Aufgabe 2: Spießgesellen

Teilnahme-ID: 55627

Bearbeiter dieser Aufgabe:
Elia Doumerc

18. April 2021

Inhaltsverzeichnis

A. Hilfe für Donald

Man kann die Informationen als Mengen darstellen, wobei die Buchstaben die Initialen der Obstsorten sind (Die Brombeeren wurden mit dem Buchstaben C abgekürzt.):

$$Micky = \{A, B, C, 1, 4, 5\} \qquad Minnie = \{B, P, W, 3, 5, 6\}$$
$$Gustav = \{A, C, E, 1, 2, 4\} \qquad Daisy = \{E, P, 2, 6\}$$

Interessant ist die Schnittmenge von jeder möglichen Mengenkombination. Im Fall von Mickey und Minnie sieht sie folgenderweise aus:

$$Micky \cap Minnie = \{B, 5\}$$

Beide haben aus der Schüssel 5 ein Stück Obst genommen, und beide haben ein Stück Banane am Spieß. Da sie ansonsten an völlig verschiedenen Schüsseln waren und auch verschiedene Oststücke haben, müssen in Schüssel 5 Bananenstücke liegen.

Auch bei Minnie und Daisy kommt man auf Ergebnisse.

$$Daisy \cap Minnie = \{P, 6\}$$

Die Differenzmenge von Minnie nach Abzug von Daisy und Micky lässt zudem darauf schließen, dass die Weintrauben in Schüssel 3 sind.

$$(Minnie \setminus Daisy) \setminus Micky = \{W, 3\}$$

Dies ist aber nicht immer eindeutig, wie der Fall der Schnittmenge von Mickey und Gustav zeigt.

$$Micky \cap Gustav = \{A, C, 1, 4\}$$

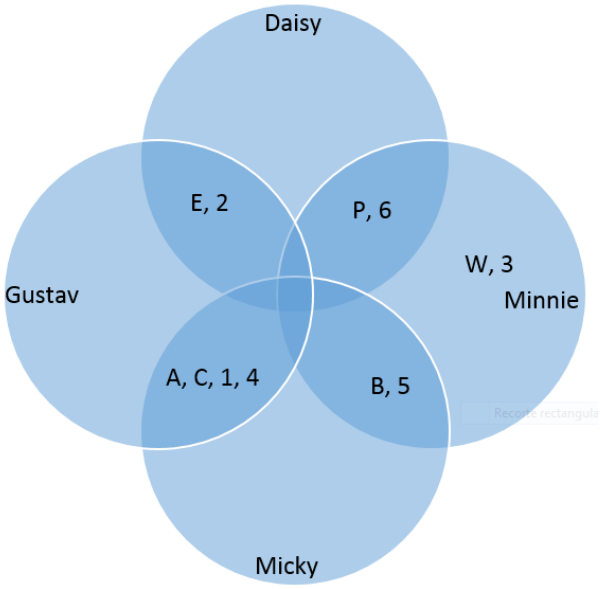
Man kann hier nicht klar erkennen, in welcher der beiden Schüsseln Äpfel sind und in welcher Brombeeren. Zudem sind alle Schnittmengen aus drei Mengen leer, können hier also nicht weiter helfen. Es bildet sich, wenn man alle Schnittmengen betrachtet, folgendes Mengendiagramm:

Jeder Kreis steht für eine Menge und dort, wo sich zwei oder mehr Kreise überlappen, sind Schnittmengen. Es lassen sich zwar nicht die Schnittmengen von Micky und Daisy und die von Gustav und Minnie darstellen (immer die gegenüberliegenden), sie sind aber glücklicherweise leer und deswegen nicht weiter von Bedeutung.

Man kann trotzdem gut erkennen, welche Obstsorten sich eindeutig einer Schüssel zuordnen lassen und welche nicht:

- Bananen → Schüssel 5
- Erdbeeren → Schüssel 2
- Pflaumen → Schüssel 6
- Weintrauben → Schüssel 3

Abbildung 1: Mengendiagramm



Nur bei den Äpfeln und Brombeeren bleiben noch Zweifel. Dies ist aber für Donald nicht schlimm, da er beide möchte und wir sicher wissen, dass diese sich in den Schüsseln 1 und 4 befinden. Donald sollte sich also aus den Schüsseln 1, 3 und 4 bedienen.

