Dokumentation zu den Aufgaben des 39. Bundeswettbewerbs Informatik, 1. Runde

von Jan Ehehalt, Jonathan Hager

Inhaltsverzeichnis

1 Allgemeine Informationen

Da einige Methoden zu lang für eine Seite sind, wurde der Quellcode teilweise gekürzt. Die wichtigen Teile sind weiterhin enthalten. Auslassungen sind entsprechend mit „[…]“ gekennzeichnet und mit einer kurzen Erklärung zur Funktion des ausgelassenen Codes versehen.

2 Wörter aufräumen

2.1 Lösungsidee

Es werden zwei Listen angelegt, die jeweils alle Lücken und alle einzufügenden Wörter getrennt speichern. Das Programm geht dann alle Lücken durch und fügt für jede Lücke, bei der bereits ein Buchstabe gegeben ist, ein eindeutig zuzuordnendes Wort ein. Das Programm achtet hierbei darauf, dass der gegebene Buchstabe, mit dem im Wort übereinstimmt, und dass das Wort genauso lang ist wie die Lücke. Sollte für eine Lücke mehr als ein Wort potenziell passen, so wird diese Lücke vorerst übersprungen. Dieses Verfahren wird dann so oft ausgeführt, bis entweder alle Lücken gefüllt sind oder alle noch leeren Lücken keinen gegebenen Buchstaben mehr haben. Sobald es keine gegebenen Buchstaben mehr gibt werden die restlichen Wörter noch anhand ihrer Länge den übrigen Lücken zugeordnet. So gibt es am Ende keine leeren Lücken mehr.

2.2 Umsetzung

Jede Lücke wird im Array *sentence* als String gespeichert. Hierbei werden sowohl die Lücken, welche aus „_“ und verschiedenen Buchstaben bestehen können, als auch die Satzeichen, als eigene Elemente im Array gespeichert. Der *words* Array speichert die Wörter, die in den Satz eingefügt werden müssen. Dann wird eine Schleife gestartet, die nun den Lücken Array mit Elementen aus dem *words* Array füllen soll. Um zu erkennen welche Lücken bereits gefüllt wurden, wird ein Array aus *booleans* erstellt, der speichert welche Lücke bereits gefüllt wurde. Die Satzzeichen werden hier automatisch von Beginn an auf *true* gesetzt, da sie nicht gefüllt werden müssen. Dann iteriert das Programm über alle Wörter und zählt in wie viele noch nicht gefüllte Lücken das Wort anhand des gegebenen Buchstaben und der Länge passen würde. Sollte das Wort in nur eine Lücke passen, so wird es auch in diese eingefügt und aus dem *words* Array entfernt. Sollte das Wort in mehrere Lücken passen, so wird es zunächst übersprungen. Sobald einmal alle Wörter durchgegangen wurden wird noch einmal geprüft ob eines der Wörter in eine oder mehr Lücken passt. Sollte irgendein Wort noch eine Lücke finden, der es eindeutig zuzuordnen ist, so wird die Schleife noch nicht abgebrochen und startet erneut. Dieses ganz Verfahren läuft also so lange, bis sich kein Wort mehr einer Lücke mithilfe eines Buchstaben zuordnen lässt. Sobald die Schleife beendet wurde kann es keine Lücke mehr geben, die einen gegebenen Buchstaben hat. Das einzige Mittel, um die Wörter zuzuordnen, ist nun also noch die Länge. Also iteriert das Programm wieder über alle

Wörter und sucht für jedes Wort eine noch nicht gefüllte Lücke, die genauso lang ist wie das Wort. Wenn die Lücke gefunden wurde, wird das Wort hier eingefügt und aus dem *words* Array gelöscht. Nach diesem Loop kann es nun keine ungefüllte Lücke mehr geben.

Um zu prüfen, ob ein Wort anhand des gegebenen Buchstaben und der Länge in eine Lücke passt, wird die Methode *fits()* benutzt. Diese bekommt zwei Übergabeparameter. Einmal das Wort, welches in die Lücke eingefügt werden soll, und die Lücke, in die das Wort eingefügt werden soll. Diese Methode läuft zunächst einmal über die Lücke und sucht nach einem gegebenen Buchstaben, dessen Position im String der Lücke dann gespeichert wird. Sollte kein Buchstabe gefunden werden, so gibt die Methode *false* zurück, da das Wort ja nicht eindeutig zuwiesen werden kann. Sollte ein Buchstaben gefunden worden sein, prüfen wir noch im einzufügenden Wort, ob es an der gleichen Position den gleichen Buchstaben hat und ob die Länge der beiden Strings übereinstimmen. Da jeder Lücke eindeutig ein Wort zuzuordnen sein muss, muss das Wort also in diese Lücke eingefügt werden, die Methode gibt *true* zurück.

2.3 Dokumentation der Beispiele

2.3.1 Selbsterstelltes Beispiel

Die Funktion des Programms soll nochmal an folgendem Beispiel erläutert werden:

- _a_ ___s_ w___ _____ ___d_____ K_____ !
- Pudels also war des Das Kern

Das Programm liest nun den Lückentext in den *sentence* Array ein. Dieser hätte nun an der Stelle 2 zum Beispiel „___s_“ und an der Stelle 3 „w___“. Die einzufügenden Wörter werden im *words* Array gespeichert. An der Stelle 2 zum Beispiel „war“ und an der Stelle 3 „des“. Der *sentence* Array hat nun eine Länge von 7 und *words* 6. Nun läuft das Programm in der Schleife einmal über alle Wörter und zählt, wie vielen Lücken die Worte zugeordnet werden könnten. Das erste Wort „Pudels“ findet für sich nur genau eine Lücke. Da es also eindeutig zuzuordnen ist wird es in Lücke 5 eingefügt und aus *words* entfernt. Das Wort „also“ ist eindeutig der zweiten Lücke zuzuordnen. Das Wort „war“ hat nun zwei Lücken, in die es anhand der gegebenen Buchstaben passen würde (Lücke 1 und 3). Deshalb wird es vorerst übersprungen. Das Wort „des“ lässt sich mithilfe der gegebenen Buchstaben keiner Lücke eindeutig zuordnen, weshalb es auch übersprungen wird. Das Wort „Das“ lässt sich nun wieder eindeutig der ersten Lücke zuordnen, wird also auch eingefügt. „Kern“ ist auch eindeutig der Lücke 6 zuzuordnen, wird also eingefügt. Nach diesem ersten Durchgang der Schleife sehen die Arrays also wie folgt aus:

sentence Das also w___ _____ Pudels Kern !

