Aufgabe 2: Simultane Labyrinthe

Teilnahme-ID: 74749

Bearbeiter/-in dieser Aufgabe:
Christian Krause

28. April 2025

Inhaltsverzeichnis

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 Labyrinthe ohne Gruben geeignet.

1.1 Modellierung

Im Folgenden arbeite ich mit der Annahme, dass die Labyrinthe 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 $G = (V, E)$ dargestellt werden, dessen Knoten alle Möglichen Zustände sind.

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

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.

```
function BFS(Labyrinth1, Labyrinth2)
    visited ← leeres Dictionary
    visited[S_0] ← null
    queue ← leere Warteschlange
    queue.ENQUEUE(S_0)
    while queue nicht leer do
        S ← queue.POPFIRST
        if S = S_end then
            break
        end if
        for jede Anweisung a ∈ A do
            S' ← (f(S[0], a), f(S[1], a))
            if S' = S then
                continue
            end if
            if S' ∈ visited then
                continue
            end if
            visited[S'] ← S
            queue.ENQUEUE(S')
        end for
    end while
    if S_end ∉ visited then
        PRINT("No path found")
    else
        return                          ▷ Verfolge Pfad anhand der visited Hashmap zurück
    end if
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.

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

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

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

2.2 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

```
Visited_1 ← {S_0}
Visited_2 ← {S_end}
Frontier_1 ← {S_0}
Frontier_2 ← {S_end}
while |Frontier_1| > 0 ∧ |Frontier_2| > 0 do
    if |Frontier_1| ≤ |Frontier_2| then
        Frontier_1 ← Expand_by_one_1(Frontier_1)
        Visited_1 ← Visited_1 ∪ Frontier_1
        if |Frontier_1 ∩ Visited_2| > 0 then
            exit                                    ▷ Pfad gefunden!
        end if
    else
        Frontier_2 ← Expand_by_one_2(Frontier_2)
        Visited_2 ← Visited_2 ∪ Frontier_2
        if |Frontier_2 ∩ Visited_1| > 0 then
            exit                                    ▷ Pfad gefunden!
        end if
    end if
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 Breitensuche von S_0 aus oder die von S_{end} aus um eine Ebene weitergeführt wird. 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 Breitensuche 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.

2.2.1 Laufzeit

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

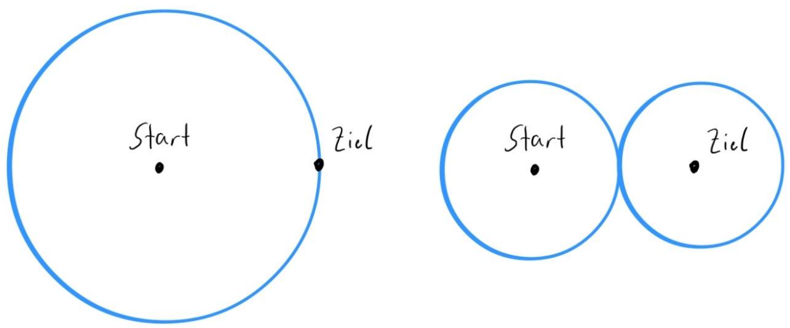

Abbildung 1: Links: normale Breitensuche, Rechts: Bidirektionale Breitensuche

Intuitiv ist Bidirektionale Breitensuche aber schneller, da meistens weniger Knoten besucht werden müssen (siehe Bild 2.2.1).
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 $m^2n^2 \leq 4^{\frac{n+m}{2}}$ für alle $n, m \in \mathbb{N}$ ist $O(n^2m^2)$ 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 hinter dem Schlüssel 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 wurden.
Die *backward* Funktion muss für einen Knoten alle Knoten finden, von denen aus dieser Knoten mit einer Anweisung erreicht werden kann:

