

34. BUNDESWETTBEWERB INFORMATIK

RUNDE 1
01.09. - 30.11.2015

Aufgabe 1

Kassiopeias Weg

25. September 2015

Eingereicht von: *Der Skript-Tim*

Tim Hollmann
ich@tim-hollmann.de

Verwaltungs-Nr.: 34.00003

Ich versichere hiermit, die vorliegende Arbeit ohne unerlaubte fremde Hilfe entsprechend den Wettbewerbsregeln des Bundeswettbewerb Informatik angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Tim Hollmann, den 25. September 2015

INHALTSVERZEICHNIS

Inhaltsverzeichnis	2
1 Lösungsidee	3
1.1 Abstraktion des Problems	3
1.1.1 Junioraufgabe 2	3
1.1.2 Aufgabe 1	3
1.2 Algorithmus zum Lösen des Problems	4
1.2.1 Junioraufgabe 1	4
1.2.2 Aufgabe 1	4
1.3 Warum der Algorithmus richtig ist	5
1.3.1 Junioraufgabe 2	5
1.3.2 Aufgabe 1	5
1.4 Laufzeit- und Speicherplatzkomplexität - Theoretisch	6
1.4.1 Junioioraufgabe 2	6
1.4.2 Aufgabe 1	6
2 Umsetzung	6
2.1 Kommandozeilenargumente	6
2.1.1 Junioraufgabe 2	6
2.1.2 Aufgabe 1	6
2.2 Repräsentation des Spielfeldes	6
2.3 Tiefensuche	7
2.4 Funktionen und ihre Aufgaben	7
2.4.1 Junioraufgabe 2	7
2.4.2 Aufgabe 1	7
2.5 Auswertung	7
2.6 Kompilat	8
3 Beispiele	8
4 Quelltext	11
4.1 Junioraufgabe 2	11
4.2 Aufgabe 1	13
Literatur	17
Abbildungsverzeichnis	17

1 LÖSUNGSIDEE

1.1 ABSTRAKTION DES PROBLEMS

Quadranten sei definiert als Summe der weißen und schwarzen Felder Q_w und Q_s . Die Anzahl der weißen Felder Q_w sei definiert als n . Kassiopeias Startpunkt sei $S = (S_x|S_y)$.

1.1.1 JUNIORAUFGABE 2

Konkretes Ziel ist es, herauszufinden, ob alle weißen Felder Q_w Quadranten erreichbar sind. Zunächst definiere ich jedes weiße Feld als einen Zustand; jeder dieser Zustände ist durch seine x - und y -Koordinaten eindeutig identifizierbar. Von einem Zustand lassen sich im Normalfall ein oder mehrere weitere Zustände erreichen; analog zu der Bewegung auf dem Feld. Diese Möglichkeiten der Zustandsveränderung lassen sich in einer Adjazenzmatrix und daraus folgend einem Graphen darstellen.

In einen Graphen übertragen bedeutet die Tatsache, dass alle Zustände/Felder erreichbar sind, dass kein Zustand isoliert sein darf (so wie z.B. in `⊠ kassiopeia1.txt`).

Ich definiere deshalb:

Definition 1 (*Erreichbarkeit im Graphen*)

Erreichbar ist ein Feld, wenn eine Kantenfolge existiert, die seinen Zustand mit dem Startpunkt-Zustand verbindet. Diese Kantenfolge ist maximal so lang, wie die Anzahl der weißen Felder n beträgt.

Für eine Ausgabe „Ja“ müssen alle Felder erreichbar sein; für ein „Nein“ mindestens eines nicht.

Um dies zu überprüfen, eignet sich eine Tiefensuche, die vom Startpunkt S ausgeht und deren Tiefe auf die Anzahl der weißen Felder n begrenzt ist. Die Tiefensuche vermeidet bereits erreichte Zustände. Wenn die Anzahl der so erreichten Zustände der Anzahl der weißen Felder auf dem Spielfeld entspricht, sind alle weißen Felder erreichbar.

