# 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 7
6. ist nun nur noch Weintraube nicht-zugeordnet: Weintraube -> 3
7. 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 | |  | Spieß B | |
| Apfel  Erdbeere  Clementine  Pflaume  Birne | 6  3  5  9  1 |  | Himbeere  Kiwi  Birne  Orange  Apfel | 8  9  3  7  4 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Spieß A | |  | Spieß B | |  | Schnittspieß | |
| Erdbeere  Clementine  Pflaume | 6  5  1 |  | Himbeere  Kiwi  Orange | 8  7  4 |  | Apfel  Birne | 3  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

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;

}

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.

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

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 |  |

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

## Quantencomputer

FLEISCH JUFO INTRO

Um das Problem auch 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. 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 script 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 script liegt. Das script 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 script 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 pythonfile verwendeten libraries von D-Wave (installiert in einer virtual environment, deren Pfad im pythonfile angegeben wird), sowie eine Internetverbidung.

## 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.

**Begriffserklärung:**

spießeNum: Anzahl an Spießen : n

avSpießLen: durchschnittliche Länge eines Spießes

wunschLen: Länge des Wunschspießes

resultsNum: Anzahl an Ergebnissen vom Quantencomputer

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

|  |  |
| --- | --- |
| 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

## Quantencomputer

# Code