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 vollständigen Wörter getrennt speichern. Das Programm geht die Liste der Lücken durch und fügt für jede Lücke, der sich nur ein Wort eindeutig zuordnen lässt, das entsprechende Wort ein. Dieser Durchlauf wird so oft wiederholt, bis entweder alle Lücken gefüllt sind, oder in einem Durchlauf kein Wort mehr eingefügt werden konnte. Diese Lücken müssen anhand der Länge dem entsprechenden Wort zugeordnet werden. Dieses Verfahren muss funktionieren, da in jede Lücke nur ein Wort passen kann.

2.2 Umsetzung

3 Dreieckspuzzle

3.1 Lösungsidee

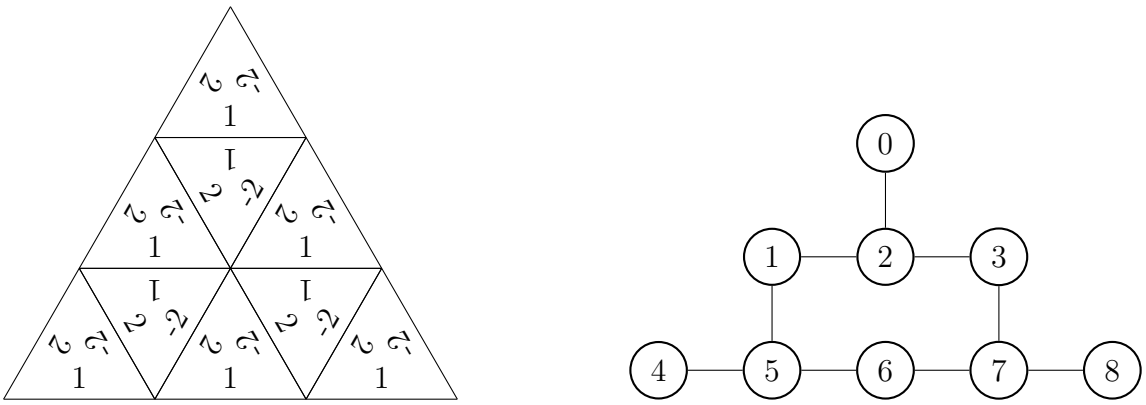
Ein Algorithmus soll durch systematisches Probieren alle sinnvollen Möglichkeiten testen. Das Programm legt erst das erste Puzzleteil, versucht dann dessen anliegende Kanten mit Teilen zu füllen. Sollte es für ein Puzzleteil keine verfügbaren Teile mehr geben, die an die Kante passen, geht der Algorithmus 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 9 gezeigt.

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.



(a) Das Puzzle (b) Darstellung als Graph

Abbildung 1: Interne Speicherung der Puzzlestruktur

3.2.1 Klassen

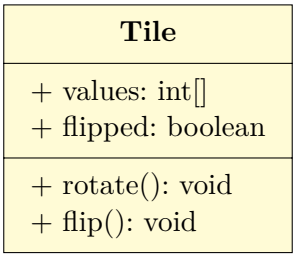


Abbildung 2: Die Klasse *Tile*

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°.

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.

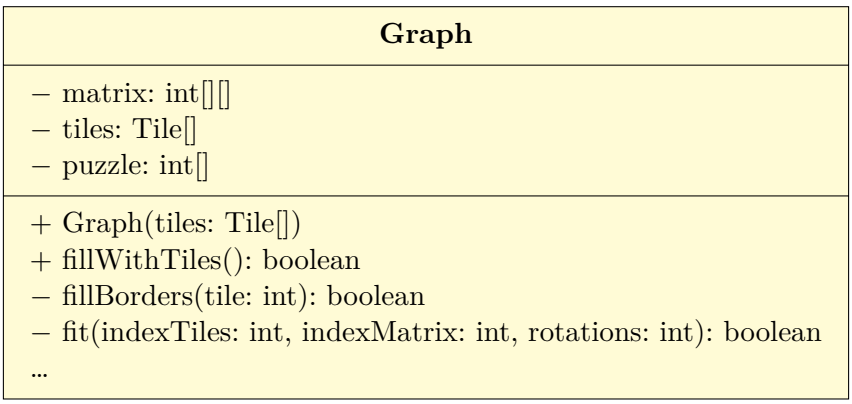


Abbildung 3: Die Klasse *Graph*

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.

```
1  if(fit(k, j, 0)) {
2     tileFound = true;
3     resetTile(j);
4     puzzle[k] = j;
5     updateTrueLink(tile, j);
6     placedTiles.add(j);
7
8     if(fillBorders(puzzle[k])) {
9        break;
10    }
11     else {
12       resetTile(j);
13       updateFalseLink(tile, j);
14       tileFound = false;
15        continue;
16     }
17 }
18 if(!tileFound) {
19     // [...] alle vom Tile erstellten Tiles werden ebenfalls gelöscht
20     return false;
21 }
22 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.