1.1.2 AUFGABE 1

Die Erweiterung im Bezug auf Junioraufgabe 2 besteht bei dieser Aufgabe konkret darin, dass eine Punktfolge gesucht ist, die exakt die Länge der Anzahl der weißen Felder besitzt und jeden Zustand exakt ein mal beinhaltet; eine solche Punktfolge wird auch Hamiltonweg genannt.

Um diesen Pfad – falls er denn existiert – zu ermitteln, muss ein intelligent geschriebener Algorithmus für Junioraufgabe 2 (der bereits erreichte Zustände vermeidet) nur minimal verändert werden; er muss lediglich bei jeder Rekursion der Tiefensuche überprüfen, ob er sich auf maximaler Tiefe (n -ter Ebene) befindet. Wenn er dies ist, handelt es sich im aktuellen Zustand um eine Lösung, da der Algorithmus – bereits erreichte Zustände

vermeidend – die Ebene erreicht hat, die der Anzahl der weißen Felder entspricht und somit alle weißen Felder-Zustände exakt einmal beinhaltet.

1.2 ALGORITHMUS ZUM LÖSEN DES PROBLEMS

1.2.1 JUNIORAUFGABE 1

Der Algorithmus ist eine Tiefensuche, die die Zustände in einem globalen Stack speichert und vermeidet. Implementiert ist die Tiefensuche als rekursive Funktion.

Algorithmus 1 : Junioraufgabe 2

Eingabe : \emptyset
Daten : Spielfeld $Q\{Q_w, Q_s\} \Rightarrow \text{field}$, Startposition $\Rightarrow \text{startZustand}$
Ausgabe : **bool** Alle Felder erreichbar

```

1 Prozedur Tiefensuche(aktuellerZustand, int tiefe)
2   if tiefe >  $|Q_w|$  oder aktuellerZustand bereits in erreichteZustaende then
3     return;
4   else
5     aktuellerZustand in erreichteZustaende speichern.
6     Bewegung nach oben;
7     if aktuellerZustand.x < field.x UND field[aktuellerZustand.x + 1,
      aktuellerZustand.y]  $\in Q_w$  then
8       tempzustand  $\leftarrow$  aktuellerZustand;
9       tempzustand.x++;
10      Tiefensuche(tempZustand, tiefe + 1);
11     Bewegung nach unten, rechts, links;
12     [...]
1 erreichteZustaende = Stack von Typ Zustand;
2 Tiefensuche(startZustand, 1);
3 if  $|erreichteZustaende| = |Q_w|$  then
4   return true;
5 else
6   return false;

```

1.2.2 AUFGABE 1

Dieser Algorithmus unterscheidet sich im wesentlichen von dem von Junioraufgabe 2 darin, dass die Zustände nicht in einem globalen Stack gespeichert werden, sondern in einer temporären Schlange, die den „Pfad“ der Zustände beschreibt, der von der Tiefensuche genommen wurde. Außerdem erfolgt die Überprüfung, ob alle Zustände erreicht wurden aufgrund der temporären Schlangen nicht mehr nach Beendigung der Tiefensuche,

sondern in jeder Iteration.

Algorithmus 2 : Aufgabe 1

Eingabe : \emptyset

Daten : Spielfeld $Q\{Q_w, Q_s\} \Rightarrow \text{field}$, Startposition $S \Rightarrow \text{startZustand}$

Ausgabe : Liste der möglichen Pfade

```

1 Prozedur Tiefensuche(pfad, int tiefe)
2   if tiefe =  $|Q_w|$  then
3     endzustände  $\leftarrow$  pfad;
4     return;
5   aktuellerZustand  $\leftarrow$  letzter Eintrag von pfad;
6   //Bewegung nach oben;;
7   if aktuellerZustand.x < field.x UND field[aktuellerZustand.x + 1,
    aktuellerZustand.y]  $\in Q_w$  then
8     tempzustand  $\leftarrow$  aktuellerZustand;
9     tempzustand.x++;
10    tempzustand.action = 'N';
11    if Nicht tempzustand in pfad enthalten then
12      t  $\leftarrow$  pfad;
13      t.add tempzustand;
14      Tiefensuche(t, tiefe + 1);
15  Bewegung nach unten, rechts, links;
16  [...]

1 endzustände = Liste;
2 Tiefensuche(startZustand, 1);
3 return endzustände;