Algorithm 2 BACKWARD

```
function BACKWARD(l1, l2, current, forward_visited, backward_visited)
    new_backward_queue ← [ ]
    for all inst ∈ {UP, DOWN, LEFT, RIGHT} do
        temp_queue ← [ State(l1.shift'(current.pos1, inst),
                      l2.shift'(current.pos2, inst))]
        if l1.hasWallInDirection(current.pos1, inst^-1) then
            temp_queue.add(State(current.pos1, l2.shift'(current.pos2, inst)))
        end if
        if l2.hasWallInDirection(current.pos2, inst^-1) then
            temp_queue.add(State(l1.shift'(current.pos1, inst), current.pos2))
        end if
        for all new_pos ∈ temp_queue do
            if new_pos = current then
                continue
            end if
            if new_pos ∈ backward_visited then
                continue
            end if
            if State(l1.shift(new_pos.pos1, inst^-1),
                        l2.shift(new_pos.pos2, inst^-1)) ≠ current then
                continue
            end if
            backward_visited[new_pos] ← current
            if new_pos ∈ forward_visited then
                return Connection(new_pos)
            end if
            new_backward_queue.add(new_pos)
        end for
    end for
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 inv^{-1} konstant bleibt, da Anton gegen die Wand läuft. 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. Deshalb 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 Labyrinthe, 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 zu speichern. Um den Speicherplatzverbrauch dieses Arrays zu minimieren, sollte aber nicht für jeden Zustand der Herkunftszustand gespeichert werden. Für ein Labyrinth mit den Seitenlängen n und m und 16-Bit integer Variablen für die Koordinaten der Positionen hätte dieses Array die Größe:

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

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.
Eine Möglichkeit wäre es, in A nur die Anweisung zu speichern, durch die ein bestimmter Zustand erreicht wurde, welche 4-Bits einnimmt. Für die Bidirektionale Suche ist aber auch wichtig, von welcher Richtung (rückwärts oder vorwärts) aus der Zustand erreicht wurde, was auch in dem Array gespeichert werden muss. Bei der Rückverfolgung des Pfades ergeben sich aber einige Probleme: Wenn Anton oder Bea in ein Loch gefallen sind, kann allein durch die Anweisung nicht zurückverfolgt werden kann, in welches Loch die Person gefallen ist. Das heißt, es müsste im Array gespeichert werden, ob Anton oder Bea vor der Zustandsänderung in ein Loch gefallen sind. In einer getrennten Hashmap könnte dann gespeichert werden, in welches Loch die Person gefallen ist. Das größere Problem liegt aber darin, dass die Rückverfolgung des Pfades selbst ohne Löcher anhand der Anweisungen nicht eindeutig ist. Im Prinzip können hier die gleichen Fälle auftreten, wie bei der *backward* Funktion. Wenn rechts von den Positionen von Anton und Bea eine Wand liegt und man weiß, dass sie diese Positionen durch die Anweisung „rechts"erreicht haben, gibt es mehrere Fälle: Es könnte sein, dass beide nach rechts gelaufen sind oder dass einer der beiden bereits an der Position war und gegen die Wand gelaufen ist. Man könnte entweder noch mehr Informationen in dem Array speichern, oder alle Möglichkeiten, die sich ergeben wieder mit einer Breitensuche durchsuchen, aber am einfachsten ist es einfach die Aufgabenstellung zu ändern. Der Ansatz mit den Arrays ist nämlich gut dafür geeignet, die Pfadlänge zu bestimmten, wenn man den Pfad nicht mehr zurückverfolgen muss. Für jede Zelle muss dann nur gespeichert werden, ob diese noch nicht, rückwärts oder vorwärts besucht wurde. Die Array-Breitensuche ist also eine Erweiterung des Problems auf eine leicht andere Aufgabenstellung, die eine andere, effizientere Lösung ermöglicht.

