text{Apfel, Brombeere, Weintraube}}$

$\boxed{1, 3, 4}$

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. 4a). Jede Kante steht für eine mögliche Zuweisung eines Index einer Obstsorte.

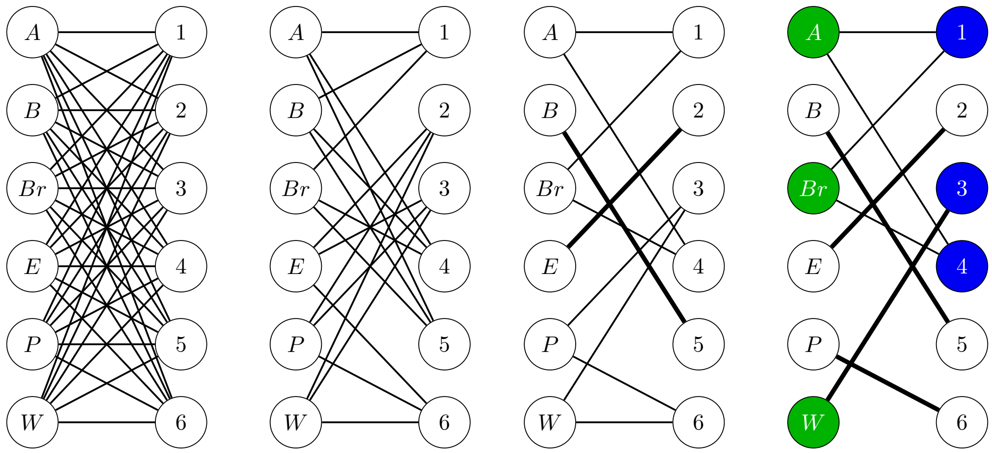
Wir analysieren die 1. Spießkombination: $\boxed{F_1 = \{\text{Apfel, Banane, Brombeere}\}, Z_1 = \{1, 4, 5\}}$.
Wir stellen fest, dass Apfel, Banane und Brombeere jeweils keine Indizes 2, 3, 6 haben können, weil jeder dieser Obstsorten ein Index aus der Menge Z_1 zugewiesen wird. Gleichzeitig merken 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. 4b).

Wir analysieren die 2. Spießkombination: $\boxed{F_2 = \{\text{Banane, Pflaume, Weintraube}\}, Z_2 = \{3, 5, 6\}}$.
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. Wir entfernen 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, Brombeere 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 steht fest: $o(\text{Erdbeere}, 2)$. Auf der Abbildung 4c 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: $\boxed{F_3 = \{\text{Apfel, Brombeere, Erdbeere}\}, Z_3 = \{1, 2, 4\}}$.
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: $\boxed{F_4 = \{\text{Erdbeere, Pflaume}\}, Z_4 = \{2, 6\}}$.
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: $\{\text{Apfel, Brombeere}\}$ und $\{1, 4\}$.

An dieser Stelle schauen wir die Wunschliste an: $\boxed{\text{Apfel, Brombeere, Weintraube}}$.
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 führen. Auf der Abbildung 4d wurden alle gefundenen Zuweisungen durch fette Kanten dargestellt und alle Wünsche mit ihren Indizes wurden entsprechend grün und blau markiert.



(a) (b) (c) (d)

Abbildung 4

3.1 Beispiel 1 (BWINF)

Textdatei: `spiesse1.txt`

Wünsche: Clementine, Erdbeere, Grapefruit, Himbeere, Johannisbeere

1, 2, 4, 5, 7

3.2 Beispiel 2 (BWINF)

Textdatei: `spiesse2.txt`

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

1, 5, 6, 7, 10, 11

3.3 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
```

3.4 Beispiel 4 (BWINF)

Textdatei: `spiesse4.txt`

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

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

3.5 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

3.6 Beispiel 6 (BWINF)

Textdatei: `spiesse6.txt`

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

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

3.7 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
```

3.8 Beispiel 8

Textdatei: `spiesse8.txt`
Besonderheit: Ein Beispiel für den Fall (P1) → (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.