```

1.3 WARUM DER ALGORITHMUS RICHTIG IST

1.3.1 JUNIORAUFGABE 2

Der Algorithmus ist richtig, da er alle Bewegungsmöglichkeiten auf dem Spielfeld vom Startpunkt aus simuliert. Die nicht erreichbaren Felder müssen zwangsläufig übrig bleiben.

1.3.2 AUFGABE 1

Gesucht ist ein Kantenpfad, in dem sämtliche Zustände exakt einmal vorkommen. Wenn die Tiefensuche unter Vermeidung bereits erreichter Zustände auf n -ter Ebene ist, muss sie alle Zustände genau ein mal beinhalten und ist eine Lösung.

1.4 LAUFZEIT- UND SPEICHERPLATZKOMPLEXITÄT - THEORETISCH

1.4.1 JUNIORAUFGABE 2

Da durch den globalen Stapel der erreichten Zustände die Tiefe der Tiefensuche stark (auf annähernd die Anzahl der weißen Felder) beschränkt wird, nehme ich auch die Laufzeitkomplexität als annähernd linear an. $\mathcal{O}(k * n)$

1.4.2 AUFGABE 1

Das Finden eines Hamiltonweges ist NP-vollständig [1, Kap.2]. Demzufolge hatte auch der von mir implementierte Algorithmus von Anfang an wenig Chancen. Ob/in wiefern meine Tiefensuche einen Approximationsalgorithmus darstellt, kann ich nicht beurteilen. Die geringen Spielfeldgrößen der Angabe und der Beispiele stellen für meinen Algorithmus keinerlei Problem dar; vergrößert man das Spielfeld aber erheblich, so ist die Bearbeitungszeit meines Algorithmus stark von der konkreten Form des Spielfeldes abhängig, sofern man alle möglichen Lösungswege ermitteln will; wird lediglich eine mögliche Route gefordert (mit dem Kommandozeilenparameter `-o` zu erreichen), ist die Zunahme der Bearbeitungszeit bei stark zunehmender Spielfeldgröße erst erheblich später spürbar, als wenn man alle möglichen Hamiltonwege ermitteln wollte.

Da laut Aufgabenstellung eigentlich nur ein Lösungsweg gefordert ist, empfiehlt sich bei Aufgabe 1 grundsätzlich die Verwendung des Kommandozeilenparameters `-o`; bei den geringen Spielfeldgrößen der Angabe ist dies aber - wie gesagt - unerheblich.

2 UMSETZUNG

Die Umsetzung erfolgte in C++ (Standard C++-11) unter Visual C++ 2015.

2.1 KOMMANDOZEILENARGUMENTE

2.1.1 JUNIORAUFGABE 2

`-f` `--file` [filename] Pfad/Dateiname der Datei mit Syntax der BwInf-Materialvorlagen

2.1.2 AUFGABE 1

`-f` `--file` [filename] Pfad/Dateiname der Datei mit Syntax der BwInf-Materialvorlagen
`-o` `--justOneSolution` Falls nur eine mögliche Lösung erwünscht ist

2.2 REPRÄSENTATION DES SPIELFELDES

Die per Kommandozeilenparameter übergebene Datendatei wird ausgelesen und in einer zweidimensionalen `bool`-Matrix gespeichert.

```
1 std::vector<std::vector<bool>> > field;
```

Dabei repräsentiert `true` ein weißes Feld und `false` entsprechend ein schwarzes.

2.3 TIEFENSUCHE

Die Implementierung der Tiefensuche erfolgte in beiden Fällen in Form einer rekursiven Funktion `Application::tiefensuche`, deren Initialisierung gleich nach Einlesen des Spielfeldes in `Application::main()` geschieht.

