

Aufgabe 2: Simultane Labyrinth

Teilnahme-ID: 74749

Bearbeiter/-in dieser Aufgabe:
Christian Krause

27. April 2025

Inhaltsverzeichnis

1	Lösungsidee	1
1.1	Modellierung	1
2	Breitensuche	2
2.0.1	Laufzeitkomplexität	3
2.1	Bidirektionale Breitensuche	4
2.1.1	Optimalität	5
2.1.2	Laufzeit	5
3	Umsetzung	5
3.1	Breitensuche	5
3.2	Bidirektionale Breitensuche	6
3.2.1	Optimierung	7
3.3	Weitere Optimierungsmöglichkeiten	7
4	Beispiele	8
4.1	Interessante Beispiele	8
5	Erweiterung	10

1 Lösungsidee

Ich habe mich damit beschäftigt, einen Algorithmus zu entwickeln, der eine optimale Lösung für die Aufgabenstellung berechnen kann. Eine optimale Lösung ist hier die kürzeste Anweisungssequenz, die Anton und Bea ins Ziel bringt. In diesem Abschnitt wird direkt der Aufgabenteil b) behandelt, der Algorithmus ist aber auch für Labyrinth ohne Gruben geeignet.

1.1 Modellierung

Im folgenden arbeite ich mit der Annahme, dass die Labyrinth von Wänden umgeben sind. Jeder Zustand, in dem sich Anton und Bea befinden, kann als Tupel der jeweiligen Koordinaten dargestellt werden: $((x_0, y_0), (x_1, y_1))$. (x_0, y_0) ist hier die Position von Anton (der sich in Labyrinth 1 befindet), (x_1, y_1) beschreibt die Position von Bea im zweiten Labyrinth. Am Anfang herrscht der Zustand $S_0 = ((0, 0), (0, 0))$, da sich beide auf ihrem Startfeld befinden. Chris kann nun vier mögliche Anweisungen geben, die einen neuen Zustand herbeiführen würden. Wenn Anton und Bea beide ihr Zielfeld erreicht haben, befindet wir uns in dem Zustand $S_{end} = ((n - 1, m - 1), (n - 1, m - 1))$. Formal können die verschiedenen Positionen von Anton und Bea als Graph dargestellt werden, dessen Knoten alle Möglichen Zustände sind.

$$V = \{((x_0, y_0), (x_1, y_1)) \in ((\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N})) \mid 0 \leq x_0, x_1 < n \wedge 0 \leq y_0, y_1 < m\}$$

1/10

Von jedem Knoten gehen vier Kanten aus, eine für jede Anweisung, die Chris geben könnte. Die Menge der Anweisungen A kann formal als Menge an Funktionen dargestellt werden, die eine Position $p = (x_0, y_0)$ in eine neue Position p' (die oben, unten, rechts oder links von p ist) überführt:

$$A = \{(x_0, y_0) \mapsto (x_0 + 1, y_0), (x_0, y_0) \mapsto (x_0 - 1, y_0), (x_0, y_0) \mapsto (x_0, y_0 + 1), (x_0, y_0) \mapsto (x_0, y_0 - 1)\}$$

Ob Anton und Bea diese Anweisung ausführen können, hängt natürlich davon ab, ob sie die neue Position erreichen können, ohne gegen eine Wand zu stoßen oder in eine Grube zu fallen. Formal führen Anton und Bea an der Position p bei jeder Anweisung $a \in A$ folgende Funktion aus, um ihre neue Position p' zu bestimmen:

$$p' = f(p, a) = \begin{cases} p & \text{Falls zwischen } p \text{ und } a(p) \text{ eine Wand ist oder} \\ & \text{das Zielfeld erreicht ist (also } p = (n-1, m-1)). \\ (0, 0) & \text{Falls } a(p) \text{ eine Grube ist.} \\ a(p) & \text{ansonsten} \end{cases}$$

Da wir annehmen, dass das Labyrinth von Wänden umgeben ist, gibt es keine Position p , von der aus man mit einer Anweisung $a \in A$, eine Position $f(p, a)$ erreichen kann, die außerhalb des Labyrinths liegt. Um auf die zwei Positionen eines Zustands $S \in V, S = (p_0, p_1)$ zuzugreifen, schreibe ich ab jetzt $S[0] = p_0$ für die Position von Anton und $S[1] = p_1$ für die Position von Bea.

Die Zustandsänderung des Zustands $S \in V$, die durch die Anweisung $a \in A$ hervorgerufen wird, ist also:

$$S' = (f(S[0], a), f(S[1], a)).$$

Zwischen zwei Zuständen $S \in V$ und $S' \in V$ existiert also eine Kante, wenn S' durch die Anwendung einer Anweisung $a \in A$ auf S erreicht werden kann:

$$E = \{(S, S') \in (V \times V) \mid \exists a \in A, S' = (f(S[0], a), f(S[1], a))\}$$

Eine Kante zwischen dem Startknoten S und dem Endknoten S' wird hier als Tupel (S, S') dargestellt. Alle Kanten haben die Länge 1, da sie genau einer Anweisung entsprechen.