words war des

Nun wird geprüft ob es noch Wörter gibt, die sich durch gegebene Buchstaben noch einer Lücke zuordnen lassen. Da „war“ noch eine Lücke findet wird die Schleife wiederholt. „war“ lässt sich nun eindeutig der dritten Lücke zuordnen, da die andere Lücke, für die es auch gepasst hätte, ja nun bereits gefüllt ist. Deshalb kann es nun in den Satz eingefügt werden. Das letzte Wort „das“ ist nun keiner Lücke mehr zuzuordnen, weshalb die Schleife nun abbricht. Jetzt sehen die beiden Arrays so aus:

sentence Das also war _____ Pudels Kern !

words des

Nun folgt noch der Loop, der die Wörter lediglich anhand ihrer Länge den Lücken zuordnet. Das Wort „war“ ist nun das einzige Wort im *words* Array. „war“ hat eine Länge von 3. Nun wird im Satz nach einer noch nicht gefüllten Lücke der Länge 3 gesucht, welche an Stelle 4 gefunden wird. Somit gibt es nun kein Wort mehr im *words* Array und *sentence* sieht wie folgt aus:
Das also war des Pudels Kern !
Der Satz ist also fertig und wird ausgegeben.

2.3.2 raetsel0

Der fertige Satz lautet:
oh je, was für eine arbeit!

2.3.3 raetsel1

Der fertige Satz lautet:
Am Anfang wurde das Universum erschaffen. Das machte viele Leute sehr wütend und wurde allenthalben als Schritt in die falsche Richtung angesehen.

2.3.4 raetsel2

Der fertige Satz lautet:
Als Gregor Samsa eines Morgens aus unruhigen Träumen erwachte, fand er sich in seinem Bett zu einem ungeheuren Ungeziefer verwandelt.

2.3.5 raetsel3

Der fertige Satz lautet:
Informatik ist die Wissenschaft von der systematischen Darstellung, Speicherung, Verarbeitung und Übertragung von Informationen, besonders der automatischen Verarbeitung mit Digitalrechnern.

2.3.6 raetsel4

Der fertige Satz lautet:
Opa Jürgen blättert in einer Zeitschrift aus der Apotheke und findet ein Rätsel. Es ist eine Liste von Wörtern gegeben, die in die richtige Reihenfolge gebracht werden sollen, so dass sie eine lustige Geschichte ergeben. Leerzeichen und Satzzeichen sowie einige Buchstaben sind schon vorgegeben.

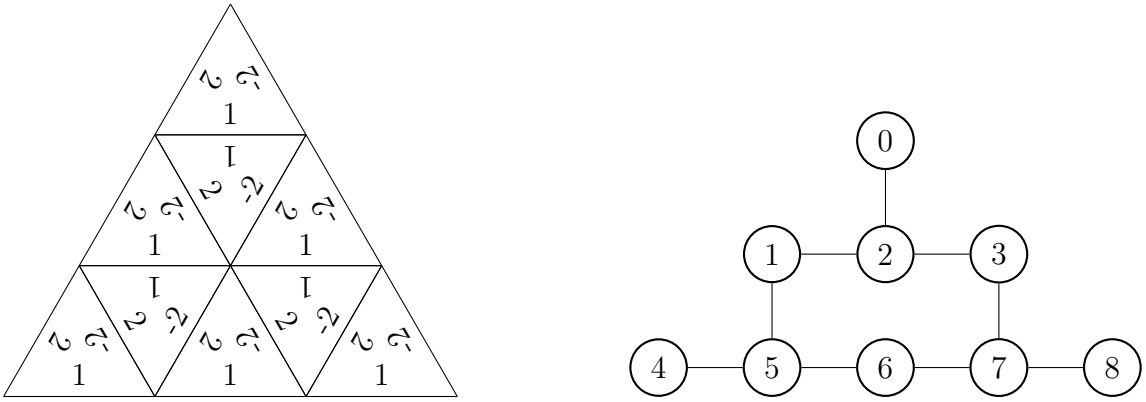
3 Dreieckspuzzle

3.1 Lösungsidee

Ein Algorithmus soll durch systematisches Probieren alle sinnvollen Möglichkeiten testen. Das Programm legt erst das erste Puzzleteil, versucht dann dessen anliegende Kanten mit Teilen zu füllen. Sollte es für ein Puzzleteil keine verfügbaren Teile mehr geben, die an die Kante passen, geht der Algorithmus mittels Backtracking zurück. Sobald er wieder zu einem Teil kommt, bei dem er andere Teile noch nicht probiert hat, versucht er das Puzzle ab hier wieder neu aufzubauen.

3.2 Umsetzung

Zur Darstellung der Struktur des Puzzle wird ein ungerichteter und ungewichteter Graph verwendet. Der Aufbau wird in Abbildung 1 gezeigt.



(a) Das Puzzle (b) Darstellung als Graph

Abbildung 1: Interne Speicherung der Puzzlestruktur

Die Knoten bilden die einzelnen Puzzleteile. Wenn eine Kante im Graph existiert, dann berühren sich die beiden Puzzleteile auch im Puzzle, es muss also an den Stellen geprüft werden, ob sie zusammenpassen. Das ist der Fall, wenn beispielsweise die rechte Kante des einen Teils addiert mit der linken Kante des anderen Teils 0 ergibt.

Im Folgenden wird erst auf die einzelnen Klassen und deren Funktion eingegangen, anschließend wird die Vorgehensweise des Algorithmus anhand von Zeichnungen erläutert.

3.2.1 Klassen

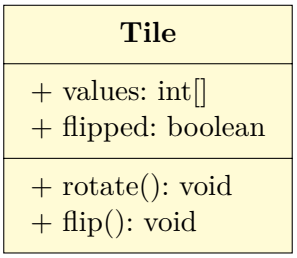


Abbildung 2: Die Klasse *Tile*

Die Klasse *Tile* (Abbildung 2) repräsentiert ein Puzzleteil. Dazu müssen die drei Kantenwerte gespeichert werden. Dies geschieht durch den Array *values*. Die Seiten haben die Indizes wie folgt: 0: links, 1 mitte, 2 rechts. Zudem muss gespeichert werden, ob ein Tile geflippt ist oder nicht. Dies ist nur bei den Stellen 2, 5, 7 im Graph der Fall. Wenn ein Tile geflippt wird, müssen die Werte der linken und rechten Seite getauscht werden. Zudem muss das Tile in die andere Richtung rotiert werden. Die *rotate()* Methode rotiert das Tile einmal um 120°.

