Bundeswettbewerb Informatik 39 Runde 2 Aufgabe 2

Spießgesellen

Dokumentation

Jakov David Wallbrecher

19.4.2021

Inhaltsverzeichnis

Aufgabe Beispiel

Bevor ich die Funktionsweise meines Computerprogramms erkläre möchte ich meine Lösung der Teilaufgabe a vorstellen. Die Vorgangsweise die ich beim manuellen Lösen verwendet habe, habe ich auch auf das Programm übertragen. Meine Grundlegende Idee ist, dass die Menge an Elementen (Obstsorten und Schüsselnummern), die in zwei Spießen enthalten sind, eine feinere Zuordnung ermöglichen.

1. Vergleiche Micky und Gustav: Apfel, Brombeere -> 1,4
2. Bei Micky ist nur noch Banane übrig: Banane -> 5
3. Bei Gustav ist nur noch Erdbeere übrig: Erdbeere -> 2
4. Wenn bei Daisy Erdbeere und 2 zugeordnet ist gilt: Pflaume -> 6
5. Bei Minnie ist nun nur noch Weintraube nicht-zugeordnet: Weintraube -> 3
6. Wunschschüsseln sind also 1, 4 und 3.

Micky: Apfel, Banane, Brombeere -> 1, 4, 5
Minnie: Banane, Pflaume, Weintraube -> 3, 5, 6
Gustav: Apfel, Brombeere, Erdbeere -> 1, 2, 4
Daisy: Erdbeere, Pflaume -> 2, 6

Wunsch: Weintraube, Brombeere, Apfel -> 1, 4, 3

Lösungsidee

Meine Lösungsidee war es, alle Spieße jeweils miteinander zu vergleichen aus den übereinstimmenden Obstsorten und Schüsselnummern neue, kleinere Spieße zu bilden. Die zugrundeliegende Idee ist, dass sich zwei Spieße, die mehrere gleiche Obstsorten beinhalten, auch in den Schüsselnummern überschneiden müssen. Nun kann man diese Schnittmengen einander zuordnen (bzw. einen neuen Spieß erstellen). Der Erkenntnisgewinn liegt nun darin, dass man eine kleinere Untergruppe mit einer spezifischeren Sorte->Schüssel Zuordnung hat (während im beobachteten Spieß mit 5 beobachteten Sorten 25 Sorte-Schüssel-Kombinationen möglich sind, sind bei einer Aussonderung von 2 Sorten auf einen neuen Spieß nur noch 9+4 Kombinationen möglich).

Spieß A	
Apfel	6
Erdbeere	3
Clementine	5
Pflaume	9
Birne	1

Spieß B	
Himbeere	8
Kiwi	9
Birne	3
Orange	7
Apfel	4

->

Spieß A	
Erdbeere	6
Clementine	5
Pflaume	1

Spieß B	
Himbeere	8
Kiwi	7
Orange	4

Schnittspieß	
Apfel	3
Birne	9

Wenn die Spieße so weit wie möglich reduziert sind, kann man aus dieser Spießzuordnungen den gewünschten Spieß kombinieren.

Dabei muss außerdem noch der Fall ausgeschlossen werden, dass nicht-beobachtete Sorten gewünscht sind.

Umsetzung

Ich habe mich für die Implementierung in der objektorientierten Programmiersprache C# entschieden.

Ich möchte den Programmablauf chronologisch beschreiben.

Anfangs allerdings einige Bemerkungen zur Programmstruktur. Weil ich das Problem mit verschiedenen Ansätzen gelöst habe (siehe Erweiterungen), die sich einige Funktionsweisen teilen, habe ich die Basisklasse `basisAlgorithmus` verwendet. Sie beinhaltet das Zusammensetzen des Wunschspießes und das Finden der unbeobachteten Spieße (eine ausführliche Erklärung folgt später) und einige Methoden zur Ergebnisausgabe in der Konsole.

Außerdem habe ich die Klasse Spieß entworfen, die hauptsächlich der Speicherung einzelner Spieße dient. Ein Spieß besteht aus einer Liste mit Obstsorten, einer Liste mit Schüsseln und der Länge. Die Methode `printSpieß()` gibt den Spieß in der Konsole aus. Die Methode `vergleicheSpieße()` gibt sowohl die Überschneidung zwischen zwei Spießen als neuen Spieß zurück, als auch die beiden verglichenen Spieße ohne deren Überschneidungsmenge.

Der im Kapitel „Lösungsidee“ beschriebene Algorithmus wird von der Methode `spießeAufspalten()` durchgeführt. Daraufhin findet die Methode `unbeobachteteObstsortenFinden()` alle unbeobachteten Sorten und Schüsseln und fügt sie als neuen Spieß hinzu. Zum Schluss werden die Spieße in der Funktion `wunschspießZusammensetzen()` zum Wunschspieß zusammengesetzt.

spießeAufspalten()

