

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

von Jan Ehehalt, Jonathan Hager

Inhaltsverzeichnis

1	Wörter aufräumen	3
1.1	Lösungsidee	3
1.2	Umsetzung	3
2	Dreieckspuzzle	3
2.1	Lösungsidee	3
2.2	Umsetzung	3
2.2.1	Klassen	3
2.2.2	Algorithmus	5
2.3	Beispiele	6
3	Tobis Turnier	6
4	Streichholzrätsel	6
5	Wichteln	6
5.1	Lösungsidee	6
5.2	Umsetzung	7
5.3	Beispiele	8

1 Wörter aufräumen

1.1 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.

1.2 Umsetzung

2 Dreieckspuzzle

2.1 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.

2.2 Umsetzung

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

2.2.1 Klassen

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

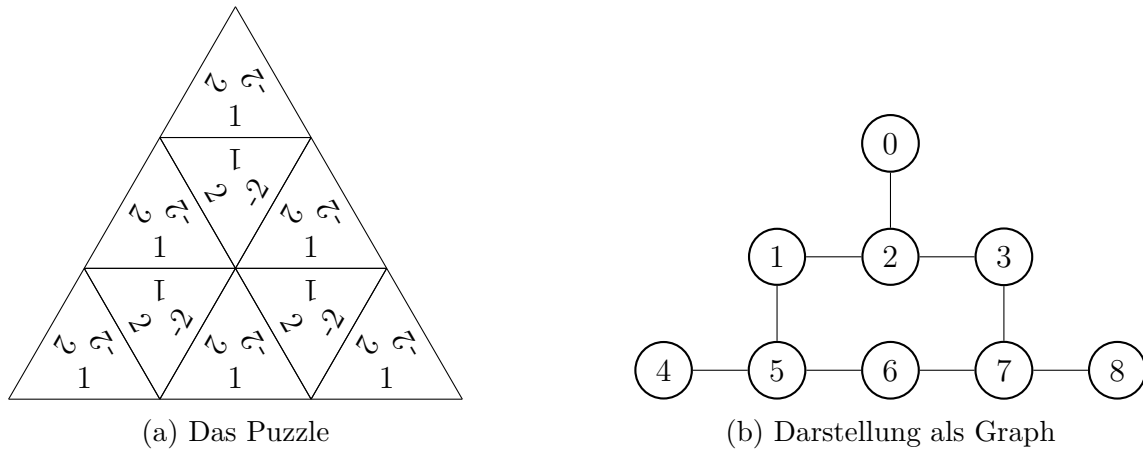


Abbildung 1: Interne Speicherung der Puzzlestruktur

Graph
<ul style="list-style-type: none"> – matrix: int[][] – tiles: Tile[] – puzzle: int[]
<ul style="list-style-type: none"> + Graph(tiles: Tile[]) + fillWithTiles(): boolean – fillBorders(tile: int): boolean – fit(indexTiles: int, indexMatrix: int, rotations: int): boolean ...

Abbildung 2: Die Klasse *Graph*

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.

2.2.2 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 3.

```
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
14         // Puzzle konnte nicht gelöst werden, Tile wird rotiert
15         resetPuzzle();
16         puzzle[i] = 0;
17         tiles[i].rotate();
18
19         if(fillBorders(0)) {
20             return true;
21         }
22
23         resetPuzzle();
24         puzzle[i] = 0;
25         tiles[i].rotate();
26
27         if(fillBorders(0)) {
28             return true;
29         }
30     }
31     return false;
32 }
```

Abbildung 3: 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 eine Teil bereits probiert wurde, wird beim

visited Array geschaut. 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 ist in Abbildung 4 zu sehen.

Die Methode *fit()* prüft für eine gegebene Stelle im Graphen, ob ein gegebenes Puzzle-teil passt. 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. Der Quellcode ist in den Abbildungen 5 bis 7 zu finden.

2.3 Beispiele

3 Tobis Turnier

4 Streichholzrätsel

Im Folgenden sieht man ein Codebeispiel für Code, der beispielhaft geschrieben wurde.

```
1 // Kommentare sind ne geile Sache
2 import com.badlogic.ShitRenderer;
3
4 public static void main(String[] args){
5     Controller.control(theWorld);
6     Math.atan(1/0);
7 }
```

Im Codebeispiel sieht man, wie der JavaScript Compiler intern arbeitet. Besondere Achtung sollte hierbei dem Math-Befehl gegeben werden, denn Math wurde nicht importiert und deshalb crasht es bereits deshalb, nicht wegen der ZeroDivision, da JavaScript hier einfach das Ergebnis würfeln wurde. # Ehre

5 Wichteln

5.1 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.

5.2 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.

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 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 solange 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 11 zu finden.