Aufgabe 1 unterscheidet sich nur in sofern von Junioraufgabe 2, als dass bei der Rekursion ein Entscheidungs-Pfad als wachsendes Argument mitgeführt wird und in jeder Iteration der Tiefensuche nun eine Überprüfung stattfindet, ob die sich in tiefster Ebene befindet.

2.4 FUNKTIONEN UND IHRE AUFGABEN

Allgemein

Funktion	Aufgabe
<code>Application::main()</code>	Hauptfunktion; Auswerten der Kommandozeilenparameter, Initiieren des Ladens der Datendatei und der Tiefensuche, Ausgabe der Lösungen nach der Tiefensuche
<code>Application::loadFromFile()</code>	Auslesen der angegebenen Datendatei
<code>Application::tiefensuche()</code>	Rekursive Tiefensuch-Funktion

2.4.1 JUNIORAUFGABE 2

Funktion	Aufgabe
<code>Application::warSchonDa()</code>	Gibt an, ob ein gegebener Zustand bereits erreicht wurde (Element in <code>erreichteZustaende</code> ist).

2.4.2 AUFGABE 1

Funktion	Aufgabe
<code>Application::warSchonDa()</code>	Gibt an, ob ein gegebener Zustand in einem gegebenen temporären Entscheidungspfad bereits erreicht wurde
<code>Application::goTo()</code>	Helferfunktion bei den Bewegungssimulationen

2.5 AUSWERTUNG

Im Gegensatz zu Aufgabe 1 ist in Junioraufgabe 2 nach der Ausführung der Tiefensuche eine Auswertung der ermittelten Zustände nötig; sind weniger Zustände erreicht worden als weiße Felder auf dem Spielfeld, sind nicht alle Felder erreichbar und die Ausgabe lautet: „Nein“, andernfalls „Ja“.

In Aufgabe 1 müssen die Ergebnisse (bzw. die in `Coord::action` gespeicherten Bewegungen N, S, W, O) lediglich ausgegeben werden.

2.6 KOMPILAT

Ich habe für beide Aufgaben den Quelltext unter Windows sowohl für 32- als auch für 64-Bit kompiliert; sie sind unter `☞ [aufgabe] ▶ bin ▶ WinX[Bit] ▶ [aufgabe].exe` zu finden.

Laufzeitumgebung

Die von mir kompilierten Programme erfordern standardmäßig die „Microsoft Visual C++ Redistributable für Visual Studio 2015“-Laufzeitumgebung. Ich habe alternative Versionen der Programme beigelegt, die ohne diese relativ neue Runtime funktionieren, deren Funktionalität dadurch aber eingeschränkt sein könnte. Sie sind unter `☞ [aufgabe] ▶ bin ▶ WinX[Bit] ▶ [aufgabe]Alternativ.exe` zu finden.

3 BEISPIELE

Hinweis

Alle Bearbeitungen meiner Programme zu allen aufgeführten Spielfeldern können Sie auch als direkt abgespeicherte Ausgabe im Verzeichnis der jeweiligen Aufgabe unter `☞ [aufgabe] ▶ data ▶ [quelldatei]_bearbeitet.txt` finden.

Datei	Spielfeld	Junioraufgabe 2 - Ausgabe	Aufgabe 1 - Ausgabe
kassiopeia0.txt	##### # # # # # # # K # # # # # #####	Ja	1 : +WNNWSSSOOONNNNOOSSSONNN 2 : +WNNWSSSOOONNNNOOSSSWNN 3 : +WNNWSSSOOONNNNOOSWSSON 4 : +WNNWSSSOOONNNNOOSWSOSW
kassiopeia1.txt	##### # # # # # # # # # # K # # # # # # #####	Nein	- Kein Weg gefunden. -
kassiopeia2.txt	##### # K # # # # # # # # #####	Ja	- Kein Weg gefunden. -
kassiopeia3.txt	##### # K # # # # # # # # #####	Ja	1 : +WSSSOOOONOSONNWWW 2 : +WSSSOOOOONNWSWNW 3 : +OOSONOSSWWWWWNNO 4 : +OOOSSWNWSWWWNNO
kassiopeia4.txt	##### # K # #####	Ja	- Kein Weg gefunden. -
kassiopeia5.txt	##### # K# #####	Ja	1 : +WWWWWWWWWWW
kassiopeia6.txt	##### #K # # # # # # # #####	Ja	- Kein Weg gefunden. -
kassiopeia7.txt	##### #K # # # # # # # # # # # #####	Ja	- Kein Weg gefunden. -