B. Programmdokumentation

B.1. Lösungsidee

Wie im Aufgabenteil a beschrieben wurde, ist es das Ziel, möglichst viele Werte durch das Betrachten verschiedener Schnittmengen und Restmengen zu isolieren. So kann man Schüsseln eine Obstsorte zuordnen. Offen bleibt noch die Frage, wie man diese Mengen möglichst effizient erstellen und darstellen kann, da wir vorhin gesehen haben, dass nicht alle gebraucht werden.

Aus der Mengenlehre wissen wir, dass die Potenzmenge $P(X)$ einer Menge X mit n Elementen [1] 2^n Elemente enthält. Die Zahl aller Schnittmengen von X entspricht der ihrer Teilmengen. [2] Beobachtet Donald also zum Beispiel 10 Spieße, gibt es schon $2^{10} = 1024$ Möglichkeiten, diese Spieße zu kombinieren. Dazu kommt, dass es für jede Schnittmenge mindestens zwei Restmengen gibt, die auch noch in Betrachtung gezogen werden müssen. Folglich scheint es keine gute Idee zu sein, ein Programm zu schreiben, dass alle möglichen Schnitt- und Restmengen generiert und überprüft, ob irgendwelche Schüsselnummern und Obststücke isoliert worden sind.

Der Lösungsweg, den ich für diese Aufgabe verwendet habe, ist wie folgt: Da eine Schüssel einer Obstsorte zugeordnet wird, kann man solche Zuordnungen als Kanten zwischen beiden sehen. Eine Matrix *kanten* besteht aus Spalten, die einer Obstsorte entsprechen, und Zeilen, die jeweils zu einer Schüsselnummer gehören. Das Feld a_{ij} enthält so Information über den Zustand von der Kante, die die Schüssel i mit der Obstsorte j verbindet. Ein Feld kann entweder den Wert *wahr*, wenn die Kante möglich ist, oder $falsch$ haben, wenn die Kante unmöglich ist.

Zu Beginn haben alle Felder den Wert *wahr*, da wir nicht wissen, welche Sorten sich in welchen Schüsseln befinden. Das Ziel ist, so viele Felder wie möglich auf $falsch$ zu setzen, damit wir eine möglichst eindeutige Aussage über die Position der Obstsorten geben können. Dazu werden wie im Algorithmus 1 die Spieße eingelesen und bearbeitet.

Algorithmus 1 Kanteneliminierung

```
    procedure ANALYSIERE SPIESS(S)                    ▷ S ist ein Spieß.
2:      spSch ← {x|x ∈ S ∧ x ∈ schuesselnummern}
        spObst ← {x|x ∈ S ∧ x ∈ obstsorten}
4:      andereSch ← alleSch \ spSch        ▷ Alle Schüsselnummern, die nicht im Spieß sind.
        andereObst ← alleObst \ spObst          ▷ Alle Obstsorten, die nicht im Spieß sind.
6:
        for i in spSch do
8:          for j in andereObst do
                kanten[i][j] ← falsch
10:         end for
        end for
12:     for i in andereSch do
            for j in spObst do
14:             kanten[i][j] ← falsch
            end for
16:     end for
    end procedure
```

Alle Kanten, die durch das Betrachten des Spießes unmöglich werden, müssen auf $falsch$ gesetzt werden. Dies sind alle, denen Kanten im Spieß ähnlich sind. [3] „Ähnlich“ bedeutet, dass genau einer der zwei Knoten im Spieß vorhanden ist. Sind stattdessen die zwei Knoten im Spieß vorhanden, handelt es sich um eine mögliche Zuordnung. Ist keiner der Knoten vorhanden, können wir (noch) nichts über die Kante sicher wissen.

[1] In unserem Fall sind das die Spieße, die ab jetzt als Mengen, bestehend aus Obstsorten und Schüsselnummern, betrachtet werden.
[2] Ich vernachlässige hier die leere Menge.
[3] Mit Kanten im Spieß sind alle Kanten gemeint, die sich durch das Kombinieren von Schüsselnummern und Obstsorten des Spießes erstellen lassen.