Jeder Pfad von einem Knoten $A \in V$ zu einem anderen Knoten $B \in V$ repräsentiert eine Sequenz von Anweisungen, die Anton und Bea von ihren Positionen bei A zu ihren Positionen bei B bringt. Die Länge eines solchen Pfades entspricht der Anzahl der durchlaufenen Anweisungen.

Der kürzeste Pfad von S_0 zu S_{end} entspricht also einer kürzesten Anweisungssequenz, die Anton und Bea ins Ziel bringt. Die Aufgabenstellung lässt sich also darauf reduzieren, den kürzesten Pfad von S_0 zu S_{end} in dem oben beschriebenen Graph zu finden.

Da der Graph nicht gewichtet ist, kann dieser Pfad mit einer Breitensuche gefunden werden.

2 Breitensuche

In der Implementierung können alle Schleifen, also Kanten die einen Knoten mit sich selbst verbinden, ignoriert werden, da sie mit Anweisungen zusammenhängen, die den Zustand nicht ändern (z.B. da Anton und Bea beide gegen eine Wand laufen).

Außerdem muss man beachten, dass Anton und Bea warten, wenn sie ihr Zielfeld bereits erreicht haben. Wenn also eine der Koordinaten die Position $(n-1, m-1)$ erreicht hat, wird diese von den Anweisungen von Chris nicht mehr verändert. Da nun nur noch eine Person ihr Ziel finden muss, würde es keinen Sinn machen, Anwendungen auszuführen, mit denen diese Person gegen eine Wand laufen würde. Dies muss aber in der Praxis nicht extra überprüft werden, da sich der Gesamtzustand in diesen Fällen nicht ändern würde (da eine Person gegen die Wand läuft und die andere wartet). Solche Anweisungen werden sowieso herausgefiltert.

```

1: function BFS(Labyrinth1, Labyrinth2)
2:   visited  $\leftarrow$  leeres Dictionary
3:   visited[ $S_0$ ]  $\leftarrow$  null
4:   queue  $\leftarrow$  leere Warteschlange
5:   queue.ENQUEUE( $S_0$ )
6:   while queue nicht leer do
7:      $S \leftarrow$  queue.POPFIRST
8:     if  $S = S_{end}$  then
9:       break
10:    end if
11:    for jede Anweisung  $a \in A$  do
12:       $S' \leftarrow (f(S[0], a), f(S[1], a))$ 
13:      if  $S' = S$  then
14:        continue
15:      end if
16:      if  $S' \in$  visited then
17:        continue
18:      end if
19:      visited[ $S'$ ]  $\leftarrow S$ 
20:      queue.ENQUEUE( $S'$ )
21:    end for
22:  end while
23:  if  $S_{end} \notin$  visited then
24:    PRINT("No path found")
25:  else
26:    return ▷ Verfolge Pfad anhand der visited Hashmap zurück
27:  end if
28: end function

```

Am Anfang werden die Variablen *visited* und *queue* initialisiert. Die *visited*-Variable ist eine Hashmap, die alle Knoten speichert, die bereits besucht wurden. Der Schlüssel ist der Knoten, der besucht wurde und der Wert ist der Knoten, von dem aus dieser Knoten erreicht wurde. Die *queue* ist eine Warteschlange, die alle Knoten speichert, die noch besucht werden müssen.

In der While-Schleife wird immer der erste Knoten S aus der Warteschlange entfernt. Falls dieser der Zielknoten ist, ist die Suche abgeschlossen.

Ansonsten wird für jede Anweisung der Zustand S' berechnet, der durch die Anwendung der Anweisung auf S erreicht werden kann. Falls dieser Zustand S nicht ändert (weil Anton und Bea gegen die Wand laufen) oder schon besucht wurde, wird die Anweisung übersprungen.

Ansonsten wird S' in die *visited*-Hashmap eingetragen und zur Warteschlange hinzugefügt.

Wenn die Warteschlange leer ist, aber der Zielknoten nicht besucht wurde, gibt es keinen Pfad von S_0 zu S_{end} und die Suche wird abgebrochen. Falls ein Pfad gefunden wurde, kann er anhand der *visited* Hashmap zurückverfolgt werden.

2.0.1 Laufzeitkomplexität

Im Worst-Case besucht die Breitensuche jeden Knoten genau einmal, die While-Schleife wird also $O(\|V\|)$ mal ausgeführt. Die For-Schleife in der While-Schleife läuft für jede Anweisung einmal, also $O(4) = O(1)$ mal. Da die Operationen in der For-Schleife (mit der Hashmap und der Warteschlange) mit einer theoretischen Laufzeit von $O(1)$ ausgeführt werden können, ist die Laufzeit der For-Schleife insgesamt $O(1)$. Damit hat der gesamte Algorithmus eine Laufzeit von $O(\|V\|)$.

