Aufgabe 2: Simultane Labyrinthe

Teilnahme-ID: 74130

Bearbeiter/-in dieser Aufgabe:
Matthew Greiner

April 28, 2025

Contents

1 Lösungsidee **1**
1.1 Breitensuche (BFS) . . . 2
1.2 A* . . . 2

2 Umsetzung **3**
2.1 Breitensuche (BFS) . . . 3
2.2 A* . . . 5

3 Komplexitätsanalyse **6**
3.1 BFS . . . 6
3.2 A* . . . 7

4 Beispiele **7**

5 Vergleich der Methoden **11**

6 Quellcode **12**
6.1 BFS . . . 12
6.2 A* . . . 15

1 Lösungsidee

Die Grundidee dieser Aufgabe ist es, eine möglichst kurze Anweisungssequenz zu bestimmen, mit der zwei Personen gleichzeitig von ihrem jeweiligen Startpunkt $(0, 0)$ zum Zielpunkt $(n-1, m-1)$ in zwei unterschiedlichen Labyrinthen gelangen. Beide Labyrinthe haben die gleiche Grundgröße ($n \times m$), unterscheiden sich jedoch in ihrer Struktur (Platzierung der Wände). Die Herausforderung dieses Problems liegt in der Synchronisation, denn beide Personen müssen dieselbe Anweisungssequenz befolgen. Das bedeutet, dass optimale Pfade für die beiden Spieler nicht ausreichen, da eine separate Betrachtung normalerweise zu einer insgesamt längeren Abfolge von Anweisungen führen würde. Daher ist es notwendig, die beiden Labyrinthe in einem kombinierten System zu betrachten, um eine für beide gültige und gleichzeitig global minimale Sequenz zu finden. Die Teilaufgabe B wird auch direkt mit betrachtet, da bei dieser nur spezielle Felder, Gruben, hinzugefügt werden. Die Lösungsidee und Umsetzung des Algorithmus, der die Gruben betrachtet, kann auch die Labyrinthe ohne Gruben (Teilaufgabe A) lösen.

Um dieses Problem zu lösen, wird es in ein Kürzeste-Wege-Problem überführt. Dazu wird ein Zustandsgraph modelliert, der alle möglichen gültigen Positionen der beiden Spieler zu jedem Zeitpunkt enthält. Ein Zustand S im Graph wird dabei definiert als ein Tupel (x_1, y_1, x_2, y_2), wobei (x_1, y_1) die Koordinaten von Spieler 1 und (x_2, y_2) die Koordinaten von Spieler 2 in ihrem Labyrinth darstellt. In diesem Graph repräsentiert eine Kante die Anwendung einer einzelnen Richtungsanweisung ($\leftarrow$, $\rightarrow$, $\uparrow$, $\downarrow$). Wenn eine dieser Anweisung für einen Spieler gegen eine Wand führt, bleibt der Spieler laut der

Aufgabenstellung auf seiner aktuellen Position stehen. In der wird auch definiert, dass ein Spieler auf seinem Zielfeld bleibt und weitere Bewegungsanweisungen ignoriert, sobald er es erreicht hat. Außerdem wird ein Spieler auf sein Startfeld $(0, 0)$ zurückgesetzt, wenn er auf eine Grube trifft.

Das Ziel dieser Modellierung ist, den kürzesten Pfad im Zustandsgraphen vom Startzustand $(0, 0, 0, 0)$ zum Zielzustand $(n-1, m-1, n-1, m-1)$ zu finden. Für diese Suche bietet sich die Breadth-First-Search Methode (BFS) an, da sie garantiert den kürzesten Weg in einem ungewichteten Graphen (alle Bewegungen kosten gleich viel) findet. Allerdings ist der A-Star Suchalgorithmus auch passend, der theoretisch das Ergebnis schneller finden sollte, da bei der Suche eine Heuristik verwendet wird. Im Folgenden werden beide dieser Algorithmen umgesetzt:

1.1 Breitensuche (BFS)

Die Breitensuche (BFS) ist ein Graphsuchalgorithmus, der systematisch den Graph erkundet, hier also die möglichen Bewegungen in den Labyrinthen. Sie beginnt beim Startzustand und untersucht alle erreichbaren Nachbarzustände. Anschließend werden für jeden dieser Nachbarn deren unbesuchte Nachbarn untersucht, und so weiter. Dieses Vorgehen erfolgt "Schicht für Schicht", basierend auf der Anzahl der Schritte (Kanten) vom Startzustand. Eine FIFO-Warteschlange (Queue) verwaltet die Reihenfolge der zu besuchenden Zustände. Da BFS die Zustände in der Reihenfolge ihrer Entfernung vom Start bearbeitet, findet sie garantiert den Pfad mit der geringsten Anzahl an Schritten, wenn solch ein Pfad existieren sollte. Um Endlosschleifen (u.a. aufgrund der Gruben) und Zyklen bei der Suche im Graphen zu verhindern, muss gespeichert werden, welche Zustände bereits besucht wurden. Bevor ein neuer Zustand zur Queue hinzugefügt wird, wird geprüft, ob dieser spezifische Zustand bereits besucht wurde. Wenn dies der Fall ist, wird dieser Zustand ignoriert und nicht erneut zur Queue hinzugefügt.

Die Einführung von Gruben stellt auf den ersten Blick eine Herausforderung dar, da sie Zyklen im Pfad von einem Spieler erzeugen. Für BFS ist dies allerdings kein Problem, da sie den kombinierten Zustand (x_1, y_1, x_2, y_2) betrachtet. Wenn ein Spieler in eine Grube fällt, ändert sich der kombinierte Zustand (z.B. zu $(0, 0, x_2, y_2)$). Sollte BFS durch weitere Züge (evtl. nach öfterem Fallen in Gruben) später wieder denselben Zustand erreichen, wäre dies kein Problem, da gespeichert wurde, dass dieser Zustand schon erreicht wurde.

1.2 A*

Als Alternative zur Breitensuche kann auch der A* Algorithmus verwendet werden, um den kürzesten Pfad im Zustandsgraphen zu finden. A* ist ein informierter Suchalgorithmus, der zusätzlich zu den bereits zurückgelegen Kosten ($g(n)$), eine Schätzung der verbleibenden Kosten bis zum Ziel ($h(n)$, die Heuristik) berücksichtigt, um potenziell weniger Zustände durchsuchen zu müssen, um die Lösung zu finden.
Es wird eine PriorityQueue verwendet, um zu entscheiden, welcher Zustand als nächstes besucht wird. Sei

- $g(n)$ die tatsächlichen Kosten (Anzahl der Schritte/Kanten/Anweisungen) vom Startzustand bis zum Zustand n.
- $h(n)$ eine heuristische Schätzung der minimalen Kosten (minimale Anzahl weiterer Schritte), um von Zustand n zum Zielzustand zu gelangen. Diese Heuristik muss *zulässig* (admissible) sein, d.h. um zu garantieren, dass die Lösung optimal ist, darf sie die tatsächlichen Kosten zum Ziel nie überschätzen.

Dann ist die Priorität eines Zustand n ist durch die Funktion

$$f(n) = g(n) + h(n) \tag{1}$$

bestimmt. $f(n)$ sind die geschätzten Gesamtkosten des Pfades vom Start zum Ziel über den Zustand n. Mit Hilfe der PriorityQueue wählt A* immer den Zustand mit dem niedrigsten $f(n)$-Wert zur Expansion aus. Für diese Aufgabe wird eine Heuristik benötigt, die die Distanz beider Spieler berücksichtigt. Eine geeignete zulässige Heuristik ist das Maximum der Manhatten-Distanzen der beiden Spieler zu ihren Zielen:

$$h(n) = max(|x_1 - \text{zielX}| + |y_1 - \text{zielY}|, |x_2 - \text{zielX}| + |y_2 - \text{zielY}|) \tag{2}$$

Die Manhatten-Distanz ist die minimale Anzahl von Schritten in einem Gitter ohne Berücksichtigung von Wänden oder Gruben. Da die gemeinsame Anweisungssequenz mindestens so lang sein muss wie der

(theoretisch) kürsteste Weg des Spielers, der am weitesten entfernt ist, überschätzt diese Heuristik nie die tatsächliche Anzahl der noch benötigen Schritte und ist daher zulässig.

