Portfolioprüfung - Werkstück A

im Modul

Betriebssysteme & Rechnernetze

Studiengang International Business Information Systems

Thema

Alternative 7 – Labyrinth – Dokumentation zum Code

Vorgelegt von:

Tim Fichtner	Muhammed Erdal Akkoc	Emmanuel David
Matrikel-Nr.: 1220889	Matrikel-Nr.: 1221147	Matrikel-Nr.: 1369740

Themensteller: Prof. Dr. Christian Baun

Abgabedatum: 28.06.2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Die nachfolgende Arbeit ist eine Dokumentation zum Bash-Code für das Werkstück A in dem Modul Betriebssysteme & Rechnernetze für die Alternative 7 „Labyrinth“. Die Anforderungen umfassen ein zufällig generiertes Labyrinth zu entwickeln, durch welches man einen Spieler mit Tastatureingaben bewegen kann. Dabei soll es nicht möglich sein durch Wände zu laufen. Der Anwender soll die Größe des Labyrinths bestimmen können. Es soll einen Eingang und einen Ausgang besitzen. Ist der Ausgang erreicht, soll eine Gewinnnachricht ausgegeben werden, welche auch die Spielzüge beinhaltet.

Die Dokumentation ist in zwei Abschnitte unterteilt. Der erste Abschnitt behandelt die Initialisierung und Generierung des Labyrinths. Hierbei sind die wesentlichen Funktionen

- die Bestimmung der Breite und Höhe des Labyrinths durch Benutzereingabe
- die Erstinitiierung des Labyrinth-Arrays
- die zufällige Generierung des Labyrinths mithilfe des Algorithmus des rekursiven Backtrackings
- und die Ausgabe des Labyrinths

Im zweiten Abschnitt wird auf die Bewegung des Spielers, die Kollision mit den Wänden und die Prüfung der Gewinnkondition eingegangen.

Anzumerken ist, dass diese Dokumentation nicht in Gänze auf jede Funktionalität des Programmes eingeht. Je Kapitel werden die entsprechenden Code-Ausschnitte dargestellt und erläutert. Dieser Ansatz wird dann genutzt, um die Entscheidungen für die dargelegten Vorgehensweisen zu begründen.

2. Initialisierung und Generierung des Labyrinths

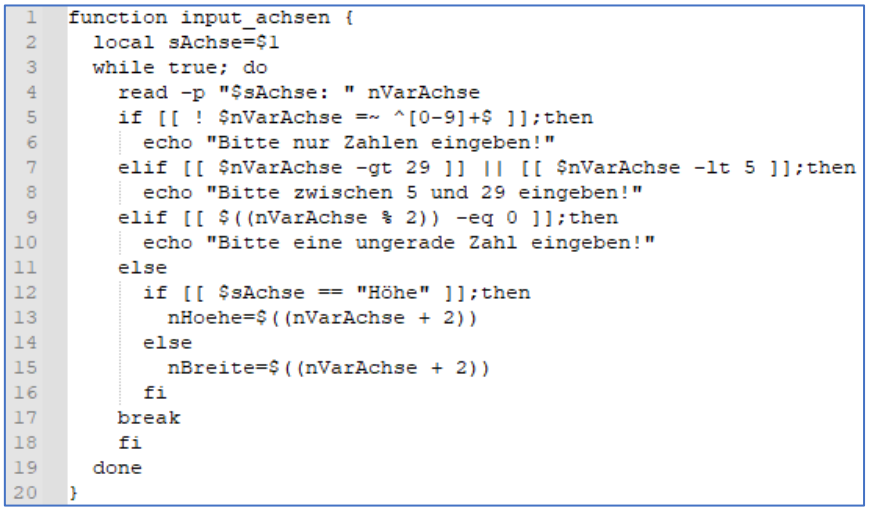
Zu Beginn des Programmes erhält der Benutzer eine Ausgabe, über die er aufgefordert wird, die Größe des Labyrinths zu bestimmen. Hierfür wird die Funktion "input_achsen" aufgerufen (Abb. 1 Zeile 5 und 6). Anschließend wird das Array initialisiert (Zeile 8) und das Labyrinth in der Funktion "rekursives_backtracking" erstellt (Zeile 9). Die genannten Funktionen werden in den Abschnitten 2.1 - 2.3 ausführlich beschrieben.

```
1  clear
2  echo -e "Portfolioprüfung - Werkstück A - Alternative 7 - Labyrinth\n"
3  echo -e "Bitte die Höhe und Breite des Labyrinths angeben.\nNur ungerade Zahlen zwischen 5 und 29 sind gültig.\n"
4
5  input_achsen "Höhe"
6  input_achsen "Breite"
7
8  lab_array_initiieren
9  rekursives_backtracking  $((2 * nBreite + 2))
```