eigen1.txt	##### # K # ## # # # # # # # #####	Nein	- Kein Weg gefunden. -
eigen2.txt	##### # K # # #### # # # # # # # # #####	Ja	1: +SSWWNONWWWWWWSSWNNWSSSOOONOSOOOONNN 2: +SSWWNONWWWWWWSWNWSSSONOSOOONOSOOOONNN 3: +SSWWNONWWWWWWSSSONNOSOOONOSOOOONNN 4: +SSWWNONWWWWWWWSOOSWWSOOONOSOOOONNN 5: +WSSWNNWWWWSSWNNWSSSOOONOSOOOONWNON 6: +WSSWNNWWWWSWNWSSSONOSOOONOSOOOONWNON 7: +WSSWNNWWWWSSSONNOSOOONOSOOOONWNON 8: +WSSWNNWWWWWSOOSWWSOOONOSOOOONWNON 9: +WSWNWWWWSSWNNWSSSOOONOSOOONOSOOONWNON 10: +WSWNWWWWSWNWSSSONOSOOONOSOOONOSOOONWNON 11: +WSWNWWWWWWSSSONNOSOOONOSOOONOSOOONWNON 12: +WSWNWWWWWSOOSWWSOOONOSOOONOSOOONWNON 13: +WSOSWNNWWWWSSWNNWSSSOOONOSOOOONNN 14: +WSOSWNNWWWWSWNWSSSONOSOOONOSOOOONNN 15: +WSOSWNNWWWWWWSSSONNOSOOONOSOOOONNN 16: +WSOSWNNWWWWWSOOSWWSOOONOSOOOONNN 17: +WWWWWWSSWNNWSSSOOONOSOOONNOSOOONWNON 18: +WWWWWWSSWNNWSSSOOONOSOOONNOOSWOOONNN 19: +WWWWWWSSWNNWSSSOOONOSOOONWNOOSSONNN 20: +WWWWWWSSWNNWSSSOOONOSOOOONWWWN000N [...]
eigen3.txt	##### # # # # # ## # # # # # # # ## # # # # # # K# #####	Ja	- Kein Weg gefunden. -
eigen4.txt	##### # # # # # ### # ### # #K ## # # #####	Ja	1: +OONNWWNOOOSOOONNOOOOSSSWNWWWSW 2: +OONNWWNOOOSOOSONOOSONNNWWWS


4 QUELLTEXT

4.1 JUNIORAUFGABE 2

```

6 struct Application
7 {
8
9     struct Coord
10    {
11        unsigned int x, y;
12    };
13
14    auto main(const std::vector<std::string>& arguments) -> int;
15    auto loadFromFile(void) -> bool;
16
17    auto Application::tiefensuche(Coord path, const int deep) -> void;
18    auto Application::warSchonDa(Coord f) -> bool;
19
20    std::string _filename;
21
22    std::vector<std::vector<bool> > field;
23    unsigned int startX, startY = -1;
24    int anzahlWeisseFelder;
25
26    std::vector<Application::Coord> erreichteZustaende;
27
28    std::vector< std::vector<Application::Coord> > endzustaende;
29 };