3.9 Beispiel 9

Textdatei: `spiesse9.txt`
Besonderheit: Ein Beispiel für den Fall (P2) → (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.

3.10 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

3.11 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
```

3.12 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 eideutige 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
```

3.13 Beispiel 13

Textdatei: `spiesse13.txt`
Besonderheit: Es gibt $m = n$ Spießkombinationen. Die Mächtigkeiten der Mengen dieser Spießkombinationen entsprechen: $n, n-1, ..., 1$. 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 GpOrapefruit 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

3.14 Beispiel 14

Textdatei: `spiesse14.txt`
Besonderheit: Ein Beispiel für den Fall (P1) → (F2), s. Teil 1.5.

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

Wünsche: Apfel, Clementine, Feige

Error: Es gibt Fehler in der Eingabedatei.

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.

3.15 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 behandeln 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: Apfel, Dattel, Feige

2, 3, 4

3.16 Beispiel 16

Textdatei: `spiesse16.txt`
Besonderheit: Alle n Obstsorten beginnen mit „M“. Im worst–case läuft die Suche nach einem String in der Hashtabelle in $O(n)$.

```
11
Moosbeere Maulbeere Marone Mandarine Mango
6
11 9 6 4 8
Mandarine Mirabelle Moosbeere Moltebeere Mispel
1 7 5 3 2
Mango Maracuja Marille Maulbeere Melone
7 6 3 9 11 1 4
Mango Marille Maulbeere Mirabelle Mispel Moosbeere Moltebeere
2 6 9 8 5 3
Mango Moosbeere Moltebeere Maracuja Mandarine Melone
9 11 2 5 8
Mispel Maracuja Mandarine Moltebeere Melone
1 11 3 5 2 8 9
Marille Moltebeere Mango Mandarine Maracuja Melone Mispel
```