Im Algorithmus, der für jeden Spieß ausgeführt wird, werden deswegen zunächst alle Schüsselnummern und alle Obstsorten, die nicht im Spieß sind, bestimmt. In Folge werden alle ähnliche Kanten blind auf *falsch* gesetzt. Diese findet man, indem man durch die Schüsseln im Spieß iteriert, und in dieser Schleife wiederum durch alle Obstsorten, die sich nicht im Spieß befinden, geht. Dasselbe passiert mit den Schüsselnummern, die sich nicht im Spieß befinden, und den Obstsorten im Spieß.

Hier ist eine mögliche Matrix *kanten* zu sehen, links in Startform und rechts nach Verarbeitung des Spießes $\{1, B\}$ (B ist hier eine Obstsorte, und die zu ihr führenden Kanten befinden sich in der zweiten Spalte).

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Sobald alle Spieße verarbeitet wurden, muss man durch jede Zeile der Matrix gehen und die Kanten finden, die noch möglich sind. Diese können dann ausgegeben werden.

Die Laufzeit des Programms wächst mit der Zahl der Elemente in einem Spieß quadratisch, liegt also bei $O(S * n^2)$, wobei S die Anzahl der Spieße ist. Mehr dazu gibt es im nächsten Abschnitt.

B.2. Implementierung und Laufzeitbestimmung

Ich habe das Programm in C++ mit Hilfe der Klassen der Standard-Bibliothek geschrieben. Für die Ein- und Ausgabe habe ich außerdem einige Klassen der *Boost*-Bibliothek verwendet.

B.2.1. Datenstrukturierung

Der Hauptbestandteil des Programms ist die Klasse `Graph`, die den zweidimensionalen Vektor `kanten` und Funktionen zu dessen Bearbeitung enthält. Die Funktion `main` ruft mit der eingelesenen Sortenzahl n den Konstruktor von `Graph` auf, der unter anderem `kanten` erstellt und alle Felder (es sind n^2) davon auf `true` setzt. Hier werden auch die Listen `schuesseln` und `sorten` erstellt, die einfach Zahlen von 0 bis $n-1$ enthalten.

Danach werden in der Funktion `setze_praeferenzen` Donalds Präferenzen in einem `unordered_set` gespeichert. Das ist eine Menge, deren Elemente *gehasht* werden, was die Suche nach einem Element in konstanter Zeit ermöglicht (dies wird in den Ausgabefunktionen benötigt). Die Laufzeit der bisherigen Prozesse ist vernachlässigbar. [4]

Sowohl in diesem Schritt als auch bei der Einlesung der Spieße wird für die Obstsorten eine *bidirektionale Map* `Aliasse` angelegt, [5] in der zu jedem Namen eine einzigartige Zahl zwischen 0 und $n-1$ gehört und so gut in Kombination mit `kanten` verwendet werden kann. Das Programm ist so völlig unabhängig von den Obstsortennamen (siehe B.2.4). Bei den Schüsselnamen gehe ich allerdings davon aus, dass es sich um Zahlen im Bereich zwischen 1 und n handelt. Andere Werte führen zum Abbruch des Programms. Das Einsetzen eines Elements in `Aliasse` dauert $O(2e)$, wobei e die aktuelle Anzahl von Elementen in `Aliasse` ist.

[4] Dies zeigen auch die Ergebnisse der integrierten Zeitmessung. Siehe dazu den Abschnitt B.3

[5] Ich habe hierfür die Klasse `Boost.Bimap` verwendet.

B.2.2. Spießverarbeitung

In der Funktion `add_spieß(string zahlen, string namen)` werden für jeden Spieß beide Zeilen als Eingabe genommen und aufgetrennt. Für die Sortennamen wird die vorhin beschriebene Ersetzung durchgeführt. Darauf kommt die Information in zwei Vektoren, die am Ende der Funktion aufsteigend sortiert werden.

Ein Spieß hat l (mit $l \leq n$) Sorten. Das Füllen von `Aliasse` dauert in Folge $O(l*2e)$, wobei e die Zahl der Elemente in `Aliasse` zum Zeitpunkt des Einsetzens ist und deswegen durchschnittlich gilt: $e = \frac{n}{2}$. Das anschließende Sortieren der zwei Vektoren braucht $2 * O(l * log(l))$ Schritte. Insgesamt kommt man so auf $O(l * n + 2l * log(l))$.