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.

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

Der Term $(n - x_0 + m - y_0 - 2)$ repräsentiert die minimale Anzahl an Anweisungen, die benötigt werden, dass Anton sein Ziel erreicht. Die Heuristik nimmt das Maximum der minimalen Anweisungen für Bea und Anton, was den Pfad nicht überschätzt. Um zu überprüfen, ob die Berechnung der kürzesten Pfade durch diese Heuristik optimiert werden kann, habe ich den $A^\star$ Algorithmus anhand des Wikipedia Artikels implementiert[1]. In der Praxis stellte sich aber heraus, dass der Algorithmus die Beispieldateien zwar korrekt lösen kann, aber trotzdem deutlich langsamer ist als die Breitensuche. Das liegt vermutlich daran, dass die Heuristik die Pfade stark unterschätzt, da die Pfade selbst in einem gewöhnlichen Labyrinth deutlich Länger wären als die Manhattan-Distanz, die ich als Heuristik verwende. In dieser Aufgabe kommt dann noch dazu, dass die Bea und Anton nicht den Pfad wählen können, der für sie optimal ist, sonder dass die Pfade aufeinander abgestimmt sein müssen, wodurch die Heuristik noch weiter von der tatsächlichen Lösung entfernt ist. Meine Implementierung des $A^\star$ Algorithmus ist im Quellcode-Teil zu finden.

Eine andere Möglichkeit zur optimierung wäre noch, 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 Program abgebrochen. Ich habe in dieser Tabelle auch die Laufzeit der Array-Implementierung mit einbezogen, es ist allerdings zu beachten, dass sie nur die Länge des optimalen Pfads bestimmt. Die Anweisungssequenzen sind in separaten Textdateien in der Abgabe enthalten.

Nummer des Labyrinths	Anweisungen	Laufzeit BFS	Laufzeit BidiBFS	Laufzeit Array BidiBFS
0	8	$9\mu s$	$25\mu s$	14
1	31	$75\mu s$	$120\mu s$	50
2	66	$2.4ms$	$2.6ms$	$1.5ms$
3	164	$89ms$	$56ms$	$26ms$
4	14384	$113s$	$37s$	$10s$
5	1308	(timeout)	1782	$1723s$
6	1844	$66s$	$107s$	$94s$
7	-	$14ms$	$30ms$	$14ms$
8	472	(timout)	$106s$	$24s$
9	1012	(timout)	$150s$	$35s$

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 Labyrinthe 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

[1] `https://de.wikipedia.org/wiki/A*-Algorithmus`

Pfaden erkennen kann, wann Anton und Bea sich an welchem Punkt befunden haben. Ein gutes Beispiel dafür ist Beispieldatei 4:

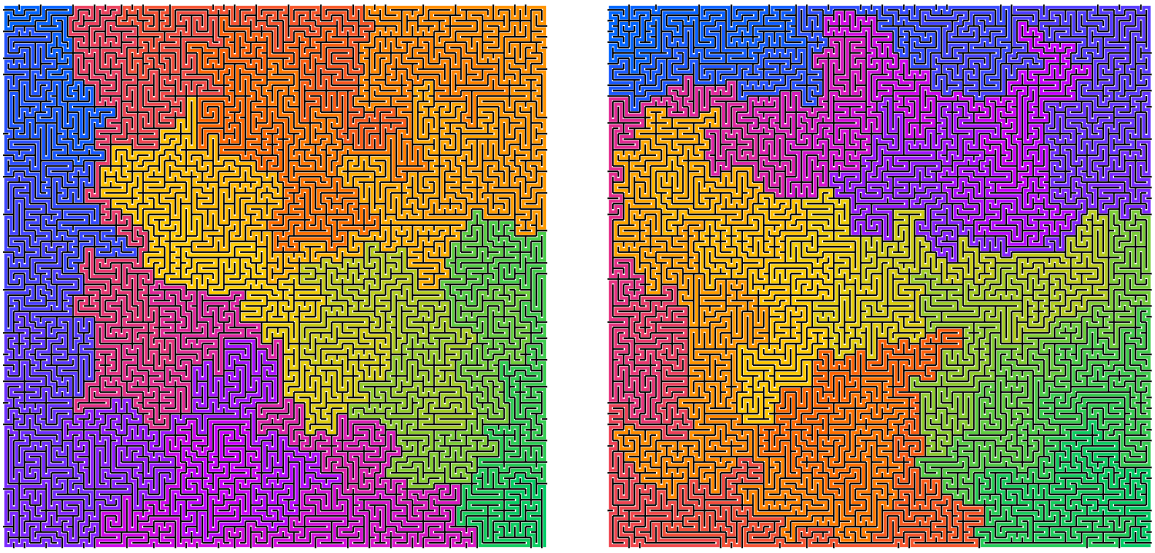

Abbildung 3: Die Labyrinthe aus Beispieldatei 4

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:

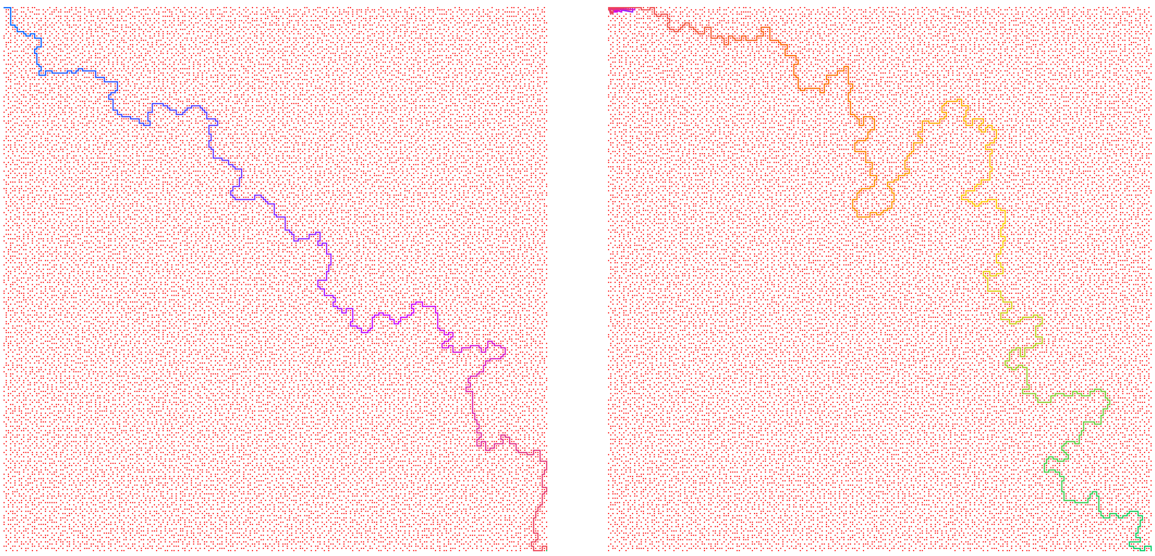

Abbildung 4: Die Labyrinthe aus Beispieldatei 6

Man sieht, dass die Labyrinthe in dieser Beispieldatei ausschließlich aus Löchern bestehen. 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.

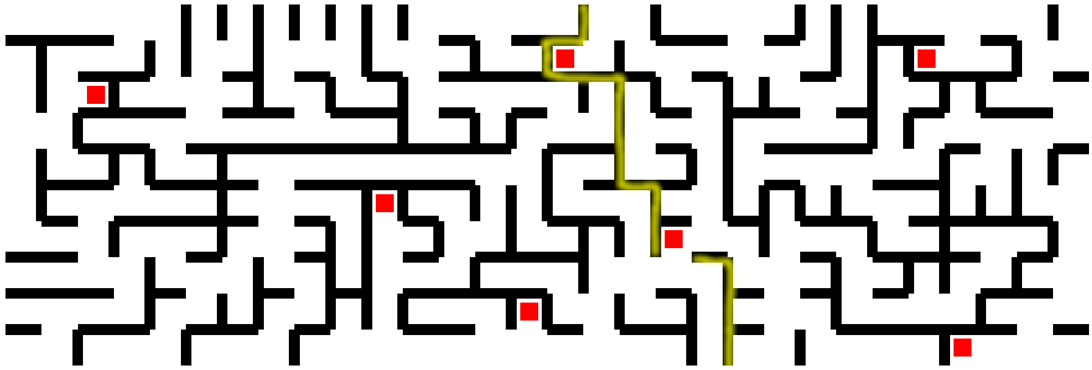

Abbildung 5: Das zweite Labyrinth der Beispeldatei 7

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.

5 Quellcode

Der vollständige Quellcode ist in einer separaten Datei in der Abgabe enthalten.

5.1 Definitionen