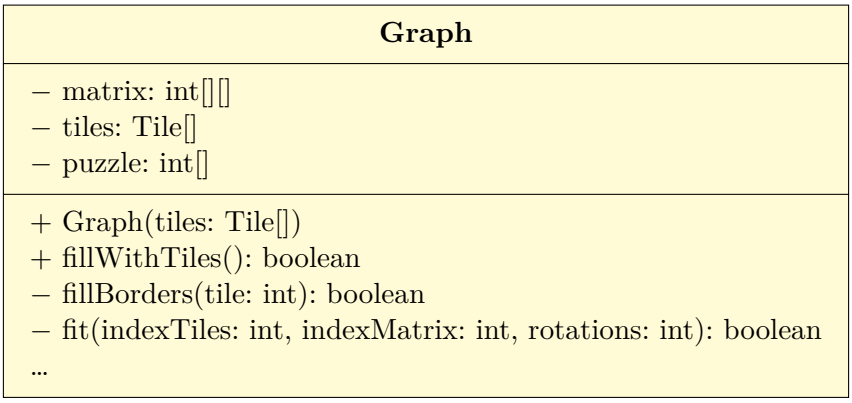


Abbildung 3: Die Klasse *Graph*

Der Graph speichert seine Adjazenzmatrix, die als Wert -1, 0 und 1 annehmen kann. -1 bedeutet, dass diese Kante nicht existiert. Das ist der Fall, wenn sich die zwei Teile im echten Puzzle nicht berühren können, z.B. die rechte Kante der 0 und die untere Kante der 4. Den Wert 0 nimmt eine Kante an, wenn diese aktuell nicht auf beiden Seiten besetzt ist, z.B. wenn in Abbildung 1b Teil 5 fehlen würde, hätten die Kanten 4-5, 1-5 und 5-6 den Wert 0. Der Wert 1 bedeutet, dass die Kante besetzt ist. Ist das Puzzle gelöst, haben also alle Kanten den Wert -1 oder 1. 0 würde bedeuten, dass es

noch eine Kante gibt, die besetzt werden muss. Der Graph speichert im *Tile* Array alle Puzzleteile, die es gibt. Der puzzle-Array speichert die Position jedes Teils im Graphen. Ist das Teil nicht im Graphen, hat puzzle an der Stelle des Teils den Wert -1.

Es wurden bewusst nicht alle Methoden in das Klassendiagramm übernommen, die wichtigen Methoden sind allerdings vorhanden. Die restlichen Methoden sind zur Erklärung des Algorithmus nicht von Bedeutung, sie fügen beispielsweise eine Kante am Anfang ein oder setzen das Puzzle zurück.

3.2.2 Algorithmus

Die Methode *fillWithTiles()* wird aufgerufen, um das Puzzle zu lösen. Diese geht über jedes Puzzleteil und versucht das Puzzle damit an Stelle 0 im Graphen zu lösen. Dazu wird *fillBorders(0)* aufgerufen. Für den Quellcode der Methode, siehe Abbildung 4.

```
1  public boolean fillWithTiles() {
2     for(int i = 0; i < this.tiles.length; i++) {
3        // puzzle-Array wird zurueckgesetzt
4        resetPuzzle();
5
6          // Das Tile wird an die Stelle 0 im Graph gesetzt
7          puzzle[i] = 0;
8
9          if(fillBorders(0)) {
10             // Puzzle wurde gelöst
11             return true;
12         }
13         // [...] fillBorders() wird für jede Rotation des Tile wiederholt
14      }
15    return false;
16 }
```

Abbildung 4: Die Methode *fillWithTiles()*

Die Methode *fillBorders(int)* versucht alle Kanten, die an die übergebene Stelle im Graphen, anliegen, zu füllen. Ob alles besetzt werden konnte, lässt sich am Rückgabewert *true* oder *false* erkennen. Dazu wird erst über die entsprechende Zeile in der Adjazenzmatrix iteriert. Nur beim Wert 0 ist noch keine Kante vorhanden, also wird jetzt für diese Kante ein passendes Teil gesucht. Ob ein Teil bereits probiert wurde, wird im *visited* Array geprüft. Dort wurde bereits die eigene Stelle und alle Teile, die im Graph sind, als *true* markiert, damit diese nicht erneut probiert werden. Wenn ein passendes Teil gefunden wurde, wird dieses vorläufig eingefügt. Anschließend versucht dieses Teil seine Kanten zu füllen. Scheitert es, wird es entfernt und ein neues Teil gesucht. Ist es erfolgreich, geht die Methode zum nächsten Eintrag in der Zeile der Matrix und versucht diese ebenfalls zu füllen. Der Quellcode in Abbildung 5 ist ein Teil der Methode. Dort wird zuerst mit *fit()* überprüft, ob das Tile an die Stelle passt. Anschließend wird es

dort eingefügt und versucht alle Kanten zu füllen, indem es *fillBorders()* mit der eigenen Stelle im Graphen aufruft (*puzzle* speichert für jedes Tile die Position im Graphen oder -1). k ist das aktuelle Tile, j die aktuelle Kante.

```
if(fit(k, j, 0)) {
   tileFound = true;
   resetTile(j);
   puzzle[k] = j;
   updateTrueLink(tile, j);
   placedTiles.add(j);

   if(fillBorders(puzzle[k])) {
      break;
   }
    else {
      resetTile(j);
      updateFalseLink(tile, j);
      tileFound = false;
       continue;
    }
}
if(!tileFound) {
    // [...] alle vom Tile erstellten Tiles werden ebenfalls gelöscht
    return false;
}
return true;
```

