

Aufgabe 2: Spießgesellen

Teilnahme-Id: 55628

Bearbeiter dieser Aufgabe:
Michal Boron

April 2021

Inhaltsverzeichnis

1	Lösungsidee	1
1.1	Formulierung des Problems	1
1.2	Bipartiter Graph	2
1.3	Logik	3
1.4	Zusammenhangskomponenten	4
1.5	Prüfung auf Korrektheit der Eingabe	7
1.6	Laufzeit	7
1.7	Speicherplatz	8
2	Umsetzung	9
2.1	Klasse Solver	9
2.2	Klasse Graph	10
3	Beispiele	11
3.1	Beispiel 0 (Aufgabenstellung)	11
3.2	Beispiel 1 (BWINF)	11
3.3	Beispiel 2 (BWINF)	11
3.4	Beispiel 3 (BWINF)	11
3.5	Beispiel 4 (BWINF)	11
3.6	Beispiel 5 (BWINF)	11
3.7	Beispiel 6 (BWINF)	12
3.8	Beispiel 7 (BWINF)	12
3.9	Beispiel 8	12
4	Quellcode	12

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 Zahlen $Z_i \subseteq B$ 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.

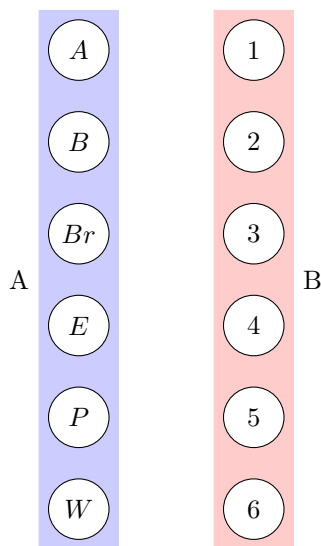
Nach Axiom 1 gehört jeder Obstsorte aus A genau ein Index aus B . Dennoch man kann am Anfang keiner Obstorte einen Index zuweisen. Deshalb wird zunächst jeder Knoten aus A mit jedem Knoten aus B durch eine Kante verbunden:

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

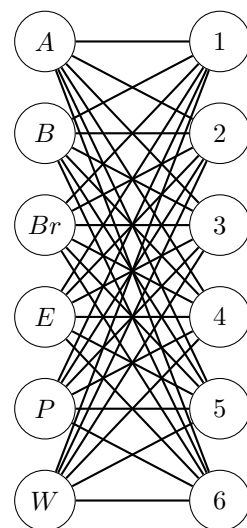
Am Anfang ist M dementsprechend voll mit 1-en. Bei der Erstellung der Adjazenzmatrix kann man den Vorteil nutzen, dass die jeweilige Liste von Nachbarn des jeden Knotens $x \in A$ nur aus 0-en und 1-en besteht, indem diese Liste als Bitmasken dargestellt werden kann (mehr dazu in der Umsetzung).

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).



(a) Die entsprechenden Mengen des Graphen



(b) Der Graph am Anfang

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 $y \in B \setminus Z$ 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 aufgegliedert wird, schrumpft die Anzahl an möglichen Zuweisungen zwischen jedem x und jedem y . \square

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 $y \in R_c \setminus Z$ führen, aus E entfernt.

Da Bitmasken für die Darstellung jeder Liste M_i ($i \in A$) verwendet werden, kann die Laufzeit bei der Analyse der jeweiligen Spießkombination optimiert werden (mehr dazu im Teil 1.6), weil man für die Operation des Entfernens Logikgatter verwenden kann.

1.3 Logik

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:

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

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.

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 zu Z_s nicht gehört und an der Stelle $M_{x,y}$ 0 steht, bleibt es auch 0.
 - d) 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.
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 zu Z_s nicht gehört, aber an der Stelle $M_{x,y}$ 0 steht, bleibt es auch 0.
 - b) Falls der Knoten y zu nicht 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, aber an der Stelle $M_{x,y}$ 1, wird die Stelle $M_{x,y}$ zu 0.
 - d) 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 $N(X) = \{y \in \mathcal{V} \mid \exists x \in X : (x, y) \in \mathcal{E}\}$. [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 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 ¹ zu finden. □

Wir können Folgendes feststellen.

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.

¹Anup 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>

Beweis.