```
#[derive(EnumIter, Clone, Copy)]
#[repr(u8)]
pub enum Instruction {
    UP,
    DOWN,
    LEFT,
    RIGHT,
}

impl Instruction {
    pub fn inv(&self) -> Instruction {
        // Invertiert eine Instruction
        match self {
            Instruction::UP => Instruction::DOWN,
            Instruction::DOWN => Instruction::UP,
            Instruction::LEFT => Instruction::RIGHT,
            Instruction::RIGHT => Instruction::LEFT,
        }
    }
}

#[derive(Eq, Hash, PartialEq, Debug)]
pub struct Point(pub i16, pub i16);
impl Point {
    pub fn shift(&self, direction: &Instruction) -> Point {
        // Bewegt den Punkt in die Richtung der Instruction
        match direction {
            Instruction::UP => Point(self.0, self.1 - 1),
            Instruction::DOWN => Point(self.0, self.1 + 1),
            Instruction::LEFT => Point(self.0 - 1, self.1),
            Instruction::RIGHT => Point(self.0 + 1, self.1),
        }
    }
    pub fn copy(&self) -> Point {
        Point(self.0, self.1)
    }
}

#[derive(Eq, Hash, PartialEq, Debug)]
```

```
pub struct State(pub Point, pub Point);

impl State {
    pub(crate) fn copy(&self) -> State {
        State(Point(self.0.0, self.0.1), Point(self.1.0, self.1.1))
    }
}

pub struct Labyrinth {
    pub(crate) width: i16,
    pub height: i16,
    pub vertical_walls: Vec<Vec<bool>>,
    pub horizontal_walls: Vec<Vec<bool>>,
    pub holes: Vec<Point>,
}

impl Labyrinth {

    /*
    ....
    */


    /// Bewegt einen Punkt und beachtet dabei Wände, Löcher und Zielfelder
    pub fn shift_point(&self, point: &Point, direction: &Instruction) -> Point {
        if self.is_wall_in_direction(&point, direction) {
            return Point(point.0, point.1);
        }
        if point.0 == self.width - 1 && point.1 == self.height - 1 {
            return Point(point.0, point.1);
        }

        if self.is_hole(&point) {
            return Point(0, 0);
        }
        point.shift(direction)
    }

    /// Bewegt einen Punkt wie die shift_point Funktion, außer dass am Ende nicht
    /// stillgestanden wird. Das wird für die Breitensuche rückwärts benötigt
    pub fn shift_point_without_end_fix(&self, point: &Point, direction: &Instruction) -> Point
        if self.is_wall_in_direction(&point, direction) {
            return Point(point.0, point.1);
        }
        if self.is_hole(&point) {
            return Point(0, 0);
        }
        point.shift(direction)
    }
}
```

5.2 Breitensuche

```
/// Findet den optimalen Pfad mit Breitensuche
fn double_path_bfs(
    l1: &Labyrinth,
    l2: &Labyrinth,
    mut path: (Vec<Point>, Vec<Point>),
) -> (Vec<Point>, Vec<Point>) {
    assert!(l1.width == l2.width && l1.height == l2.height);

    let start = State(Point(0, 0), Point(0, 0));
    let end = State(
        Point(l1.width - 1, l1.height - 1),
        Point(l2.width - 1, l2.height - 1),
    );
```

```
    // Key: Alle Zustände, die bereits besucht wurden
    // Value: Der Zustand, von dem aus der aktuelle Zustand erreicht wurde
    let mut visited: HashMap<State, State> = HashMap::new();
    visited.insert(start.copy(), start.copy());

    // Warteschlange für die Breitensuche
    let mut queue: VecDeque<State> = VecDeque::from(vec![start.copy()]);

    while queue.len() > 0 {
        let current = queue.pop_front().unwrap();
        if current == end {
            // Ein Pfad ist gefunden
            break;
        }
        for inst in Instruction::iter() {
            // Die neue Position in jede Richtung wird bestimmt
            let new_pos = State(
                l1.shift_point(&current.0, &inst),
                l2.shift_point(&current.1, &inst),
            );
            if new_pos == current {
                // Die Position hat sich nicht verändert
                continue;
            }
            if visited.contains_key(&new_pos) {
                // Die Position wurde schon besucht
                continue;
            }
            visited.insert(new_pos.copy(), current.copy());
            queue.push_back(new_pos);
        }
    }
    if (!visited.contains_key(&end)) {
        // Das Ende wurde nicht erreicht
        println!("No path found");
        return path;
    }
    // Der Pfad wird zurückverfolgt
    path.0.push(Point(l1.width - 1, l1.height - 1));
    path.1.push(Point(l1.width - 1, l1.height - 1));
    let mut current = visited.get(&end).unwrap();
    while current != &start {
        path.0.push(Point(current.0.0, current.0.1));
        path.1.push(Point(current.1.0, current.1.1));
        current = visited.get(current).unwrap();
    }
    path.0.push(Point(0, 0));
    path.1.push(Point(0, 0));
    path.0.reverse();
    path.1.reverse();
    path
}
```

5.3 Bidirektionale Breitensuche

```
/// Expandiert einen Knoten entlang der Pfeile und
/// hängt alle neuen Knoten an die Warteschlange
fn forward(
    l1: &Labyrinth,
    l2: &Labyrinth,
    current: &State,
    new_forward_queue: &mut Vec<State>,
    forward_visited: &mut HashMap<State, State>,
    backward_visited: &mut HashMap<State, State>,
) -> Option<State> {
    for inst in Instruction::iter() {
```