Abbildung 1: Programmalgorithmus

2.1 Funktion „input_achsen“

In Abbildung 2 ist die Funktion „input_achsen“ zu sehen. Diese dient der manuellen Bestimmung der Höhe und Breite des Labyrinths.



```
1  function input_achsen {
2    local sAchse=$1
3    while true; do
4      read -p "$sAchse: " nVarAchse
5      if [[ ! $nVarAchse =~ ^[0-9]+$ ]];then
6        echo "Bitte nur Zahlen eingeben!"
7      elif [[ $nVarAchse -gt 29 ]] || [[ $nVarAchse -lt 5 ]];then
8        echo "Bitte zwischen 5 und 29 eingeben!"
9      elif [[ $((nVarAchse % 2)) -eq 0 ]];then
10       echo "Bitte eine ungerade Zahl eingeben!"
11     else
12       if [[ $sAchse == "Höhe" ]];then
13         nHoehe=$((nVarAchse + 2))
14       else
15         nBreite=$((nVarAchse + 2))
16       fi
17     break
18     fi
19   done
20 }
```

Abbildung 2: Funktion „input_achsen"

Der Anwender wird vor der Funktion auf die Limitationen der Benutzereingabe hingewiesen (Abb. 1 Zeile 3). Beim Aufruf der Funktion, wird ein Parameter als String übergeben („Höhe“ oder „Breite“). Dieser Wert wird an die lokale Variable „sAchse“ übergeben (Abb.2 Zeile 2). Anschließend wird eine while-Schleife initiiert. Hierbei wird der Anwender zur Eingabe der Größe des Labyrinths aufgefordert (Zeile 4). Diese Eingabe wird in der Variablen „nVarAchse“ gespeichert. In den folgenden if-Bedingungen wird geprüft, ob die Eingabe den definierten Begrenzungen entspricht. Im ersten Fall wird geprüft, ob die Eingabe eine Zahl ist (Zeile 5). Als Zweites wird geprüft, ob die Eingabe im definierten Zahlenbereich liegt. Die Untergrenze ist ein Wert von 5 und die Obergrenze ein Wert von 29. Erst ab einem Mindestwert von 5 hat der Anwender die Möglichkeit, den Spieler in jede der Richtungen zu bewegen. Die Obergrenze ist wichtig, damit die Ausführung des Programms performant bleibt.

Ein Performance-Problem tritt im Zusammenhang mit der Spielerbewegung auf, welche im Abschnitt 3.2 erklärt wird. Nach jeder Bewegung des Spielers in eine neue Zelle wird die Ausgabe des Labyrinths mittels „clear“-Command gelöscht. Anschließend wird das Labyrinth erneut ausgegeben mit der neu berechneten Spielerposition. Je größer das Labyrinth ist, desto länger dauert dieser Schritt. Dieses Problem erkennt man in abgeschwächter Form bei der Obergrenze von 29. Zwischen den Bewegungsschritten gibt es kurze Verzögerung, die für den Anwender wie ein Flackern erscheint.

In Zeile 9 ist die letzte Prüfung der gültigen Benutzereingabe. Diese legt fest, dass nur ungerade Zahlen eingegeben werden sollen. Dies ist notwendig, damit ein logisches von Wänden umgebenes Labyrinth ausgegeben wird. Genauer wird dies im Abschnitt 2.3 begründet. Bei einer gültigen Eingabe wird die Variable „nVarAchse“ um 2 erhöht (Zeile 12 bis 16), da ansonsten bei der Ausgabe die für den Anwender sichtbare Höhe oder Breite um 2 kleiner wäre. Dies wird im Abschnitt 2.3 nochmal visuell erläutert. Dieser Wert wird je nach Funktions-Parameter („Höhe“ oder „Breite“) den Variablen „nHoehe“ oder „nBreite“ zugewiesen.