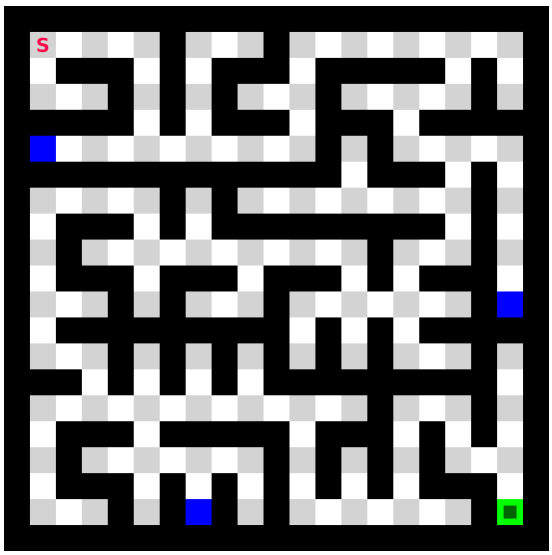
2 Umsetzung

2.1 Breitensuche (BFS)

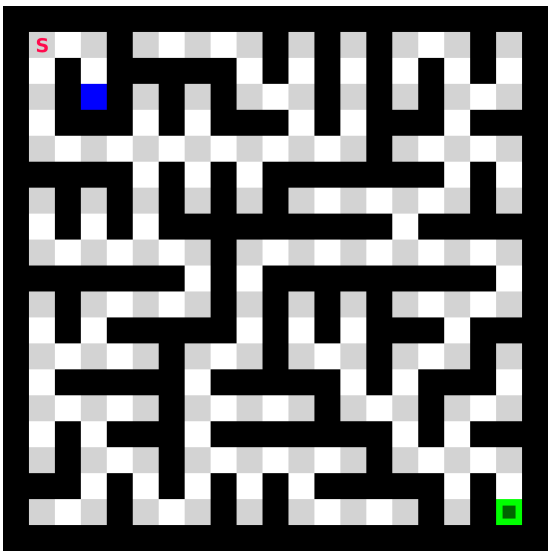
Bevor der Algorithmus angewendet werden kann, müssen die Labyrinthe aus den gegebenen Textdateien gelesen werden und in einer passenden Datenstruktur gespeichert werden. Die folgende Implementierung verwendet für das Speichern eines Labyrinths eine *Maze*-Klasse, die das Labyrinth intern in einem *byte[][]*-Array speichert. In den Eingabebeispielen der BwInf-Webseite werden die Labyrinthe der Größe $(n \times m)$ so dargestellt, dass sich die Spieler auf Koordinaten im Bereich von $(0, 0)$ bis $(n-1, m-1)$ befinden können. Es wird gegeben, zwischen welchen Koordinaten sich Wände befinden. Um in der Implementierung die begehbaren Felder und auch die Wände zwischen den Feldern explizit darzustellen, wird die Dimension des Arrays verdoppelt und um eins erhöht. Ein Labyrinth der Größe $(n \times m)$ wird also in einem Array der Größe $(height \cdot 2 + 1) \times (width \cdot 2 + 1)$ repräsentiert, wobei $width = n$ und $height = m$. Daraus folgt:

- Logische Felder (x, y) (mit $0 \leq x < width$, $0 \leq y < height$) haben den Index $(x \cdot 2 + 1, y \cdot 2 + 1)$ im Array
- Wände oder Freiräume zwischen den logischen Feldern haben gerade Indizes. Beispielsweise befindet sich die potenzielle vertikale Wand rechts vom Feld (x, y) bei $(x \cdot 2 + 2, y \cdot 2 + 1)$ und die potenzielle horizontale Wand unterhalb von (x, y) bei $(x \cdot 2 + 1, y \cdot 2 + 2)$

Der Wert 1 im Array markiert eine Wand, 0 einen freien Weg, und 2 eine Grube. Somit kann das Labyrinth leicht visuell dargestellt werden. Beispielsweise sehen die Labyrinthe aus der Beispieleingabe *labyrinthe2.txt* wie folgt aus:



(a) Labyrinth für Spieler 1



(b) Labyrinth für Spieler 2

Figure 1: **Schwarz**: Wand (1 im Array), **grau**: valides logisches Feld (0 im Array), **weiß**: Freiraum zwischen zwei logischen Feldern (kann nicht betreten werden), **grün**: Zielfeld, **rotes S**: Startfeld

Um diesen Algorithmus umzusetzen und somit die Aufgabe zu lösen, wird wie folgt vorgegangen (die Nummerierung entspricht auch der Nummerierung im Quellcode):

1. **Einlesen der Labyrinthe und deren Modellierung**
   Zur Einlesung der gegebenen Labyrinthe wird die Datei an den einzelnen Zeilen aufgespaltet. Die erste Zeile wird gelesen, um die logischen Dimensionen (*width, height*) der Labyrinthe zu bekommen. Basierend darauf werden zwei *Maze*-Objekte erstellt.
   Zunächst werden *height* Zeilen gelesen, die die Position der vertikalen Wände definieren. Der Wert 1 an der x-ten Position einer Zeile markiert eine Wand rechts vom logischen Feld (x, y), was intern

an der Array-Position $(x \cdot 2 + 2, y \cdot 2 + 1)$ gespeichert wird.
Anschließend folgen $height - 1$ Zeilen, die die horizontalen Wände beschreiben. Eine 1 an der x-ten Position stellt eine Wand unterhalb des logischen Feldes (x, y) dar, gespeichert an $(x \cdot 2+1, y \cdot 2+2)$. Schließlich wird die Anzahl der Gruben gelesen, gefolgt von den Zeilen mit den jeweiligen 0-basierten logischen Koordinaten (x, y) jeder Grube. Die Methode $setHole(x, y)$ von $Maze$ markiert die Zelle $(x \cdot 2 + 1, y \cdot 2 + 1)$ im internen Array als Grube mit dem Wert 2.

2. **Implementierung der Breitensuche (BFS)**

   a) **Kernkomponenten**
   Zu Beginn wird eine FIFO-Warteschlange *queue* (mit dem Java Datentyp *ArrayDeque*) angelegt, in der die zu besuchende Zustände sind. Die Klasse *State* stellt den Zustand der beiden Spieler zu einem Zeitpunkt dar und speichert zusätzlich zu den Koordinaten der Spieler, die Anzahl an benötigten Schritten vom Startknoten bis zu dem Zustand in *steps*, den Vorgängerknoten *parent* und die Anweisung, die zu diesem Zustand geführt hat in *move*. Dies wird gemacht, da die Rückkonstruktion des Pfads dadurch stark vereinfacht wird.

   In den Beispiellabyrinthen der BwInf-Webseite sind die größten Labyrinthe ca. (250×250) groß. Das bedeutet, dass im worst Case 250^4 Zustände besucht werden müssen und somit auch für jeden Zustand gespeichert werden muss, ob dieser bereits erkundet wurde. Das kann bei ca. 4 Milliarden gespeicherten Zuständen zu einem großen Speicherproblem werden. Daher wird ein Array aus BitSets zur Verwaltung der besuchten Zustände verwendet. BitSets werden verwendet, da sie intern ca. 1 Bit pro möglichen Zustand benötigen. Nun müssen die Zustände nur noch auf die Indizes der Bits im BitSet gemapped werden. Dann entspricht eine 1, dass dieser Zustand bereits besucht wurde, und eine 0, dass sie noch nicht erkundet wurde. Ob ein Zustand bereits erforscht wurde, kann somit in $\mathcal{O}(1)$ überprüft werden, was bei 250^4 Zuständen einen großen Unterschied im Speicherplatzverbrauch und der Laufzeit macht. Ein einzelnes BitSet verwendet intern int-Werte zur Indizierung der Bits. Dies begrenzt die maximale Anzahl direkt adressierbarer Zustände auf ca. 2,1 Milliarden. Wie bereits erwähnt, kann die Anzahl der besuchbaren Zustände in userem Problem, diese Grenze überschreiten. Daher reicht ein einzelnen BitSet nicht und um dieses Problem zu umgehen, wird der gesamte Zustandsraum logisch in mehrere Segmente unterteilt. Jedes Segment wird von einem eigenen BitSet-Objekt verwaltet, das zusammen in einem Array (*BitSet*[] *visited*) gespeichert wird.
   Die Zuweisung von den möglichen Zuständen auf eine Zahl erfolgt nach folgender Formel:

$$\text{index} = ((\texttt{long})x_1 \cdot \text{height} + y_1) \cdot \text{stateSpacePerMaze}) + ((\texttt{long})x_2 \cdot \text{height} + y_2) \quad (3)$$

   wobei stateSpacePerMaze = width · height. Das Casten auf `long` verhindert Überläufe des Darstellungsbereich von 32-bit Integern. Dann wird durch eine Division bestimmt, welches *BitSet* im Array den Zustand verwaltet:

$$\text{arrayIndex} = \left\lfloor \frac{\text{index}}{\text{BITSET\_SIZE}} \right\rfloor \quad (4)$$

   wobei (BITSET_SIZE) die maximale Größe eines einzelnen *BitSets* angibt. Der konkrete Bit-Index innerhalb des ausgewählten *BitSets* wird mit den Modulo-Operator berechnet:

$$\text{bitIndex} = \text{index} \bmod \text{BITSET\_SIZE}. \quad (5)$$

   b) **Start der Suche**
   Der Startzustand mit den beiden Startpositionen $(0, 0)$ der Spieler wird im BitSet-Array als besucht markiert und der FIFO-Warteschlange hinzugefügt.

   c) **Hauptschleife der BFS**
   Die while-Schleife wird ausgeführt, solange die *queue* noch Zustände zur Untersuchung enthält.

      i. **Zustand entnehmen**
      In jeder Iteration wird der vorderste Zustand, *currentState* aus der Warteschlange geholt. Dies ist der nächste unbesuchte Knoten auf der aktuellen Ebene.

      ii. **Zielprüfung**
      Der Algorithmus prüft direkt danach, ob im *currentState* beide Spieler ihre Zielkoordinaten $(width - 1, height - 1)$ erreicht haben. Wenn dies der Fall ist, wurde der kürzeste Pfad gefunden, und dieser *currentState* wird zurückgegeben, womit die Suche endet.

iii. **Nachfolgererkundung**
Wenn das Ziel noch nicht erreicht ist, werden alle vier möglichen Nachfolgezustände generiert, die durch die Anweisungen von (←, →, ↑, ↓) entstehen:

A. Für jede Richtung werden zunächst die potenziellen neuen Koordinaten berechnet.

B. Anschließend wird für jeden Spieler einzeln geprüft, ob die Bewegung auf eine Wand treffen würde; falls ja, bleibt der Spieler stehen.

C. Danach wird für die resultierende Position jedes Spielers geprüft, ob sie eine Grube ist; falls ja, wird die Position dieses Spielers auf den Start $(0, 0)$ zurückgesetzt.

Der so ermittelte Nachfolgezustand (nx_1, ny_1, nx_2, ny_2) wird dann betrachtet. Für diesen Nachfolgezustand wird geprüft, ob diese exakte Kombination von Spielerpositionen schon zuvor erreicht und im *visited*-BitSet markiert wurde. Wenn der Zustand neu ist, wird er als besucht markiert. Dann wird ein neues *State*-Objekt wird erstellt, das die neuen Koordinaten, die inkrementierte Schrittzahl, eine Referenz auf *currentState* als Vorgängerknoten und die verwendete Anweisung (*move*) enthält.
Dieses neue *State*-Objekt wird am Ende der FIFO-Warteschlange eingefügt, um später untersucht zu werden.

3. **Pfadrekonstruktion**
Wenn die Suche erfolgreich war, wird die Methode $reconstructPath(State finalState)$ aufgerufen. Sie iteriert vom $finalState$ rückwärts, indem sie dem *parent*-Link folgt, und sammelt dabei die *move*-Zeichen in einem *StringBuilder*. Am Ende wird der *StringBuilder* umgedreht, um die kürzeste Anweisungssequenz für die Spieler zu erhalten.

2.2 A*

Zur Umsetzung des A* Algorithmus werden die Methoden zum Einlesen und die Darstellung der Labyrinthe und Zustände, sowie die Methode zur Rekonstruktion des Pfads wiederverwendet. Daher muss nur die Speicherung der bereits besuchten Zustände und die Hauptmethode aus der BFS angepasst werden. Es wird wie folgt vorgegangen:

2. **Kernkomponenten**
Anstelle einer FIFO-Warteschlange wird eine Prioritätswarteschlange (PriorityQueue) verwendet. Diese sortiert die Zustände nach einem Schätzwert $f(n)$ (berechnet mit 1) für die Gesamtkosten des Pfades durch diesen Zustand.
Im Gegensatz zur BFS, die nur wissen muss, ob ein Zustand besucht wurde, muss A* die minimalen Kosten ($g(n)$) speichern, mit denen jeder Zustand bisher erreicht wurde. Dies ist notwendig, da A* einen Zustand über einen längeren Pfad finden könnte, bevor später ein kürzerer Pfad zum selben Zustand entdeckt wird. Nur der kürzeste Pfad soll weiterverfolgt werden.
Zur Speicherung dieser minimalen Kosten wird in dieser Implementierung ein Integer-Array *costArray* verwendet. Die Größe dieses Arrays ($stateSpaceSize = width \cdot height \cdot width \cdot height$) entspricht der Gesamtzahl möglicher kombinierter Zustände. Jeder Index im Array repräsentiert einen einzigartigen Zustand. Das Array wird zu Beginn mit Integer.MAX_VALUE initialisiert, um anzuzeigen, dass noch kein Pfad zu diesen Zuständen gefunden wurde.[1]
Die Formeln 3, 4, 5 funktionieren analog für das int-Array wie für das BitSet Array aus der Breitensuche.
Zudem werden die Methoden $isBetterPath(x1, y1, x2, y2, newCost)$ und $updateCost(x_1, y_1, x_2, y_2)$ definiert. $isBetterPath()$ prüft, ob der neu gefundene Pfad zu einem Zustand (mit Kosten *newCost*) besser ist als der bisher beste Pfad zu diesem Zustand. Sie tut dies, indem sie *newCost* mit dem Wert vergleicht, der aktuell für diesen Zustand im *costArray* gespeichert ist. $UpdateCost()$ schreibt die Kosten (*cost*, also $g(n)$) für den gegebenen Zustand an den entsprechenden Index im costArray. Außerdem wird die Heuristik mit der Gleichung 2 berechnet.

3. **Start der Suche** Das *costArray* wird vollständig mit Integer.MAX_VALUE gefüllt. Der Startzustand *startState* wird erstellt und zur PriorityQueue hinzugefügt.

[1] Anmerkung: Diese Implementierung mit einem einzelnen int[] costArray und int-Indizierung funktioniert nur korrekt, solange stateSpaceSize den Wert Integer.MAX_VALUE (ca. 2,147 Mrd.) nicht überschreitet. Für größere Labyrinthe wäre eine Aufteilung in mehrere Arrays (int[][] costArrays) und die Verwendung von long-Indizes sinnvoll, analog zur BitSet-Implementierung bei BFS. Allerdings wird dann so viel Speicher verbraucht, dass dies nicht mehr rentabel ist und somit nicht mehr umgesetzt wurde.

4. **Hauptschleife von A***
   Analog zur Schleife von BFS.

3 Komplexitätsanalyse

Für die folgende Komplexitätsanalyse sei

- n die Breite des Labyrinths.
- m die Höhe des Labyrinths.
- V die Gesamtzahl der möglichen kombinierten Zustände (x_1, y_1, x_2, y_2). $V = (n \cdot m)^2$. Dies entspricht der Anzahl der Knoten im Zustandsgraphen.
- E die Gesamtzahl der möglichen Kanten zwischen Zuständen im Graphen.
- P die Länge des gefundenen kürzesten Pfades (Anzahl der Anweisungen).