```
        let new_pos = State(
            l1.shift_point(&current.0, &inst),
            l2.shift_point(&current.1, &inst),
        );
        if &new_pos == current {
            continue;
        }
        if forward_visited.contains_key(&new_pos) {
            continue;
        }
        forward_visited.insert(new_pos.copy(), current.copy());
        if backward_visited.contains_key(&new_pos) {
            // Ein Pfad wurde gefunden
            return Some(new_pos);
        }
        new_forward_queue.push(new_pos);
    }
    None // Nichts besonderes ist passiert,
    // die Knoten wurden an die new_forward_queue angehängt
}

enum BackwardResult {
    None, // Nichts Besonderes ist passiert
    //(Die Knoten wurden an die new_backward_queue angehängt)
    Connection(State), // Pfad gefunden
    CannotContinue,    // ein Punkt ist (0,0),
    // d.h. die Suche rückwärts wird nicht weitergeführt
}


/// Expandiert einen Knoten entgegen der Pfeilrichtungen
fn backward(
    l1: &Labyrinth,
    l2: &Labyrinth,
    current: &State,
    new_backward_queue: &mut Vec<State>,
    forward_visited: &mut HashMap<State, State>,
    backward_visited: &mut HashMap<State, State>,
) -> BackwardResult {
    let mut can_continue = true;
    for inst in Instruction::iter() {
        // Alle möglichen Zustände in Richtung inst werden betrachtet
        let mut temp_queue: Vec<State> = vec![];
        if l1.is_wall_in_direction(&current.0, &inst.inv()) {
            temp_queue.push(State(
                current.0.copy(),
                l2.shift_point_without_end_fix(&current.1, &inst),
            ));
        }
        if l2.is_wall_in_direction(&current.1, &inst.inv()) {
            temp_queue.push(State(
                l1.shift_point_without_end_fix(&current.0, &inst),
                current.1.copy(),
            ));
        }
        temp_queue.push(State(
            l1.shift_point_without_end_fix(&current.0, &inst),
            l2.shift_point_without_end_fix(&current.1, &inst),
        ));
        for new_pos in temp_queue {
            if &new_pos == current {
                continue;
            }
            if backward_visited.contains_key(&new_pos) {
                continue;
            }
            if &State(
```

```
                l1.shift_point(&new_pos.0, &inst.inv()),
                l2.shift_point(&new_pos.1, &inst.inv()),
            ) != current
            {
                continue;
            }

            backward_visited.insert(new_pos.copy(), current.copy());
            if forward_visited.contains_key(&new_pos) {
                return BackwardResult::Connection(new_pos);
            }
            if new_pos.0 == Point(0, 0) || new_pos.1 == Point(0, 0) {
                can_continue = false;
            }
            new_backward_queue.push(new_pos);
        }
    }
    if !can_continue {
        BackwardResult::CannotContinue
    } else {
        BackwardResult::None
    }
}

fn double_path_bidibfs(
    l1: &Labyrinth,
    l2: &Labyrinth,
    mut path: (Vec<Point>, Vec<Point>),
) -> (Vec<Point>, Vec<Point>) {
    assert!(l1.width == l2.width && l1.height == l2.height);
    let start = State(Point(0, 0), Point(0, 0));
    let end = State(
        Point(l1.width - 1, l1.height - 1),
        Point(l2.width - 1, l2.height - 1),
    );
    // Die Punkte die in Vorwärts- und in Rückwärtsrichtung besucht wurden
    let mut forward_visited: HashMap<State, State> = HashMap::new();
    let mut backward_visited: HashMap<State, State> = HashMap::new();
    forward_visited.insert(start.copy(), start.copy());
    backward_visited.insert(end.copy(), start.copy());

    // Jeweils zwei Warteschlangen für jede Richtungen, zwischen denen hin und her gewechselt
    // wird (immer eine aus der gelesen wird und eine in der die Ergebnisse gespeichert werden)
    let mut forward_queue1: Vec<State> = vec![start.copy()];
    let mut backward_queue1: Vec<State> = vec![end.copy()];
    let mut forward_queue2: Vec<State> = vec![start.copy()];
    let mut backward_queue2: Vec<State> = vec![end.copy()];
    let mut forward_queue = &mut forward_queue1;
    let mut forward_queue_result = &mut forward_queue2;
    let mut backward_queue = &mut backward_queue1;
    let mut backward_queue_result = &mut backward_queue2;

    // Speichert, ob die Suche in Rückwärtsrichtung vorgeführt werden kann.
    let mut can_continue_backwards = true;

    // Speichert, ob ein Pfad gefunden wurde
    let mut found_path = false;

    // Der Knoten, der die Vorwärts und Rückwärtssuchen ”verbindet”,
    // also von beiden besucht wird
    let mut conn = State(Point(0, 0), Point(0, 0));

    // Die aktuelle suchtiefe
    let mut depth = 0;

    while (forward_queue.len() > 0 && backward_queue.len() > 0) && !found_path {
        if depth % 100 == 0 {
```

```
            println!(
                "Depth: {}, Forward Stack {}, Backward Stack {},
                        Forward Visited {}, Backward Visited {}",
                depth,
                forward_queue.len(),
                backward_queue.len(),
                forward_visited.len(),
                backward_visited.len()
            );
        }
        depth += 1;