2.2 Funktion „lab_array_initiieren“

Der Gedankengang zur Darstellung des Labyrinths umfasst, dass dieses aus einer Anzahl an Zellen besteht, welche durch „Höhe mal Breite“ definiert wird. Aufgrund dieser Idee war es naheliegend, das Labyrinth als Array zu deklarieren, bei dem die Obergrenze an Indexeinträgen auf „Höhe mal Breite“ begrenzt ist. Jeder Indexeintrag stellt eine Zelle dar und erhält im Folgenden einen Wert, um zu bestimmen, welche Art von Zelle ausgeben wird. Die Darstellung der Zellen erfolgt im Abschnitt 2.3. Die möglichen zugewiesenen Werte sind entweder 0 (eine Wand), 1 (ein leeres begehbares Feld) oder 2 (ein Token, welches den Spieler darstellt). Die folgende Funktion in Abbildung 3 stellt die erstmalige Initiierung des Arrays dar.

```
1  function lab_array_initiieren {
2    for ((y=0; y<nHoehe; y++)) ; do
3      for ((x=1; x<$((nBreite-1)); x++)) ; do
4        lab_array[$((y * nBreite + x))]=0
5      done
6      lab_array[$((y * nBreite + 0))]=1
7      lab_array[$((y * nBreite + (nBreite - 1)))]=1
8    done
9    for ((x=0; x<nBreite; x++)) ; do
10     lab_array[$x]=1
11     lab_array[$(((nHoehe - 1) * nBreite + x))]=1
12   done
13   lab_array[$((nBreite + 2))]=2
14   lab_array[$(((nHoehe - 2) * nBreite + nBreite - 3))]=1
15 }
```

Abbildung 3: Funktion „lab_array_initiieren"

In der Shell Bash ist es nicht möglich, nativ ein mehrdimensionales Array zu deklarieren. Um die Darstellung eines zweidimensionalen Arrays zu simulieren, wird in der Funktion eine verschachtelte for-Schleife verwendet. Dabei wird je y-Wert (Zeile) jeder x-Wert (Spalte) initiiert. In der inneren for-Schleife (Zeile 3 bis 5) wird dabei je Zeile jeder x-Wert auf 0 (Wand) gesetzt, ausgenommen vom ersten und letzten x-Wert jeder Zeile. Diese werden auf 1 (leeres Feld) gesetzt nach der inneren for-Schleife (Zeile 6 und 7). Zweidimensional verbildlicht besteht das Labyrinth nun komplett aus Wänden ausgenommen der ersten und letzten Spalte, welche leere Felder sind. In der zweiten for-Schleife wird daraufhin die letzte und erste Zeile auf 1 gesetzt (Zeile 9 bis 12). Das Labyrinth ist nun einmal rundum von leeren Feldern umgeben. Im letzten Schritt werden zwei bestimmte Indexeinträge auf 1 und 2 gesetzt. Diese stellen im Späteren die anfängliche Spielerposition im Eingangsfeld (Zeile 13) und den Ausgang des Labyrinths dar (Zeile 14). Bei einem Array mit der Höhe und Breite 7 würde dieses nach der Funktion nun so ausgegeben werden (ohne Leerzeichen):

lab_array: 1111111 1020001 1000001 1000001 1000001 1000101 1111111

Damit dieses noch „zweidimensional“ dargestellt wird, muss nach jeder Zeile ein Zeilenumbruch geschehen. Diese Prozedur wird erst in Abschnitt 3.1 näher beschrieben. Für ein besseres Verständnis, wie das Labyrinth dargestellt wird, wird dieser Schritt jedoch vorgezogen. Im Folgenden ist links in Tabelle 1 das Labyrinth mit Indexwerten je Zelle zu sehen. Rechts in Tabelle 2 ist dieses zu sehen nach Zuweisung der Indexwerte zu Symbolen, welches ebenso bei der Funktion „labyrinth_ausgeben“ geschieht. Dabei ist zu erkennen, dass bei einer Höhe und Breite von 7 das für den Anwender sichtbare Feld sich auf 5 mal 5 beschränkt, da dieses von leeren Feldern umgeben ist. Aus diesem Grund ist es wichtig, dass die Werte bei der manuellen Benutzereingabe im Abschnitt 2.1 um 2 erhöht werden, damit das sichtbare Feld mit den Eingaben des Anwenders übereinstimmt. Die Entscheidung für die leeren Felder und für die Position des Eingangs und Ausgangs wird im nächsten Abschnitt begründet.