Zuerst werden die Spieße wie beschrieben „zerkleinert“. Dazu vergleicht die Funktion jeden Spieß mit jedem anderen Spieß. Wenn sich zwei Spieße überschneiden (also gleiche Sorten und Schüsseln beinhalten) wird der Schnittspieß zur Liste aller Spieße hinzugefügt. Von den betrachteten Spießen wird die Überschneidungsmenge jeweils entfernt.

```
List<Spieß> spießeAufspalten(List<Spieß> spieße) {
        for (int i = 0; i < spieße.Count; i++) {
            for (int j = i; j < spieße.Count; j++) {
                if (i != j) {
                    (Spieß spieß2neu, Spieß schnittSpieß) =
                     spieße[i].vergleicheSpieße(spieße[j]);
                    if (schnittSpieß.length > 0) {
                        spieße[j] = spieß2neu;
                        spieße.Add(schnittSpieß);
                    }
                }
            }
        }
    spieße.RemoveAll(sp => sp.length == 0);
    return spieße;
}
```

Da Schleife durch die Länge von spieße begrenzt ist kann sichergestellt werden, dass auch Schnittspieße nochmals mit allen anderen Spießen verglichen werden.

unbeobachteteObstsortenFinden()

Da es möglich ist, dass eine gewünschte Sorte nicht beobachtet wurde, überprüft die Funktion zuerst, ob alle gewünschten Sorten beobachtet wurden. Eine solche Obstsorte unterscheidet sich insofern von Sorten die unbeobachtet und nicht gewünscht sind als das ihr Name bekannt ist (er ist schließlich im Wunschspieß aufgeführt).

Nun sucht die Funktion alle Schüsselnummern, die nicht beobachtet wurden. Abschließend gibt die Funktion einen Spieß, bestehend aus den unbeobachteten, aber gewünschten, Sorten, den unbeobachteten und nicht gewünschten Sorten und den unbeobachteten Schüsselnummern zurück. Unbeobachtete und nicht gewünschte Sorten werden dabei „unbeobachtete Obstsorte x“ genannt (x wird hochgezählt).

Es ist möglich, diesen kombinierten Spieß zu der Spieß-liste hinzuzufügen, da sie den gleichen Informationsgehalt haben wie ein beobachteter Spieß haben (es ist sicher, dass jede dieser Sorten in einer der Schüsseln enthalten ist).

wunschspießZusammensetzen()

Um den Wunschspieß zusammenzusetzen, iteriert die Funktion über alle (mittlerweile zerkleinerte) Spieße. Dabei wird überprüft, ob der betrachtete Spieß ganz gewünscht ist, oder ob nur ein Teil der Sorten gewünscht sind. Wenn alle vorkommenden Sorten gewünscht sind, werden die Schüsselnummern des beobachteten Spießes den Schüsselnummern des Wunschspießes hinzugefügt. Sollte nur ein Teil der Sorten des beobachteten Spießes gewünscht sein, wird der Spieß der Liste `spießeHalbfalsch` hinzugefügt, die gesondert ausgegeben wird.

Zum Schluss werden die Ergebnisse (alle Spieße, wunschSpieß, …) in der Konsole ausgegeben.

Erweiterungen

Tabelle

In einer ersten Erweiterung habe ich einen anderen Ansatz zum Zuordnen der Sorten zu einer Schüssel verwendet. Die Lösungsidee war hier, in einer Tabelle mit den Achsen Obstsorten und Schüsseln an jeder Position (Sorte|Schüssel) die Anzahl dieser Kombination in den Spießen zu hinterlegen. Der Schüsselwert der Felder der Tabelle, die der größte Wert einer Reihe und einer Spalte

Beispiel:

Spieß1= {[apfel, birne],[1,2]}

Spieß2={[apfel,clementine],[2,3]}

	Apfel	Birne	Clementine	
1	sp1	sp1		A1, B1
2	sp1, sp2	sp1	sp2	A2
3	sp2		sp2	A3, C3
	A2	B1, B2	C2, C3	

sind, kann nun der Obstsorte, die ihre Spalte repräsentiert, zugeordnet werden.

Dieser Algorithmus wird in der Funktion `spießeAufspalten2` umgesetzt. Abgesehen von der Funktion `spießeAufspalten2` funktioniert diese Erweiterung jedoch genauso wie der erste beschriebene Lösungsweg.

Quantencomputer

Quantencomputer gehören zu den größten technischen Innovationen des letzten Jahrzehnts. Noch sind sie nicht besonders Leistungsfähig, doch wenn die Entwicklung weiterhin in so rasantem Tempo vorangeht könnten sie schon bald einen festen Platz in der angewandten Informatik einnehmen. Einer der wenigen kommerziell vertriebenen Quantencomputer wird von der kanadischen Firma D-Wave entwickelt. Er ist kein gatterbasierter Quantencomputer, wie ihn Google, IBM und Microsoft bauen, sondern ein adiabatischer Quantenannealer. Auch jetzt ist dieser schon in der Lage, kleinere praxisrelevante Probleme zu lösen. Auch ich habe schon auf ihm gearbeitet – im Rahmen eines Jugend-forscht Projekts im Jahr 2019, mit dem ich bundesweit den vierten Platz erreicht habe. Die Forschungsarbeit hat das Interesse verschiedener Unternehmen und Forschungsinstituten erregt, darunter die Krones AG, das Max-Planck-Institut für Quantenoptik (MPQ) und der International Supercomputing Conference (ISC).

Ich habe mir die Herausforderung gestellt, auch dieses Problem von einem Quantencomputer lösen zu lassen. Dank meinem bereits erworbenen Know-How ist mir dies auch gelungen.