Der oben beschriebene Graph besitzt einen Knoten für jede Mögliche Kombination an Positionen von Anton und Bea. Bea und Anton können jeweils $n \cdot m$ verschiedene Positionen einnehmen, d.h. insgesamt hat der Graph $\|V\| = n^2 m^2$ Knoten, d.h. die Laufzeit ist $O(n^2 m^2)$.

Das stimmt auch mit der allgemein bekannten Laufzeit für Breitensuche von $O(\|E\| + \|V\|)$ überein, da jeder Knoten höchstens vier ausgehende Kanten hat: $\|E\| \leq 4 \cdot n^2 m^2$. Die Worst-Case Laufzeit ist also:

$$O(n^2 m^2 + 4 \cdot n^2 m^2) = O(n^2 m^2).$$

Die Breitensuche hat im Wort-Case einen Speicherplatzverbrauch von $O(\|V\|) = O(n^2m^2)$, da alle besuchten Knoten in der *visited*-Hashmap gespeichert werden müssen.

2.1 Bidirektionale Breitensuche

Die Breitensuche hat die bestmögliche asymptotische Laufzeit für einen Algorithmus, der einen optimalen Pfad in einem Graphen findet, der nicht gewichtet ist.

In der Praxis lässt sie sich aber noch weiter optimieren, nämlich durch das Verfahren der Bidirektionalen Breitensuche. Dabei wird die Breitensuche nicht nur von dem Startknoten S_0 aus gestartet, sondern gleichzeitig auch „rückwärts“ von S_{end} aus. Sobald die Breitensuche aus der einen Richtung einen Knoten besucht, der von der anderen Richtung aus schon besucht wurde, ist ein kürzester Pfad gefunden und die Suche ist abgeschlossen. Dabei muss aber genauer auf die Reihenfolge geachtet werden, in der die Knoten besucht werden. Die normale Breitensuche funktioniert nämlich mit einer Datenstruktur, die wie eine Warteschlange (Queue) nach dem First-in Last-out prinzip funktioniert. Das bedeutet, dass ein neuer Knoten, der zu der Warteschlange hinzugefügt wurde erst besucht wird, wenn alle anderen Knoten, die vorher hinzugefügt wurden bereits aus der Warteschlange entfernt sind. Das bedeutet, dass zuerst alle Knoten, die i Kanten von S_0 entfernt sind, besucht werden, bevor ein Knoten in der Entfernung $i + 1$ besucht wird. Daraus folgt auch die Optimalität der Breitensuche, da so sicher der kürzeste Pfad gefunden wird.

Würde man bei der bidirektionalen Breitensuche dieses Verfahren vom Startknoten aus und vom Zielknoten aus rückwärts anwenden, könnte es passieren, dass ein Pfad gefunden wird, der um eine Kante zu lang ist.

Darum müssen die Knoten von jeder Breitensuche aus „Ebene für Ebene“ besucht werden:

Algorithm 1 Bidirektionale Suche

```

1:  $Visited_1 \leftarrow \{S_0\}$ 
2:  $Visited_2 \leftarrow \{S_{end}\}$ 
3:  $Frontier_1 \leftarrow \{S_0\}$ 
4:  $Frontier_2 \leftarrow \{S_{end}\}$ 
5: while  $|Frontier_1| > 0 \wedge |Frontier_2| > 0$  do
6:   if  $|Frontier_1| \leq |Frontier_2|$  then
7:      $Frontier_1 \leftarrow \text{Expand\_by\_one}_1(Frontier_1)$ 
8:      $Visited_1 \leftarrow Visited_1 \cup Frontier_1$ 
9:     if  $|Frontier_1 \cap Visited_2| > 0$  then
10:      exit ▷ Pfad gefunden!
11:   end if
12: else
13:    $Frontier_2 \leftarrow \text{Expand\_by\_one}_2(Frontier_2)$ 
14:    $Visited_2 \leftarrow Visited_2 \cup Frontier_2$ 
15:   if  $|Frontier_2 \cap Visited_1| > 0$  then
16:     exit ▷ Pfad gefunden!
17:   end if
18: end if
19: end while
```

Als erstes werden die Variablen $Visited_1$ und $Visited_2$ initialisiert, um die Knoten zu speichern, für die bereits ein Pfad von S_0 bzw von S_{end} aus bekannt ist. Die Variablen $Frontier_1$ und $Frontier_2$ speichern alle Knoten, die genau i Pfadlängen von S_0 bzw S_{end} entfernt sind. In der Schleife wird entschieden, ob die Binärsuche von S_0 aus oder die von S_{end} aus um eine Ebene weitergeführt werden. In welcher Reihenfolge dies geschieht ist irrelevant, der Algorithmus ist in jedem Fall optimal. Hier wird die Seite weitergeführt, die zurzeit weniger „aktive“ Knoten hat, die erweitert werden sollen.