1	1	1	1	1	1	1
1	0	2	0	0	0	1
1	0	0	0	0	0	1
1	0	0	0	0	0	1
1	0	0	0	0	0	1
1	0	0	0	1	0	1
1	1	1	1	1	1	1

Tabelle 1: Array Ausgabe

	■	•	■	■	■	
	■	■	■	■	■	
	■	■	■	■	■	
	■	■	■	■	■	
	■	■	■		■	

Tabelle 2: Ausgabe mit Symbolen

2.3 Rekursives Backtracking zur Generierung des Labyrinths

Nachdem das Array initialisiert wurde, erfolgt nun die eigentliche Generierung des Labyrinths. Hierfür wurde der Algorithmus des rekursiven Backtrackings verwendet. Im Folgenden wird diese Entscheidung begründet anhand der Erklärung der Funktionsweise des Algorithmus und der Gegenüberstellung zur Ausgabe von anderen Algorithmen. Um nicht den Rahmen dieser Arbeit zu überschreiten, werden die Funktionsweisen weiterer Algorithmen außen vorgelassen. Im Abschnitt 2.3.2 wird die Umsetzung des Algorithmus im Programm beschrieben. Hierbei werden die Besonderheiten bei der Umsetzung in Bash erklärt.

2.3.1 Funktionsweise des Algorithmus

Bei der Durchführung des Algorithmus werden die folgenden Schritte vollzogen, um ein zufälliges Labyrinth zu generieren:

1. Wähle eine zufällige Zelle aus und betrachte diese als aktuelle Zelle
2. Markiere diese Zelle als „besucht"
3. Solange diese Zelle „unbesuchte“ Nachbarzellen hat:
   3.1. Wähle eine der „unbesuchten“ Nachbarzellen zufällig aus
   3.2. Entferne die Wand zwischen der aktuellen Zelle und der Nachbarzelle
   3.3. Die ausgewählte Nachbarzelle wird nun als aktuelle Zelle betrachtet
   3.4. Wiederhole die Schritte rekursiv

Der Algorithmus des rekursiven Backtrackings kann als langer Pfad verstanden werden, welcher sich durch das Labyrinth "schlängelt". Dabei erfolgt bei jedem Schritt zu einer nächsten Zelle ein weiterer rekursiver Aufruf der Funktion. Diese Prozedur wiederholt sich fortlaufend, bis kein weiteres Feld "besucht" werden kann. Ist dies der Fall, geht der Algorithmus auf die vorherige Zelle zurück („Backtracking“) und prüft, ob eine andere Nachbarzelle „unbesucht“ ist. Wenn eine „unbesuchte“ Zelle ermittelt wird, wird ein neuer Pfad eröffnet mit weiteren Rekursionsinstanzen. Dies garantiert, dass alle Zellen des Labyrinths am Ende des Algorithmus „besucht“ werden.

Nachteilig beim rekursiven Backtracking ist die Performanz bezüglich Verarbeitungsleistung. Jede geöffnete Instanz muss solange im Arbeitsspeicher des PCs gespeichert bleiben bis sie vollständig abgearbeitet wurde. Je größer das Labyrinth, desto länger ist der Pfad, und desto mehr Rekursionsinstanzen befinden sich zur selben Zeit im Stack. Dies ist einer der Gründe für die Festlegung der Obergrenze der Achsen in Abschnitt 2.1.

Im Gegensatz zu anderen Algorithmen wird beim rekursiven Backtracking ein besonders langer Pfad durch das Labyrinth garantiert, der für den Anwender auf den ersten Blick nicht erkennbar ist. Damit soll der Anwender mental herausgefordert werden und somit der Spielspaß erhöht werden. Außerdem weist dieser Algorithmus weniger „Dead-Ends“ (Sackgassen) als zum Beispiel der „Kruskal-Algorithmus“ auf. Der Pfad wirkt „zufälliger“ im Vergleich zu dem Algorithmus „rekursive Division“, da bei letzterem das Labyrinth in mehrere Sektoren aufgeteilt wird und diese wiederum in mehrere Untersektoren. Im Folgenden ist eine graphische Gegenüberstellung der drei Labyrinthtypen beispielartig zu sehen.