Um das Problem auf einem adiabatischen Quantenannealer lösen zu können, muss man es als QUBO (Quadratic Unconstrained Binary Optimization) - Problem formulieren. Man muss das Problem als ungerichteten Graph darstellen, bei dem jeder Knoten zwei Zustände annehmen kann. In diesem Fall ist jeder Knoten ein mögliches (in der Spieß-liste vorkommendes) Sorte-Schüssel Paar. Die Kanten beschreiben Belohnungen und Bestrafungen, also negative oder positive Kosten, die dann eintreten, wenn beide Knoten, die durch die Kante verbunden sind, den Wert 1 annehmen. Um dieses Problem also auf einem echten Quantencomputer zu lösen, muss man einen Algorithmus entwickeln, der auf die richtigen Kanten die richtigen Werte schreibt. Dazu muss ich zuerst einige Regeln formulieren, die ich dann später im Programm umsetzen kann.

1. Nur eine Schüssel pro Sorte (-> bestrafen, wenn zwei betrachtete Felder zur gleichen Sorte gehören)
2. Nur eine Sorte pro Schüssel (-> bestrafen, wenn zwei betrachtete Felder zur gleichen Schüssel gehören)
3. Möglichst viele Sorte-Schüssel-Kombinationen (sonst werden alle Knoten 0, weil nur bestraft wird)

Dieser Graph kann in einer Adjazenzmatrix dargestellt werden. Da diese Matrix meist recht groß ist, habe ich sie nicht hier abgedruckt, aber im Anhang als .txt Datei inkludiert. Nach dem Erstellen einer solchen Matrix iteriert das Programm über alle Felder-paare. Jedes Feld der Matrix wird so mit jedem Anderen Feld in der Matrix verglichen. Wenn beide Felder zur gleichen Sorte gehören (aber zu einer anderen Schüssel) und andersrum wird an der Stelle (Feld1|Feld2) eine Bestrafung von +2 angetragen. Zusätzlich wird in jedem Feld, das eine mögliche, in den Spießen vorkommende, Kombination aus Sorte und Schüssel darstellt, eine Belohnung von -2 eingetragen.

Um das Problem auf einem Quantenannealer, einer besonderen Art der Quantencomputer die besonders für Optimierungsprobleme ausgelegt sind, auszuführen, braucht es einen Zugang zum sogenannten „Leap Portal“ der Firma D-Wave, die die Computer schon herstellen, verkaufen und vermieten. Meinen Zugang habe ich über einen Kontakt am Forschungszentrum Jülich bekommen. Über eine Python API von D-Wave kann ich dem Quantencomputer die Matrix schicken. Ein Vorgang, bei dem der Quantencomputer versucht das Problem zu lösen, dauert per Default 20 Mikrosekunden. Da meist keine optimale Lösung gefunden wird, kann man festlegen, wie viele solcher Durchläufe gemacht werden sollen (Default: 2000). Als Ergebnis gibt die API für jeden der Durchläufe eine Reihe von Einsen und Nullen zurück. Wenn an der Stelle der Kombination Apfel|2 im Ergebnis eine 1 steht, zählt die Kombination als „akzeptiert“.

Da das Hauptprogramm in C# geschrieben ist und ich darüber aber nicht mit dem Quantencomputer kommunizieren kann, musste eine Möglichkeit her, die Matrix und die Parameter aus dem C#-Programm an ein python Skript zu übermitteln. Dies habe ich so gelöst, dass man diese Dateien (Matrix und Parameter) in einem Speicherdialog im dem Ordner ablegen muss, indem auch das python Skript liegt. Das Skript muss nun „manuell“ ausgeführt werden. Sobald es fertig ist und die Ergebnisse vom Quantencomputer in der Datei results gespeichert wurden, kann man das C#-Programm mit der Eingabetaste weiterlaufen lassen. Dort muss zuerst die soeben vom Skript gespeicherte results Datei lokalisiert und geladen werden.

Weil bei einigen Beispieldateien keine eindeutige Zuordnung von Sorte zu Schüssel möglich ist, der Quantencomputer aber versucht eine möglichst eindeutige Lösung zu finden, kombiniere ich die besten Ergebnisse miteinander. Abschließend muss das Ergebnis dekodiert werden.

An einigen Stellen greife ich im C#-Programm auf die Selbstgeschriebene Library `QA_Communication` zurück. Sie beinhaltet einige Helfer-Funktionen, die die Arbeit mit den gewonnenen Daten vereinfacht.

Um diese Erweiterung selber zu testen braucht man einen Zugangs-token vom D-Wave Leap-Portal, die im python Skript verwendeten libraries von D-Wave (installiert in einer

virtual environment, deren Pfad im python Skript angegeben wird), sowie eine Internetverbindung.

Plausibilitätsprüfung

Nachdem die Daten eingelesen werden, prüft das Programm sie auf Plausibilität. Folgende Punkte werden geprüft:

- Jeder Spieß hat die gleiche Anzahl an beobachteten Sorten wie beobachtete Schüsseln
- Jeder Spieß hat jeden Wert (Obstsorte oder Schüsselnummer) höchstens einmal

Außerdem wird in der Funktion __ überprüft, ob die Anzahl der Sorten gleich groß ist wie die Anzahl der Schüsseln des aktuellen Spießes. So kann sichergestellt werden, dass die Schnittmenge zweier Spieße in Sorten und Schüsseln gleich groß ist. Wenn das nicht der Fall ist, wird eine Fehlermeldung ausgegeben und das Programm kann manuell beendet werden.