Die Expand_by_one_1 Funktion nimmt eine Menge an Knoten und gibt alle Knoten zurück, die genau eine Pfadlänge von einem der Knoten in $Frontier$ entfernt liegen, was einem Schritt der Binärsuche entspricht. Die Funktion Expand_by_one_1 geht dabei entlang der Pfeile des Graphen und Expand_by_one_2 geht „Rückwärts“ in die entgegengesetzte Richtung.

Anschließend werden in beiden Fällen die neuen Knoten zur jeweiligen $Visited$ -Menge hinzugefügt. Am Ende wird überprüft, ob einer der neuen Knoten in der jeweiligen $Frontier$ -Variable bereits aus der

anderen Richtung besucht wurde. Wenn dies der Fall ist, dann ist der kürzeste Pfad gefunden, da zu dem Knoten, der nun z.B. in $Frontier_1$ und $Visited_2$ enthalten ist, ein Pfad von S_0 aus und von S_{end} aus bekannt ist. Im Umsetzungsteil gehe ich genauer darauf ein, welche Möglichkeiten es gibt, diesen Pfad zu speichern und zurückzuverfolgen.

2.1.1 Optimalität

Warum??

2.1.2 Laufzeit

Asymptotisch ist die Laufzeit von bidirektionaler Breitensuche gleich wie die der normalen Breitensuche, im Wort-Case muss jeder Knoten und jede Kante besucht werden, also $O(\|V\| + \|E\|)$.

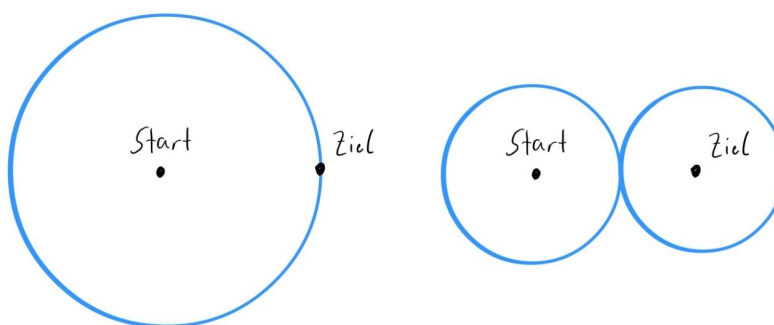


Abbildung 1: Links: normale Breitensuche, Rechts: Bidirektionale Breitensuche

Intuitiv ist Bidirektionale Breitensuche aber schneller, da meistens weniger Knoten besucht werden müssen (siehe Bild 2.1.2).

Mathematisch lässt sich das schwer quantifizieren, man kann allerdings eine andere Obergrenze der Laufzeit der Breitensuche betrachten. Sei d die Länge des Pfades, der gefunden werden muss und b die Durchschnittliche Anzahl an ausgehenden Kanten eines Knotens, in unserem Fall $d = 4$. Die Laufzeit von Breitensuche lässt sich durch $O(b^d)$ begrenzen, da, bis der Zielknoten gefunden wurde, auf jeder „Ebene“ die Anzahl der besuchten Knoten um den Faktor b steigt.

Die zwei Breitensuchen der Bidirektionalen Breitensuche treffen sich aber schon nach $D/2$ Schritten in der Mitte (wenn man beide Richtungen immer abwechselnd um eine Ebene erweitert). Die Laufzeit der Bidirektionalen Breitensuche lässt sich also durch $O(b^{\frac{d}{2}})$ begrenzen.

Für die Pfadlänge in unserem Graphen gilt allerdings $d > m + n$, und da $m^2 n^2 \leq 4^{\frac{n+m}{2}}$ für alle $n, m \in \mathbb{N}$ ist $O(n^2 m^2)$ eine deutlich bessere Abschätzung für die Laufzeit der Bidirektionalen Breitensuche als $O(b^{\frac{d}{2}})$. Im Beispielteil wird sich zeigen, dass Bidirektionale Breitensuche in den meisten Fällen trotzdem deutlich schneller ist.

3 Umsetzung

3.1 Breitensuche

Ich verwende eine *HashMap* um zu speichern, welche Knoten bereits besucht wurden. Der Schlüssel entspricht dem Knoten der Besucht wurde und der Wert, der hinter dem Schlüssel hinterlegt wird, ist der Knoten, von dem aus der Schlüssel erreicht wurde. Das hat den Vorteil, dass Werte mit konstanter Laufzeit gesetzt werden können und dass mit konstanter Laufzeit überprüft werden kann, ob ein Knoten bereits besucht wurde. Wenn dann der Zielknoten gefunden wurde, kann der Pfad mit der *HashMap* leicht zurückverfolgt werden.