Danach werden diese neuen Vektoren der Funktion `eliminiere_kanten(vector<int>* zahlen, vector<int>* namen)` übergeben. Sie erstellt mit der Funktion `std::set_difference()` für beide Vektoren die in der Lösungsidee beschriebenen Restmengen. Dies hat eine Laufzeit von $2 * O(2 * (l + n))$ (l ist die Vektorlänge, also höchstens n). Die durch die Kombination der verschiedenen Mengen entstehenden Werte werden in der zugehörigen Position von `kanten` auf `false` gesetzt. Hierfür liegt die Laufzeit bei etwa $O(2 * (n - l) * l)$.

B.2.3. Erweiterung: verteilte Obstsorten

Bis jetzt sind wir immer davon ausgegangen, dass es genauso viele Schüsseln wie Obstsorten gibt. Was wäre aber, wenn Minnie auf einmal beschließt, mehr Schüsseln aufzustellen, und die Sorten darauf zu verteilen? Für so einen Fall hat das Programm die Option `--mehrere` oder `-m`. Damit sie richtig funktioniert, muss in der Eingabedatei nach der Sortenzahl in eine neue Zeile die Schüsselzahl geschrieben werden (siehe die Beispiele in B.3).

Um die korrekten Zuordnungen zu finden, wird eine etwas veränderte Form der Funktion `eliminiere_kanten` ausgeführt. Und zwar werden hier *nicht* die Kanten auf `false` gesetzt, die von einer Obstsorte auf Schüsseln außerhalb des Spießes führen. Allerdings reicht dies nicht aus, da nicht alle unmöglichen Kanten entfernt werden. Zum Beispiel kann das Programm nur mit dieser Funktion bei einer Datei mit 3 Schüsseln, 2 Sorten und den Spießen $S_1 = \{A, B, 1, 2\}$ und $S_2 = \{B, 2, 3\}$ nicht mit Sicherheit sagen, welche Sorte sich in der Schüssel 2 befindet. Der Computer weiß nämlich nicht, dass jede Sorte in mindestens einer Schüssel sein muss.

Das Programm wird mit der Funktion `void eliminiere_Kanten_2`, die nach der Verarbeitung aller Spieße aufgerufen wird, ergänzt. Sie überprüft für jede Sorte, ob sie zu mehr als einer Schüssel zugeordnet werden kann. Wenn nicht, dann wird die einzige mit ihr verbundene Schüssel für sie festgelegt. Das heißt, alle anderen Kanten, die zu dieser Schüssel führen, werden auf `false` gesetzt. Dies wird so lange wiederholt, bis alle Zuordnungen eindeutig sind oder so oft, wie es Sorten gibt. Die Laufzeit beträgt im schlechtesten Fall etwa $O(n^2 * S)$, wobei S die Schüsselzahl ($S > n$) ist. Im besten Fall ist sie $\Omega(n * S)$.

B.2.4. Ausgabe

Für die Ausgabe der Dateien habe ich zwei Funktionen implementiert. `void drucke_resultate_v()` sorgt für eine verbose Ausgabe, d.h., sie iteriert durch jede Zeile von `kanten` (entspricht einer Schüssel) und druckt die Schüsselnummer und die Aliasse aller Kanten mit dem Wert `true` in der Zeile. Ist beispielsweise die Kante `Kanten[0][6]` möglich, wird der Text „Schüssel 1: `aliasse.at(6)`“

gedruckt. Letzteres wird durch den Namen der Obstsorte ersetzt, der in `aliasse` dem Schlüssel 6 entspricht. Wird für den Schlüssel kein Wert in `aliasse` gefunden, gilt die Sorte als *unbekannt*. `drucke_resultate_v` hat eine Laufzeit $O(n^2)$.

Die Funktion `void drucke_resultate()` ist dagegen bei der Ausgabe nicht so ausführlich, beantwortet die Aufgabenstellung aber präziser. Dazu geht sie zwar auch alle Zeilen von `kanten` durch und sucht die möglichen Kanten heraus, unterscheidet aber zwischen verschiedenen Fällen:

- Sind alle möglichen Obstsorten für eine Schüssel auch Wunschsorten, werden sie in die erste Ausgabezeile gedruckt, die die Form `Schüssel(n) {x, y, z}` hat. Dies sind alle Schüsseln, aus denen Donald sich bedienen sollte.
- Gibt es unter den möglichen Sorten welche, die zu vermeiden sind, wird die Schüsselnummer mit allen Möglichkeiten wie in `drucke_resultate_v` gedruckt.
- Gibt es unter den möglichen Sorten keine Wunschsorten, wird für diese Sorte nichts gedruckt.