```

Listing 1:  j2\inc\Application.hpp - Application-Headerdatei

```

7 #include "../inc/Application.hpp"
8
9 auto Application::main(const std::vector<std::string>& arguments) -> int
10 {
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50 if (!Application::loadFromFile()) return EXIT_FAILURE;
51
52 std::cout << "\nGeladenes Spielfeld ( " << field.size() << " x " << ↵
53     field[0].size() << "): \n\n";
54
55 for (auto zeile : Application::field) {
56     for (auto spalte : zeile) {
57         std::cout << ((spalte) ? " " : "#");
58     }
59     std::cout << std::endl;
60 }
61
62 std::cout << "\nStartposition: ( " << startX + 1 << " | " << startY ↵
63     + 1 << " )";

```

```
62
63 Application::Coord startPosition;
64 startPosition.x = startX;
65 startPosition.y = startY;
66
67 std::cout << "\nAnzahl der Weißen Felder: " << anzahlWeisseFelder;
68
69 std::cout << "\n\nStarte Tiefensuche ...";
70
71 // Tiefensuche starten
72 tiefensuche(startPosition, 1);
73
74 std::cout << "fertig.\n\n" << "Können alle Blätter erreicht werden? ↵
75 : ";
76 // Wenn Anzahl der weißen Felder gleich Anzahl der Felder ist, die ↵
77 // als einzigartiges Feld gefunden wurden, ist die Ausgabe "Ja"
78 if (erreichteZustaende.size() == anzahlWeisseFelder) {
79     std::cout << "Ja.";
80     return EXIT_SUCCESS;
81 }
82 else {
83     std::cout << "Nein.";
84     std::cout << "\nKassiopeia stirbt! :(";
85     return EXIT_SUCCESS;
86 }
87
88 }
89
90 auto Application::loadFromFile(void) -> bool // Laden des Spielfeldes ↵
91     aus einer Datei
92 {
93     [...]
94 }
95
96
97 auto Application::warSchonDa(Application::Coord f) -> bool
98 {
99     for (auto p : erreichteZustaende)
100         if (p.x == f.x && p.y == f.y) { return true; }
101     return false;
102 }
103
104 auto Application::tiefensuche(Coord currentField, const int deep) -> void
105 {
106
107     if (deep > anzahlWeisseFelder) return;
108
109     erreichteZustaende.push_back(currentField);
110
111     // nach oben
112     if (currentField.y > 0 && field[currentField.y - ↵
113         1][currentField.x]) {
114         Coord temp = currentField;
115         temp.y--;
116     }
117 }
```

```

168     if (!warSchonDa(temp))
169         tiefensuche(temp, deep + 1);
170 }
171
172 // nach unten
173 if (currentField.y + 1 < field.size() && field[currentField.y + 1][currentField.x]) {
174     Coord temp = currentField;
175     temp.y++;
176
177     if (!warSchonDa(temp))
178         tiefensuche(temp, deep + 1);
179 }
180
181 // nach links
182 if (currentField.x > 0 && field[currentField.y][currentField.x - 1]) {
183     Coord temp = currentField;
184     temp.x--;
185
186     if (!warSchonDa(temp))
187         tiefensuche(temp, deep + 1);
188 }
189
190 // nach rechts
191 if (currentField.x + 1 < field[0].size() && field[currentField.y][currentField.x + 1]) {
192     Coord temp = currentField;
193     temp.x++;
194     if (!warSchonDa(temp))
195         tiefensuche(temp, deep + 1);
196 }
197 }
198 }

```

Listing 2: `j2\src\Application.cpp` - Application - Quelldatei

4.2 AUFGABE 1

```

6 struct Application
7 {
8
9     struct Coord
10    {
11        unsigned int x, y;
12        char action;
13    };
14
15    auto main(const std::vector<std::string>& arguments) -> int;
16    auto loadFromFile(void) -> bool;
17
18    auto Application::tiefensuche(std::vector<Coord> path, int deep) -> bool;

```

```

19  auto Application::warSchonDa(std::vector<Application::Coord> path, ↵
    Application::Coord f) -> bool;
20  auto Application::goTo(Application::Coord pos, std::vector<Coord> ↵
    path, const int deep) -> bool;
21
22  std::string _filename;
23
24  bool _justOneWay = false;
25
26  std::vector<std::vector<bool> > field;
27  unsigned int startX, startY = -1;
28  int anzahlWeisseFelder;
29
30  std::vector< std::vector<Application::Coord> > endzustaende;
31  };