Für die Warteschlange verwende ich eine *Deque* (Double-ended Queue), die es mit konstanter Laufzeit ermöglicht, Knoten hinten anzuhängen und vorne zu entfernen.

3.2 Bidirektionale Breitensuche

Für die Bidirektionale Breitensuche benötigt man zwei Funktionen; die *forward* und die *backward* Funktion. Die *forward* „erweitert“ einen Knoten entlang der Richtung der Pfeile wie bei der normalen Breitensuche und gibt alle Knoten zurück, die eine Pfadlänge von dem Knoten entfernt sind und noch nicht besucht wurde.

Die *backward* Funktion muss für einen Knoten alle Knoten finden, von denen aus dieser Knoten mit einer Anweisung erreicht werden kann:

Algorithm 2 BACKWARD

```

1: function BACKWARD( $l_1, l_2, current, forward\_visited, backward\_visited$ )
2:    $new\_backward\_queue \leftarrow []$ 
3:   for all  $inst \in \{UP, DOWN, LEFT, RIGHT\}$  do
4:      $temp\_queue \leftarrow [State(l_1.shift'(current.pos1, inst),$ 
5:                            $l_2.shift'(current.pos2, inst))]$ 
6:     if  $l_1.hasWallInDirection(current.pos1, inst^{-1})$  then
7:        $temp\_queue.add(State(current.pos1, l_2.shift'(current.pos2, inst)))$ 
8:     end if
9:     if  $l_2.hasWallInDirection(current.pos2, inst^{-1})$  then
10:       $temp\_queue.add(State(l_1.shift'(current.pos1, inst), current.pos2))$ 
11:    end if
12:    for all  $new\_pos \in temp\_queue$  do
13:      if  $new\_pos = current$  then
14:        continue
15:      end if
16:      if  $new\_pos \in backward\_visited$  then
17:        continue
18:      end if
19:      if  $State(l_1.shift(new\_pos.pos1, inst^{-1}),$ 
20:               $l_2.shift(new\_pos.pos2, inst^{-1})) \neq current$  then
21:        continue
22:      end if
23:       $backward\_visited[new\_pos] \leftarrow current$ 
24:      if  $new\_pos \in forward\_visited$  then
25:        return Connection( $new\_pos$ )
26:      end if
27:       $new\_backward\_queue.add(new\_pos)$ 
28:    end for
29:  end for
30: end function

```

In Algorithmus 3.2 sieht man den wichtigsten Teil der *backward* Funktion. Für einen Knoten *current* müssen für jede mögliche Anweisung *inst* alle Knoten gefunden werden, die durch eine Anwendung von $inst^{-1}$ wieder zu *current* werden. Dafür gibt es verschiedene Möglichkeiten, die in *temp_queue* gespeichert werden. Die erste Möglichkeit ist, beide Positionen des Knotens in Richtung *inst* zu verschieben. Falls aber in Richtung $inst^{-1}$ von der ersten Position (also *current.pos1*) eine Wand liegt, kann es auch sein, dass diese Position durch die Anwendung von $inst^{-1}$ konstant bleibt, da Anton gegen die Wand läuft. (TODO Bild zum erklären) Das selbe gilt auch für *current.pos2*. Es gibt also in manchen Fällen bis zu drei mögliche Knoten, von denen aus *current* durch eine Anwendung von $inst^{-1}$ erreicht werden könnte. Für jeden dieser Knoten muss nun überprüft werden, ob das tatsächlich der Fall ist und ob der Knoten in der Suche beachtet werden muss. Dafür wird zuerst überprüft, ob der Knoten *new_pos* überhaupt unterschiedlich zu *current* ist. Wenn z.B. eine Wand in Richtung *inst* liegt, könnte es sein, dass Anton und Bea gegen die Wand gelaufen sind bei dem Versuch, *current* in Richtung *inst* zu verschieben. Als nächstes wird überprüft, ob die Position schon besucht wurde, da sie in diesem Fall nicht beachtet werden muss. Schließlich wird überprüft ob man von *new_pos* aus *current* tatsächlich durch einen Schritt in $inst^{-1}$ erreichen kann.

Wenn alle notwendigen Bedingungen gegeben sind, kann *new_pos* in die *backward_visited* Hashmap eingetragen werden. Als Wert wird *current* eingetragen, wodurch später der Pfad zurückverfolgt werden

kann.

Anschließend wird überprüft, ob der Knoten *new_pos* bereits aus der anderen Richtung besucht wurde. Wenn dies der Fall ist, ist ein kürzester Pfad gefunden und der verbindende Knoten wird zurückgegeben. Von ihm aus kann nun durch die HashMaps *forward_visited* und *backward_visited* der Pfad in beide Richtungen zurückverfolgt werden.