Die Komplexität von `drucke_resultate()` ist gleich der von `drucke_resultate_v`. In der Praxis sollte die erste allerdings etwas schneller sein, da weniger Ausgaben getätigt werden müssen. Zu beiden Funktionen gibt es im Abschnitt B.3 Beispielausgaben.

B.2.5. gesamte Laufzeit

Fasst man die Laufzeit aller Schleifen zur Spießverarbeitung zusammen (Spießzahl $*$[(`add_spieß`) $*$ (`eliminiere_kanten`)]+ Ausgabe) , erhält man folgende Gleichung:

$$\begin{aligned} &O(S * [(l * n + 2l * log(l)) + (2 * 2 * (l + n) + (2 * (n - l) * l))] + n^2) \\ =&O(S * [l * n + 2l * log(l) + 2ln - 2l^2 + 4l + 4n] + n^2) \end{aligned}$$

Im Worst-Case-Szenario ist die durchschnittliche Spießlänge $l = n$. Die daraus entstehende Laufzeit ist quadratisch.

$$\begin{aligned} &O(S * [n^2 + 2n * log(n) + 2n^2 - 2n^2 + 4n + 4n] + n^2) \\ =&O(S * [n^2 + 8n + 2n * log(n)] + n^2) \end{aligned}$$

Im Best-Case-Szenario ist die durchschnittliche Spießlänge $l = 1$. In Folge gilt:

$$\begin{aligned} &\Omega(S * [n + 2 * log(1) + 2n - 2 + 4 + 4n] + n^2) \\ =&\Omega(S * [7n + 2] + n^2) \end{aligned}$$

Sieht man die durchschnittliche Spießlänge l als $\frac{n}{2}$, kommt man auf:

$$\begin{aligned} &\Theta(S * [\frac{n^2}{2} + n * log(\frac{n}{2}) + 6n + \frac{n^2}{2}] + n^2) \\ =&\Theta(S * [n^2 + 6n + n * log(\frac{n}{2})] + n^2) \end{aligned}$$

Dies lässt sich, bei optimistischer Betrachtung, auf $\Theta(S * n^2)$ reduzieren. Zieht man auch die Funktion `eliminiere_kanten_2` in Betracht, werden höchstens $O(S * n^2)$ addiert.

B.3. Beispiele

Das Programm druckt einen Hilfetext, falls die Argumente nicht stimmen oder die Option `--help` angegeben wurde.

```
elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 -h
Nutzung: Aufgabe2 Eingabedatei [Optionen]
Erlaubte Optionen:
  -h [ --help ]         Erstellt diese Ausgabe
  -f [ --file ] arg     Die Datei mit der Eingabeinformation
  -v [ --verbose ]      Erstellt die verbose Ausgabe.
  -z [ --zeit ] arg     Ausgabedatei für Benchmarks. Werden standardmäßig
                        nicht ausgegeben.
  -m [ --mehrere ]      Erlaubt das Vorkommen einer Obstsorte in mehreren
                        Schüsseln. Für die korrekte Funktionsweise muss in
                        der Eingabedatei die Schüsselzahl in eine neue Zeile
                        unter die Sortenzahl geschrieben werden.
```