Wünsche: Moosbeere, Maulbeere, Marone, Mandarine, Mango

3, 6, 7, 8, 10

4 Quellcode

```
//diese Methode ist eine Hashfunktion, die einem String
//  eine Stelle in der Hashtabelle zuordnet
int Hashing::hash(string s) {
  //die Stelle in der hashtable wird anhand des
  //  ersten Buchstabne ermittelt
  char letter = s[0];
  letter -= 65;

  //falls der erste Buchstabe klein ist
  if (letter > 25)
    letter -= 32;

  //die Stelle wird ermittelt
  return letter % hashtable.size();
}

//diese Methode speichert einen String an der
//  durch die Hashfunktion ermittelte Stelle
void Hashing::saveHash(string s) {
  int pos = hash(s);

  //falls die zugeordnete Stelle nicht besetzt ist
  if (hashtable[pos].first == "") {
    hashtable[pos].first = s;
    return;
  }

  //falls die zugeordnete Stelle bereits besetzt ist,
  //  wird nach der ersten freien Stellen gesucht
  int it = (pos + 1) % hashtable.size();
  while (hashtable[it].first != "") {
    it++;
    if (it == pos)
      return;
  }

  //der String wird an der gefundenen Stelle gespeichert
  hashtable[it].first = s;
}

//diese Methode gibt die Stelle in der hashtable zurueck,
//  falls ein String sich in der hashtable bereits befindet.
//  Sonst wird -1 ausgegeben
int Hashing::getHash(string s) {
  int pos = hash(s);

  //falls der gesuchte String sich an der zugeordneten
  //  Stelle befindet
  if (hashtable[pos].first == s)
    return pos;

  //es wird weiteriteriert, um die Stelle mit dem gegebenen String zu finden
  int i = (pos + 1) % hashtable.size();
  for (; i != pos; i = (i + 1) % hashtable.size()) {
    if (hashtable[i].first == "")
      return -1;
    if (hashtable[i].first == s)
      return i;
  }

  return -1;
}

//diese Methode gibt den internen Index einer Obstsorte zurueck
int Hashing::getIndex(string s) {
  int pos = getHash(s);
  if (pos > -1)
    return hashtable[pos].second;
  return -1;
}
```

```cpp
//diese Methode liest Informationen aus der Datei ein
void Hashing::readFile(string path) {
  fstream file;
  file.open(path, ios::in);
  if (file.is_open()) {

    //Anzahl der Obstsorten
    file >> n;

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

    //Erstellung der hashtable
    hashtable = vector<pair<string, int>> (MAXN, {"", -1});

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

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

    //in dieser Menge werden alle Obstnamen gespeichert, die in der
    //  Datei auftreten
    set<string> all_fruits(wishes_words.begin(), wishes_words.end());

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

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

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

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

        //ein Index (aus der Aufgabenstellung) wird
        //  als verwendet merkiert
        used[stoi(x)+n-1] = true;
      }

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

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

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

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

      //eine Spiesskombination als ein Menge von Zahlen und
      //  eine Menge von Strings wird gespeichert
      tempInfos.pb({currNum, currFruits});
```

```
    }

    file.close();

    //alle Obstsorten werden durch die Hashfunktion verarbeitet
    for (auto x: all_fruits) {
      int id = getHash(x);
      //falls der Obstname x noch nicht aufgetreten ist
      if (id == -1)
        saveHash(x);
    }

    //Zuweisung der internen Indizes 0..(n-1) jeder Obstsorte
    //  mithilfe einer Hasfunktion
    int it = 0;
    for (int i = 0; i < hashtable.size(); i++) {
      if (hashtable[i].first != "") {
        //ein interner Index wird einer Obstsorte zugeordnet
        hashtable[i].second = it;

        //der Obstname wird in die Liste ID2Fruit hinzugefuegt,
        //  dessen Stelle in der Liste dem internen Index
        //  der Obstsorte entspricht
        ID2Fruit.pb(hashtable[i].first);

        //der interne Index einer Obstsorte wird als verwendet merkiert
        used[it++];
      }
    }

    //die gewuenschten Obstsorten werden als Indizes gespeichert
    for (auto x: wishes_words)
      wishes.insert(getIndex(x));

    //die Spiesskombinationen als Mengen der Indizes der Obstsorten
    //  einer Spiesskombination und Mengen der internen Indizes der Obstsorten
    for (auto x: tempInfos) {
      set<int> currFruits;
      for (auto y: x.second)
        currFruits.insert(getIndex(y));
      infos.pb({x.first, currFruits});
    }
  }
  //falls die Textdatei nicht geoeffnet werden kann
  else {
    cerr << "Error:␣File␣could␣not␣be␣opened.␣Abort.\n";
    exit(0);
  }
}


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

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

  //eine Bitmaske mit 1-en auf allen n Stellen