Laufzeit

Nun will ich die (maximale) Laufzeit des Programms analysieren. Da sie von vielen Parametern abhängt (Anzahl an Spießen, durchschnittliche Länge eines Spießes, …) werde ich die einzelnen Funktionen gesondert analysieren und jeweils erst am Schluss die O-Notation vereinfachen. Um besser die tatsächliche Laufzeit abschätzen zu können, habe ich die Laufzeit der komplexeren Funktionen erst allgemein formuliert (mit Verweisen auf andere Funktionen), dann die Laufzeiten der verwiesenen Funktionen eingesetzt und vereinfacht, und erst zum Schluss für Variablen deren Obergrenzen eingesetzt.

Zusammenfassend lässt sich sagen, dass die deterministischen Algorithmen alle eine Laufzeitobergrenze von O(n) haben. Allerdings lässt sich dabei nur bedingt die faktische Dauer des Programms ableiten, da in der faktischen Laufzeit große Multiplikatoren dazukommen (gesamtobst²) und n bei „realistischen“ Spießanzahlen nicht groß genug wird um größeren Einfluss auf die Laufzeit zu haben als diese Multiplikatoren.

Interessant ist auch, dass `spießeAufspalten2()` besser skaliert als `spießeAufspalten()`, während `spießeAufspalten()` für n<3*gesamtobst^2 eine deutlich bessere Laufzeit hat.

Begriffserklärung:

`spießeNum`: Anzahl an Spießen : n

`avSpießLen`: durchschnittliche Länge eines Spießes

`wunschLen`: Länge des Wunschspießes

`resultsNum`: Anzahl an Ergebnissen vom Quantencomputer

findeLeereReihen()	gesamtobst^4
verkleinereMatrix()	gesamtobst^4
vergrößereErgebnis()	gesamtobst^2
decodiereQCErgebnis()	resultsNum + besteErgs* (vergrößereErgebnis + gesamtobst^2) + gesamtobst^2 = resultsNum*(gesamtobst^2)
Gesamtobst, avSpießLen, wunschLen	Worst case: 26

unbeobachteteObstsortenFinden()	spießeNum + wunschLen* spießeNum* avSpießLen + unbeobachteteSorten.Count + gesamtobst* spießeNum* avSpießLen = gesamtobst^2* spießeNum
spießeAufspalten()	spießeNum^2
spießeAufspalten2()	spießeNum* avSpießLen^2 + gesamtobst + gesamtobst^2 + gesamtobst* (gesamtobst + spießeNum) = spießeNum* gesamtobst^2
quantenannealing()	sp* avSpießLen^2 + gesamtobst^4 + findeLeereReihen() + verkleinereMatrix() + resultsNum + decodiereQCErgebnis() + wunschspießZusammensetzen() = sp* avSpießLen^2 + gesamtobst^4 + resultsNum* (gesamtobst^2) + spießeNum* avSpießLen* wunschLen = sp* gesamtObst^2 + resultsNum* gesamtObst^2 + gesamtObst^4
wunschspießZusammensetzen()	wunschLen + spießeNum* avSpießLen* wunschLen = spießeNum* gesamtObst^2

Ergebnisse

deterministische Ansätze

Da sowohl `spießeAufspalten()` als auch `spießeAufspalten2()` deterministische Algorithmen sind, kommen jeweils die gleichen, richtigen Ergebnisse raus. Hier ist nun zu jeder Eingabedatei die Einzelzuordnung und der Wunschspieß abgedruckt.

Datensatz 1

```
AUFGESPLITETE SPIESSE:
Banane  -> 3
Clementine  -> 1
Feige  -> 10
Ingwer  -> 6
Apfel  -> 8
Dattel  -> 9
Johannisbeere  -> 5
Erdbeere Himbeere  -> 4 2
Grapefruit  -> 7

WUNSCHSPIESS:
Clementine Erdbeere Grapefruit Himbeere
Johannisbeere  -> 1 5 4 2 7
```

Datensatz 2

```
AUFGESPLITETE SPIESSE:
Kiwi  -> 6
Litschi  -> 7
Ingwer  -> 12
Erdbeere  -> 8
Apfel  -> 1
Dattel Feige  -> 2 9
Banane Clementine Himbeere  -> 11 5 10
Grapefruit  -> 3
Johannisbeere  -> 4

WUNSCHSPIESS:
Apfel Banane Clementine Himbeere Kiwi
Litschi  -> 6 7 1 11 5 10
```

Datensatz 3

```
AUFGESPLITETE SPIESSE:
Apfel Banane  -> 14 6
Erdbeere  -> 8
Feige Ingwer  -> 7 10
Kiwi  -> 12
Clementine  -> 5
Dattel  -> 13
Himbeere  -> 1
Nektarine  -> 4
Grapefruit Litschi  -> 11 2
Johannisbeere  -> 9
Orange  -> 3
unbekannte Obstsorte 0  -> 15

WUNSCHSPIESS:
Clementine Erdbeere Feige Himbeere
Ingwer Kiwi Litschi  -> 8 7 10 12 5 1

TEILWEISE PASSENDE:
Grapefruit Litschi  -> 11 2
```