```
1  int difference = indexMatrix - i;
2  int side1 = 1;
3  int side2 = 1;
4  int tile2 = getIndexTiles(i);
5
6  if(difference == -1) {
7      side1 = 2;
8     side2 = 0;
9  }
10 else if(difference == 1) {
11    side1 = 0;
12    side2 = 2;
13 }
14
15 if(tiles[indexTiles].values[side1] + tiles[tile2].values[side2] != 0) {
16     fits = false;
17     break;
18 }
```

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. 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

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.

3.3.4 Puzzle3

Laut dem Algorithmus ist dieses Puzzle nicht lösbar.

4 Tobis Turnier

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

Abbildung 10: Teil 5 wird von 1 angelegt

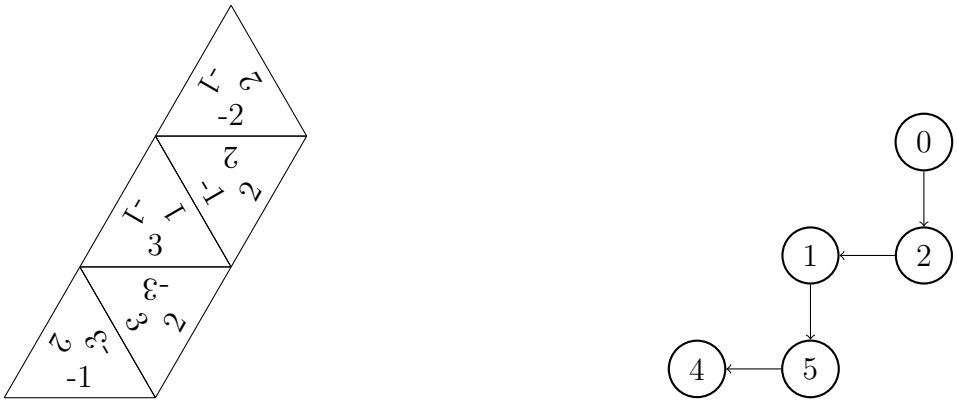


Abbildung 11: Teil 4 wird von 5 angelegt

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.

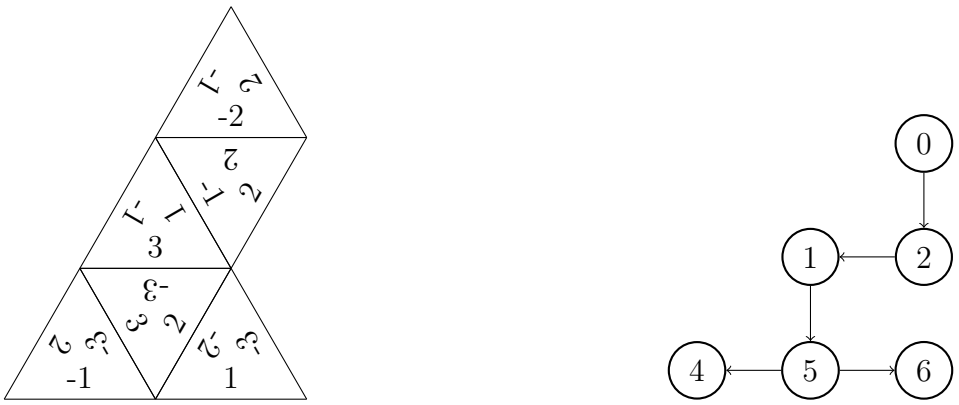


Abbildung 12: Teil 6 wird von 5 angelegt

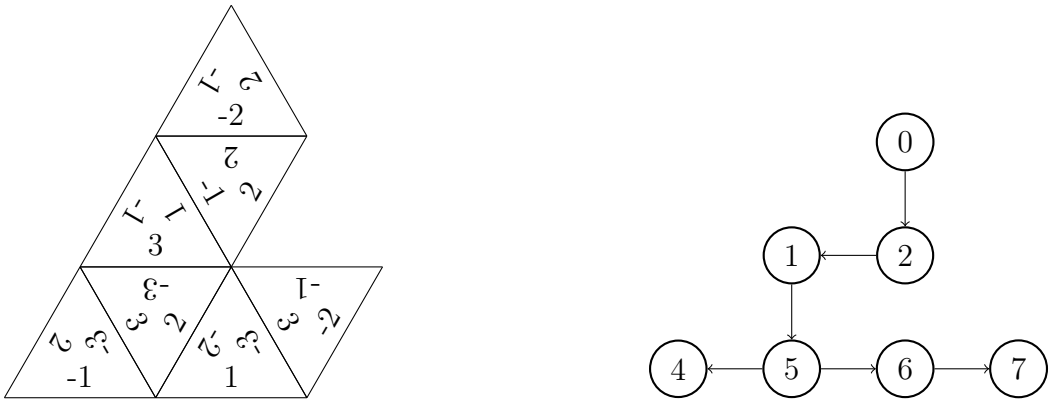


