Aufgabe 2: Simultane Labyrinthe

Teilnahme-ID: 74749

Bearbeiter/-in dieser Aufgabe:
Christian Krause

26. April 2025

Inhaltsverzeichnis

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 Aufgabennamen 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 Breitensuche

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.

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

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:

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

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:

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

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.
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 $\|V\| = n^2m^2$ Knoten. Jeder Knoten hat höchstens vier ausgehende Kanten (da Schleifen nicht betrachtet werden müssen): $\|E\| \leq 4 \cdot n^2m^2$. Damit entspricht die worst-case Laufzeit $O(n^2m^2 + 4 \cdot n^2m^2) = O(n^2m^2)$.

1.1.2 Speicherplatzverbrauch

TODO $O(\|V\|) = O(n^2m^2)$ -> 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, der einen optimalen Pfad findet (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 entfert 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. Darus 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

```
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 Binärsuche von S_0 aus oder die von S_{end} aus um eine Ebene weitergeführt werden. In welcher Reihenfolge dies geschieht ist irrelevant, der Algorithmus ist in jedem Fall optimal. Hier wird die Seite weitergeführt, die zurzeit weniger „aktive“ Knoten hat, die erweitert werden sollen.
Die $Expand\_by\_one_1$ Funktion nimmt eine Menge an Knoten und gibt alle Knoten zurück, die genau eine Pfadlänge von einem der Knoten in *Frontier* entfernt liegen, was einem Schritt der Binärsuche entspricht. Die Funktion $Expand\_by\_one_1$ geht dabei entlang der Pfeile des Graphen und $Expand\_by\_one_2$ geht „Rückwärts“ in die entgegengesetzte Richtung.
Anschließend werden in beiden Fällen die neuen Knoten zur jeweiligen *Visited*-Menge hinzugefügt. Am

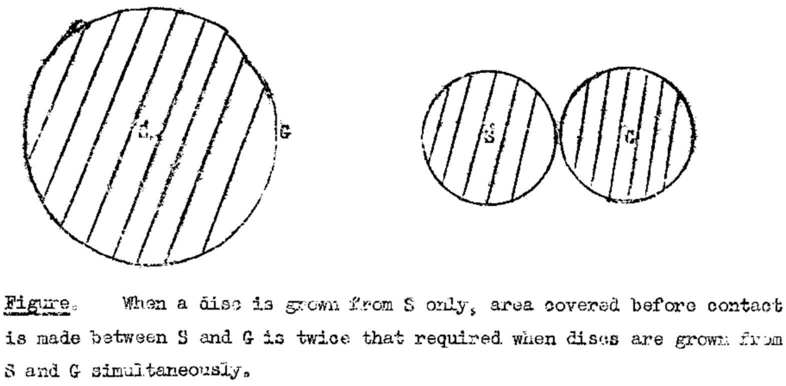

Abbildung 1: TODO QUelle:

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.

1.2.1 Optimalität

Warum??

1.2.2 Laufzeit

Asymptotisch ist die Laufzeit von bidirektionaler Breitensuche gleich wie die der normalen Breitensuche, im Wort-Case muss jeder Knoten und jede Kante besucht werden, also $O(\|V\| + \|E\|)$. TODO Quelle. `https://en.wikipedia.org/wiki/Bidirectional_search#/media/File:Bidirectional_Search_in_1966.png`

Intuitiv ist Bidirektionale Breitensuche aber schneller, da meistens weniger Knoten besucht werden müssen (siehe Bild 1.2.2).
Mathematisch lässt sich das schwer quantifizieren, man kann allerdings eine andere Obergrenze der Laufzeit der Breitensuche betrachten. Sei d die Länge des Pfades, der gefunden werden muss und b die Durchschnittliche Anzahl an ausgehenden Kanten eines Knotens, in unserem Fall $d = 4$. Die Laufzeit von Breitensuche lässt sich durch $O(b^d)$ begrenzen. Die zwei Breitensuchen 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. (TODO Quelle)
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.

2 Umsetzung

2.1 Breitensuche

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

2.2 Bidirektionale Breitensuche

Für die Bidirektionale Breitensuche benötigt man zwei Funktionen; die *forward* und die *backward* Funktion. Die *forward* „erweitert“ einen Knoten entlang der Richtung der Pfeile wie bei der normalen Breitensuche und gibt alle Knoten zurück, die eine Pfadlänge von dem Knoten entfernt sind und noch nicht besucht wurde.
Die *backward* Funktion muss für einen Knoten alle Knoten finden, von denen aus dieser Knoten mit einer Anweisung erreicht werden kann:

Algorithm 2 BACKWARD

