Aufgabe 2: Simultane Labyrinthe

Teilnahme-ID: 74993

Bearbeiter dieser Aufgabe:
Mathieu de Borman

28. April 2025

Inhaltsverzeichnis

1 Lösungsidee

Jedes Labyrinth kann als Graph mit Knoten und gerichteten Kanten betrachtet werden. Abbildung 1 und 2 zeigen die Graphen der Labyrinthe aus "labyrinthe0.txt". Die Positionen sind von oben nach unten und von links nach rechts durchnummeriert. Vom Start aus unmögliche Bewegungen wurden ausgelassen.

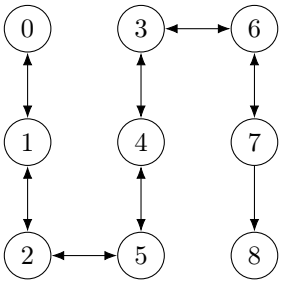

Abbildung 1: Antons Labyrinth

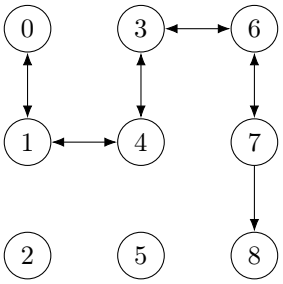

Abbildung 2: Beas Labyrinth

Das Problem, welches entsteht, wenn beide Labyrinthe simultan durchlaufen werden, kann auch als Graph dargestellt werden. Bei einem solchen Graphen stellt ein Knoten eine Kombination an Positionen dar und die Kanten mögliche Bewegungen. Abbildung 3 zeigt einen solchen Graphen für "labyrinthe0.txt". In der Aufgabe geht es nun darum, den kürzesten Weg durch diesen Graphen zu finden. Das könnte zwar durch Dijkstra oder A* gelöst werden, allerdings reicht schon Breadth-first search (BFS) aus, da alle Kanten das gleiche Gewicht haben.

Sei n die Breite und m die Höhe der Labyrinthe. Der Graph hat also n^2m^2 Knoten, was selbst bei der linearen Berechnungsdauer von BFS zu einer langen Laufzeit und einem großen Speicherplatzbedarf führt. Um das zu verhindern, wird die Anzahl an überprüften Elementen limitiert. Statt über alle Pfade einer Distanz zu iterieren, wird nur über die besten k iteriert. Als Heuristik um zu ermitteln, ob über einem Knoten iteriert werden soll, werden die Summen der Distanzen zwischen Knoten und den Zielfeldern innerhalb der Labyrinthe verwendet. Z.B. ergibt sich in unserem Beispiel für $(1, 6)$ der Wert $7 + 2 = 9$, da der kürzeste Weg von Position 1 zu 8 in Antons Labyrinth über 7 Kanten geht und der kürzeste Weg von Position 6 zu 8 in Beas Labyrinth über 2 Kanten geht. Für $(7, 3)$ wäre der Wert $1 + 3 = 4$. Dieser Algorithmus erlaubt es einem, durch die Wahl von k zu wählen, ob der kürzeste Pfad eher schnell oder eher Präzise berechnet werden soll.