Datensatz 4

```
AUFGESPLITETE SPIESSE:
Apfel  -> 9
Clementine  -> 15
Litschi  -> 3
Dattel  -> 10
Johannisbeere  -> 5
Ingwer  -> 6
Quitte  -> 4
Feige  -> 13
Himbeere  -> 11
Nektarine  -> 7
Grapefruit  -> 8
Banane  -> 17
Pflaume  -> 12
Orange  -> 14
Kiwi  -> 2
Mango  -> 1
Erdbeere  -> 16

WUNSCHSPIESS:
Apfel Feige Grapefruit Ingwer Kiwi
Nektarine Orange Pflaume  -> 9 6 13 7 8
12 14 2
```

Datensatz 5

```
AUFGESPLITETE SPIESSE:
Nektarine  -> 14
Apfel Grapefruit Mango  -> 1 19 4
Clementine  -> 20
Erdbeere  -> 8
Dattel  -> 6
Rosine  -> 17
Pflaume  -> 10
Tamarinde  -> 12
Ingwer  -> 11
Orange Sauerkirsche  -> 16 2
Banane Quitte  -> 3 9
Johannisbeere  -> 13
Himbeere  -> 5
Kiwi Litschi  -> 15 7
Unbekannte_Obstsorte_0  -> 18

WUNSCHSPIESS:
Apfel Banane Clementine Dattel
Grapefruit Himbeere Mango Nektarine
Orange Pflaume Quitte Sauerkirsche
Tamarinde  -> 14 1 19 4 20 6 10 12 16 2
3 9 5
```

Datensatz 6

```
AUFGESPLITETE SPIESSE:
Vogelbeere  -> 6
Sauerkirsche  -> 16
Rosine Ugli  -> 11 15
Ingwer  -> 14
Tamarinde  -> 19
Himbeere  -> 18
Banane  -> 8
Litschi  -> 13
Feige  -> 22
Quitte  -> 4
Mango  -> 1
Clementine  -> 7
Erdbeere  -> 10
Grapefruit  -> 17
Johannisbeere  -> 12
Weintraube  -> 21
Orange  -> 20
Kiwi  -> 23
Nektarine  -> 2
Apfel  -> 3
Pflaume  -> 9
Dattel  -> 5

WUNSCHSPIESS:
Clementine Erdbeere Himbeere Orange
Quitte Rosine Ugli Vogelbeere  -> 6 11
15 18 4 7 10 20
```

Datensatz 7

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

WUNSCHSPIESS:
Apfel Clementine Dattel Grapefruit
Mango Sauerkirsche Tamarinde Ugli
Vogelbeere Xenia Yuzu Zitrone  -> 5 23
14 8 24 6 16 17

TEILWEISE PASSENDE:
Banane Ugli  -> 25 18
Apfel Grapefruit Litschi Xenia  -> 10
20 3 26
```

Quantencomputer

Hier die Ergebnisse des Quantencomputers. Da die aktuellen Quantencomputer eine sehr begrenzte Rechenkapazität haben, ist es ihm nur gelungen, die erste, kleinste Eingabedatei richtig zu lösen. Schon für die zweite Datei, die nur zwei Sorten mehr hat, kamen keine guten Ergebnisse mehr raus. Auch für die weiteren Beispieldateien ist es mir nicht gelungen, Parameter zu finden, die richtige Ergebnisse zur Folge haben. Nichtsdestotrotz ist es ein Machbarkeitsbeweis! Es lässt hoffen, dass zukünftige, bessere Quantencomputer diese Probleme viel besser lösen können und bei einer bestimmten Größe vielleicht sogar schneller sind.

Datensatz 1

```
SOLUTION QANTUM
a : 8
b : 3
c : 1
d : 9
e : 2,4
f : 10
g : 7
h : 2,4
i : 6
j : 5

WUNSCHSPIESS
Clementine Erdbeere Grapefruit Himbeere
Johannisbeere  -> 1 2 2 4 4 5 7

QA_ARGUMENTS:
        annealing_time      =  40
        Durchläufe    =  10000
        chain_strength      =  4
```

Datensatz 2

```
SOLUTION QANTUM
a : 1
b : 10,11
c : 8
d : 2
e :
f : 9
g : 3
h : 5
i : 12
j : 4
k : 2
l : 6,7

WUNSCHSPIESS
Apfel Banane Clementine Himbeere Kiwi
Litschi  -> 1 2 5 6 7 8 10 11

QA_ARGUMENTS:
        annealing_time   =  90
        num_reads        =  10000
        chain_strength   =  5
```

Code

Deterministischer Ansatz

```
/*ANSATZ:
/* Fall abfangen, dass eine Obstsorte gewünscht ist, die nicht beobachtet wurde.
    * Der Vollständigkeit halber werden unbeobachtete Obstsorten auch unabhängig davon ob sie
gewünscht wurden ergänzt.
    * ANSATZ:
    * 1: unbeobachtete Obstsorten sammeln (entweder mit Namen von wunschSpieß oder als
"unbeobachtetes Obst n")
    * 2: unbeobachtete Schüsselnummern sammeln
    * 3: neuer Spieß aus obstsorten und schüsseln
    * */
    /// <summary>
    /// findet noch nicht beobachtete Obstsorten und Schüsseln, die existieren müssen.
    /// </summary>
    /// <returns>erweiterte Liste mit allen Spießen</returns>
protected List<Spieß> unbeobachteteObstsortenFinden(List<Spieß> spieße) {
    //überprüfen ob alle Spieße beobachtet wurden
    int beobachteteSorten = 0;
    foreach (Spieß sp in spieße) {
        beobachteteSorten += sp.länge;
    }
    if (beobachteteSorten != gesamtObst) {
        //gewünschte Sorten, die nicht beobachtet wurden ausfindig machen
        List<string> unbeobachteteSorten = new List<string>();
        foreach (string wunschObst in orgWunschSpieß.obstSorten) {
            bool wunschObstBeobachtet = false;
            foreach (Spieß sp in spieße) {
                if (sp.obstSorten.Contains(wunschObst.ToLower()[0] + "") ||
sp.obstSorten.Contains(wunschObst)) { wunschObstBeobachtet = true; break; }
            }
            if (!wunschObstBeobachtet) { unbeobachteteSorten.Add(wunschObst); }
        }
        //sollten Sorten weder gewünscht noch beobachtet sein, aber existieren, werden sie als
unbekannte Obstsorte hinterlegt
        for (int i = 0; unbeobachteteSorten.Count < gesamtObst - beobachteteSorten; i++) {
            unbeobachteteSorten.Add("unbekannte_Obstsorte_" + i);
        }