3.1 BFS

Schritt	**Laufzeit**	**Speicherbedarf**
Einlesen der Datei	$\mathcal{O}(n \cdot m)$	$\mathcal{O}(n \cdot m)$
BFS	$\mathcal{O}(V)$	$\mathcal{O}(V)$
Rekonstruktion des Pfads	$\mathcal{O}(P)$	$\mathcal{O}(P)$
Gesamt	$\boldsymbol{\mathcal{O}((n \cdot m)^2)}$	$\boldsymbol{\mathcal{O}((n \cdot m)^2)}$

Table 1: Laufzeit- und Speicherkomplexität BFS

- **Einlesen der Datei**: Für das Einlesen der Datei müssen die Dimensionen eingelesen werden, die Maze-Objekte (mit byte-Arrays der Größe $\mathcal{O}(n \cdot m)$) initialisiert werden und die Wand- und Grubeninformationen geparsed werden. Die Anzahl der Wand-/Grubeninformationen ist proportional zu $n \cdot m$. Somit liegt die Laufzeit bei $\mathcal{O}(n \cdot m)$.
- **BFS**: Zur Initialisierung der *visited*-BitSets wird $\mathcal{O}(V)$ Zeit benötigt, da für jedes der $\mathcal{O}(\frac{V}{\text{BITSET\_SIZE}})$ BitSets jeweils $\mathcal{O}(1)$ Aufwand nötig ist. Allerdings ist der konstante Faktor durch BITSET_SIZE sehr klein.
  Die while-Schleife wird maximal V-mal durchgelaufen, da jeder Zustand höchstens einmal aus der *queue* genommen wird. Das Entnehmen und Einfügen in die ArrayDeque benötigt amortisiert $\mathcal{O}(1)$. Der Zieltest ist in $\mathcal{O}(1)$ sowie auch die Generierung der Nachfolger (4 Richtungen), da die Wand-/Grubenprüfung in konstanter Zeit erfolgt und die Prüfung, ob ein Zustand bereits besucht wurde, wegen der Optimierung mit den BitSets im Durchschnitt $\mathcal{O}(1)$ benötigt.
  Insgesamt braucht die Verarbeitung eines Zustands und seiner (maximal 4) Nachfolger konstante Zeit pro Kante. Da jeder Zustand und jede Kante im schlimmsten Fall einmal betrachtet wird, entspricht die Laufzeit der Standard-BFS-Komplexität $\mathcal{O}(V + E)$. Da in diesem Graphen E direkt proportional zur Gesamtzahl der Knoten V ist $(E \leq 4 \cdot V)$, gilt: $E = \mathcal{O}(V)$. Daher vereinfacht sich die gesamte Laufzeit für BFS zu $\mathcal{O}(V)$ oder $\mathcal{O}((n \cdot m)^2)$.
  Der Speicherbedarf wird durch das visited-Array und die maximale Größe der Queue dominiert. Das Array speichert den Besucht-Status für jeden der V möglichen Zustände. Sie benötigt ca. 1 Bit pro Zustand, also insgesamt O(V) Bits. Die *queue* kann im Extremfall bis zu $\mathcal{O}(V)$ Zustände halten, die jeweils konstanten Speicherplatz brauchen (für Koordinaten, Referenzen etc.).
- **Rekonstruktion des Pfads**: Die Schleife läuft P Mal und die Operationen in der Schleife sind $\mathcal{O}(1)$. Das Umdrehen des StringBuilders ist in $\mathcal{O}(P)$.

3.2 A*

Schritt	Laufzeit	Speicherbedarf
Einlesen der Datei	$\mathcal{O}(n \cdot m)$	$\mathcal{O}(n \cdot m)$
A*	$\mathcal{O}(V \log V)$	$\mathcal{O}(V \log V)$
Rekonstruktion des Pfads	$\mathcal{O}(P)$	$\mathcal{O}(P)$
Gesamt	$\boldsymbol{\mathcal{O}((n \cdot m)^2 \log(n \cdot m))}$	$\boldsymbol{\mathcal{O}((n \cdot m)^2)}$

Table 2: Laufzeit- und Speicherkomplexität A*

Die Implementierung des A*-Algorithmus ist nur eine kleine Änderung der Breitensuche, weshalb viele Methoden wiederverwendet werden und somit die gleiche Laufzeit wie bei BFS haben. Entscheidende Unterschiede sind:

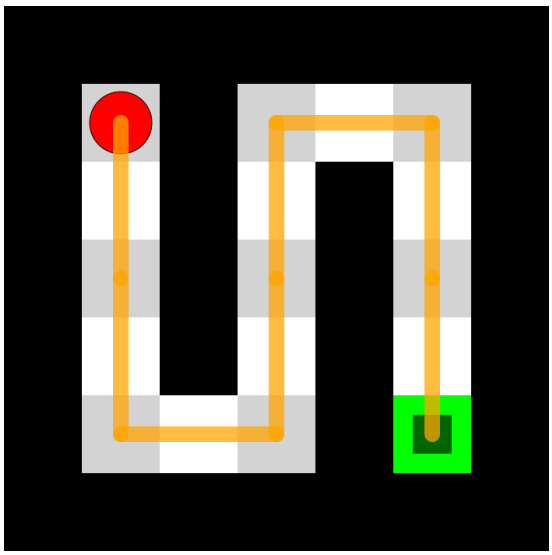
- A* verwendet eine PriorityQueue anstatt der ArrayDeque (Einfüge- und Entnehmoperationen in $\mathcal{O}(1)$). Diese priorisiert Zustände nach dem geschätzten Gesamtkostenwert $f = g+h$. Zustände mit einem niedrigeren f-Wert werden zuerst bearbeitet, was intern mit einem Min-Heap erreicht wird, der für *add*- und *poll* Aufrufe $\mathcal{O}(\log Q)$ braucht, wobei Q die Größe der Queue ist (bis zu $\mathcal{O}(V)$). Daher ist die Laufzeit im worst-Case $\mathcal{O}(V \log V)$. Die tatsächliche Anzahl an besuchten Zuständen hängt sehr stark von der gewählten Heuristik ab.
- Da durch die Priorisierung ein Zustand evtl. über einen suboptimalen Pfad zuerst erreicht wird, reicht es nicht, nur zu speichern, ob ein Zustand besucht wurde (wie im BitSet der BFS). Sondern A* muss die minimalen Kosten speichern, mit denen jeder Zustand bisher erreicht wurde. Das wird in einem int-Array gemacht, der 32 Bits für jeden der V Zustände braucht. Der Speicherbedarf ist somit trotzdem in $\mathcal{O}(V)$, braucht in der Praxis aber 32-mal soviel wie BFS.

4 Beispiele

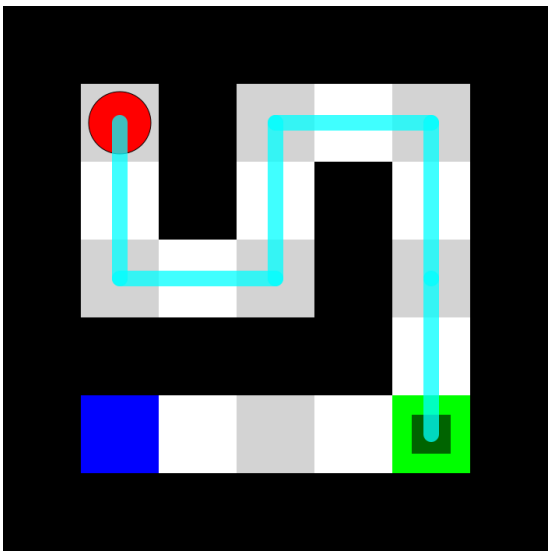
Hier sind die Ausgaben der Programme mit den Beispiellabyrinthen von der BwInf-Webseite. Die gesamte Programmausgabe ist in **Aufgabe2/Ausgaben/ProgrammausgabenBFS.txt** und **Aufgabe2/Ausgaben/ProgrammausgabenAStar.txt** zu finden. Die SVGs für die verwendeten Wege können in dem Folder **Aufgabe2/Ausgaben/ BFS oder AStar** gefunden werden. Zusätzlich ist hier die Ausführungszeit angegeben (intel i3-12100f, mit 32GB Ram).