Das Pflichtargument ist die Eingabedatei. Das Programm lässt sich so auf die Beispieleingaben anwenden, wie die Abbildung 3 zeigt. Die unerwünschten Sorten werden in roter Farbe ausgegeben.

```
elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesse1.txt
Schüssel(n): { 1, 2, 4, 5, 7 }

elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesse2.txt
Schüssel(n): { 1, 5, 6, 7, 10, 11 }

elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesse3.txt
Schüssel(n): { 1, 5, 7, 8, 10, 12 }

Diese Schüsseln können nicht eindeutig einer Wunschsorte zugewiesen werden:
Schüssel 2: Litschi, Grapefruit
Schüssel 11: Litschi, Grapefruit

elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesse4.txt
Schüssel(n): { 2, 6, 7, 8, 9, 12, 13, 14 }

elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesse5.txt
Schüssel(n): { 1, 2, 3, 4, 5, 6, 9, 10, 12, 14, 16, 19, 20 }

elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesse6.txt
Schüssel(n): { 4, 6, 7, 10, 11, 15, 18, 20 }

elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesse7.txt
Schüssel(n): { 5, 6, 8, 14, 16, 17, 23, 24 }

Diese Schüsseln können nicht eindeutig einer Wunschsorte zugewiesen werden:
Schüssel 3: Apfel, Grapefruit, Xenia, Litschi
Schüssel 10: Apfel, Grapefruit, Xenia, Litschi
Schüssel 18: Ugli, Banane
Schüssel 20: Apfel, Grapefruit, Xenia, Litschi
Schüssel 25: Ugli, Banane
Schüssel 26: Apfel, Grapefruit, Xenia, Litschi
```

Die verbose Ausgabe sieht bei der Eingabedatei `spiesse3.txt` zum Beispiel so aus:

```
elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesse3.txt -v
Öffne die Datei spiesse3.txt...
Schüsssel 1: Himbeere
Schüsssel 2: Litschi Grapefruit
Schüsssel 3: Orange
Schüsssel 4: Nektarine
Schüsssel 5: Clementine
Schüsssel 6: Apfel Banane
Schüsssel 7: Feige Ingwer
Schüsssel 8: Erdbeere
Schüsssel 9: Johannisbeere
Schüsssel 10: Feige Ingwer
Schüsssel 11: Litschi Grapefruit
Schüsssel 12: Kiwi
Schüsssel 13: Dattel
Schüsssel 14: Apfel Banane
Schüsssel 15: Unbekannt
```

Die Datei `spiesseX.txt` ist für das Programm kein Problem, obwohl sie beliebige Obstsortennamen enthält.

```
elia@elia-ubuntu:~/Aufgabe2$ cat spiesseX.txt
3
Apfel
2
1 2
Apfel Ananas
2 3
Ananas %&/(13
```

```
elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesseX.txt
Schüssel(n): { 1 }

elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesseX.txt -v
Öffne die Datei spiesseX.txt...
Schüsssel 1: Apfel
Schüsssel 2: Ananas
Schüsssel 3: %&/(13
```

Hier noch eine Datei, die sehr einfach ist und die Funktionsweise der Option `--mehrere (-m)`, die die Verteilung einer Sorte auf mehrere Schüsseln erlaubt, verdeutlichen soll.

```
elia@elia-ubuntu:~/Aufgabe2$ cat spiesseZ.txt
3
4
Apfel
2
4 2
Apfel Banane
3 2
Banane
```

```
elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesseZ.txt -m
Schüssel(n): { 4 }

elia@elia-ubuntu:~/Aufgabe2$ ./Aufgabe2 spiesseZ.txt -vm
Öffne die Datei spiesseZ.txt...
Schüsssel 1: Unbekannt
Schüsssel 2: Banane
Schüsssel 3: Banane
Schüsssel 4: Apfel
```

Schließlich ist hier die Ausgabe der integrierten Zeitmessung (Ich habe sie implementiert, bevor ich das nützliche Profiling-Tool Callgrind kannte) nach Ausführung des Befehls `./Aufgabe2 spiesse7.txt -z Zeit.txt` zu sehen.

```
elia@elia-ubuntu:~/Aufgabe2$ cat Zeit.txt

spiesse7.txt ============= Mon Mar 29 10:10:50 2021
Dateieinlesung: 25 us
Konstruktor: 2 us
Preferenzen: 14 us
Spießverarbeitung: 51 us
Ausgabe: 51 us
Gesamt 146 us
```

B.4. Quelltext

Der interessante Teil sind die Funktionen der Klasse `Graph`. In der Datei `main.cpp` sind eigentlich nur Befehle zur Eingabeverwaltung, Dateieinlesung und zum Aufrufen von Funktionen, die zu `Graph` gehören.

