

Aufgabe 2: Simultane Labyrinth

Teilnahme-ID: 74749

Bearbeiter/-in dieser Aufgabe:
Christian Krause

23. April 2025

Inhaltsverzeichnis

1	Lösungsidee	1
1.1	Optimale Lösung	1
1.1.1	Laufzeitkomplexität	2
1.1.2	Speicherplatzverbrauch	2
1.2	Bidirektionale Breitensuche	3
1.2.1	Optimalität	4
1.2.2	Laufzeit	4
2	Umsetzung	4
3	Erweiterung	4

Anleitung: Trage oben in den Zeilen 8 bis 10 die Aufgabennummer, die Teilnahme-ID und die/den Bearbeiterin/Bearbeiter dieser Aufgabe mit Vor- und Nachnamen ein. Vergiss nicht, auch den Aufgabennummern anzupassen (statt „ \LaTeX -Dokument“)!
Dann kannst du dieses Dokument mit deiner \LaTeX -Umgebung übersetzen.

Die Texte, die hier bereits stehen, geben ein paar Hinweise zur Einsendung. Du solltest sie aber in deiner Einsendung wieder entfernen!

1 Lösungsidee

... Einleitung

Annahmen: Das Labyrinth ist von Wänden umgeben ...

1.1 Optimale Lösung

Zuerst habe ich 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 (TOOD besser formulieren).

TODO vlt section Modellierung hier?

Jeder Zustand, in dem sich Anton und Bea befinden, kann als Tupel der jeweiligen Koordinaten dargestellt werden: $((x_0, y_0), (x_1, y_1))$. (x_0, y_0) ist hier die Position von Anton (der sich in Labyrinth 1 befindet), (x_1, y_1) beschreibt die Position von Bea im zweiten Labyrinth. Am Anfang herrscht der Zustand $S_0 = ((0, 0), (0, 0))$, da sich beide auf ihrem Startfeld befinden. Chris kann nun vier mögliche Anweisungen geben, die einen neuen Zustand herbeiführen würden. Wenn Anton und Bea beide ihr Zielfeld erreicht haben, befindet wir uns in dem Zustand $S_{end} = ((n - 1, m - 1), (n - 1, m - 1))$.

Formal können die verschiedenen Positionen von Anton und Bea als Graph dargestellt werden, dessen Knoten alle Möglichen Zustände sind.

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

1/4

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:

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

Da wir annehmen, dass das Labyrinth von Wänden umgeben ist, kann gibt es keine Position p , von der aus man mit einer Anweisung $a \in A$, eine Position $f(p, a)$ außerhalb des Labyrinths erreichen kann.

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:

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

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.

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

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

1.1.1 Laufzeitkomplexität

(TODO Löcher abziehen, weil da kann man ja nicht sein??)(TODO Pseudocode zum Laufzeit erklären)
Die Breitensuche hat eine Zeitkomplexität von $O(\|E\| + \|V\|)$, wobei $\|E\|$ die Anzahl der Kanten und $\|V\|$ die Anzahl der Knoten im Graphen ist. (TOOD Quelle)

Der oben beschriebene Graph besitzt einen Knoten für jede Mögliche Kombination an Positionen von Anton und Bea. Bea und Anton können jeweils $n \cdot m$ verschiedene Positionen einnehmen, d.h. insgesamt hat der Graph $\|V\| = n^2 m^2$ Knoten. Jeder Knoten hat höchstens vier ausgehende Kanten (da Schleifen nicht betrachtet werden müssen): $\|E\| \leq 4 \cdot n^2 m^2$. Damit entspricht die worst-case Laufzeit $O(n^2 m^2 + 4 \cdot n^2 m^2) = O(n^2 m^2)$.

1.1.2 Speicherplatzverbrauch

TODO $O(\|V\|) = O(n^2 m^2)$ -> viel

TODO Datenstrukturen für visited (array ist schlecht, hashtable (Python set ist besser))

1.2 Bidirektionale Breitensuche

Der oben beschriebene Algorithmus hat zwar die bestmögliche Asymptotische Laufzeit für einen Algorithmus (TODO Stimmt das?), lässt sich aber in der Praxis 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. (TODO bild queue) 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 (TODO Quelle die eien internetseite).

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

Algorithm 1 Bidirektionale Suche

```

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

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

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

Anschließend werden in beiden Fällen die neuen Knoten zur jeweiligen $Visited$ -Menge hinzugefügt. Am Ende wird überprüft, ob einer der neuen Knoten in der jeweiligen $Frontier$ -Variable bereits aus der anderen Richtung besucht wurde. Wenn dies der Fall ist, dann ist der kürzeste Pfad gefunden, da zu dem Knoten, der nun z.B. in $Frontier_1$ und $Visited_2$ enthalten ist, ein Pfad von S_0 aus und von S_{end} aus bekannt ist. Im Umsetzungsteil gehe ich genauer darauf ein, welche Möglichkeiten es gibt, diesen Pfad zu speichern und zurückzuverfolgen.

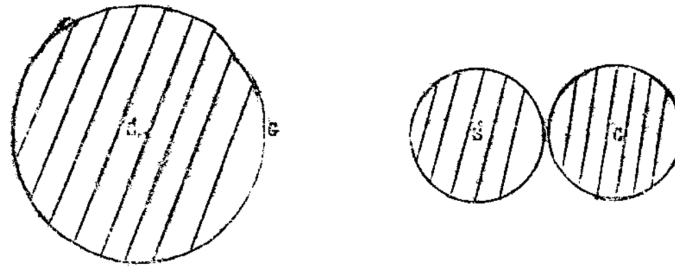


Figure: When a disc is grown from S only, area covered before contact is made between S and G is twice that required when discs are grown from S and G simultaneously.

Abbildung 1: TODO Quelle:

1.2.1 Optimalität

Warum??

1.2.2 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 $O(\|V\| + \|E\|)$.

TODO Quelle. https://en.wikipedia.org/wiki/Bidirectional_search#/media/File:Bidirectional_Search_in_1966.png Bidirektionale Suche ist schneller weil (intuition : des eine Bild mit zwei Kreisen))

2 Umsetzung

TODO man kann Bidirektionale suche Parallelisieren

3 Erweiterung

Mehrere Pfade? -> Ist die Intersection zwischen Frontier und Visited größer 1?

TODO gibt es mehrere mögliche kürzeste Anweisungssequenzen TODO die Personen bleiben im Zielfeld nicht stehen TODO nicht nur zwei labyrinth TODO mehrdimensionale Labyrinth