        // Die Richtung mit der kürzeren Warteschlange wird ausgewählt
        if forward_queue.len() <= backward_queue.len() || !can_continue_backwards {
            forward_queue_result.clear();
            for current in &*forward_queue {
                // Alle Knoten der forward_queue werden vorwärts Erweitert
                let result = forward(
                    l1,
                    l2,
                    &current,
                    &mut forward_queue_result,
                    &mut forward_visited,
                    &mut backward_visited,
                );
                match result {
                    None => {}
                    Some(state) => {
                        found_path = true;
                        conn = state;
                        break;
                    }
                }
            }
            // Die Referenzen zu den Warteschlangen werden vertauscht
            let temp = forward_queue_result;
            forward_queue_result = forward_queue;
            forward_queue = temp;
        } else {
            backward_queue_result.clear();
            for current in &*backward_queue {
                // Alle Knoten der backward_queue werden rückwärts erweitert
                let result = backward(
                    l1,
                    l2,
                    current,
                    backward_queue_result,
                    &mut forward_visited,
                    &mut backward_visited,
                );

                match result {
                    BackwardResult::None => {}
                    BackwardResult::Connection(state) => {
                        found_path = true;
                        conn = state;
                        break;
                    }
                    BackwardResult::CannotContinue => {
                        can_continue_backwards = false;
                    }
                }
                if (found_path) {
                    break;
                }
            }
            // Die Warteschlangen werden vertauscht
```

```
            let temp = backward_queue_result;
            backward_queue_result = backward_queue;
            backward_queue = temp;
        }
    }
    if !found_path {
        println!("No path found");
        return path;
    }
    // Der Pfad wird zurückverfolgt
    path.0.push(conn.0.copy());
    path.1.push(conn.1.copy());
    let mut current = forward_visited.get(&conn).unwrap();
    while current != &start {
        path.0.insert(0, Point(current.0.0, current.0.1));
        path.1.insert(0, Point(current.1.0, current.1.1));
        current = forward_visited.get(current).unwrap();
    }
    path.0.insert(0, start.1.copy());
    path.1.insert(0, start.0.copy());
    current = backward_visited.get(&conn).unwrap();
    while current != &start {
        path.0.push(Point(current.0.0, current.0.1));
        path.1.push(Point(current.1.0, current.1.1));
        current = backward_visited.get(current).unwrap();
    }
    path
}
```

5.4 Bidirektionale Array Breitensuche

```
#[derive(Clone)]
#[repr(u8)]
enum Cell {
    NotVisited,      // Der Zustand wurde noch nicht besucht
    ForwardVisited,  // Der Zustand wurde von der Breitensuche vorwärts besucht
    BackwardVisited, // Der Zustand wurde von der Breitensuche rückwärts besucht
}

// Das Macro macht es einfacher, das Element für einen bestimmten zustand in dem Array zu finden
macro_rules! state_at {
    ($states:expr, $state:expr, $l1_width:expr, $l1_height:expr, $l2_width:expr) => {
        $states[($state.0.1 as usize * $l1_width as usize + $state.0.0 as usize)
            * $l2_width as usize
            * $l1_height as usize
            + $state.1.1 as usize * $l2_width as usize
            + $state.1.0 as usize]
    };
}

fn forward_array(
    l1: &Labyrinth,
    l2: &Labyrinth,
    current: &State,
    new_forward_queue: &mut Vec<State>,
    states: &mut Vec<Cell>,
    l1_width: i16,
    l1_height: i16,
    l2_width: i16,
) -> Option<State> {
    for inst in Instruction::iter() {
        let new_pos = State(
            l1.shift_point(&current.0, &inst),
            l2.shift_point(&current.1, &inst),
        );
        if &new_pos == current {
            continue;
```

```rust
        }
        match state_at!(states, new_pos, l1_width, l1_height, l2_width) {
            Cell::NotVisited => {
                // Der Zustand wurde noch nicht besucht und wird daher als
                // vorwärts besucht markiert
                state_at!(states, new_pos, l1_width, l1_height, l2_width) =
                                                            Cell::ForwardVisited;
                new_forward_queue.push(new_pos);
            }
            // Der Zustand wurde schon von der anderen
            // Breitensuche besucht, d.h. ein Pfad ist gefunden
            Cell::BackwardVisited => return Some(new_pos),
            // Der Pfad wurde schon von der eigenen Breitensuche besucht,
            //  d.h. der Knoten wird nicht an die new_forward_queue angehängt
            Cell::ForwardVisited => continue,
        }
    }
    None
}

fn backward_array(
    l1: &Labyrinth,
    l2: &Labyrinth,
    current: &State,
    new_backward_queue: &mut Vec<State>,
    states: &mut Vec<Cell>,
    l1_width: i16,
    l1_height: i16,
    l2_width: i16,
) -> BackwardResult {
    let mut can_continue = true;
    for inst in Instruction::iter() {
        // Alle möglichen Punkte in Richtung inst werden gesucht
        let mut temp_queue: Vec<State> = vec![];
        if l1.is_wall_in_direction(&current.0, &inst.inv()) {
            temp_queue.push(State(
                current.0.copy(),
                l2.shift_point_without_end_fix(&current.1, &inst),
            ));
        }
        if l2.is_wall_in_direction(&current.1, &inst.inv()) {
            temp_queue.push(State(
                l1.shift_point_without_end_fix(&current.0, &inst),
                current.1.copy(),
            ));
        }
        temp_queue.push(State(
            l1.shift_point_without_end_fix(&current.0, &inst),
            l2.shift_point_without_end_fix(&current.1, &inst),
        ));
        for new_pos in temp_queue {
            if &new_pos == current {
                continue;
            }
            match state_at!(states, new_pos, l1_width, l1_height, l2_width) {
                Cell::NotVisited => {
                    if &State(
                        l1.shift_point(&new_pos.0, &inst.inv()),
                        l2.shift_point(&new_pos.1, &inst.inv()),
                    ) != current
                    {
                        continue;
                    }
                    state_at!(states, new_pos, l1_width, l1_height, l2_width) =
                        Cell::BackwardVisited;
                    if new_pos.0 == Point(0, 0) || new_pos.1 == Point(0, 0) {
                        // Anton oder Bea sind am Startfeld und könnten
```