Abbildung 5: Die Methode *fillBorders*

Die Methode *fit()* prüft für eine gegebene Stelle im Graphen, ob ein gegebenes Puzzleteil passt. Zuerst wird geprüft, ob sich das Puzzleteil an der Stelle 2, 5 oder 7 befindet. Ist dies der Fall, muss es geflippt werden. Im echten Puzzle äußert sich das an der Kante oben anstatt einer Spitze. Um zu wissen, welche Seiten bei zwei Teilen verglichen werden müssen, werden die beiden Stellen im Graph voneinander abgezogen. Kommt hier der Wert -1 raus, befindet sich das gefundene Teil rechts, bei 1 links und bei jedem anderen Wert müssen die beiden mittleren Werte verglichen werden. Ist die Summe der beiden Kantenwerte der Teile 0, passt das Puzzleteil. Der Quellcode, der eine Seite des Teils mit einer vorhandenen Kante vergleicht, ist in Abbildung 6 zu finden. i ist der Index des zweiten Teils, mit dem verglichen werden muss.

```
int difference = indexMatrix - i;
int side1 = 1;
int side2 = 1;
int tile2 = getIndexTiles(i);

if(difference == -1) {
    side1 = 2;
   side2 = 0;
}
else if(difference == 1) {
   side1 = 0;
   side2 = 2;
}

if(tiles[indexTiles].values[side1] + tiles[tile2].values[side2] != 0) {
    fits = false;
    break;
}
```

Abbildung 6: Ausschnitt von *fit()*

3.3 Beispiele

3.3.1 Puzzle0

In der folgenden Reihe an Abbildungen wird das Beispiel von *puzzle0.txt* verwendet. Es wird gezeigt, wie sich die richtige Lösung stückweise aufbaut. Allerdings wird jegliche Rotation und Backtracking, wegen der Übersichtlichkeit, nicht dargestellt. Man soll lediglich erkennen können, wie sich der Graph bzw. das Puzzle stückweise aufbauen. Die Pfeile am Graphen stehen nicht für eine Richtung oder alle Kanten, sie sollen lediglich veranschaulichen, welches Teil welches gelegt hat.