1. **labyrinthe0.txt**

   BFS



(a) Labyrinth für Spieler 1



(b) Labyrinth für Spieler 2

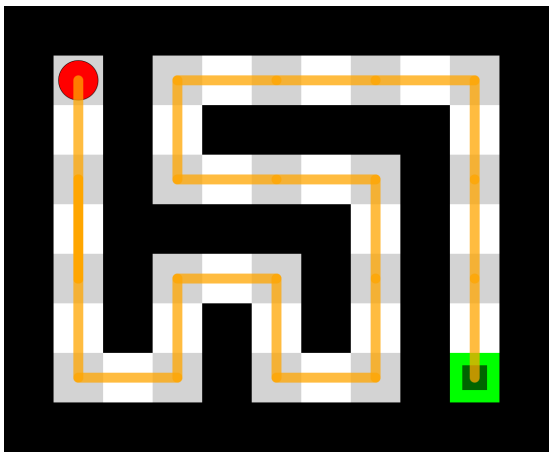
```
States visited: 29
Ausführungszeit in Sekunden für BFS: 0.035
Anweisungsfolge der Länge 8: ↓↓→↑↑→↓↓
```

A*

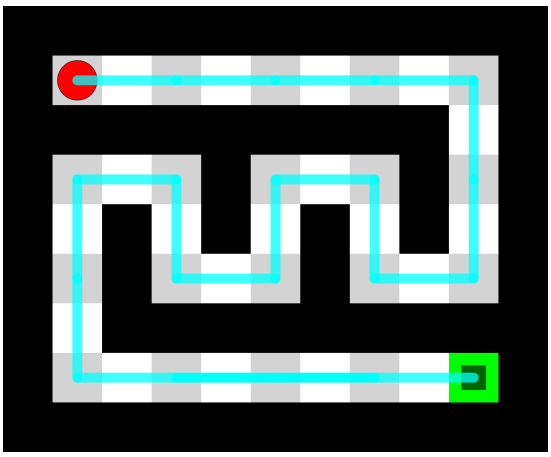
```
States visited: 23
Ausführungszeit in Sekunden für A*: 0.019
Anweisungsfolge der Länge 8: ↓↓→↑↑→↓↓
```

2. **labyrinthe1.txt**

BFS



(a) Labyrinth für Spieler 1



(b) Labyrinth für Spieler 2

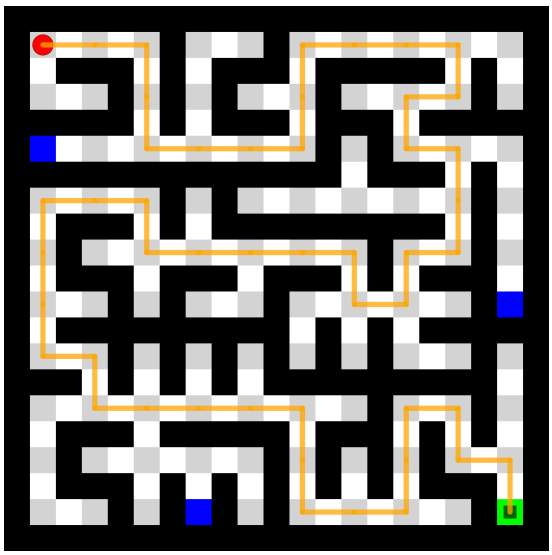
```
States visited: 342
Ausführungszeit in Sekunden für BFS: 0.031
Anweisungsfolge der Länge 31: →→→→↓↓←↑←↓←↑←↓↓→↑→↓→↑↑←←↑→→→↓↓↓
```

A*

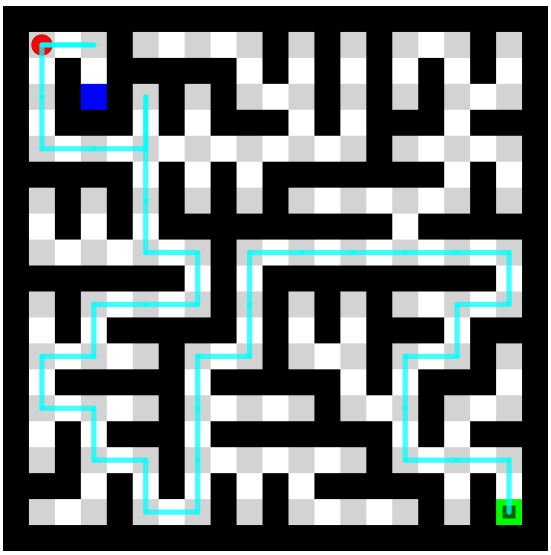
```
States visited: 320
Ausführungszeit in Sekunden für A*: 0.016
Anweisungsfolge der Länge 31: →→→→↓↓←↑←↓←↑←↓↓→↑→↓→↑↑←←↑→→→↓↓↓
```

3. **labyrinthe2.txt**

   BFS



(a) Labyrinth für Spieler 1



(b) Labyrinth für Spieler 2

```
States visited: 8,921
Ausführungszeit in Sekunden für BFS: 0.043
Anweisungsfolge der Länge 65: →→↓↓→↓→→↑↑→→→↓↓←↓→↓↓←↓←↑←←←↑←←↓↓←↓→↓→→→→↓↓→→↑↑↑→↑↑
→→→→→↓←↓←↓↓→→↓
```

   A*

```
States visited: 8,876
Ausführungszeit in Sekunden für A*: 0.036
Anweisungsfolge der Länge 65: →→↓↓↓→→→↑↑→→→↓←↓↓→↓↓←↓←↑←←←↑←←↓↓←↓→↓→→→→↓↓→→↑↑↑→↑↑
→→→→→↓←↓←↓↓→→↓
```

4. **labyrinthe3.txt**

```
States visited: 235,651
Ausführungszeit in Sekunden für BFS: 0.101
Anweisungsfolge der Länge 164: ↓↓↓↓→↓↓→↑→→→↓↓↓↓→→→→↓←↓←↓↓↓↓←←↓←↓←↓↓↓↓→→↓↓→→↓↓→←↓
↓↓←↓→↓←←↓←↑←↓↓→→↓↓→→↓←↓↓→→→→↓↓→→↓→→↓→↓↓←←↓↓←↓←↓↓→↑→↓↓←↓←↓←↑←↑←↑←↓←↓→→↓↓↓→→↑→→↑→↑→
↓↓←↓↓←←←↑→↑←↓↓↓↓←↓↓↓→↑→↓→↓→↓→→→→→
```

   A*

```
States visited: 243,733
Ausführungszeit in Sekunden für A*: 0.104
Anweisungsfolge der Länge 164: ↓↓↓↓→↓↓→↑→→→↓↓↓↓→→→→↓←↓↓↓←↓↓↓←←↓←↓←↓↓↓↓→→↓↓→→↓↓→←↓
↓↓←↓→↓←←↓←↑←↓↓→→↓↓→→↓←↓↓→→→→↓↓→→↓→→↓→↓↓←←↓↓←↓←↓↓→↑→↓↓←↓←↓←↑←↑←↑←↓←↓→→↓↓↓→→↑→→↑→↑→
↓↓↓←↓←←←↑↑←↓→↓↓↓←↓↓↓→↑→↓→↓→↓→→→→→
```

5. **labyrinthe4.txt**

   BFS

```
States visited: 91,540,327
Ausführungszeit in Sekunden für BFS: 7.641
Anweisungsfolge der Länge 14384: →→→↓↓←↑←←↓→↓←↓→↓←↓→→↓→↑↑←↑↑→↓→↑→↓←↓←↓↓↓←←←↑←↓↓↓→↑
→→→↓↓↓→↑↑→→↑←←↑→→→→↑↑↑↑↑↓←←↑←↓←↓→↓←←↓→→→→↓←←←↑→→↑↑↑↑→↑←←↓←↑↑→→→→↓↓↓↓→↑→→→→→→...
```