  for (int i = 0; i < n; i++) mask_full ^= (1 << i);

  //die Bitmaske fuer die Obstsorten aus der Menge
  for (auto x: nums) mask_nums ^= (1 << (x-1));

  //die Bitmaske fuer die Obstsorten ausserhalb der Menge
  mask_rev = ~(mask_nums) & mask_full;

  //es wird ueber die Menge der Obstsorten der Spiesskombination
  //  iteriert
  auto it = fruits.begin();

  //es wird ueber die Obstsorten in der Adjazenzmatrix iteriert
```

```
  for (int i = 0; i < n; i++) {
    //falls eine Obstsorte zur Menge der Obstsorten der
    //  Spiesskombination gehoert
    if (it != fruits.end() && *it == i) {
      matrix[i] &= mask_nums;
      it++;
    }
    //falls eine Obstsorte ausserhalb der Menge der Obstsorten
    //  der Spiesskombination liegt
    else
      matrix[i] &= mask_rev;
  }
}

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

  //alle noch uebrigen Kanten werden in den Graphen hinzugefuegt
  for (int i = 0; i < ID2Fruit.size(); i++) {
    for (int j = matrix[i]._Find_next(-1); j < MAXN; j = matrix[i]._Find_next(j))
      G.addEdge(i, j);
  }
}

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

      //wir nehmen den ersten Nachbarn von i
      int neigh = G.getFirstNeighbor(i);

      //wir vergleichen die Kardinalitaeten
      if (G.deg(neigh) != G.deg(i))
        return false;
    }

  return true;
}

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

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

  //die Liste der Knoten der gewuenschten Obstsorten
  vector<int> todo(n);

  //es wird durch die Menge der Wuensche iteriert
  for (auto x: wishes) {
    //falls der Grad des Knotens der gewuenschten Obstsorte
    // gleich 1 ist
    if (G.deg(x) == 1) {
      int neigh = G.getFirstNeighbor(x);

      //der einzige Nachabr wird in die Ergebnismenge hinzugefuegt
      result.insert(neigh - n + 1);
    }
    //falls der Grad des Knotens der gewuenschten Obstsorte
    // mehr als 1 betraegt
```

```
    else {
      //der gewuenschte Knoten wird in die Menge
      //  fuer die Untersuchung der Komponente hinzugefuegt
      multip.insert(x);

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

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

  //diese Variable zeigt an, ob eine Loesung fuer die Eingabe existiert
  bool solv = true;

  //eine Liste mit Mengen der Obstsorten der Komponenten,
  //  zu deren Knoten gehoeren, die nicht auf der Wunschliste stehen
  vector<set<int>> problems;

  //es wird durch die Menge der Knoten der gewuenschten Obstsorten iteriert,
  // deren Grad groesser als 1 betraegt
  for (auto x: multip) {
    //diese Variable zeigt an, ob mind. ein Knoten zur Komponente
    // gehoert, aber nicht gewuenscht ist
    bool prob = false;

    //falls die Komponente eines Knotens noch nicht geprueft wurde
    if (!ready[x]) {
      //die Menge der Knoten der Indizes der Obstsorten der jeweiliger Komponente
      vector<int> setB = G.getNeighbors(x);
      //die Menge der Knoten der Obstsorten der jeweiliger Komponente
      vector<int> setA = G.getNeighbors(setB[0]);

      //es wir durch die Knoten der Obstsorten dieser Komponente
      //  iteriert
      for (auto y: setA) {
        //falls die Obstsorte x nicht gewuenscht ist,
        //  gibt es keine eindeutige Loesung duer die Eingabe
        if (!todo[y]) {
          solv = false;
          prob = true;
        }
        //falls die Obstsorte x gewuenscht ist,
        //  wird sie als geprueft markiert
        else
          ready[y] = 1;
      }

      //falls mind. ein Knoten zur Komponente gehoert, aber nicht
      //  gewuenscht ist
      if (prob) {
        //die Menge der Obstsorten der Komponente enthaelt
        //  mind. einen Knoten einer Obstsorte, die nicht gewuenscht ist
        set<int> currSet;
        for (auto y: setA)
          currSet.insert(y);
        problems.pb(currSet);
      }
      //falls alle Knoten, die zur Komponente gehoeren, gewuenscht sind
      else
        //alle Indizes der Obstsorte der Komponente werden
        //  in die Menge der Indizes der gewuenschten Obstsorten hinzugefuegt
        for (auto y: setB)
          result.insert(y - n + 1);
    }
  }

  //falls eine Loesung fuer die Eingabe existiert
  if (solv) {
    return true;
```

```
  }
  //falls es keine eindeutige Loesung fuer die Eingabe gibt
  else {
    cout << "Fuer␣die␣folgenden␣Obstsorten␣konnte␣keine␣eindeutige␣"
      << "Zuweisung␣gefunden␣werden.\n";
    for (auto x: problems) {
      cout << "Komponente:␣";
      //Aufzaehlen der Obstsorten, die zur Komponente gehoeren
      for (auto y: x)
        cout << ID2Fruit[y] << "␣";

      //Aufzaehlen der Obstsorten, die zur Komponente gehoeren,
      //   aber nicht gewuenscht sind
      cout << "\n\t-->␣Nicht␣auf␣der␣Wunschliste:␣";
      for (auto y: x)
        if (!todo[y])
          cout << ID2Fruit[y] << "␣";
      cout << "\n";
    }
    return false;
  }
}
```