```
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 2.2 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. (TODO Bild zum erklären) Das selbe gilt auch für *current.pos2*. Es gibt also in manchen Fällen bis zu drei mögliche Knoten, von denen aus *current* durch eine Anwendung von $inst^{-1}$ erreicht werden könnte. Für jeden dieser Knoten muss nun überprüft werden, ob das tatsächlich der Fall ist und ob der Knoten in der Suche beachtet werden muss. Dafür wird zuerst überprüft, ob der Knoten *new_pos* überhaupt unterschiedlich zu *current* ist. Wenn z.B. eine Wand in Richtung *inst* liegt, könnte es sein, dass Anton und Bea gegen die Wand gelaufen sind bei dem Versuch, *current* in Richtung *inst* zu verschieben.
Als nächstes wird überprüft, ob die Position schon besucht wurde, da sie in diesem Fall nicht beachtet werden muss. Schließlich wird überprüft ob man von *new_pos* aus *current* tatsächlich durch einen Schritt in $inst^{-1}$ erreichen kann.
Wenn alle notwendigen Bedingungen gegeben sind, kann *new_pos* in die *backward_visited* Hashmap eingetragen werden. Als Wert wird *current* eingetragen, wodurch später der Pfad zurückverfolgt werden

kann.
Anschließend wird überprüft, ob der Knoten *new_pos* bereits aus der anderen Richtung besucht wurde. Wenn dies der Fall ist, ist ein kürzester Pfad gefunden und der verbindende Knoten wird zurückgegeben. Von ihm aus kann nun durch die HashMaps *forward_visited* und *backward_visited* der Pfad in beide Richtungen zurückverfolgt werden.
Bei der *backward* Funktion müssen einige edge-cases beachtet werden. Die *shift* Funktion (in der Lösungsidee als $f(p, a)$ definiert für eine Anweisung $a \in A$ und eine Position p) verändert eine Position normalerweise nicht, falls sie bereits den Endzustand $p = (n-1, m-1)$ erreicht hat. Wenn in der *backward* Funktion allerdings Positionen „Rückwärts“ vom Endzustand aus verschoben werden sollen, ist das ein Problem. Drum verwende ich im Pseudocode die *shift'* Funktion, die diese Bedingung nicht beachtet. Im Quellcode heißt diese Funktion *shift_without_end_fix*.
Wenn Anton oder Bea in ein Loch fallen, gehen sie direkt zum Startfeld zurück. Das bedeutet, wenn die *backward* Funktion alle Felder ermitteln müsste, von denen Anton oder Bea mit einer Anweisung zum Startfeld gelangen können, müsste sie jede Position ausgeben, von der aus ein Loch mit einer Anweisung erreichbar ist. Da das für die meisten Beispieldateien eine sehr große Anzahl an Feldern ist, verwende ich die *backward* Funktion nicht mehr, sobald Anton oder Bea die Position $(0, 0)$ Rückwärts besucht haben. Von dort an wird dir 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.

TODO man kann Bidirektionale suche Parallelisierne TODO Bidirektionale Suche Array

3 Beispiele

Nummer des Labyrinths	Länge der Optimalen Anweisungssequenz	Laufzeit BFS	Laufzeit BidiBFS
0	8	$9\mu s$	$25\mu s$
1	31	$75\mu s$	$120\mu s$
2	66	$2.4ms$	$2.6ms$
3	164	$89ms$	$56ms$
4	14384	$113s$	$37s$
5	TODO	TODO	TODO
6	1844	$66s$	$107s$
7	TODO	TODO	TODO
8	TODO	TODO	TODO
9	TODO	TODO	TODO

4 Erweiterung

Bis jetzt ging es immer darum einen beliebigen optimalen Pfad in dem Graphen $G = (V, E)$ zu finden. Eine interessante Frage ist aber auch, wie viele optimale Pfade es gibt.

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 labyrinthe TODO mehrdimensionale Labyrinthe