A*

```
States visited: 98,109,451
Ausführungszeit in Sekunden für A*: 26.092
Anweisungsfolge der Länge 14384: →→→↓↓←↑←←↓→↓←↓→↓←↓→→↓→↑↑←↑↑→↓→↑→↓↓←←↓↓↓←←←↑←↓↓↓→↑
→→→↓↓↓→↑↑→→↑←←↑→→→→↑↑↑↑↓←↑↑←↓←←↓→↓←←↓→→→→↓←←←↑→↑↑↑↑→→↑←←↓←↑↑→→→↓↓↓↓→↑→→→→→→...
```

6. **labyrinthe5.txt**

   BFS

```
States visited: 412,785,957
Ausführungszeit in Sekunden für BFS: 42.501
Anweisungsfolge der Länge 1302: →→↓↓→→↑→↑→→↓←←↓↓←←↑↓→↓→↓→→↓←↓←←←↓→↓←↓→↓↓←↓↓→→↓→↓→↑
→↓↓↓←↓→→↓→→↓↓↓↓↓←←↓→→→↓→→→→↑→↓↓↓↓←←↓↓↓↓→↓↓←↓↓↓→↓→→↑↑→↑→↑↑→→↓→↑↑←↑→→↓→↑→→↑→↓→↑→→...
```

   A*

```
States visited: 441,430,843
Ausführungszeit in Sekunden für A*: 183.765
Anweisungsfolge der Länge 1302: →→↓↓→→↑→↑→→↓←←↓↓←←↓→↓→↓←↓←↑→→↓←←↓→↓←↓→↓↓←↓↓→→↓→↓→↑
→↓↓↓←↓→→↓→→↓↓↓↓↓←←↓→→→↓→→→→↑→↓↓↓↓←←↓↓↓↓→↓↓←↓↓↓→↓→→↑↑→↑↑↑→→↑→→↓→↑→→↑→↓→↑→→↑↑→↓↓↓...
```

7. **labyrinthe6.txt**

   BFS

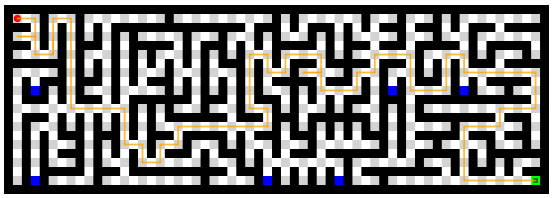
```
States visited: 395,670,022
Ausführungszeit in Sekunden für BFS: 41.325
Anweisungsfolge der Länge 776: →→→↓↓↓↓↓↓↓→→↓→↓↓→→→↓→↓→→↑→→↓↓→→→↓↓↓↓←←↓←↓↓→↓↓↓↓↓→↓
→↓↓←↓↓→→→↑↑→→→↓→→→→→→↑→→↓→→↑→↑→→↓→→→→→↓↓↓→→→↓↓→→→→→↓↓↓↓←↓←←↓←↓↓→→↓↓↓↓↓→→↓↓→...
```

   A*

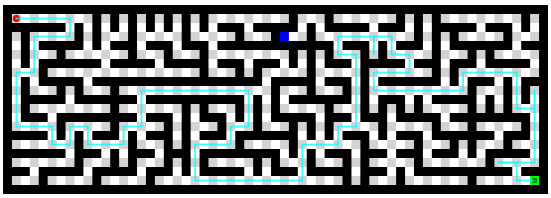
```
Labyrinth zu groß für diese A* Implementierung, verwende BFS! (ist schneller)
```

8. **labyrinthe7.txt**

   BFS



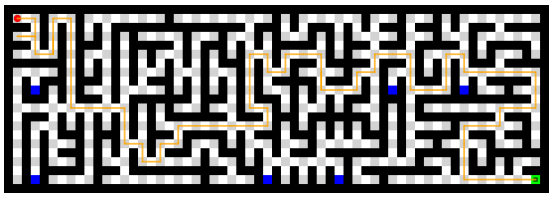
(a) Labyrinth für Spieler 1



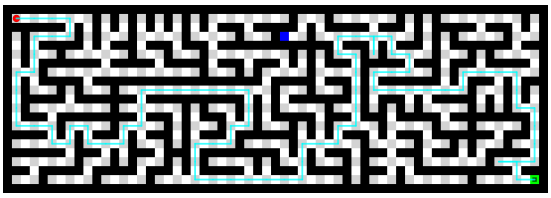
(b) Labyrinth für Spieler 2

```
States visited: 83,438
Ausführungszeit in Sekunden für BFS: 0.075
Anweisungsfolge der Länge 115: →→→↓←←↓↓←↓↓↓→→↓→↑→↓→→↑→↑↑→→→→→↓↓←↓←←↓↓→→→↓↓→↓→↑→↑→
→↑→→↑→↑↑←↑↑←↑→→↓→↑→↓→↓←←↓→↓→→↑→→↑→→→↓↓→→↑↑→↓→→→↓→↓↓←←↓←←↓↓→↓→→→→
```

   A*



(a) Labyrinth für Spieler 1



(b) Labyrinth für Spieler 2

```
States visited: 86,512
Ausführungszeit in Sekunden für A*: 0.077
Anweisungsfolge der Länge 115: →→→↓←←↓↓←↓↓↓→→↓→↑→↓→→↑→↑↑→→→→→↓↓←↓←←↓↓→→→↓↓→↓→↑→↑→
→↑→→↑→↑↑←↑↑←↑→↓→↓↑→↓→↓↓←←↑↓→→↑→↑→→↓↓→→↑↑→↓→→↓↓→↓→↓↓←←↓←←↓↓↓→↓→→→
```

9. **labyrinthe8.txt**

   BFS

```
States visited: 371,718,411
Ausführungszeit in Sekunden für BFS: 37.147
Anweisungsfolge der Länge 472: ↓→→↓↓↓↓→↓↓→↓↓↓→↓↓→→→↓→→→↓→→↑→↓→↓←↓→→↓→↓↓↓→↓↓↓→→→↓↓↓
→→→↑→→↑→↓→↓↓→↑→→→→→↓↓→→→→↓→↓→→↓→→→↓↓↓→→↓↓↓→→→↓↓←↓→↓←↓↓↓↓→→↓→↓↓→↓→→→↓→↓↓↓→→→↓...
```

   A*

```
States visited: 249,556,289
Ausführungszeit in Sekunden für A*: 114.49
Anweisungsfolge der Länge 472: ↓→↓↓↓→↓↓→↓↓↓→↓↓→↓→→→↓→→→↓→→↑↑→↓→↓←↓→↓→↓↓↓→↓↓↓→→↓↓↓→
↓→↑→→→↑→↓→↓↓→↑→→→→→↓↓→→→↓→↓→→→→↓→→↓↓↓→↓→↓↓→↓→→↓→↓←↓←↓↓→↓↓↓→→↓→↓↓→↓→↓→↓→→↓→→↓→↓...
```

10. **labyrinthe9.txt**

    BFS

```
States visited: 760,293,232
Ausführungszeit in Sekunden für BFS: 79.598
Anweisungsfolge der Länge 1012: ↓→↓→→↓→→↑→→↓↓↓→↑→↑→→↓↓↓↓→→↑↑→→↓→↓→↓↓→↓←↓←↓←←↓↓→↓↓→
↓→→→↑→↓↓↓↓↓↓←↓←↓→↓↓→↓→↑→→↑→→↑↑→↑→↑←↑→→↓↓→↓↓→→↑→→↓↓→→↓↓→→↓←↑↓←←←↓→↑→↓→↓←←←↓→→→↓←...
```

    A*

```
States visited: 644,735,858
Ausführungszeit in Sekunden für A*: 299.032
Anweisungsfolge der Länge 1012: ↓→↓→→↓→→↑→→↓↓↓→↑→↑→→↓↓↓↓→→↑↑→→↓→↓→↓↓→↓←↓←↓←←↓↓→↓↓→
↓→→→↑→↓↓↓↓↓↓←↓←↓→↓↓→↓→↑→→↑→→↑↑→↑→↑↑→↓→↓↓→↓↓→↓→↑←→→↓↓↓↓→→↓←↓←←←↑↓→↑→↓→↓←←←↓→→→↓←...
```