```
                        // aus jedem Loch gekommen sein
                        can_continue = false;
                    }
                    new_backward_queue.push(new_pos);
                }
                // Der Zustand wurde schon vorwärts besucht,
                // d.h. ein Pfad wurde gefunden
                Cell::ForwardVisited => return BackwardResult::Connection(new_pos),
                // Der Zustand wurde schon rückwärts besucht,
                // d.h. er wird nicht an die new_backward_queue angehängt
                Cell::BackwardVisited => continue,
            }
        }
    }
    if !can_continue {
        BackwardResult::CannotContinue
    } else {
        BackwardResult::None
    }
}
fn double_path_bidibfs_array(
    l1: &Labyrinth,
    l2: &Labyrinth,
    path: (Vec<Point>, Vec<Point>),
) -> (Vec<Point>, Vec<Point>) {
    assert!(l1.width == l2.width && l1.height == l2.height);
    let start = State(Point(0, 0), Point(0, 0));
    let end = State(
        Point(l1.width - 1, l1.height - 1),
        Point(l2.width - 1, l2.height - 1),
    );

    let total_states =
        (l1.width as usize) * (l1.height as usize) * (l2.width as usize) * (l2.height as usize)
    let mut states = vec![Cell::NotVisited; total_states];
    print!(
        "Creating state array of size {} (Size {} mb)\n",
        total_states,
        (total_states * size_of::<Cell>()) / 1024 / 1024
    );
    print!("Size of one Cell: {} bytes\n", std::mem::size_of::<Cell>());
    state_at!(states, start, l1.width, l1.height, l2.width) = Cell::ForwardVisited;
    state_at!(states, end, l1.width, l1.height, l2.width) = Cell::BackwardVisited;

    // Jeweils zwei Warteschlangen für jede Richtungen, zwischen denen hin und her gewechselt wird
    let mut forward_queue1: Vec<State> = vec![start.copy()];
    let mut backward_queue1: Vec<State> = vec![end.copy()];
    let mut forward_queue2: Vec<State> = vec![start.copy()];
    let mut backward_queue2: Vec<State> = vec![end.copy()];
    let mut forward_queue = &mut forward_queue1;
    let mut forward_queue_result = &mut forward_queue2;
    let mut backward_queue = &mut backward_queue1;
    let mut backward_queue_result = &mut backward_queue2;

    // Speichert, ob die Suche in Rückwärtsrichtung vorgeführt werden kann.
    let mut can_continue_backwards = true;

    let mut found_path = false;
    let mut conn = State(Point(0, 0), Point(0, 0));
    let mut depth = 0;
    while (forward_queue.len() > 0 && backward_queue.len() > 0) && !found_path {
        if depth % 100 == 0 {
            println!(
                "Depth: {}, Forward Stack {}, Backward Stack {}",
                depth,
                forward_queue.len(),
                backward_queue.len(),
```

```
            );
        }
        depth += 1;
        // Es wird ausgewählt, in welche Richtung der nächste Schritt gehen soll
        // wenn noch rückwärts weitergemacht werden kann, wird die Richtung mit
        // der kürzeren Warteschlange gewählt
        if forward_queue.len() <= backward_queue.len() || !can_continue_backwards {
            // Die neue Warteschlange wird geleert
            forward_queue_result.clear();
            for current in &*forward_queue {
                // Jeder Knoten wird vorwärts expandiert
                let result = forward_array(
                    l1,
                    l2,
                    current,
                    forward_queue_result,
                    &mut states,
                    l1.width,
                    l1.height,
                    l2.width,
                );
                match result {
                    None => {}
                    Some(state) => {
                        // Ein Pfad wurde gefunden
                        found_path = true;
                        conn = state;
                        continue;
                    }
                }
            }
            // Die Warteschlangen werden ausgetauscht
            let temp = forward_queue_result;
            forward_queue_result = forward_queue;
            forward_queue = temp;
        } else {
            // Die neue Warteschlange wird geleert
            backward_queue_result.clear();
            for current in &*backward_queue {
                // Jeder Knoten wird rückwärts expandiert
                let result = backward_array(
                    l1,
                    l2,
                    current,
                    backward_queue_result,
                    &mut states,
                    l1.width,
                    l1.height,
                    l2.width,
                );
                match result {
                    BackwardResult::None => {}
                    BackwardResult::Connection(state) => {
                        // Ein Pfad wurde gefunden
                        found_path = true;
                        conn = state;
                        break;
                    }
                    BackwardResult::CannotContinue => {
                        // Ein Punkt ist (0,0), d.h. die Suche
                        // rückwärts wird nicht weitergeführt
                        can_continue_backwards = false;
                    }
                }
            }
            // Die Warteschlangen werden ausgetauscht
            let temp = backward_queue_result;
```