Bei der *backward* Funktion müssen einige edge-cases beachtet werden. Die *shift* Funktion (in der Lösungsidee als $f(p, a)$ definiert für eine Anweisung $a \in A$ und eine Position p) verändert eine Position normalerweise nicht, falls sie bereits den Endzustand $p = (n - 1, m - 1)$ erreicht hat. Wenn in der *backward* Funktion allerdings Positionen „Rückwärts“ vom Endzustand aus verschoben werden sollen, ist das ein Problem. Drum verwende ich im Pseudocode die *shift'* Funktion, die diese Bedingung nicht beachtet. Im Quellcode heißt diese Funktion *shift_without_end_fix*.

Wenn Anton oder Bea in ein Loch fallen, gehen sie direkt zum Startfeld zurück. Das bedeutet, wenn die *backward* Funktion alle Felder ermitteln müsste, von denen Anton oder Bea mit einer Anweisung zum Startfeld gelangen können, müsste sie jede Position ausgeben, von der aus ein Loch mit einer Anweisung erreichbar ist. Da das für die meisten Beispieldateien eine sehr große Anzahl an Feldern ist, verwende ich die *backward* Funktion nicht mehr, sobald Anton oder Bea die Position (0,0) Rückwärts besucht haben. Von dort an wird die Suche nur noch mit der *forward* Funktion fortgeführt. In der Praxis wird aber meistens ein Pfad gefunden, bevor *backward* in einem Labyrinth das Startfeld erreicht.

3.2.1 Optimierung

In meiner Implementierung der Bidirektionalen Breitensuche verwende ich für jede Richtung eine Hashmap, um zu speichern welche Knoten bereits besucht wurden (Entspricht den *visited* variablen in Algorithmus 1). Um den Pfad später wieder zurückzuverfolgen, speichere ich in der Hashmap den Knoten, von dem aus der aktuelle Knoten erreicht wurde. Um zu überprüfen, ob bereits ein Pfad gefunden wurde, muss für jeden neuen Knoten überprüft werden, ob dieser bereits in der *visited*-Hashmap der anderen Richtung enthalten ist. Die HashMap-Operationen haben zwar eine theoretische Laufzeit von $O(1)$, in der Praxis sind diese vor allem für große Labyrinth, bei denen viele Knoten besucht werden aber langsamer. Eine Möglichkeit, den Bidirektionalen BFS-Algorithmus zu optimieren, ist daher ein Array A zu verwenden, um die besuchten Knoten und deren Herkunft. Um den Speicherplatzverbrauch dieses Arrays zu minimieren, kann sollte aber nicht für jeden Zustand der Herkunftszustand gespeichert werden. Für ein Labyrinth mit den Seitenlängen n und m und 16 Variablen für die Koordinaten der Positionen hätte dieses Array die Größe:

$$|A| = n^2 m^2 \cdot ((2 + 2) + (2 + 2)) = n^2 m^2 \cdot 16$$

Für das größte Labyrinth mit $n = m = 250$ hätte dieses Array eine Größe von über 62gb, was für die meisten Computer nicht mehr in den Arbeitsspeicher passt.

Daher speichere ich in A nur die Anweisung, durch die ein bestimmter Zustand erreicht wurde, die 4-Bits einnimmt. Für die Bidirektionale Suche ist aber auch wichtig, von welcher Richtung aus der Zustand erreicht wurde, was auch in dem Array gespeichert werden muss. Anhand der Anweisungen kann der Pfad nach der Suche theoretisch zurückverfolgt werden. Wenn Anton und Bea in ein Loch gefallen sind, ergibt sich allerdings ein Problem, da allein durch die Anweisung nicht zurückverfolgt werden kann, in welches Loch die Person gefallen ist. D.h. es muss im Array gespeichert werden, ob Anton oder Bea vor der Zustandsänderung in ein Loch gefallen sind. In einer getrennten Hashmap wird dann gespeichert, in welches Loch die Person gefallen ist. Da solche Fälle aber selten sind, wäre diese Hashmap kein signifikanter zusätzlicher Speicherplatzverbrauch.

3.3 Weitere Optimierungsmöglichkeiten

Eine Möglichkeit wäre, den A* Algorithmus zu verwenden, um die Suche mit einer Heuristik zu beschleunigen. Dabei könnte man folgende Heuristik verwenden.

$$h((x_0, y_0), (x_1, y_1)) = \max\{(n - x_0 + m - y_0 - 2), (n - x_1 + m - y_1 - 2)\}$$

Der Term $(n - x_0 + m - y_0 - 2)$ repräsentiert die minimale Anzahl an Anweisungen, die benötigt sind, dass Anton sein Ziel erreicht. Die Heuristik nimmt das maximum der Minimalen Anweisungen für Bea und Anton, was den Pfad nicht überschätzt. In der Praxis ist der Pfad aber deutlich länger, weshalb die Heuristik nicht sehr gut ist und die Suche nicht signifikant beschleunigen würde.