5 Vergleich der Methoden

Obwohl sowohl die Breitensuche (BFS) und auch der A* Algorithmus implementiert wurden, um die kürsteste Anweisungssequenz für dieses Labyrinthproblem zu finden, unterscheiden sie sich in ihren praktischen Eigenschaften.
Der grundlegende Unterschied ist die Suchstrategie. BFS ist eine Art "uninformierte" Suche, die den Zustandsraum systematisch Ebene für Ebene exploriert, was durch eine einfache FIFO-Warteschlange (ArrayDeque) effizient umgesetzt wird. Im Gegensatz dazu ist A* eine "informierte" Suche, die eine Heuristik (hier das Maximum der Manhattan-Distanzen) nutzt, um Zustände in der Suche zu priorisieren. Mit einer Prioritätswarteschlange (PriorityQueue) werden Zustände bevorzugt, die laut der Schätzung $f = g + h$ näher am Ziel liegen.
Diese Priorisierung kann potenziell dazu führen, dass das Ziel mit der Exploration von weniger Zuständen als BFS gefunden wird. Dieser potenzielle Vorteil ist aber nicht garantiert. Die Verwaltung der PriorityQueue und die Berechnung der Heuristik für jeden Zustand führt zu einem höheren Rechenaufwand pro Schritt im Vergleich zu BFS. Die Effektivität von A* hängt außerdem stark von der Problemstruktur ab. Beispielsweise könnten die Gruben die Heuristik etwas irreführend machen: Ein Zustand nahe am Ziel kann durch eine Grube plötzlich zu einem Zustand mit einem sehr hohem Heuristikwert führen, was A* unter Umständen dazu verleitet, ungünstigere Pfade zu verfolgen. Es ist deswegen durchaus möglich, dass A* in manchen Situationen nicht weniger oder sogar ähnlich viele oder mehr Zustände wie BFS untersuchen muss, um den optimalen Pfad zu finden.
Noch ein großer Unterschied liegt im Speicherbedarf. Asymptotisch gesehen ist er gleich, allerdings reicht

es für BFS zu wissen, ob ein Zustand bereits besucht wurde oder nicht. Dies wird sehr speichereffizient in $BitSet[]visited$ gespeichert (ca. 1 Bit pro Zustand). A* hingegen muss sich für jeden erreichten Zustand die bisher minimalen Kosten (g) merken, um sicherzustellen, dass nur der beste Pfad weiterverfolgt wird. Diese Speicherung der Kosten im $int[]costArray$ braucht deutlich mehr Speicher (ca. 32 Bit pro Zustand). Bei sehr großen Labyrinthen wie das Beispiel *labyrinthe6.txt* kann dies der entscheidende Unterschied sein, ob der Algorithmus verwendet werden kann oder nicht.
Außerdem ist die Laufzeit im worst-Case ($\mathcal{O}(V \log V)$ vs. $\mathcal{O}(V)$), was wegen der aufwändigeren Operationen ($\mathcal{O}(\log V)$) der Prioritätswarteschlange von A* zustande kommt.
Zusammenfassend ist die implementierte Verson von BFS oft der bessere/effizientere Ansatz für dieses Problem mit diesen Eingabebeispielen. Die Vorteile von BFS liegen in der Einfachheit, sie garantiert Optimalität und vor allem wird deutlich weniger Speicher benötigt. A* kann durch die Heuristik zwar potenziell weniger Zustände besuchen (siehe z.B. *labyrinthe8.txt*, oder *labyrinthe9.txt*), aber der Rechen-Overhead pro Zustand (Priority Queue) gleicht diesen Vorteil wieder aus. Da das Einfügen und Entnehmen in die Queue für beide Algorithmen einer der meist gebrauchten Operationen ist, macht der log-Faktor in der Praxis einen großen Unterschied.
Daher ist BFS für dieses Problem die bessere Wahl.

Zudem anzumerken, dass, obwohl beide Algorithmen garantiert eine Lösung mit der minimalen Anzahl an Schritten finden, der gefundene Pfad nicht eindeutig sein muss. Wenn mehrere Pfade gleicher minimaler Länge existieren, hängt der spezifisch gefundene Pfad von der Reihenfolge der Zustände in der *queue* ab. Dies ist z.B. in *labyrinthe7.txt* zu sehen, wo am Ende der Sequenz ein kleiner Unterschied ist.

6 Quellcode

6.1 BFS

Hier die wichtigsten Teile des Quellcodes:

Einlesen der Labyrinthe

```
// 1. Parse input
public static StateMazes parseInput(String input) {
    // Save every line in Array of Strings
    String[] lines = input.split("\n");
    int index = 0;

    // Read width and height of the mazes
    String[] parts = lines[index++].trim().split("␣");
    int width = Integer.parseInt(parts[0]);
    int height = Integer.parseInt(parts[1]);
    Maze maze1 = new Maze(new byte[height * 2 + 1][width * 2 + 1], width, height);
    Maze maze2 = new Maze(new byte[height * 2 + 1][width * 2 + 1], width, height);

    // Read first maze
    // Num of height lines each with num of width - 1 elements -> vertical walls
    for (int y = 0; y < height; y++) {
        parts = lines[index].trim().split("␣");
        for (int x = 0; x < parts.length; x++) {
            if (Integer.parseInt(parts[x]) == 1) {
                maze1.set(x * 2 + 2, y * 2 + 1, 1);
            }
        }
        index++;
    }

    for (int y = 0; y < height - 1; y++) {
        parts = lines[index].trim().split("␣");
        for (int x = 0; x < parts.length; x++) {
            if (Integer.parseInt(parts[x]) == 1) {
                maze1.set(x * 2 + 1, y * 2 + 2, 1);
            }
        }
        index++;
    }
    maze1.numHoles = Integer.parseInt(lines[index++].trim());
```

```
    for (int i = 0; i < maze1.numHoles; i++) {
        parts = lines[index++].trim().split("␣");
        maze1.setHole(Integer.parseInt(parts[0]), Integer.parseInt(parts[1]));
    }

    // Read second maze
    // Num of height lines each with num of width - 1 elements -> vertical walls
    for (int y = 0; y < height; y++) {
        parts = lines[index].trim().split("␣");
        for (int x = 0; x < parts.length; x++) {
            if (Integer.parseInt(parts[x]) == 1) {
                maze2.set(x * 2 + 2, y * 2 + 1, 1);
            }
        }
        index++;
    }

    for (int y = 0; y < height - 1; y++) {
        parts = lines[index].trim().split("␣");
        for (int x = 0; x < parts.length; x++) {
            if (Integer.parseInt(parts[x]) == 1) {
                maze2.set(x * 2 + 1, y * 2 + 2, 1);
            }
        }
        index++;
    }
    maze2.numHoles = Integer.parseInt(lines[index++].trim());

    for (int i = 0; i < maze2.numHoles; i++) {
        parts = lines[index].trim().split("␣");
        maze2.setHole(Integer.parseInt(parts[0]), Integer.parseInt(parts[1]));
    }

    return new StateMazes(maze1, maze2);
}
```