```
            backward_queue_result = backward_queue;
            backward_queue = temp;
        }
    }
    if !found_path {
        println!("No path found");
        return path;
    } else {
        println!("Found path at level {}", depth);
        path
    }
}
```

5.5 A*

```
/// Die Heuristik für den A* Algorithmus
fn heuristic(state: &State, width: i16, height: i16) -> i16 {
    ((width - 1) - state.0.0 + (height - 1) - state.0.1)
        .max((width - 1) - state.1.0 + (height - 1) - state.1.1)
}

// Ein Zustand wird gemeinsam mit seinem Wert von g gespeichert
// Dadurch spart man sich eine separate HashMap
struct AStarState(State, i16);


// Der Wert von g wird für die Gleichheit nicht berücksichtigt
impl PartialEq for AStarState {
    fn eq(&self, other: &Self) -> bool {
        self.0 == other.0
    }
}

// Der Wert von g wird beim Hashing nicht berücksichtigt
impl std::hash::Hash for AStarState {
    fn hash<H: std::hash::Hasher>(&self, state: &mut H) {
        self.0.hash(state);
    }
}


// Die Information, die sonst in der closed liste gespeichert wird
// wird in den Elementen der CameFrom HashMap gespeichert
struct CameFrom {
    state: State,
    is_closed: bool,
}

impl Eq for AStarState {}

fn double_path_a_star(
    l1: &Labyrinth,
    l2: &Labyrinth,
    mut path: (Vec<Point>, Vec<Point>),
) -> (Vec<Point>, Vec<Point>) {
    let start = State(Point(0, 0), Point(0, 0));

    let end = State(
        Point(l1.width - 1, l1.height - 1),
        Point(l2.width - 1, l2.height - 1),
    );
    // Diese PriorityQueue ist eine Max-Heap, daher werden die Prioritäten negiert,
    // um die Ordnung umzudrehen
    let mut open: PriorityQueue<AStarState, i16> = PriorityQueue::new();
    // Speichert, woher der Zustand erreicht wurde, und ob der Zustand teil der Closed Liste ist
    // d.h. wenn cameFrom[state].is_closed = true, dann ist state in der Closed Liste
    let mut cameFrom: HashMap<State, CameFrom> = HashMap::new();
```

```
cameFrom.insert(
    start.copy(),
    CameFrom {
        state: start.copy(),
        is_closed: false,
    },
);
open.push(AStarState(start.copy(), 0), 0);

while !open.is_empty() {
    let (current_a_star_state, _) = open.pop().unwrap();
    let current = current_a_star_state.0;
    let current_g = current_a_star_state.1;

    // Der Zustand ist "abgeschlossen"
    cameFrom.get_mut(&current).unwrap().is_closed = true;

    if current == end {
        // Ein Pfad ist gefunden
        break;
    }
    // Die neuen Kosten sind einfach zu berechnen, da jede Anweisung
    // gleich viel "kostet"
    let new_cost = current_g + 1;
    for inst in Instruction::iter() {
        let new_pos = State(
            l1.shift_point(&current.0, &inst),
            l2.shift_point(&current.1, &inst),
        );
        if (new_pos == current) {
            continue;
        }

        // Wenn die Position schon in der closed_list enthalten ist, wird sie übersprungen
        match cameFrom.get_mut(&new_pos) {
            Some(came_from) => {
                if came_from.is_closed {
                    // Die Position wurde schon besucht
                    continue;
                }
            }
            None => {}
        }

        match open.get_mut(&AStarState(new_pos.copy(), 0)) {
            Some(mut new_AStarState) => {
                // Der Zustand ist schon in der Open list
                if (new_cost) >= new_AStarState.0.1 {
                    // Es ist schon ein besser Pfad zu new_pos bekannt
                    continue;
                } else {
                    // Dieser Pfad zu new_pos ist besser, d.h. der Wert von g von new_pos
                    // wird aktualisiert
                    new_AStarState.0.1 = new_cost;
                }
            }
            None => {
                // Die Position ist noch nicht in der Open List
                open.push(
                    AStarState(new_pos.copy(), new_cost),
                    -(new_cost + heuristic(&new_pos, l1.width, l1.height)),
                );
                cameFrom.insert(
                    new_pos.copy(),
                    CameFrom {
                        state: current.copy(),
```

```
                            is_closed: false,
                        },
                    );
                    continue;
                }
            }
            cameFrom.insert(
                new_pos.copy(),
                CameFrom {
                    state: current.copy(),
                    is_closed: false,
                },
            );
            // Die Priorität in der Open List wird aktualisiert
            open.change_priority(
                &AStarState(new_pos.copy(), 0),
                -(new_cost + heuristic(&new_pos, l1.width, l1.height)),
            );
        }
    }
    // Wenn die Position schon in der closed_list enthalten ist, wird sie übersprungen
    match cameFrom.get_mut(&end) {
        Some(_) => {}
        None => {
            println!("No path found");
            return path;
        }
    }

    // Der Pfad wird zurückverfolgt
    path.0.push(end.0.copy());
    path.1.push(end.1.copy());
    let mut current = &end.copy();
    while current != &start.copy() {
        current = &cameFrom.get(&current).unwrap().state;
        path.0.push(Point(current.0.0, current.0.1));
        path.1.push(Point(current.1.0, current.1.1));
    }
    path.0.reverse();
    path.1.reverse();

    path
}
```