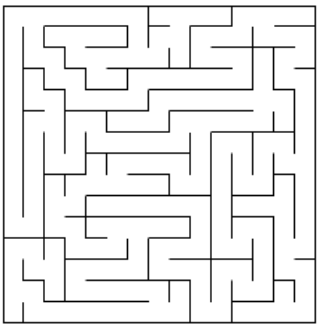


Abbildung 4: Rekursives Backtracking

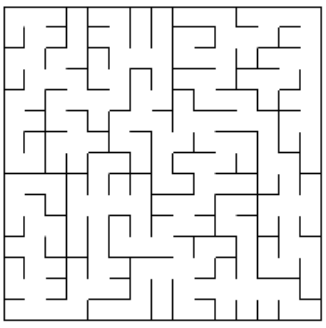


Abbildung 5: Kruskal-Algorithmus

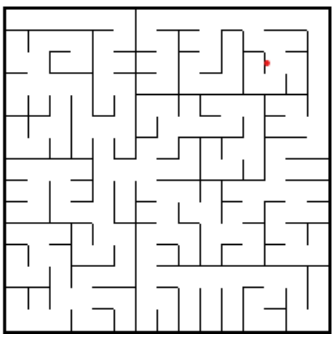


Abbildung 6: Rekursive Division

2.3.2 Funktion „rekursives_backtracking“

Die Funktion des rekursiven Backtrackings ist ähnlich, wie im Abschnitt 2.3.1 in der Erklärung nach Schritten, aufgebaut. Allerdings gibt es Besonderheiten, die bei der Programmierung in Bash umgesetzt werden mussten. Diese werden im Folgenden näher erläutert. Im Abschnitt 2 wird beim ersten Aufruf der Funktion ein fester Parameter übergeben, der die Anfangszelle der Funktion beschreibt (Abb. 8).

```
1  rekursives_backtracking  $((2 * nBreite + 2))
```

Abbildung 7: Erstmaliger Funktionsaufruf „rekursives_backtracking"

Diese Zelle ist im Gegensatz zur vorherigen Erklärung des Algorithmus nicht zufällig, sondern befindet sich immer eine Zelle unter der anfänglichen Spielerposition. Damit soll gewährleistet werden, dass der längste Pfad immer unterhalb des Spielers anfängt. Hierbei erklärt sich auch die Festlegung des Ausgangs unten rechts im Labyrinth. Der Abstand zwischen Spielerposition und Ausgang ist somit am Größten. Zu Beginn der Funktion wird der übergebene Parameter als Startposition der Variable „nAktuelle_zelle" übergeben (Abb. 7 Zeile 2). Anhand dieses Indexes wird im Array das erste leere Feld gesetzt (Zeile 5). Zunächst wird in der while-Schleife der Rest einer vorher generierten Zufallszahl anhand Modulo vier berechnet (Zeile 7). Der dadurch berechnete Rest der Zufallszahl kann nur eine von vier möglichen Ergebnissen haben. Entsprechend des berechneten Ergebnisses wird eine der vier möglichen Richtungen bestimmt (Zeile 8 bis 16). Diese sind links (-1), rechts (+1), oben (-$nBreite) und unten ($nBreite). Dadurch wird zufällig eine Nachbarzelle als nächste Zelle ausgewählt.

```
1  function rekursives_backtracking {
2    local nAktuelle_zelle=$1
3    local nZufallszahl=$RANDOM
4    local i=0
5    lab_array[$nAktuelle_zelle]=1
6      while [ $i -le 3 ] ; do
7        nModZufallszahl=$((nZufallszahl % 4))
8        if [[ $nModZufallszahl -eq 0 ]];then
9          nRichtung=1
10       elif [[ $nModZufallszahl -eq 1 ]];then
11         nRichtung=-1
12       elif [[ $nModZufallszahl -eq 2 ]];then
13         nRichtung=$nBreite
14       elif [[ $nModZufallszahl -eq 3 ]];then
15         nRichtung=$((-$nBreite))
16       fi
17       local nNaechste_zelle=$((nAktuelle_zelle + nRichtung))
18         if [[ lab_array[$nNaechste_zelle] -eq 0 ]];then
19           local nUebernaechste_zelle=$((nNaechste_zelle + nRichtung))
20           if [[ lab_array[$nUebernaechste_zelle] -eq 0 ]];then
21             lab_array[$nNaechste_zelle]=1
22             rekursives_backtracking $nUebernaechste_zelle
23           fi
24         fi
25       i=$((i + 1))
26       nZufallszahl=$((nZufallszahl + 1))
27     done
28 }
```

Abbildung 8: Funktion „rekursives_backtracking"