Hauptschleife der Breitensuche

```
// 2. Use BFS to find shortest sequence of moves
public static State bfs(Maze maze1, Maze maze2) {
    // 2a.
    int count = 0; // Counter for visited states (for progress printing)

    // FIFO queue for BFS states
    Queue<State> queue = new ArrayDeque<>();
    // Create Bitsets
    initializeBitSets();
    // 2b. Create Startstate and add starting state
    State startState = new State(0, 0, 0, 0, 0, null, '␣');
    queue.add(startState);
    setVisited(0, 0, 0, 0);

    // 2c. Main BFS loop
    while (!queue.isEmpty()) {
        if (count % 1000000 == 0 && count != 0) {
            System.out.println("States␣visited:␣" + count);
        }

        // 2ci. Get the next state from front of the queue
        State currentState = queue.poll();
        count++;

        // 2cii. Check if end state reached
        if (currentState.x1 == maze1.getGoalX() && currentState.x2 == maze2.getGoalX()
                && currentState.y1 == maze1.getGoalY() && currentState.y2 ==
    maze2.getGoalY()) {
            System.out.println("States␣visited:␣" + count);
            // Solution found
            return currentState;
        }

        // 2ciii. Explore neighbors (apply each move)
```

```
        for (int i = 0; i < 4; i++) {
            // Calculate potential next coordinates
            int nx1 = currentState.x1 + dx[i];
            int ny1 = currentState.y1 + dy[i];
            int nx2 = currentState.x2 + dx[i];
            int ny2 = currentState.y2 + dy[i];

            // Check walls if move is invalid, player stays
            if (!maze1.isValidMove(currentState.x1, currentState.y1, nx1, ny1)) {
                nx1 = currentState.x1;
                ny1 = currentState.y1;
            }
            if (!maze2.isValidMove(currentState.x2, currentState.y2, nx2, ny2)) {
                nx2 = currentState.x2;
                ny2 = currentState.y2;
            }

            // Check for holes if player lands on a hole, reset to start
            boolean reset1 = maze1.isHole(nx1, ny1);
            boolean reset2 = maze2.isHole(nx2, ny2);
            if (reset1) {
                nx1 = 0;
                ny1 = 0;
            }
            if (reset2) {
                nx2 = 0;
                ny2 = 0;
            }

            // Check if neighbor State is already visited
            if (!isVisited(nx1, ny1, nx2, ny2)) {
                // Mark as visited
                setVisited(nx1, ny1, nx2, ny2);
                // Create the new state and add it to the queue
                queue.add(new State(nx1, ny1, nx2, ny2, currentState.steps + 1,
    currentState,
                        moves[i].charAt(0)));
            }
        }
    }
    return null;
}
```

Rückkonstruktion des Pfades

```
// 3. reconstruct path from final state
public static String reconstructPath(State finalState) {
    StringBuilder path = new StringBuilder();
    // Iterate over parent chain and put into stringbuilder
    while (finalState.parent != null) {
        path.append(finalState.move);
        finalState = finalState.parent;
    }
    // Reverse the path to get final shortest path
    path.reverse();
    return path.toString();
}
```

Erstellung der BitSets

```
// 2a. create enough bitsets for maze
private static void initializeBitSets() {
    // Ensure no overflow and calculate amount of bitsets and size
    long totalStates = (long) width * height * width * height;
    numBitSets = (int) ((totalStates / BITSET_SIZE) + 1);
    visited = new BitSet[numBitSets];

    // Allocate each bitset segment
    for (int i = 0; i < numBitSets; i++) {
        visited[i] = new BitSet(BITSET_SIZE);
```

```
    }
}
```

Hilfsmethoden

```
// Map a state to index in bitsets
private static long encodeState(int x1, int y1, int x2, int y2) {
    long spacePerMaze = width * height;
    return ((long) x1 * height + y1) * spacePerMaze + ((long) x2 * height + y2);
}

// Set corresponding bit in Bitset to indicate if state already visited
private static void setVisited(int x1, int y1, int x2, int y2) {
    // Calculate which bitset and which bit
    long index = encodeState(x1, y1, x2, y2);
    int bitsetIndex = (int) (index / BITSET_SIZE);
    int bitIndex = (int) (index % BITSET_SIZE);

    // Set corresponding bit
    visited[bitsetIndex].set(bitIndex);
}

// Get value of corresponding bit from bitset to see if state already visited
private static boolean isVisited(int x1, int y1, int x2, int y2) {
    long index = encodeState(x1, y1, x2, y2);
    int bitsetIndex = (int) (index / BITSET_SIZE);
    int bitIndex = (int) (index % BITSET_SIZE);

    return visited[bitsetIndex].get(bitIndex);
}
```

Aus der Klasse Maze zur Überprüfung der Validität einer Anweisung für einen Spieler

```
// Use logic from maze represented in byte[][] array
public boolean isValidMove(int x, int y, int newX, int newY) {
    // Right
    if (x < newX) {
        if (x == width || get(x * 2 + 2, y * 2 + 1) == 1) {
            return false;
        }
        // Left
    } else if (x > newX) {
        if (x == 0 || get(x * 2, y * 2 + 1) == 1) {
            return false;
        }
    }
    // Down
    if (y < newY) {
        if (y == height || get(x * 2 + 1, y * 2 + 2) == 1) {
            return false;
        }
        // Up
    } else if (y > newY) {
        if (y == 0 || get(x * 2 + 1, y * 2) == 1) {
            return false;
        }
    }
    return true;
}
```

6.2 A*

Hauptschleife A* (fast gleich zu BFS)

```
// 2. Use A* to find shortest sequence of moves
public static State aStar(Maze maze1, Maze maze2) {
```

```
    // 2a.
    int count = 0;// Counter for visited states (for progress printing)

    // Priority queue for states
    PriorityQueue<State> queue = new PriorityQueue<>(
            Comparator.comparingInt(s -> s.steps + heuristic(s, maze1, maze2)));

    // Fill cost array with Inteer.MAX_VALUE
    Arrays.fill(costArray, Integer.MAX_VALUE);
    // 2b. Create Startstate and add starting state
    State startState = new State(0, 0, 0, 0, 0, null, ' ');
    queue.add(startState);
    updateCost(0, 0, 0, 0, 0);

    // 2c. Main A* loop
    while (!queue.isEmpty()) {
        if (count % 1000000 == 0 && count != 0) {
            System.out.println("States visited: " + count);
        }

        // 2ci. Get the next state from front of the queue
        State currentState = queue.poll();
        count++;

        // 2cii. Check if end state reached
        if (currentState.x1 == maze1.getGoalX() && currentState.x2 == maze2.getGoalX()
                && currentState.y1 == maze1.getGoalY() && currentState.y2 ==
    maze2.getGoalY()) {
            System.out.println("States visited: " + count);
            // Solution found
            return currentState;
        }
        // 2ciii. Explore neighbors (apply each move)
        for (int i = 0; i < 4; i++) {
            // Calculate potential next coordinates
            int nx1 = currentState.x1 + dx[i];
            int ny1 = currentState.y1 + dy[i];
            int nx2 = currentState.x2 + dx[i];
            int ny2 = currentState.y2 + dy[i];

            // Check walls if move is invalid, player stays
            if (!maze1.isValidMove(currentState.x1, currentState.y1, nx1, ny1)) {
                nx1 = currentState.x1;
                ny1 = currentState.y1;
            }
            if (!maze2.isValidMove(currentState.x2, currentState.y2, nx2, ny2)) {
                nx2 = currentState.x2;
                ny2 = currentState.y2;
            }

            // Check for holes if player lands on a hole, reset to start
            boolean reset1 = maze1.isHole(nx1, ny1);
            boolean reset2 = maze2.isHole(nx2, ny2);
            if (reset1) {
                nx1 = 0;
                ny1 = 0;
            }
            if (reset2) {
                nx2 = 0;
                ny2 = 0;
            }

            // Check if neighbor State is already visited
            int newCost = currentState.steps + 1;
            if (isBetterPath(nx1, ny1, nx2, ny2, newCost)) {
                // Update cost to state
                updateCost(nx1, ny1, nx2, ny2, newCost);
                // Create the new state and add it to the queue
                queue.add(new State(nx1, ny1, nx2, ny2, newCost, currentState,
                        moves[i].charAt(0)));
            }
        }
    }
```

```
    return null;
}
```

Heuristik

```
// Admissible heuristic for this problem
// Returns max ManhattenDistance from both the players
private static int heuristic(State state, Maze maze1, Maze maze2) {
    return Math.max(Math.abs(state.x1 - maze1.getGoalX()) + Math.abs(state.y1 -
    maze1.getGoalY()),
            Math.abs(state.x2 - maze2.getGoalX()) + Math.abs(state.y2 -
    maze2.getGoalY()));
}
```