Konstruktor

```
// Füllt Donalds Präferenzen
void Graph::setze_praeferenzen(string* text){
    stringstream strstm(*text);
    string tmp;
    while (strstm >> tmp)
    {
        // obst_aliasse ist ein bidirectional map <int, string>
        obst_aliasse.insert({obst_aliasse.size(), tmp});
        int x = obst_aliasse.right.at(tmp);
        praeferenzen.insert(x);
    }
}
```

Funktionen zur Kanteneliminierung

```
// Setzt unmögliche Kanten auf 0
void Graph::eliminiere_kanten(vector<ushort>* zahlen, vector<ushort>* namen){
    vector<ushort> andere_zahlen, andere_namen;

    // Falls nicht mehrere Schüsseln pro Sorte erlaubt sind:
    if (!mehrere){
        // alle Schüsseln, die nicht in der betrachteten Menge sind (kommen in
    andere_zahlen)
        set_difference(schuesseln.begin(), schuesseln.end(), (*zahlen).begin(),
    (*zahlen).end(),
                       inserter(andere_zahlen, andere_zahlen.end()));
        for (auto i : andere_zahlen){
```

```
            for (auto j : *namen) kanten[i][j] = false;
        }
    }

    // alle Sorten, die nicht in der betrachteten Menge sind (kommen in andere_namen)
    set_difference(sorten.begin(), sorten.end(), (*namen).begin(), (*namen).end(),
                   inserter(andere_namen, andere_namen.end()));

    // setze alle ähnlichen Kanten auf 0
    for (auto i : *zahlen)
    {
        for (auto j : andere_namen) kanten[i][j] = false;
    }
}

// Nur für die Option mit mehreren Schüsseln pro Sorte
void Graph::eliminiere_kanten2(){
    int einzig_moeglich; bool contains_uncertain = true;

    // Wiederhole solange es nicht eindeutige Zuordnungen gibt oder
    // oft genug um sicher zu sein, dass die maximale Gewissheit erreicht wurde.
    for (int count = 0; count < n_sorten && contains_uncertain; count++)
    {
        contains_uncertain = false;

        // Für jede Sorte
        for (auto i : sorten)
        {
            einzig_moeglich = -1; // zurücksetzen
            // Überprüfe, ob es für eine Sorte mehrere mögliche Schüsseln gibt.
            for (auto j : schuesseln)
            {
                if (kanten[j][i]){
                    if (einzig_moeglich < 0){
                        einzig_moeglich = j;
                    }
                    else{
                      einzig_moeglich = -1;
                      contains_uncertain = true;
                      break;
                    }
                }
            }
            if (einzig_moeglich < 0) continue;
            // Wenn ja, setze Alle anderen Kanten, die mit der Schüssel verbunden sind, auf 0
            for (auto k : sorten)
            {
                if (k == i) continue;
                kanten[einzig_moeglich][k] = false;
            }
```

```
        }
    }
}
```

Standard-Ausgabe

```
void Graph::drucke_resultate(){
    string ausg_eind, ausg_neind, zeile_neind, name_sorte;

    for (auto i : schuesseln)
    {
        bool nur_pref = true, keine_pref = true; // lieber true oder false?
        zeile_neind = ("Schüssel " + to_string(i+1) + ": ");

        for (auto j : sorten)
        {
            if (!kanten[i][j]) continue; // Falls Zuordnung nicht möglich, weiter

            // Differenziere zwischen Wunschsorten und anderen
            if (praeferenzen.find(j) == praeferenzen.end()){
                nur_pref = false;
                try{
                    // Wenn die Sorte nicht in einem der Spieße war, kann dies zu Fehlern führen.
                    name_sorte = obst_aliasse.left.at(j);
                } catch(...){
                    name_sorte = "Unbekannt";
                }
                zeile_neind += (ROT + name_sorte + RESET + ", ");
            }
            else{
                keine_pref = false;
                zeile_neind += (obst_aliasse.left.at(j) + ", ");
            }
        }
        if (nur_pref) ausg_eind += (to_string(i+1) + ", ");
        else if (!keine_pref){
            zeile_neind.resize(zeile_neind.size() - 2);
            ausg_neind += (zeile_neind + "\n");
        }
    }
    if (ausg_eind.size() > 1){
        ausg_eind.resize(ausg_eind.size()-2); // entfernt das letzte Komma
        cout << "Schüssel(n): { " << ausg_eind << " }\n\n";

    }
    if (ausg_neind.size() > 0) cout << "Diese Schüsseln können nicht eindeutig einer Wunschsorte zugewiesen werden:\n"
                                    << ausg_neind << "\n";
    return;
}
```