        //Schüsselnummern, die nicht genannt wurden, aber existieren müssen, werden ausfindig
gemacht
        List<int> unbeobachteteSchüsseln = new List<int>();
        for (int i = 1; i <= gesamtObst; i++) {
            bool schüsselBeobachtet = false;
            foreach (Spieß spieß in spieße) {
                if (spieß.schüsseln.Contains(i)) { schüsselBeobachtet = true; break; }
            }
            if (!schüsselBeobachtet) { unbeobachteteSchüsseln.Add(i); }
        }

        spieße.Add(new Spieß(unbeobachteteSchüsseln, unbeobachteteSorten));
    }
    return spieße;
}
```

```
    * alle zwei Spieße werden miteinander verglichen. wenn sie gemeinsame Obstsorten/Schüsseln
haben werden diese gemeinsamen als
    * neuer Spieß gespeichert und von den ursprünglichen Spießen entfernt.
    * So kann man möglichst reduzierte Spieße erzeugen, die wiederum zum Wunschspieß
zusammengesetzt werden können.
    * */
/// <summary>
/// möglichst kleinste Obstsorte->Schüssel Zuordnungen finden
/// Ansatz: Spieße vergleichen und Schnittspieße bilden
/// </summary>
/// <returns>aufgespaltete Spieße</returns>
List<Spieß> spießeAufspalten(List<Spieß> spieße) {
    for (int i = 0; i < spieße.Count; i++) {
        for (int j = i; j < spieße.Count; j++) {
            if (i != j) {
                (Spieß spieß2neu, Spieß schnittSpieß) = spieße[i].vergleicheSpieße(spieße[j]);
                if (schnittSpieß.länge > 0) {
                    spieße[j] = spieß2neu;
                    spieße.Add(schnittSpieß);
                }
            }
        }
    }
    spieße.RemoveAll(sp => sp.länge == 0);
    return spieße;
}

/// <summary>
/// kombiniert gewünschten Spieß
/// wenn ganzer Spieß in wunschSpieß enthalten ist, werden Spieß.schüsseln zu
wunschSpieß.schüsseln hinzugefügt
/// </summary>
/// <returns>wunschSpieß mit hinzugefügten Schüsseln; Liste von Spießen die nur teilweise
gewüsncht sind</returns>
protected (Spieß wunschSpieß, List<(Spieß spieß, List<string> unpassendeSorten)>
spießeHalbfalsch) wunschspießZusammensetzen(List<Spieß> spieße) {
    List<(Spieß spieß, List<string> unpassendeSorten)> spießeHalbfalsch = new List<(Spieß spieß,
List<string> unpassendeSorten)>(); //Spieße mit nur teils gewünschten Obstsorten
    Spieß wunschSpieß = orgWunschSpieß.clone();
    char[] wunschObstChar = new char[wunschSpieß.obstSorten.Count]; //flexibler mit chars statt
strings (notwendig für Quantenannealer)
    for (int i = 0; i < wunschObstChar.Length; i++) { wunschObstChar[i] =
wunschSpieß.obstSorten[i].ToLower()[0]; }

    foreach (Spieß spieß in spieße) {
        List<string> unpassendeSorten = new List<string>();
        //nicht-gewünschte Sorten im Spieß suchen
        foreach (string obst in spieß.obstSorten) {
            if (!wunschObstChar.Contains(obst.ToLower()[0])) {
                unpassendeSorten.Add(obst);
            }
        }
        if (unpassendeSorten.Count == 0) {//ganzer Spieß gewünscht
            wunschSpieß.schüsseln.AddRange(spieß.schüsseln);
        }
        else if (unpassendeSorten.Count != spieß.länge) {//wenn ein Teil des Spießes gewünscht
ist -> spießeHalbfalsch
            spießeHalbfalsch.Add((spieß, unpassendeSorten));
        }
    }
    return (wunschSpieß, spießeHalbfalsch);
}
```

Erweiterung: Quantencomputer

```
/// <summary>
/// löst das Problem auf einem adiabatischen Quantencomputer
/// </summary>
/// <returns></returns>
public (Spieß wunschSpieß, List<Spieß> neueSpieße) quantenannealing() {
    char[] alphabetAllg = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z' };
    char[] alphabet = new char[gesamtObst];
    Array.Copy(alphabetAllg, alphabet, gesamtObst);

    float[,] matrix = new float[gesamtObst * gesamtObst, gesamtObst * gesamtObst];
    List<Spieß> neueSpieße = new List<Spieß>();
    foreach (Spieß sp in orgSpieße) { neueSpieße.Add(sp.clone()); }

    neueSpieße = unbeobachteteObstsortenFinden(neueSpieße);