Nach der im Abschnitt 2.3.1 definierten Schrittweise sollte nun die Wand zwischen beiden Zellen durchbrochen werden und die nächste Zelle dann zur aktuellen Zelle umgeschrieben werden. Bei der Umsetzung in Bash wird die Wand, die durchbrochen wird, als eigene Zelle betrachtet. Dies wird in der Programmierung durch eine doppelte Prüfung gelöst. Wenn die nächste Zelle, in die der Pfad fortgeführt werden soll, eine Wand ist (Zeile 18), wird geprüft, ob die übernächste Zelle in der gleichen Richtung ebenso eine Wand ist (Zeile 19 und 20). Hierbei wird die nächste Zelle in ein leeres Feld umgewandelt (Zeile 21). Es wird eine weitere Rekursionsinstanz aufgerufen, die den Wert der übernächsten Zelle als neuen Parameter für die aktuelle Zelle an die Funktion übergibt (Zeile 22). Da am Anfang der Rekursion die aktuelle Zelle ebenso auf 1 gesetzt wird (Zeile 2), kann der Pfad je Rekursionsinstanz als zwei Schritte gleichzeitig visualisiert werden, bei denen Wandzellen auf leere Zellen gesetzt werden. Angelehnt an die visuelle Darstellung des Arrays in Abschnitt 2.2 würde ein 7 mal 7 Labyrinth nach vollständigem Durchlauf wie in Tabelle 3 aussehen:

	■	•	■	■	■	
	■	1		2	■	
	■	■	■		■	
	■	4		3	■	
	■	■	■		■	

Tabelle 3: Labyrinth nach rekursivem Backtracking

Die Zahlen in den Zellen von Tabelle 3 stellen die aktuellen Zellen der einzelnen Rekursionsinstanzen dar. Betrachtet wird nun zuerst die Zelle mit Zahl 1, welches die anfängliche Rekursionsinstanz darstellt. Angenommen der Algorithmus hat geprüft, ob es möglich ist, nach links einen Pfad zu generieren. Die erste if-Bedingung ist zugetroffen, da sich eine Wand in der nächsten Zelle befindet. In der übernächsten Zelle befindet sich jedoch ein leeres Feld, da zu Anfang bei der Funktion „lab_array_initiieren“ das Labyrinth von leeren Feldern umgeben wurde. Dies garantiert, dass die Außenwände des Labyrinths bis auf Eingang und Ausgang komplett erhalten bleiben und der Algorithmus nur im Inneren einen Pfad generiert. Die Einschränkung der Benutzereingabe auf ungerade Zahlen für die Höhe und Breite wird hiermit auch erklärt: Der Pfad umfasst von einer Rekursionsinstanz zur nächsten immer eine ungerade Anzahl an Zellen. Wenn eine gerade Zahl bei einem der beiden Achsen eingegeben wird, wird eine Außenwand des Labyrinths nicht mehr angezeigt. Somit wäre der eindeutige Ausgang nicht mehr für den Anwender erkennbar.

Wenn der Algorithmus nicht zwei Felder in eine Richtung weitergehen kann, wird keine neue Rekursionsinstanz aufgerufen. Die Zählvariable der Schleife und die Zufallszahl werden um eins erhöht (Abb. 7 Zeile 25 und 26). Da die Zufallszahl mit Modulo vier die Richtung bestimmt, die geprüft wird, wird somit garantiert, dass jede Richtung genau einmal geprüft wird. Angenommen der Algorithmus befindet sich in einer Rekursionsinstanz, in der keine der Richtungen mehr begehbar ist, wie zum Beispiel bei Instanz vier in der vorherig dargestellten Tabelle 3. Diese kehrt in die vorherige Rekursionsinstanz drei zurück. Die Bearbeitung wird fortgesetzt und es wird geprüft, ob der Pfad in eine andere Richtung fortgeführt werden kann. Ist dies nicht der Fall, wird in die Rekursionsinstanz zwei zurückgekehrt. Sobald alle Instanzen jede Richtung auf mögliche Pfade geprüft haben, wird die Funktion beendet und ein zufälliges Labyrinth ist generiert.