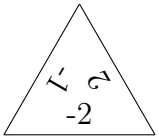


Abbildung 7: Teil 0 wird gelegt

Abbildung 8: Teil 2 wird von 0 gelegt

Abbildung 9: Teil 1 wird von 2 gelegt

Abbildung 10: Teil 5 wird von 1 angelegt

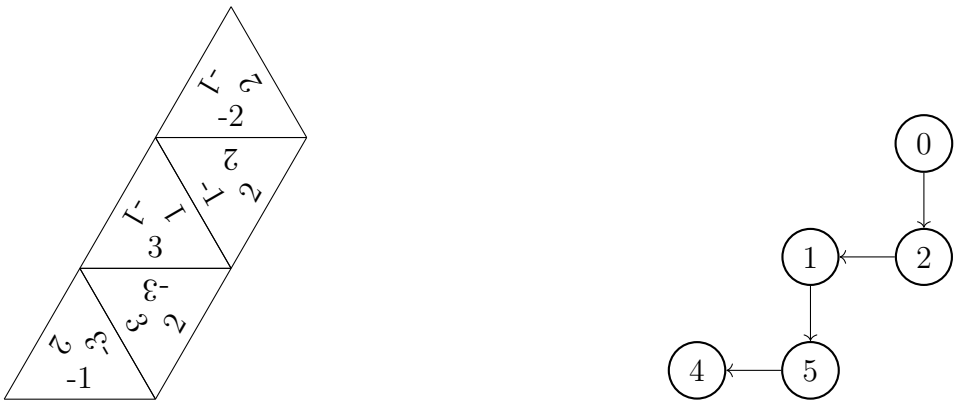


Abbildung 11: Teil 4 wird von 5 angelegt

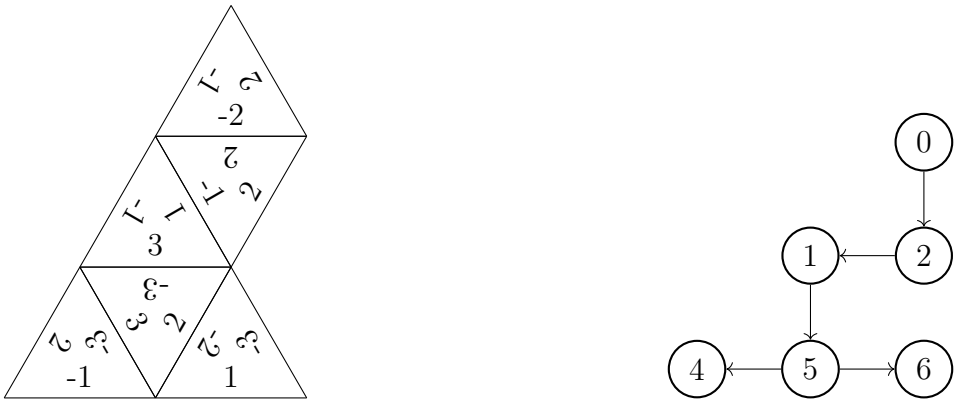


Abbildung 12: Teil 6 wird von 5 angelegt

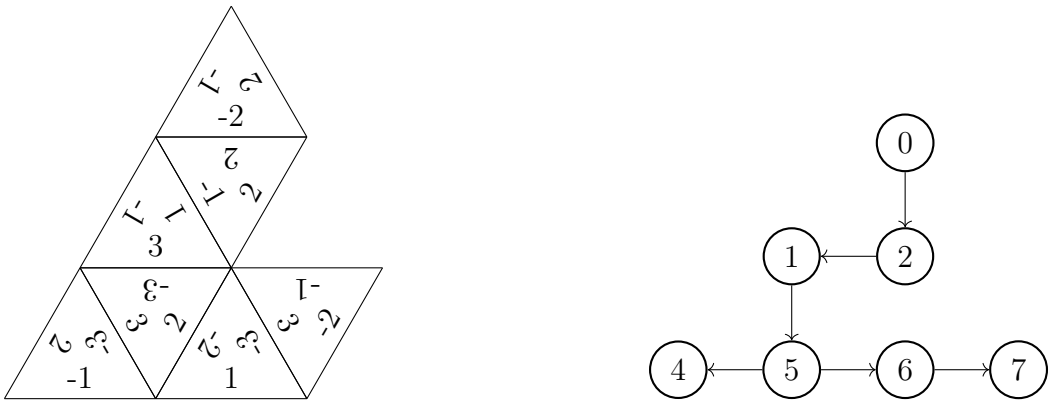


Abbildung 13: Teil 7 wird von 6 angelegt

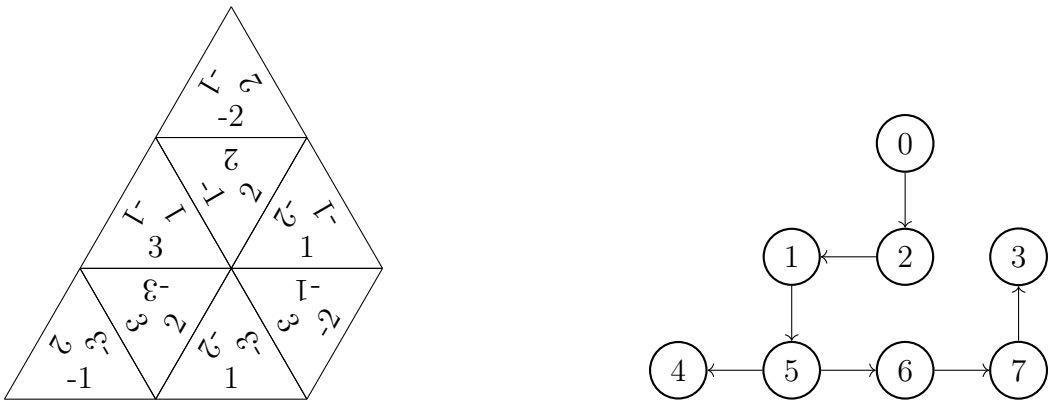


Abbildung 14: Teil 3 wird von 7 angelegt

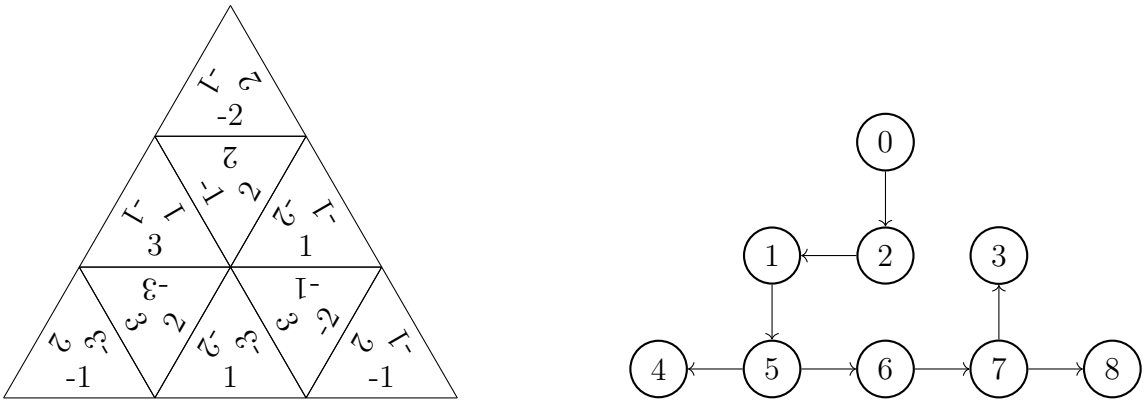


Abbildung 15: Teil 8 wird von 7 angelegt

3.3.2 Puzzle1

Laut dem Algorithmus ist dieses Puzzle nicht lösbar.

3.3.3 Puzzle2

Dieses Puzzle ist lösbar. Die Lösung ist in Abbildung 16 dargestellt.

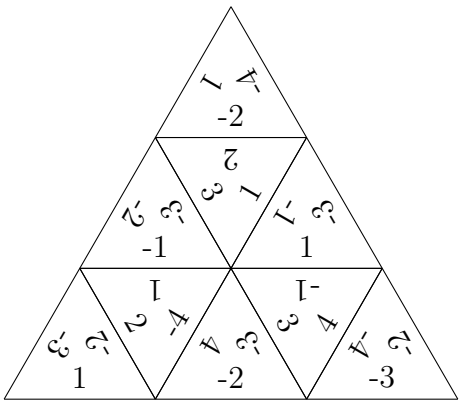


Abbildung 16: Die Lösung von Puzzle2

3.3.4 Puzzle3

Laut dem Algorithmus ist dieses Puzzle nicht lösbar.

4 Tobis Turnier

4.1 Lösungsidee

Um die verschiedenen Turniere zu Simulieren wurde für jede Turnierform eine Methode erstellt, die den Sieger des Turniers zurückgibt. So können können viele Turniere einfach durch Aufrufe dieser Methoden simuliert werden.

4.2 Umsetzung

Wie bereits erwähnt wurde für jede Turnierform eine Methode erstellt. Im folgenden Teil werden die einzelnen Methoden genauer erläutert:

4.2.1 Einlesen der Datei

Jeder Spieler wird in Form einer eigenen Klasse, die die Spielstärke und die ID speichert, in einem Array gespeichert. Das Programm geht die gegebene Datei lediglich Zeile für Zeile durch und erstellt in jeder Zeile, außer der ersten, einen neuen Spieler, der die Spielstärke hat die die Zeile angibt. Nebenbei zählen wir noch eine Variable hoch, die jedem Spieler eine neue ID zuordnet.

4.2.2 Die *match*-Methode

Die *match* Methode realisiert das Spiel RNG im Programm. Sie bekommt zwei Spieler übergeben, von denen nur der zurückgegeben wird, der das Match gewinnt. Die Methode addiert zuerst die beiden Spielstärken der Spieler und generiert dann eine zufällige Zahl, die zwischen dieser Summe und 0 liegt. Wenn diese generierte Zahl kleiner als die Stärke

von Spieler 1 ist, so hat dieser gewonnen. Sollte die Zahl größer oder gleich der Spielstärke von Spieler 2 sein, so hat dieser gewonnen.