TODO: check proof and lemma

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 die Aussage sofort für $k = 1$.) Nach der Verarbeitung von K_1 entstehen zwei Zusammenhangskomponenten: $C_1 \cup C_2 = G$, wobei o.B.d.A $C_1 = F \cup Z$. Dann sind $C_1 \cap A$ und $C_1 \cap B$ nach Definition 1 auch gleichmächtig. Ebenfalls sind dann $C_2 \cap A$ und $C_2 \cap B$ gleichmächtig. Nach 1 gilt, dass alle Kanten zwischen C_1 und C_2 aus E entfernt wurden, aber alle innerhalb von C_1 und innerhalb von C_2 beibehalten wurden. 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 $k \in \mathbb{N}$, 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, nach Korollar 1, keine Kanten aus E entfernt werden, deshalb entsteht keine neue Zusammenhangskomponente, also ist jede Zusammenhangskomponente nach der Induktionsannahme ein vollständiger, bipartiter Graph.
- (ii) Sei $D = (V_D, E_D)$ eine Zusammenhangskomponente in G . Sei $F_i \cup Z_i \subsetneq V_D \wedge (F_i \cup Z_i) \not\subseteq (A \cup B)$, also $F_i \cup Z_i$ gehört nur zu einer Zusammenhangskomponente in G . D ist laut Induktionsannahme selbst ein vollständiger, bipartiter Graph. Nach Korollar 1 werden alle Kanten zwischen allen $x \in F_i$ und allen $y \in V_D \cap Z_i$ entfernt. So entstehen zwei neue Zusammenhangskomponenten: $C_1 = F_i \cup Z_i$ 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 $1 \leq p \leq n$ beliebig, aber fest. Seien C_1, C_2, \dots, C_p 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 mögliche untersucht wurden, ist der Induktionsschritt vollzogen und die Behauptung gilt für jedes $k \in \mathbb{N}$. □

Lemma 3. Sei $C = (V_c, E_c)$ eine Zusammenhangskomponente in G . Dann existiert immer ein perfektes Matching zu C .

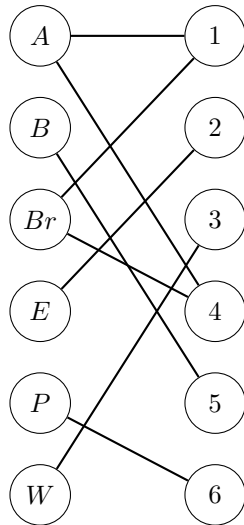
Beweis.

TODO: check details, fomulate things differently

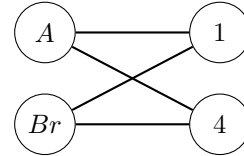
Nach Lemma 2 ist jede Zusammenhangskomponente ein vollständiger, bipartiter Graph. Nach Satz von Hall existiert ein perfektes Matching, wenn für alle Teilmengen $K \subseteq A \cap V_c$ gilt: $|K| \leq |N(K)|$. Die obere Behauptung kann für beliebig große Mächtigkeiten $|K| = k \in \mathbb{N}$ durch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

Induktionsanfang: Für $k = 1$ hat der einzelne Knoten $x \in K \subseteq A \cap V_c$ 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.

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

Induktionsschritt: Es gelte die Aussage für ein $k \in \mathbb{N}$, also für eine Teilmenge $K \subseteq A \cap V_c$, 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 $K' \subseteq A \cap V_c$ 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 Wuschliste 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 festgelegt 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 $C_x = (L_x \cup R_x, E_x)$, 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 1 bei jedem z 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 Nachricht 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.

TODO: zrobić

1.6 Laufzeit

n — die Anzahl der Obstsorten

m — die Anzahl der Spießkombinationen

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

- Einlesen: $O(m \cdot n \log n)$ (worst-case) [MB: sprawdzić bitsety]

TODO: dodać złożoności dla w

- Erstellung der Adjazenzmatrix M : [MB: $O(\frac{n}{w})$]; dodać info o bitsetach]
- Einlesen der Menge W : $O(w \log w)$
Implementierung von `set` in C++ als Rot-schwarz-Bäume²
- Einlesen der Spießkombinationen: $O(m \cdot n \log n)$ (worst-case)
Im schlimmsten Fall enthält jede Spießkombination alle Obstsorten. Die logarithmische Laufzeit ist durch Einfügen in eine Menge verursacht.
- Zuweisung der internen Indizes der Obstsorten (s. Umsetzung): $O(n \log n)$
Implementierung von `map` in C++ als Rot-schwarz-Bäume³