Eine andere Möglichkeit wäre, die Bidirektionale Suche zu parallelisieren. Da die Reihenfolge, in der die Knoten vorwärts bzw. Rückwärts expandiert werden egal ist, könnten die *Expand_by_one* in Algorithmus 1 auf mehreren Threads gleichzeitig ausgeführt werden.

4 Beispiele

In der folgenden Tabelle sieht man die Anzahl an Anweisungen des optimalen Pfads durch das entsprechende Labyrinth gemeinsam mit der Laufzeit der normalen und der bidirektionalen Breitensuche. Bei zu langen Laufzeiten des Breitensuche-Algorithmus, habe ich das Programm abgebrochen.

Nummer des Labyrinths	Anweisungen	Laufzeit BFS	Laufzeit BidiBFS
0	8	$9\mu s$	$25\mu s$
1	31	$75\mu s$	$120\mu s$
2	66	$2.4ms$	$2.6ms$
3	164	$89ms$	$56ms$
4	14384	$113s$	$37s$
5	1308	(timeout)	1782
6	1844	$66s$	$107s$
7	-	$14ms$	$30ms$
8	472	(timeout)	$106s$
9	1012	(timeout)	$150s$

4.1 Interessante Beispiele

Um einige Beispiele genauer zu betrachten, gibt das Programm ein Bild aus, in dem der Pfad in beiden Labyrinthen eingezeichnet ist:



Abbildung 2: Die Labyrinth aus Beispieldatei 2

In Abbildung 4.1 sieht man die Wände schwarz eingezeichnet, die Löcher rot und die Pfade als Farbverlauf von Blau über Orange nach Grün. Dieser Farbverlauf hat den Vorteil, dass man bei sehr langen Pfaden erkennen kann, wann Anton und Bea sich an welchem Punkt befunden haben. Ein gutes Beispiel dafür ist Beispieldatei 4:

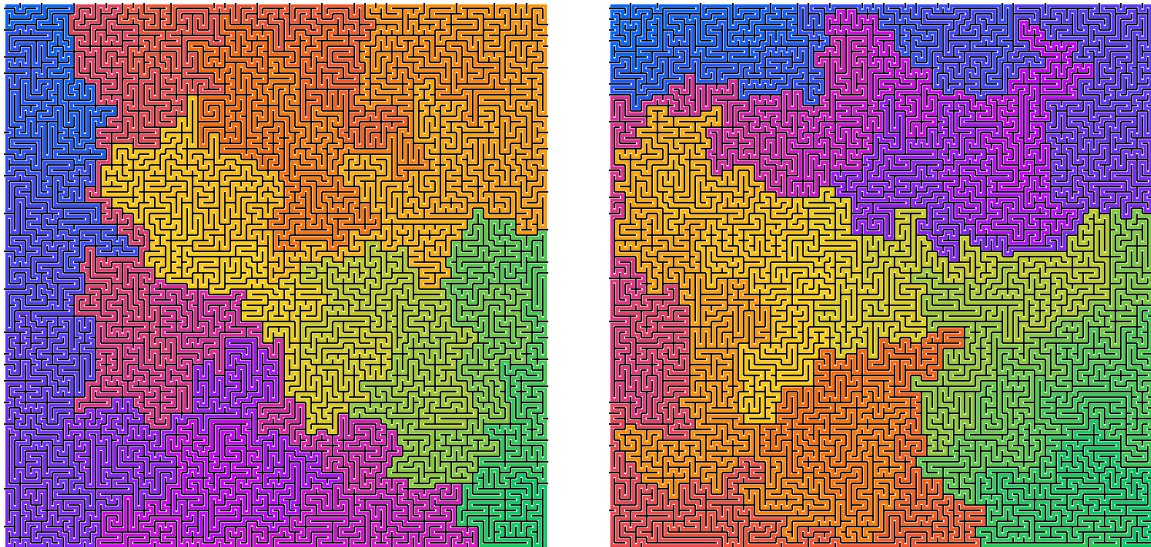


Abbildung 3: Die Labyrinth aus Beispieldatei 4

Da Anton und Bea auf ihren wegen durch dieses Labyrinth jedes Feld besucht haben, sieht man gut, wie sie parallel verschiedene Bereiche des Labyrinths durchlaufen haben. Bei Beispieldatei 6 fällt auf, dass die normale Breitensuche deutlich schneller war als die Bidirektionale Breitensuche:

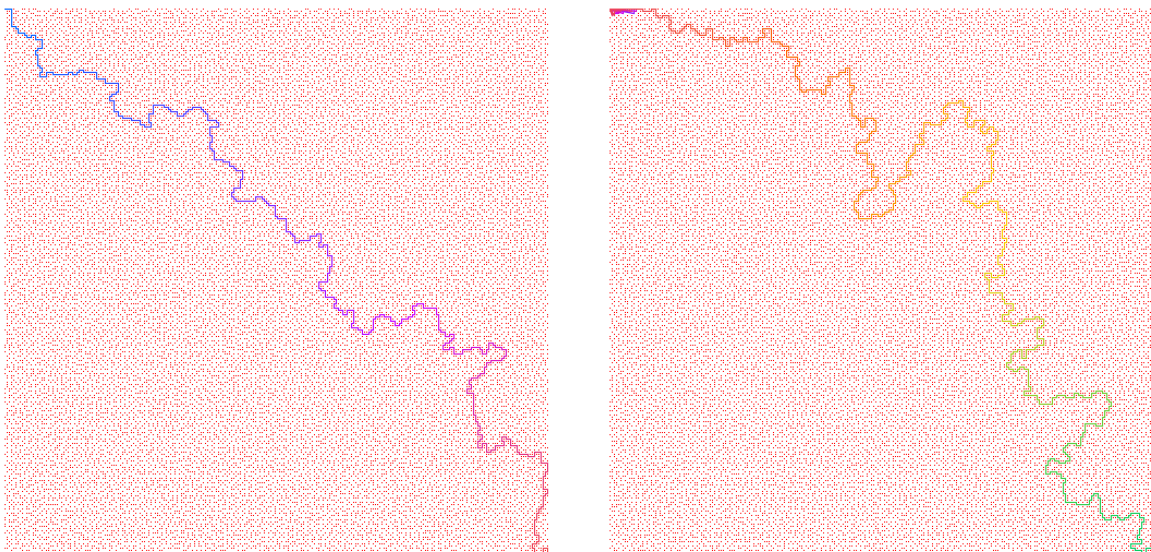


Abbildung 4: Die Labyrinth aus Beispieldatei 6

Man sieht, dass die Labyrinth in dieser Beispieldatei ausschließlich aus Löchern bestehen. Die die Rückwärts-Suche der Bidirektionalen Breitensuche abbricht, sobald sie das Startfeld erreicht hat, wurde ein großer Teil dieser Suche nur von der *Forward* Funktion ausgeführt, die aufgrund der Implementierung etwas langsamer ist als die normale Breitensuche. Das Startfeld wurde von der Rückwärts-Suche in diesem Beispiel sehr schnell erreicht, da es keine Wände gibt und die Rückwärts suche Löcher nicht beachten muss.

Aber auch die Pfade dieser Beispieldatei sind sehr interessant. Am Farbverlauf sieht man, dass Anton im ersten Labyrinth auf sehr schnellem Weg ins Ziel gekommen ist und dort gewartet hat. Bea ist dagegen für einen gewissen Zeitraum im Startbereich geblieben und ist immer wieder in Löcher gefallen (und musste dadurch zurück zum Startfeld) und ist erst Später ins Ziel gekommen.

In Beispieldatei 7 hat die Suche keine Lösung ergeben.

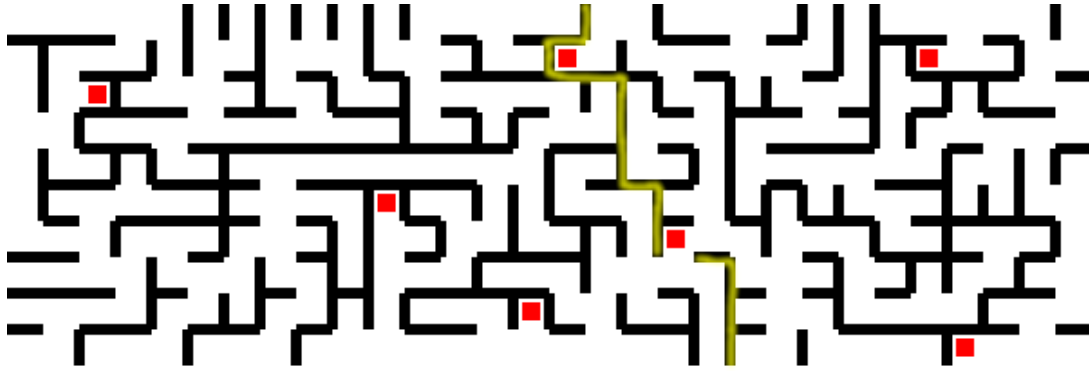


Abbildung 5: Das zweite Labyrinth der Beispeldatei 7

Man sieht in Abbildung 4.1, dass Bea nicht zum Ziel kommen kann, da der Pfad auf der Ebene der gelb eingezeichneten Linie durch Wände und ein Loch blockiert wird.