4.2.3 Liga

Die Methode *league()* bekommt einen Array aus Spielern übergeben und gibt am Ende einen Spieler zurück, der die Liga gewonnen hat. In der Liga spielt jeder Spieler genau einmal gegen jeden anderen. Um zu wissen, wer bereits gegen wen gespielt hat, speichern wir uns nebenbei einen zweidimensionalen Array aus booleans. Dieser wird dann mithilfe eines Loop an allen Stellen auf false gestellt, außer in der Diagonalen, also zum Beispiel an Stelle [1][1], damit die Spieler nicht einmal gegen sich selbst spielen. Danach startet eine Schleife die so lange läuft, bis der zweidimensionale Array kein false mehr speichert, also jeder Spieler einmal gegen jeden anderen Spieler gespielt hat. In der Schleife wird für jeden Spieler im übergebenen Array einmal über alle Spieler gegangen und anhand des zweidimensionalen Arrays geprüft, ob die beiden Spieler bereits gegeneinander gespielt haben. Sollten sie noch nicht gegeneinander gespielt haben, so spielen sie nun mithilfe der Methode match gegeneinander. Der gewinner des matches fügt dann seiner wins variable einen Sieg hinzu. Außerdem setzen wir den zweidimensionalen Array an der Stelle der beiden Spieler auf true. Sobald diese Schleife nun abbricht hat jeder Spieler nun einmal gegen jeden gespielt und jeder Spieler weiß, wie häufig er gewonnen hat. Nun wird geprüft welcher der Spieler die meisten Siege hat, also die Liga gewonnen hat. Dazu wird erst geprüft, was die höchste Anzahl an Siegen ist, die einer der Spieler erreicht hat. Dann fügen wir in einen neuen Array aus Spielern alle Spieler hinzu, die so viele Siege wie der Spieler mit den meisten Siegen hat. So haben wir nun alle Spieler, die potenziell gewinnen könnten. Sollten mehr als ein Spieler in diesem Array sein, so entscheidet die ID wer gewinnt. Wir zählen eine Variable von 0 hoch und sobald im Array ein Spieler ist, dessen ID gleich der Variable ist geben wir diesen zurück als Gewinner. Wenn wir nur einen Spieler im Array haben, so können wir diesen direkt zurückgeben.

4.2.4 Ko

In der *ko* Methode wird zunächst der Turnierbaum erzeugt. Dazu wird eine Wurzel erstellt, von der aus der Binärbaum nach unten erzeugt wird. Die Methode bekommt einen Array mit den Spielern übergeben, die in den Turnierbaum eingefügt werden müssen und gibt den Sieger des Turniers zurück. Die Wurzel startet die rekursive Methode *create()*, die den Baum erzeugt. Diese Methode bekommt den Array mit den Spielern übergeben. Die Wurzel fängt nun an. Sollte der Array nur 2 Spieler speichern, so kann die Wurzel einen der Spieler als linken Teilbaum speichern und den anderen als rechten. Sollte der Array länger sein, so erstellt er jeweils rechts und links einen Knoten, der keinen Spieler speichert, teilt die Spieler gleichmäßig in 2 neue Arrays auf und führt die Methode *create()* bei diesen, mit den neu erzeugten Arrays erneut aus. So ist der Array irgendwann nur noch 2 lang und alle Spieler sind in den Blättern des Baums. Nachdem nun der Baum erzeugt wurde kann das Turnier starten. Hierzu starten wir bei der Wurzel und gehen den Baum nach unten durch mit der rekursiven Methode *getWinner()*. Diese

prüft ob der aktuelle Knoten bereits einen Spieler besitzt. Sollte er bereits einen Spieler haben, so gibt er diesen zurück. Sollte er keinen haben, so spielen die beiden Spieler der Teilbäume gegeneinander und der Knoten speichert den Sieger des Matches als seinen Spieler. Sind alle rekursiven Methodenaufrufe abgeschlossen, speichert die Wurzel den Sieger des Turniers.

4.2.5 Kox5

Die *kox5()* Methode ist prinzipiell die gleiche wie *ko()*. Das Verfahren ist bis auf die *match()* Methode genau das gleiche. Hier wurde die Methode *matchx5()* angelegt. Diese lässt die beiden Spieler fünf mal gegeneinander antreten und es wird gespeichert, wie oft jeder Spieler gewonnen hat. Der Spieler, der häufiger gewonnen hat, wird dann als Sieger zurückgegeben.

4.3 Beispiele

4.4 Selbsterstelltes Beispiel

Im folgenden Teil wird ein Beispiel durchgespielt, in dem 4 Spieler ein Ko-Turnier veranstalten. Folgende Spieler treten an:

Spieler	Spielstärke
1	20
2	40
3	45
4	90

Nachdem diese Spieler nun in einem Array gesammelt wurden, wird die Methode *ko()* mit diesem Array ausgeführt. Nun wird zunächst eine Wurzel erstellt, die die Methode *create()* mit dem Array ausführt. Hier wird die Länge des Arrays geprüft. Da die Länge 4, und nicht 2, beträgt, erzeugt die Wurzel einen neuen linken und einen neuen rechten Teilbaum, die beide keinen Spieler speichern. Dann wird der Spieler Array aufgeteilt und der linke Knoten bekommt die ersten der beiden Spieler und der rechte Knoten die hinteren beiden Spieler. Beide führen wieder jeweils die Methode *create()* aus. Da beide nun einen Array der Länge 2 haben, erstellen beide jeweils wieder einen linken und rechten Knoten, die beide jeweils einen der beiden Spieler aus dem Array speichern.

Nun muss noch ein Sieger ermittelt werden. Hierzu wird die Methode *getWinner()* bei der Wurzel aufgerufen. Da die Wurzel noch keinen Spieler speichert, gibt sie den Sieger eines Matches zwischen den Spielern des linken und rechten Teilbaums zurück. Da diese ebenfalls keinen Spieler speichern, geben sie jeweils den Sieger eines Matches zwischen den Spielern ihrer linken und rechten Knoten bzw. Teilbäume zurück. Diese Abfolge an Spielabläufen ist in den folgenden Abbildungen dargestellt.