Abbildung 13: Teil 7 wird von 6 angelegt

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

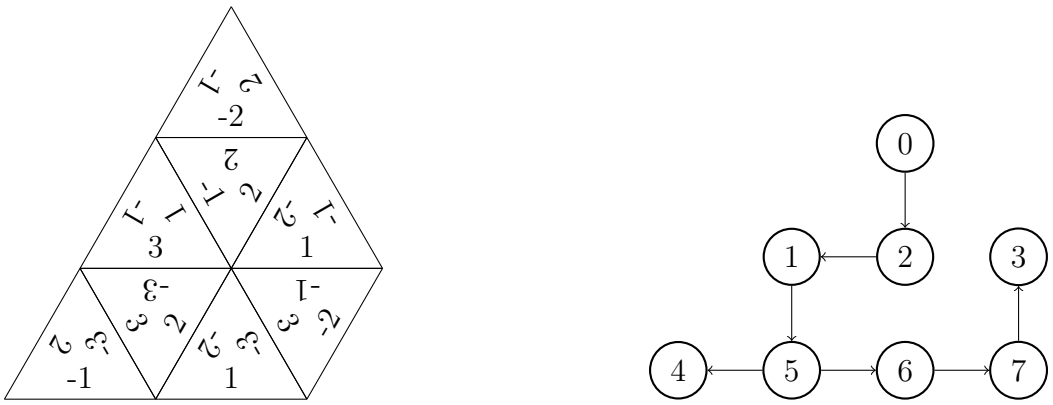


Abbildung 14: Teil 3 wird von 7 angelegt

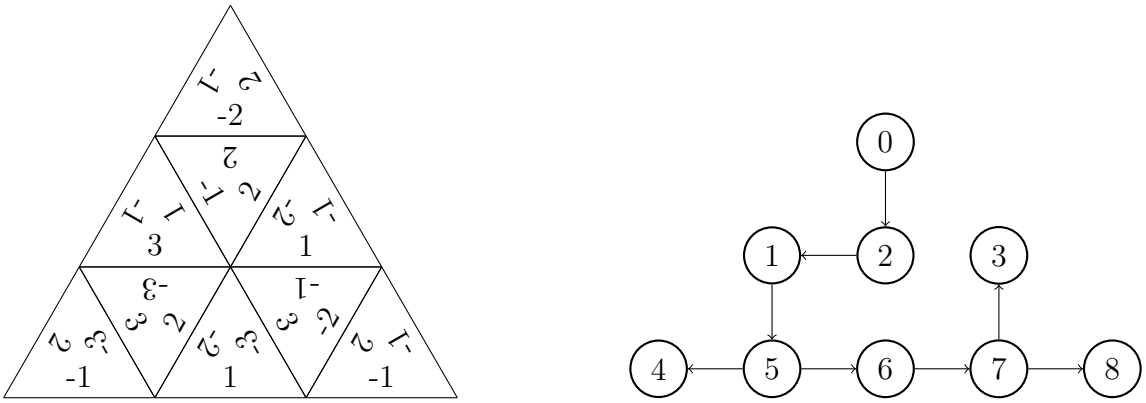


Abbildung 15: Teil 8 wird von 7 angelegt

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 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 20 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.

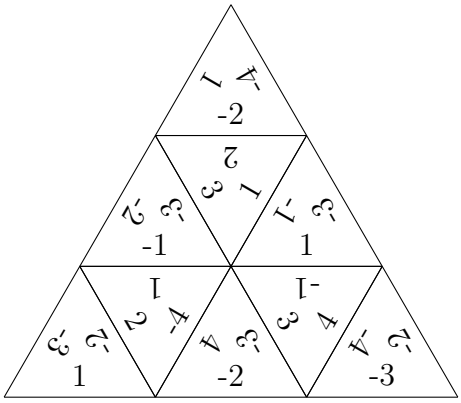


Abbildung 16: Die Lösung von Puzzle2

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

Abbildung 17: Die Klasse *Student*

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.

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

Abbildung 18: Die Klasse *Present*

```
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 19: Die Methode *changeStudent()*

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

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 20: Die Implementierung des Algorithmus