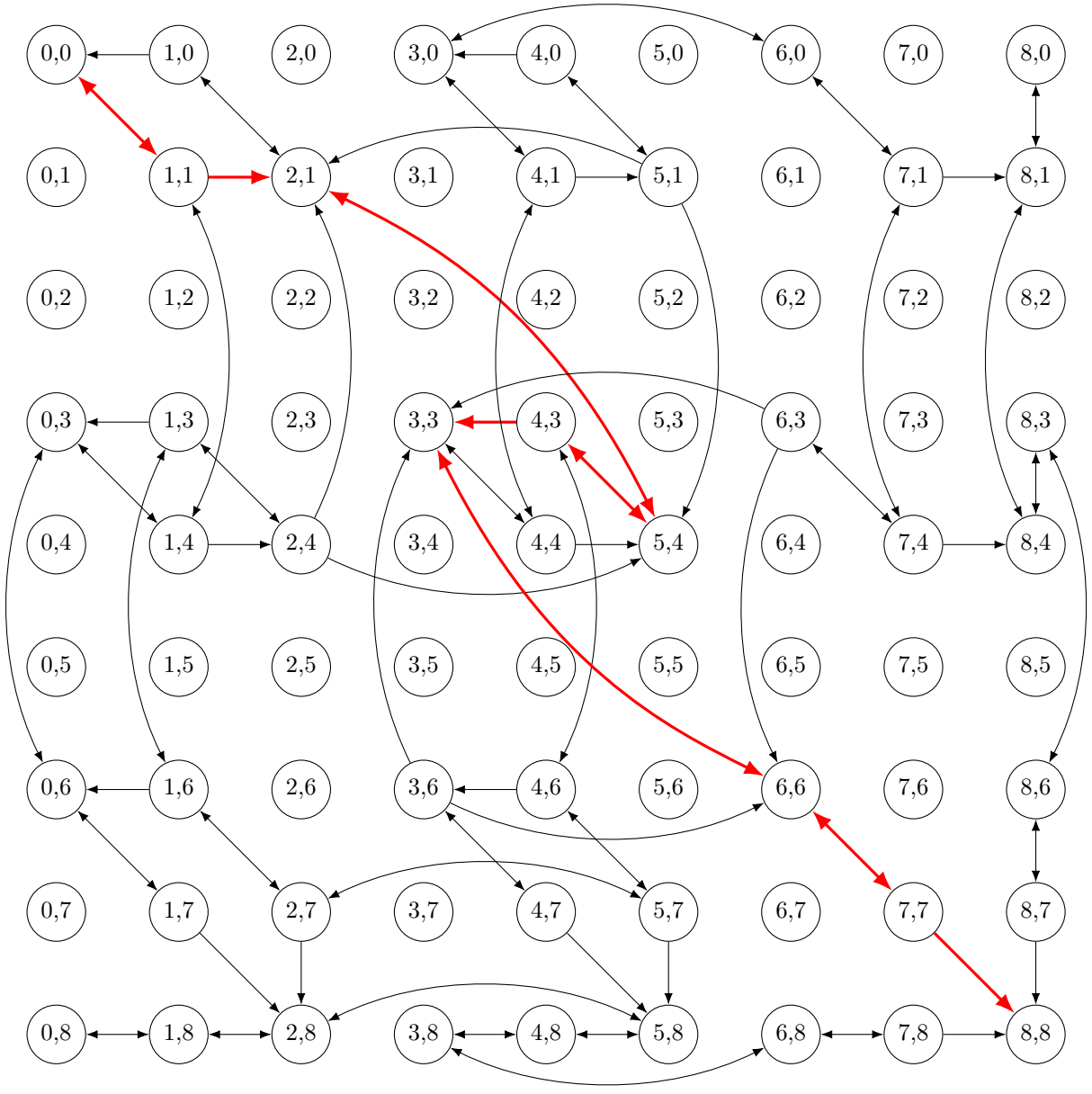

Abbildung 3: Das simultane Labyrinth
In Rot ist der kürzeste Pfad eingezeichnet

2 Umsetzung

Die Lösung wurde in JavaScript umgesetzt. Als Erstes werden die Eingabedaten konvertiert. Die Daten für ein Labyrinth werden in einer durchgehenden Liste gespeichert. Das Element an Stelle i entspricht der Position, wenn man beim Labyrinth von links nach rechts und von oben nach unten durchzählt (im Gegensatz zum Beispiel aus Abschnitt 1). Für jede Position im Labyrinth werden vier Bits gespeichert, die Aussagen in welche Richtungen gegangen werden darf. Außerdem wird ein fünftes Bit verwendet, um zu speichern, ob an der Position eine Grube ist.

Nun wird für jedes Labyrinth mithilfe der Funktion $singleBFS$ die Distanz zwischen Ziel und jeder vom Ziel erreichbaren Position ermittelt. In dem Fall, dass die Startposition nicht erreicht werden kann, gibt es für das gesamte Problem keine Lösung. Das Programm wird in dem Fall abgebrochen.