5.3 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

```
1 private boolean fillBorders(int tile) {
2
3     // [...] visited[] wird initialisiert und alle benutzten Teile werden auf
4     // true gesetzt
5
6     // Für jede anliegende Kante des Tiles wird versucht einen Partner zu
7     // finden
8     ArrayList<Integer> placedTiles = new ArrayList<>();
9     for(int j = 0; j < matrix.length; j++) {
10
11         if(matrix[tile][j] == 0) {
12
13             boolean tileFound = false;
14             for(int k = 0; k < visited.length; k++) {
15                 if(!visited[k]) {
16                     visited[k] = true;
17
18                     // Das Tile wird probiert
19                     if(fit(k, j, 0)) {
20                         tileFound = true;
21                         resetTile(j);
22                         puzzle[k] = j;
23                         updateTrueLink(tile, j);
24                         placedTiles.add(j);
25
26                         if(fillBorders(puzzle[k])) {
27                             break;
28                         }
29                     }
30                     else {
31                         resetTile(j);
32                         updateFalseLink(tile, j);
33                         tileFound = false;
34                         continue;
35                     }
36                 }
37             }
38         }
39     }
40     if(!tileFound) {
41         // [...] alle vom Tile erstellten Tiles werden ebenfalls
42         // gelöscht
43         return false;
44     }
45     return true;
46 }
```

Abbildung 4: Die Methode *fillBorders*

```
1 private boolean fit(int indexTiles, int indexMatrix, int rotations) {
2     if(indexMatrix == 2 || indexMatrix == 5 || indexMatrix == 7) {
3         if(!tiles[indexTiles].flipped) {
4             tiles[indexTiles].flip();
5         }
6     }
7     else {
8         if(tiles[indexTiles].flipped) {
9             tiles[indexTiles].flip();
10        }
11    }
12
13    if(rotations > 2) {
14        return false;
15    }
```

Abbildung 5: Das Tile wird, wenn nötig, geflippt und die Methode beendet, wenn das Tile bereits zweimal rotiert wurde

```
1 boolean fits = true;
2 for(int i = 0; i < matrix.length; i++) {
3     if(matrix[indexMatrix][i] != -1 && getIndexTiles(i) != -1) {
4         int side = indexMatrix - i;
5
6         int side1 = 0;
7         int side2 = 0;
8         int tile2 = getIndexTiles(i);
9
10        if(side == -1) {
11            side1 = 2;
12            side2 = 0;
13        }
14        else if(side == 1) {
15            side1 = 0;
16            side2 = 2;
17        }
18        else {
19            side1 = 1;
20            side2 = 1;
21        }
22
23        if(tiles[indexTiles].values[side1] + tiles[tile2].values[side2] != 0) {
24            fits = false;
25            break;
26        }
27    }
28 }
```

Abbildung 6: Alle gefundenen Kanten werden geprüft

```

1  if(fits) {
2      for(int i = 0; i < matrix.length; i++) {
3          if(matrix[indexMatrix][i] != -1 && getIndexTiles(i) != -1) {
4              updateTrueLink(indexMatrix, i);
5          }
6      }
7      return true;
8  }
9  else {
10     tiles[indexTiles].rotate();
11     return fit(indexTiles, indexMatrix, rotations + 1);
12 }

```

Abbildung 7: Das Tile wird eingefügt, wenn alles passt. Ansonsten wird es rotiert und erneut probiert

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

Abbildung 8: Die Klasse *Student*

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

Abbildung 9: Die Klasse *Present*

```
1 boolean changeStudent(Student[] students, int wish, int studentId) {
2     if(this.wish > wish){
3         if(this.studentId >= 0) {
4             students[this.studentId].hasGift = false;
5         }
6
7         this.studentId = studentId;
8         this.wish = wish;
9         students[studentId].hasGift = true;
10
11     return true;
12 }
13 else {
14     return false;
15 }
16 }
```

Abbildung 10: Die Methode *changeStudent()*

```
1 // Phase 1
2 for(int i=0; i<students.length; i++) {
3     students[i].requestPresent(presents, students, students[i].wishes[0], 0);
4 }
5
6 // Phase 2
7 boolean finished = false;
8 do {
9     // Die Schleife ist prinzipiell immer fertig, außer es wird noch ein
10    // Schüler gefunden, der kein Geschenk hat
11    finished = true;
12    for(int i = 0; i < students.length; i++) {
13        if(!students[i].hasGift) {
14            finished = false;
15
16            // Es wird versucht, einen der Wünsche zu erfüllen
17            if(!students[i].asked[students[i].wishes[0]]) {
18                students[i].requestPresent(presents, students,
19                    students[i].wishes[0], 0);
20            } else if(!students[i].asked[students[i].wishes[1]]) {
21                students[i].requestPresent(presents, students,
22                    students[i].wishes[1], 1);
23            } else if(!students[i].asked[students[i].wishes[2]]) {
24                students[i].requestPresent(presents, students,
25                    students[i].wishes[2], 2);
26            } else {
27                // Da bereits alle Wünsche probiert wurden, wird versucht
28                // irgendein Geschenk zu bekommen
29                for(int j = 0; j < presents.length; j++) {
30                    if(!students[i].asked[j]) {
31                        students[i].requestPresent(presents, students, j, 3);
32                    }
33                }
34            }
35        }
36    }
37 } while(!finished);
```

Abbildung 11: Die Implementierung des Algorithmus