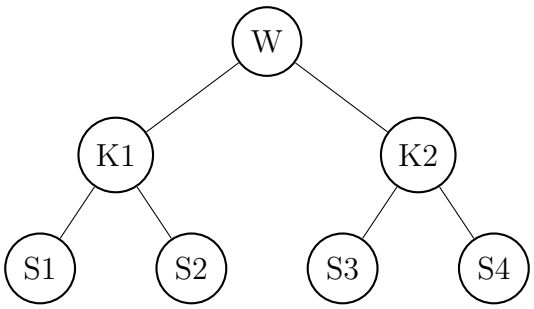


Abbildung 17: Ausgangslage des Turniers

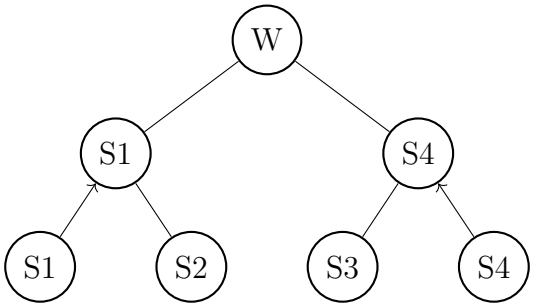


Abbildung 18: Schritt 1: S1 gewinnt gegen S2 und S4 gegen S3

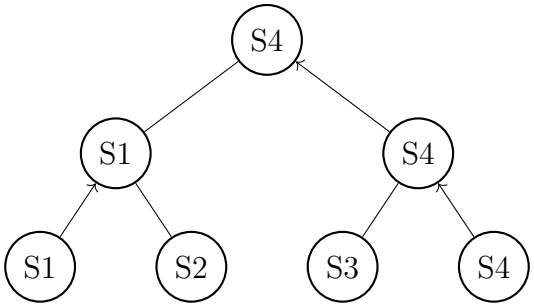


Abbildung 19: Schritt 2: S4 gewinnt gegen S1 und somit das Turnier

Im folgenden Teil werden noch die Ergebnisse der einzelnen Dateien dokumentiert. Es wird ermittelt, wie häufig der stärkste Spieler durchschnittlich gewinnt und auf dieser Basis ein Turnierformat empfohlen. Dazu werden alle drei Turnierformate 1000000 mal durchgeführt und es wird mitgezählt, wie viele Spiele der beste Spieler gewonnen hat, um daraus einen Durchschnittswert zu berechnen.

4.5 spielstaerken1

Diese Datei beinhaltet 8 Spieler, deren Spielstärke gleichmäßig verteilt zwischen 0 und 100 liegt. Der stärkste Spieler hat eine Stärke von 100. Wenn wir die Turnierformen

1000000 mal simulieren, so ergeben sich für die Siegeswahrscheinlichkeiten des stärksten Spielers folgende Werte:

Turnierform	Siegesrate
Liga	34.5 %
Ko	40.4 %
Kox5	47.4 %

Daraus lässt sich schließen, dass die Turnierform Kox5 hier die, mit kleinem Abstand, besten Ergebnisse liefert.

4.6 spielstaerken2

Diese Datei beinhaltet wieder 8 Spieler. Die Hälfte der Spieler hat eine relativ niedrige Stärke, während die andere Hälfte relativ Spielstark ist. Der stärkste Spieler hat wieder eine Stärke von 100. Wenn wir die Turnierformen 1000000 mal simulieren, so ergeben sich für die Siegeswahrscheinlichkeiten des stärksten Spielers folgende Werte:

Turnierform	Siegesrate
Liga	20.9 %
Ko	30.2 %
Kox5	32.7 %

Daraus lässt sich schließen, dass die Turnierform Kox5 hier wieder mit kleinem Abstand die besten Werte liefert.

4.7 spielstaerken3

Diese Datei beinhaltet 16 Spieler. Die Spielstärken sind relativ gleichmäßig verteilt und der stärkste Spieler hat eine Stärke von 93. Wenn wir die Turnierformen 1000000 mal simulieren, so ergeben sich für die Siegeswahrscheinlichkeiten des stärksten Spielers folgende Werte:

Turnierform	Siegesrate
Liga	31.6 %
Ko	16.6 %
Kox5	19.2 %

Hier liefert die Turnierform Liga die mit Abstand besten Werte.

4.8 spielstaerken4

Diese Datei beinhaltet 16 Spieler. Alle, bis auf ein Spieler, haben hier die Spielstärke 95. Dieser eine Spieler hat die Stärke 100, und ist somit der Stärkste der Spieler. Wenn wir die Turnierformen 1000000 mal simulieren, so ergeben sich für die Siegeswahrscheinlichkeiten des stärksten Spielers folgende Werte:

Turnierform	Siegesrate
Liga	11.5 %
Ko	0.7 %
Kox5	0.7 %

Turnierform Ko und Kox5 liefern hier sehr ähnliche Werte. Die Werte von Kox5 sind minimal besser. Liga liefert hier wieder die mit Abstand besten Werte.

5 Streichholzrätsel

6 Wichteln

6.1 Lösungsidee

Im Wesentlichen ist das Problem der Zuordnung mit dem *Stable Marriage Problem* vergleichbar. Es gibt zwei verschiedene Gruppen, die so gut wie möglich verteilt werden müssen. Deshalb lässt sich die Aufgabe mit einer leicht angepassten Variante des *Gale-Shapley* Algorithmus lösen. Allgemein fragen alle Schüler ohne Geschenk bei den Geschenken an, ob diese noch keinen Partner haben oder den neuen Partner dem aktuellen vorziehen. Dabei wird ein Erstwunsch allem vorgezogen, dann folgen Zweitwunsch, Drittwunsch und zuletzt eine Zuteilung ohne Wunsch. Jeder Schüler fragt nacheinander seine Wünsche an. Sollte er danach noch kein Geschenk haben, versucht er ein übriges Geschenk zu bekommen.

6.2 Umsetzung

Jeder Schüler wird durch ein Objekt der Klasse *Student* repräsentiert. Dabei muss jeder Schüler speichern, ob er bereits ein Geschenk hat. Zudem weiß jeder Schüler, welche Geschenk er als Erst-, Zweit- und Drittwunsch will. Für den Algorithmus ist es notwendig, dass jeder Schüler zusätzlich seine Position im Array und alle Geschenke, die er schon versucht hat zu bekommen, kennt. Der Schüler braucht *requestPresent()* als einzige Methode. Übergeben bekommt er zum einen den *Present* und *Student* Array, zum anderen den Index des entsprechenden Geschenks, sowie den Grad des Wunsches. Hierbei hat ein Erstwunsch den Grad 0, ein Zweitwunsch 1, ein Drittwunsch 2 und eine reine Zuteilung 3. Diesen Grad braucht das Geschenk, um den neuen Schüler mit dem möglicherweise aktuellen Schüler zu vergleichen und zu ermitteln, welchen Schüler es bevorzugt.