3. Ausgabe und Spielerbewegung

Nachdem das Labyrinth generiert wurde, muss dieses nun auch für den Anwender sichtbar gemacht werden. Dies geschieht in der Funktion „labyrinth_ausgeben“ (Abb. 9 Zeile 9). Des Weiteren soll es für den Anwender möglich sein, einen Token als Spieler durch das Labyrinth zu bewegen. Hierfür werden vorher notwendige Variablen deklariert (Zeile 1 bis 5).

```
1  sRichtung="0"
2  nAktuelle_position=$((nBreite + 2))
3  nAusgang=$(((nHoehe - 2) * nBreite + nBreite - 3))
4  nEingang=$((nBreite + 2))
5  nZuege="0"
6  sEingaben="a -> Nach links bewegen\ns -> Nach rechts bewegen\nd ->
7                 Nach unten bewegen\nw -> Nach oben bewegen\nq -> Spiel beenden"
8  clear
9  labyrinth_ausgeben
10 echo -e "Züge = 0\n"
11 echo -e $sEingaben
```

Abbildung 9: Ausgabe und Deklaration von Spielervariablen

3.1 Funktion „labyrinth_ausgeben“

Bis zu diesem Punkt wurden bisher lediglich die einzelnen Werte für jeden Indexeintrag des Arrays definiert. Diese bestehen wie schon vorher erwähnt aus den Werten 0, 1 oder 2. Die Darstellung dieser Werte in Symbolen erfolgt nun in der Ausgabefunktion, welche unten in Abbildung 10 dargestellt ist.

```
1  function labyrinth_ausgeben {
2    for ((y=0; y<nHoehe; y++)) ; do
3      for ((x = 0; x<nBreite; x++ )) ; do
4        if [[ lab_array[$((y * nBreite + x))] -eq 0 ]];then
5          echo -n -e "\u2588\u2588"
6        elif [[ lab_array[$((y * nBreite + x))] -eq 2 ]];then
7          echo -n -e "\u25CF "
8        else
9          echo -n "  "
10       fi
11     done
12   echo
13   done
14 }
```

Abbildung 10: Funktion „labyrinth_ausgeben"

Hierbei wird durch if-Bedingungen geprüft, welcher Wert vorliegt und weist diesen dann ein entsprechendes UTF-8-Symbol hinzu (Zeile 4 bis 7). Bei dem Wert 0 erscheint eine Wand mit dem Symbol „[redacted]“ (Zeile 5). Dies sind zwei schwarze Rechtecke, die nebeneinander ein Quadrat darstellen sollen, damit dies visuell ansprechender ist. Der Wert 2 stellt den Spieler dar mit dem Symbol „● “ (Zeile 7). Andernfalls wird ein leeres Feld ausgegeben (Zeile 9). In der vorletzten Zeile erfolgt der Zeilenumbruch durch die leere „echo“-Ausgabe (Zeile 12). Aus diesem Grund wird nochmals eine verschachtelte for-Schleife ausgeführt ähnlich wie bei der Funktion „lab_array_initiieren“, sodass nach jeder Zeile ein Zeilenumbruch geschieht. Die fertige Ausgabe des Labyrinths würde wie in Tabelle 3 in Abschnitt 2.3.2 aussehen.

3.2 Spielerbewegung

In Abbildung 11 wird in der Programmierung die Bewegung des Spielers durch Tastatureingabe ermöglicht.

```
1  while [ $sRichtung != "q" ];do
2    read  -n1 -s sRichtung
3    clear
4    labyrinth_ausgeben
5
6    if [[ $sRichtung == a ]];then
7      nNeue_position=$((nAktuelle_position - 1))
8      kollisionspruefung
9    elif [[ $sRichtung == d ]];then
10     nNeue_position=$((nAktuelle_position + 1))
11     kollisionspruefung
12   elif [[ $sRichtung == s ]];then
13     nNeue_position=$((nAktuelle_position + nBreite))
14     kollisionspruefung
15   elif [[ $sRichtung == w ]];then
16     nNeue_position=$((nAktuelle_position - nBreite))
17     if [[ $nNeue_position -lt $nEingang ]];then
18       echo "Falsche Richtung. Der Ausgang befindet sich unten rechts."
19       echo -e "Züge = $nZuege\n"
20       echo -e "Bewegung mit den Pfeiltasten: asdw\nSpiel Beenden: q"
21       continue
22     fi
23     kollisionspruefung
24   fi
25
26   if [[ $nAktuelle_position -gt $nAusgang ]];then
27     echo "Züge = $nZuege"
28     echo "Herzlichen Glückwunsch! Du hast $nZuege Züge gebraucht."
29     read
30     exit
31   fi
32
33   if [[ ! $sRichtung == *[asdwq] ]];then
34     echo "Bitte nur die Buchstaben asdw oder q eingeben!"
35   fi
36
37   echo -e "Züge = $nZuege\n"
38   echo -e $sEingaben
39 done
40
41 clear
42 labyrinth_ausgeben
43 echo -e "Züge = $nZuege\n"
44 read -p "Das Spiel wurde beendet."
```