```

Listing 3: 1►src►Application.hpp - Application - Headerdatei

```


72  auto Application::main(const std::vector<std::string>& arguments) -> int
73  {
74      [...]
75      Application::Coord startPosition;
76      startPosition.x = startX;
77      startPosition.y = startY;
78      startPosition.action = '+';
79
80      std::cout << "\nAnzahl der Weißen Felder: " << anzahlWeisseFelder;
81
82      std::cout << "\n\nStarte Ermittlung der möglichen Wege...";
83
84      std::vector<Application::Coord> path;
85      path.push_back(startPosition);
86
87      tiefensuche(path, 1);
88
89      // Endzustände ausgeben
90
91      std::cout << "\n\nErmittelte Wege: (" << endzustaende.size() << ")\n";
92
93      if (endzustaende.size() != 0) {
94          int counter = 0;
95          for (auto zust : endzustaende) {
96              std::cout << "\n" << ++counter << "\t: ";
97              for (auto p : zust) {
98                  std::cout << p.action;
99              }
100          }
101          return EXIT_SUCCESS;
102      }
103      else {
104          std::cout << "\n - Kein Weg gefunden. - ";

```

```
105     std::cout << "\n Kassiopeia stirbt! :(";
106     return EXIT_SUCCESS;
107 }
108
109 }

186 auto Application::tiefensuche(std::vector<Coord> path, const int deep) ←
    -> bool
187 {
188
189     if (deep == anzahlWeisseFelder)
190     {
191         endzustaende.push_back(path);
192         return true;
193     }
194
195     Coord currentField = path.back();
196
197     // nach oben
198     if (currentField.y > 0 && field[currentField.y-1][currentField.x])
199     {
200         Coord temp = currentField;
201         temp.y--;
202         temp.action = 'N';
203
204         if (!warSchonDa(path, temp))
205             if (goTo(temp, path, deep) && _justOneWay) return true;
206     }
207
208     // nach unten
209     if (currentField.y + 1 < field.size() && field[currentField.y + ←
        1][currentField.x])
210     {
211         Coord temp = currentField;
212         temp.y++;
213         temp.action = 'S';
214
215         if (!warSchonDa(path, temp))
216             if (goTo(temp, path, deep) && _justOneWay) return true;
217     }
218
219     // nach links
220     if (currentField.x > 0 && field[currentField.y][currentField.x - 1])
221     {
222         Coord temp = currentField;
223         temp.x--;
224         temp.action = 'W';
225
226         if (!warSchonDa(path, temp))
227             if (goTo(temp, path, deep) && _justOneWay) return true;
228     }
229
230     // nach rechts
```

```
231     if (currentField.x + 1 < field[0].size() && ↵  
232         field[currentField.y][currentField.x + 1])  
233     {  
234         Coord temp = currentField;  
235         temp.x++;  
236         temp.action = '0';  
237         if (!warSchonDa(path, temp))  
238             if (goTo(temp, path, deep) && _justOneWay) return true;  
239     }  
240  
241     return false;  
242  
243 }
```

Listing 4:  1 › src › Application.cpp - Application - Quelldatei

LITERATUR

- [1] WOTHE, MAURICE: *NP-Vollständigkeit, Proseminar Theoretische Informatik*. Freie Universität Berlin, Institut für Informatik, 2010. http://www.inf.fu-berlin.de/lehre/WS10/ProSem-ThInf/np_vollstaendigkeit_2.pdf.

ABBILDUNGSVERZEICHNIS

LISTINGS

1	✂ j2▸inc▸Application.hpp - Application-Headerdatei	11
2	✂ j2▸src▸Application.cpp - Application - Quelldatei	12
3	✂ 1▸src▸Application.hpp - Application - Headerdatei	13
4	✂ 1▸src▸Application.cpp - Application - Quelldatei	15