    //Adjazenzmatrix befüllen
    foreach (Spieß spieß in neueSpieße) {
        for (int sch = 0; sch < spieß.schüsseln.Count; sch++) {
            for (int sor = 0; sor < spieß.obstSorten.Count; sor++) {
                int posInMatrix = (spieß.schüsseln[sch] - 1) * gesamtObst +
Array.FindIndex(alphabet, c => c == spieß.obstSorten[sor].ToLower().ElementAt(0));
                matrix[posInMatrix, posInMatrix] += -2;
            }
        }
    }
    for (int schüssel1 = 0; schüssel1 < gesamtObst; schüssel1++) {
        for (int sorte1 = 0; sorte1 < gesamtObst; sorte1++) {
            int x = schüssel1 * gesamtObst + sorte1;
            for (int schüssel2 = 0; schüssel2 < gesamtObst; schüssel2++) {
                for (int sorte2 = 0; sorte2 < gesamtObst; sorte2++) {
                    int y = schüssel2 * gesamtObst + sorte2;
                    if (x != y) {
                        //nur Qubits connecten, die beide die Möglichkeit haben besetzt zu
werden (also grundsätzlich belohnt werden beziehungsweise eine Obst-schüssel kombi darstellen,
die vorkommen könnte)
                        if (matrix[x, x] != 0 && matrix[y, y] != 0) {
                            //gleiche Spalte aka schüssel
                            if (schüssel1 == schüssel2) {
                                matrix[x, y] += 2;
                            }
                            //gleiche Reihe aka sorte
                            if (sorte1 == sorte2) {
                                matrix[x, y] += 2;
                            }
                        }
                    }
                }
            }
        }
    }
    //Matrix verkleinern (Reihen und Spalten entfernen, die leer sind)
    List<int> leereReihen = findeLeereReihen(matrix);
    matrix = verkleinereMatrix(matrix, leereReihen);

    //Parameter für Quantencomputer festelegen
    List<int>[] ergebnis = new List<int>[gesamtObst];
    try {
        Dictionary<string, string> qaArguments = new Dictionary<string, string>() {
        {"annealing_time","40"},
        {"num_reads","10000"}, //max 10000 (limitation by dwave)
        {"chain_strength","4" }
        };
        Dictionary<string, string> pyParams = new Dictionary<string, string>() {
        {"problem_type","qubo"}, //qubo //ising
        {"dwave_solver", "Advantage_system1.1"}, //DW_2000Q_6 //Advantage_system1.1
        {"dwave_inspector","true" }
        };

        string qaArgumentsString = string.Join(",", qaArguments.Select(x => x.Key + "=" +
x.Value).ToArray());
        string pyParamsString = string.Join(",", pyParams.Select(x => x.Value).ToArray());
        //daten dür python speichern
```

```
        Program.saveFile(new string[] { QA_Communication.Matrix.toQUBOString(matrix) },
"qubomatrix");
        Program.saveFile( new string[]{qaArgumentsString, pyParamsString }, "data.txt");

        Console.WriteLine("jetzt die Datei embed-and-run.py ausführen und sobald sie fertig ist
auf enter drücken"); Console.ReadLine();

        //Ergebnisse lesen
        QA_Communication.qaConstellation constellation = new
QA_Communication.qaConstellation(getResults(), qaArguments, pyParams);
        constellation.printConstellation(20);
        //constellation.plotEnergyDistribution();
        (neueSpieße, ergebnis) = decodiereQCErgebnis(constellation, leereReihen, alphabet);

        (Spieß neuWunschSpieß, List<(Spieß spieß, List<string> unpassendeSorten)>
spießeHalbfalsch) = wunschspießZusammensetzen(neueSpieße);

        //Ausgabe
        Console.WriteLine("\nSOLUTION QANTUM");
        for (int s = 0; s < ergebnis.Length; s++) {
            Console.WriteLine(alphabet[s] + " : " + string.Join(",", ergebnis[s]));
        }
        Console.WriteLine("\nWUNSCHSPIESS");
        neuWunschSpieß.printSpieß();

        QA_Communication.Program.getUserInput(constellation, matrix);

        return (neuWunschSpieß, neueSpieße);
    }
    catch (Exception e) {
        Console.WriteLine("\nERROR occured:");
        Console.WriteLine(e.Message);
        Console.WriteLine(e.StackTrace);
        return (orgWunschSpieß, orgSpieße);
    }
}



/// <summary>
/// lädt results, die vom python script gespeichert wurden
/// </summary>
/// <returns></returns>
List<(float energy, int numOccurrences, float chainBreakFraction, int[] result)> getResults() {
    string[] lines = Program.loadFile("results.txt");
    List<(float energy, int numOccurrences, float chainBreakFraction, int[] result)> results =
new List<(float energy, int numOccurrences, float chainBreakFraction, int[] result)>();
    foreach(string line in lines) {
        string[] args = line.Split('\t');
        List<int> result = new List<int>();
        string[] resultString = args[3].Split(' ');
        foreach(string num in resultString) {
            result.Add(int.Parse(num.Replace("[", "").Replace("]", "").Trim()));
        }
        results.Add((float.Parse(args[0].Replace(".", ",")), int.Parse(args[2]),
float.Parse(args[1].Replace(".",",")), result.ToArray()));
    }
    return results;
}