²<https://en.cppreference.com/w/cpp/container/set>

³<https://en.cppreference.com/w/cpp/container/map>

- Umwandlung der gewünschten Obstsorten von Strings zu Integers: $O(w \log n)$
 Das Suchen in einer **map** hat logarithmische Laufzeit bezüglich der Anzahl der Obstsorten: $O(\log n)$. Das Einfügen in eine **set** hat logarithmische Laufzeit bezüglich der Anzahl der Wünsche: $O(\log w)$. Wir schätzen nach oben ab und im worst-case gilt: $w = n$, deshalb gilt für die gesamte Laufzeit: $O(w(\log n + \log w)) \in O(w(\log n + \log n)) \in O(w \log n)$. **[MB: czy to tak może być?]**
- 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: $O(\log n)$. Das Einfügen in eine **set** hat ebenfalls logarithmische Laufzeit bezüglich der Anzahl der Obstsorten: $O(\log n)$. Deshalb beträgt die Laufzeit für alle Obstsorten in einer Spießkombination höchstens: $O(n \log n)$.
- Verarbeitung der Spießkombinationen

TODO: dodać coś o bitsetach

 - Verarbeitung einer Spießkombination:
 Man geht davon aus, dass eine Spießkombination im worst-case alle Obstsorten enthält.
 - * Erstellung der Bitmaske bf : $O(n)$ (worst-case)
 - * Erstellung der Bitmaske bn : $O(w)$ (worst-case)
 - Kopieren der Adjazenzmatrix in **Graph G** (s. Umsetzung):
- Prüfung der Korrektheit der Eingabe
- Prüfung der Existenz einer Lösung

1.7 Speicherplatz

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 hinreichen.

Die Obstsorten werden als Strings eingelesen, aber in der Methode `readFile()` wird jeder Obstsorte ein interner Index⁴ 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>`, also ein Tupel entsprechend aus der Menge der internen Indizes der Obstsorten und der Menge der Indizes aus der Aufgabenstellung. Alle Spießkombinationen werden in einem **vector** gespeichert, der `infos` heißt.

In `wishes` werden als ein **set** von Integers werden die internen Indizes der gewünschten Obstsorten, also der Elemente der Menge W , gespeichert. Analog werden in `result`, also ebenfalls ein **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 Mächtigkeit der entsprechenden Menge A . Diese Markierung zeigt sich bei der Prüfung auf korrekte Eingabe hilfreich.

Die Methode `analyzeInfo()` nimmt als Argument 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 `fruits` aus dieser Spießkombination gleichzeitig mit den Obstsorten 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.

TODO: `checkCoherence()`

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 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.

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, um die Obstsorten einer Zusammenhangskomponente zu speichern, die eine nicht gewünschte Obstsorte enthält.

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` 0 steht, d.h., der Knoten gehört zu einer Zusammenhangskomponente, die noch nicht bearbeitet wurde. Falls ja, dann werden zwei Listen `setB` und

⁴Nummerierung ab 0.

`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 aufgrund von einzigen Obstsorten die Menge W' nicht eindeutig bestimmt werden kann.

Am Ende, nach der Schleife über `multi`, 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; `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

TODO: dodać dodatkowe przykłady

3.1 Beispiel 0 (Aufgabenstellung)

Textdatei: `spiesse0.txt`

Apfel, Brombeere, Weintraube

1, 3, 4

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

unlösbar: Litschi gehört zur Komponente mit Grapefruit. Dabei ist Grapefruit kein Wunsch.

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

unlösbar: Apfel, Grapefruit und Xenia gehören zur Komponente mit Litschi. Dabei ist Litschi kein Wunsch. Ugli gehört zur Komponente mit Banane. Dabei ist Banane kein Wunsch.

3.9 Beispiel 8

Textdatei: `spiesse8.txt`

```
4
Dattel Apfel Banane
3
1 2 3
Apfel Banane Dattel
1 2
Apfel Banane
1 3
Clementine Dattel
```

Wünsche: Dattel, Apfel, Banane

Error: Es gibt Fehler in der Eingabedatei.

Die erste Spießkombination legt fest, dass Apfel, Banane und Dattel einen der folgenden Indizes besitzen: $\{1, 2, 3\}$. Das bedeutet auch, dass Clementine — die einzelne übrige Obstsorte — den Index 4 besitzt. Jedoch widerspricht dieser Zuweisung die dritte Spießkombination, laut der Clementine den Index 1 oder 3 hat.

4 Quellcode

`./tex/spiesse.m`