Abbildung 11: Spielerbewegung und Gewinnkondition

Der Anwender wird in Zeile 2 aufgefordert, eine Richtungstaste einzugeben. Die möglichen Optionen werden unterhalb des Labyrinths als String ausgegeben (Zeile 38). Die Richtungstasten zur Spielerbewegung sind mit „a“ eine Zelle nach links (Zeile 6 bis 8), „d“ eine Zelle nach rechts (Zeile 9 bis 11), „s“ eine Zelle nach unten (Zeile 12 bis 14) oder „w“ eine Zelle nach oben (Zeile 15 bis 24). Mit der Taste „q“ hat er jederzeit die Möglichkeit die while-Schleife und somit das Spiel zu beenden (Zeile 1). Hierbei wird außerhalb der while-Schleife auch eine entsprechende Meldung ausgegeben (Zeile 41 bis 44). Wenn der Anwender eine andere Eingabe tätigt, wird ebenso eine entsprechende Meldung ausgegeben und die Schleife springt zurück an den Anfang (Zeile 33 bis 35).

Sollte der Anwender eine gültige Richtungseingabe getätigt haben, wird die Variable „nNeue_position“ deklariert. Diese wird mit dem Wert der Variable „nAktuelle_position“ und dem entsprechenden Richtungswert berechnet (z.B. für die Richtung links: Abb. 11 Zeile 7). Diese Variable entspricht dem Array-Wert der angrenzenden Zelle. Ob es möglich ist, den Spieler in die Zelle zu bewegen, wird in der Funktion „kollisionspruefung“ geprüft (Abbildung 12). Anzumerken ist, dass im Fall der Richtungstaste „w“ noch geprüft wird, ob sich der Anwender außerhalb des Labyrinths über den Eingang aus bewegen möchte (Zeile 17 bis 22).

```
1  function kollisionspruefung {
2    if [[ ${lab_array[$nNeue_position]} = 1 ]];then
3      lab_array[$nAktuelle_position]=1
4      lab_array[$nNeue_position]=2
5      nAktuelle_position=$nNeue_position
6      let nZuege++
7      clear
8      labyrinth_ausgeben
9    else
10     echo "Hier ist eine Mauer."
11   fi
12 }
```

Abbildung 12: Funktion „kollisionspruefung"

Befindet sich in dem Array „lab_array" an der Position „nNeue_position" der Wert 1 (leeres Feld) kann der Spieler dieses Feld betreten (Abb. 12 Zeile 2). Dabei wird die aktuelle Position des Spielers entsprechend in dem Array auf die neue Position gesetzt (Zeile 4). Der Wert des Feldes in dem Array, auf dem sich der Spieler zuvor befunden hat, wird auf 1 (leeres Feld) gesetzt (Zeile 3) Bei der erfolgreichen Bewegung zählt eine Zählervariable die Anzahl der Züge hoch (Zeile 6). Nach jeder Bewegung muss das Labyrinth neu ausgegeben werden (Zeile 7 und 8). Wie im Abschnitt 2.2 erwähnt, entstehen bei diesem Schritt die Performance-Probleme bei einem großen Labyrinth.

Ähnlich wie bei der Prüfung, ob der Spieler sich oberhalb des Eingangs bewegen will, wird geprüft, ob der Spieler sich unterhalb des Ausgangs bewegt hat. Den entsprechenden Abschnitt sieht man in Abbildung 13. Dabei wird geprüft, ob der Wert der Variablen „nAktuelle_position“ größer als der Wert der Variablen „nAusgang“ ist (Zeile 26). Wenn dies der Fall ist, hat sich der Spieler außerhalb des Labyrinths bewegt und somit erfolgreich das Labyrinth durchlaufen. Es wird die Anzahl der benötigten Züge und eine Gewinnnachricht ausgegeben (Zeile 27 und 28). Anschließend wird das Spiel beendet (Zeile 30).

```
26  if [[ $nAktuelle_position -gt $nAusgang ]];then
27    echo "Züge = $nZuege"
28    echo "Herzlichen Glückwunsch! Du hast $nZuege Züge gebraucht."
29    read
30    exit
31  fi
```

Abbildung 13: Gewinnkondition