#endregion
```

```
#region verarbeite Ergebnis/ Matrix

/// <summary>
/// findet alle leeren Reihen/Spalten in matrix
/// </summary>
List<int> findeLeereReihen(float[,] matrix) {
    List<int> leereReihen = new List<int>();
    for (int i = 0; i < matrix.GetLength(0); i++) {
        bool reiheLeer = true;
        for (int j = 0; j < matrix.GetLength(1); j++) {
            if (matrix[i, j] != 0) {
                reiheLeer = false;
                break;
            }
        }
        if (reiheLeer) {
            leereReihen.Add(i);
        }
    }
    return leereReihen;
}

/// <summary>
/// entfernt leere Reihen & Spalten aus matrix, damit Problem für Quantencomputer besser lösbar
wird
/// </summary>
/// <param name="leereReihen">Liste aller leeren Reihen (da Dreiecksmatrix entsprechen leere
Reihen den leeren Spalten)</param>
/// <returns>verkleinerte Matrix</returns>
float[,] verkleinereMatrix(float[,] matrix, List<int> leereReihen) {
    int neueLänge = matrix.GetLength(0) - leereReihen.Count;
    float[,] neueMatrix = new float[neueLänge, neueLänge];

    int countX = 0, countY = 0;
    for (int i = 0; i < matrix.GetLength(1); i++) {
        if (!leereReihen.Contains(i)) {
            for (int j = 0; j < matrix.GetLength(0); j++) {
                if (!leereReihen.Contains(j)) {
                    neueMatrix[countX, countY] = matrix[i, j];
                    countX++;
                }
            }
            countY++;
            countX = 0;
        }
    }
    return neueMatrix;
}
```

```
/// <summary>
/// aus Ergebnis in 0 und 1 Spieße ableiten
/// </summary>
/// <param name="constellation">Konstellation der Eingaben/Ausgaben des Quantencomputers</param>
/// <returns></returns>
(List<Spieß> returnSpieße, List<int>[] solution)
decodiereQCErgebnis(QA_Communication.qaConstellation constellation, List<int> leereReihen,
char[] alphabet) {
    List<int>[] solution = new List<int>[gesamtObst]; //index entspricht sorte, value entspricht
schüssel
    for (int i = 0; i < solution.Length; i++) { solution[i] = new List<int>(); }
    List<Spieß> returnSpieße = new List<Spieß>();

    int[] ergebnisKombiniert = new int[gesamtObst * gesamtObst];
    var besteErgebnisse = constellation.getLowest(1, new List<int>()); //beste Ergebnisse finden

    //kombiniere beste Ergebnisse
    foreach (int index in besteErgebnisse) {
        int[] erweitertesErgebnis = vergrößereErgebnis(constellation.results[index].result,
leereReihen);
        for (int i = 0; i < gesamtObst * gesamtObst; i++) {
            ergebnisKombiniert[i] += erweitertesErgebnis[i];
        }
    }

    //da unter den besten Ergebnissen oft ein paar falsche Zuordnungen vorkommen, sucht das
Programm für
    //jede Schüssel die Sorten raus, die am öftesten in allen besten Ergebnissen zugeordnet
wurde (bzw deren Wert in ergebnisKombiniert am höchsten ist)
    for (int sch = 0; sch < gesamtObst; sch++) {
        List<int> häufigsteSorten = new List<int>() { 0 }; //häufigsteSorten in
ergebnisKombiniert
        for (int sor = 0; sor < gesamtObst; sor++) {
            if (ergebnisKombiniert[sch * gesamtObst + sor] > ergebnisKombiniert[sch * gesamtObst
+ häufigsteSorten[0]]) {
                häufigsteSorten = new List<int>() { sor };
            }
            else if (ergebnisKombiniert[sch * gesamtObst + sor] == ergebnisKombiniert[sch *
gesamtObst + häufigsteSorten[0]] && ergebnisKombiniert[sch * gesamtObst + sor] > 0 && sor > 0) {
                häufigsteSorten.Add(sor);
            }
        }
        foreach (int sorte in häufigsteSorten) {
            solution[sorte].Add(sch + 1);
            returnSpieße.Add(new Spieß(new List<int>() { sch + 1 }, new List<string>() {
alphabet[sorte] + "" }));
        }
    }
    return (returnSpieße, solution);
}

/// <summary>
/// erweitern des Ergebnisses mit den vorher aus der Matrix herausgenommenen leeren
Reihen/Spalten
/// um eine einfachere Dekodierung zu ermöglichen
/// </summary>
/// <returns>erweitertes Ergebnis</returns>
int[] vergrößereErgebnis(int[] ergebnis, List<int> leereReihen) {
    int[] neuesErgebnis = new int[ergebnis.Length + leereReihen.Count];
    int neuePositionDifferenz = 0;
    for (int i = 0; i < neuesErgebnis.Length; i++) {
        if (!leereReihen.Contains(i)) {
            neuesErgebnis[i] = ergebnis[i - neuePositionDifferenz];
        }
        else { neuesErgebnis[i] = 0; neuePositionDifferenz++; }
    }
    return neuesErgebnis;
}
```