Student
hasGift: boolean index: int presentId: int wishes: int[] asked: boolean[]
Student(int[], int, int) requestPresent(Present[], Student[], int, int): void

Abbildung 20: Die Klasse *Student*

Present
studentId: int wish: int
Present() changeStudent(students: Student[], wish: int, studentId: int): boolean

Abbildung 21: Die Klasse *Present*

Die Geschenke werden von der Klasse *Present* repräsentiert. Ein Geschenk muss hierfür wissen, an welchen Schüler es vergeben ist. Dazu wird der Index des Schülers gespeichert. Um zwei Schüler vergleichen zu können, muss zudem bekannt sein, welchen Grad der Wunsch des aktuellen Schülers hat. Dieser wird als *int* gespeichert. Damit der Algorithmus funktioniert, muss das Geschenk seinen Schüler wechseln können. Hierfür wird der *Student* Array übergeben, sowie der Grad des Wunsches vom neuen Schüler und der Index des neuen Schülers. Die Methode gibt durch einen *boolean* zurück, ob der übergebene Schüler übernommen oder abgelehnt wurde. Die Methode arbeitet wie folgt:

Zuerst wird verglichen, ob der Grad des gespeicherten Wunsches größer ist, als der des neuen Schülers. Ein kleinerer Grad ist immer einem größeren vorzuziehen. Ist der Grad des neuen Wunsches also nicht echt kleiner als der bisherige, gibt die Methode direkt *false* zurück, da der neue Schüler abgelehnt wird. Andernfalls soll der neue Schüler jedoch den alten ersetzen. Hierfür wird dem alten Schüler erst mitgeteilt, dass er kein Geschenk mehr hat. Davor muss geprüft werden, ob es überhaupt einen alten Schüler gibt. Gibt es keinen, hat *studentId* den Standardwert -1, weshalb dieser Fall abgefangen werden muss. Anschließend werden die Werte des neuen Schülers übernommen, ihm wird mitgeteilt, dass er nun ein Geschenk hat und die Methode gibt *true* zurück. Der Algorithmus selbst ist in zwei Phasen unterteilt:

1. Jeder Schüler versucht seinen Erstwunsch zu bekommen.

2. Jeder Schüler fragt pro Durchlauf ein Geschenk, dass er noch nicht versucht hat, an. Hierbei werden nacheinander die Wünsche favorisiert. Sind bereits alle probiert

```
boolean changeStudent(Student[] students, int wish, int studentId) {
   if(this.wish > wish){
      if(this.studentId >= 0) {
         students[this.studentId].hasGift = false;
      }

      this.studentId = studentId;
      this.wish = wish;
      students[studentId].hasGift = true;

      return true;
   }
   else {
      return false;
   }
}
```

Abbildung 22: Die Methode *changeStudent()*

wurden, werden einfach alle restlichen Geschenke versucht.

Phase 1 findet nur beim ersten Durchlauf durch alle Schüler statt. Danach wird in Phase 2 übergegangen und solang über alle Schüler iteriert, bis jeder ein Geschenk bekommen hat. Endet die Schleife, wurde die bestmögliche Verteilung erreicht. Der gesamte Quellcode des Algorithmus ist in Abbildung 23 zu finden.

6.3 Beispiele

Anhand eines selbsterstellten, kleinen Beispiels wird hier die Funktion des Algorithmus dargestellt. Anschließend werden alle Ergebnisse der vorgegebenen Beispielaufgaben aufgeführt.

Schüler	Erstwunsch	Zweitwunsch	Drittwunsch
A	1	2	3
B	2	3	4
C	2	1	4
D	4	1	3

Tabelle 1: Beispielhafte Verteilung der Wünsche

Zuerst wird das Beispiel aus Tabelle 1 vom Programm eingelesen. Dabei wird für jeden Schüler, sowie für jedes Geschenk ein Objekt erstellt. Danach beginnt der Algorithmus direkt mit der Verteilung der Geschenke. Im ersten Durchlauf versucht jeder Schüler seinen Erstwunsch zu bekommen. Die Schleife beginnt mit Schüler A. Da sein Erstwunsch

frei ist, bekommt er Geschenk 1 zugeteilt. Dasselbe passiert bei Schüler B, dieser bekommt folglich Geschenk 2 zugeteilt. Der Erstwunsch von Schüler C ist bereits vergeben und da C es mit einem Erstwunsch haben will, es aber bereits an einen Erstwunsch vergeben ist, wird Schüler C abgelehnt. Schüler D bekommt Geschenk 4 als Erstwunsch zugeteilt. Jetzt wird in Phase 2 gewechselt. Schüler A und B werden übersprungen, da beide bereits ein Geschenk haben. Schüler C hat seinen Erstwunsch bereits angefragt und probiert deshalb jetzt seinen Zweitwunsch. Dieser ist ebenfalls bereits vergeben und da Schüler C mit einem Zweitwunsch anfragt, wird er abgelehnt. D wird ebenfalls übersprungen. Im nächsten Durchlauf wird ebenfalls jeder außer C übersprungen. Dieser fragt nun seinen Drittwunsch an, welcher bereits an einen höheren Wunsch vergebenen ist. Somit beginnt erneut ein neuer Durchlauf der Schleife, bei dem C nun irgendein Geschenk, das er noch nicht probiert hat, versucht. Es bleibt nur Geschenk 3 übrig, welches er zugeteilt bekommt. Damit wurde jedem Schüler ein Geschenk zugeteilt. Die ausgegebene Verteilung lässt sich Tabelle 2 entnehmen.

Schüler	Geschenk
A	1
B	2
C	3
D	4

Tabelle 2: Das Ergebnis des Programms

```java
// Phase 1
for(int i=0; i<students.length; i++) {
    students[i].requestPresent(presents, students, students[i].wishes[0], 0);
}

// Phase 2
boolean finished = false;
do {
    // Die Schleife ist prinzipiell immer fertig, außer es wird noch ein
        Schüler gefunden, der kein Geschenk hat
    finished = true;
    for(int i = 0; i < students.length; i++) {
        if(!students[i].hasGift) {
            finished = false;

            // Es wird versucht, einen der Wünsche zu erfüllen
            if(!students[i].asked[students[i].wishes[0]]) {
               students[i].requestPresent(presents, students,
                   students[i].wishes[0], 0);
            } else if(!students[i].asked[students[i].wishes[1]]) {
               students[i].requestPresent(presents, students,
                   students[i].wishes[1], 1);
            } else if(!students[i].asked[students[i].wishes[2]]) {
               students[i].requestPresent(presents, students,
                   students[i].wishes[2], 2);
            } else {
                // Da bereits alle Wünsche probiert wurden, wird versucht
                    irgendein Geschenk zu bekommen
               for(int j = 0; j < presents.length; j++) {
                   if(!students[i].asked[j]) {
                       students[i].requestPresent(presents, students, j, 3);
                   }
                }
            }
        }
    }
} while(!finished);
```

Abbildung 23: Die Implementierung des Algorithmus