Anschließend wird die Funktion $multipleBFS$ aufgerufen. In dieser wird die annähernde Lösung mithilfe von dem limitierten BFS berechnet. Um zu speichern, welche Position schon besucht wurde, wird eine Hashmap verwendet. Durch die Heuristik eliminierte Positionenkombinationen werden, für den Fall, dass der Algorithmus in eine Sackgasse gerät, gespeichert. Sobald ein Weg zum Ziel gefunden wurde, wird Backtracking verwendet, um den genauen Lösungsweg zu ermitteln.

3 Komplexitätsanalyse

Ich verwende das Wort Komplexität stellvertretend für Laufzeit- und Speicherkomplexität, um die Lesbarkeit zu erhöhen. Die Komplexität wird in Abhängigkeit zur Breite n und Höhe m der Labyrinthe sowie der BFS-Limitierung k formuliert. Wir wissen, dass der kürzeste Weg maximal $2nm$ lang ist, wenn man die Labyrinthe nacheinander löst. Der Algorithmus prüft also maximal $2nmk$ Positionenkombinationen. Das Berechnen der Lösung hat demnach eine Komplexität von $\mathcal{O}(nmk \log k)$. Das $\log k$ kommt dadurch, dass in jeder Iteration die verschiedenen Positionen sortiert werden müssen. Ohne die Limitierung wäre die Komplexität $\mathcal{O}(n^2m^2)$. Was, im Vergleich zu Dijkstra mit $\mathcal{O}(n^2m^2 \log(nm))$, immer noch leicht besser ist. Auch die Erweiterung A* macht Dijkstra in diesem Fall nicht sehr viel schneller, da alle Kanten gleich gewichtet sind.

Obwohl nicht in der Aufgabenstellung gefordert, betrachte ich außerdem die Abhängigkeit zur Anzahl an Labyrinthen a. Für eine einfache BFS-Lösung wäre die Komplexität $\mathcal{O}(n^am^a)$. Durch die Limitierung ist die Komplexität $\mathcal{O}(nma^2k \log k)$. Dieser Algorithmus kann also auch viele simultane Labyrinthe ziemlich schnell lösen. Das Programm ist so umgesetzt worden, dass es mit einer beliebigen Anzahl an Labyrinthen umgehen kann.

4 Beispiele

Wir rufen nun das JavaScript-Programm mit den verschiedenen BwInf-Eingabedateien auf. Die Beispiele "labyrinthe10.txt" und "labyrinthe11.txt" zeigen, dass das Programm auch mit beliebig vielen Labyrinthen umgehen kann. Diese Dateien sind in demselben Ordner wie die Programmdatei. Mithilfe des "Datei Hochladen" Knopf kann man Beispieleingaben auswählen. Im Textfeld kann eingegeben werden, wie groß die BFS-Limitierung sein soll. Bei dem Kontrollkästchen kann ausgewählt werden, ob Gruben betrachtet werden sollen. Die Ausgabe erscheint auf der Webseite. Das Programm terminiert mit einem BFS-Limit von 100 für alle getesteten Eingaben in weniger als 10s auf einem gewöhnlichen PC. Für alle Beispieleingaben habe ich die Visualisierung des Labyrinths und die Anweisungsfolgen weggelassen, um diesen Abschnitt kurz zu halten. Alle Ergebnisse sind auch im Ordner der Programmdatei zu finden.

Eingabedatei	Anweisungen	Laufzeit
labyrinthe0.txt	8	0.001s
labyrinthe1.txt	31	0.003s
labyrinthe2.txt	67	0.037s
labyrinthe3.txt	164	0.101s
labyrinthe4.txt	14558	4.423s
labyrinthe5.txt	1330	0.618s
labyrinthe6.txt	498	0.178s
labyrinthe7.txt	115	0.069s
labyrinthe8.txt	487	0.315s
labyrinthe9.txt	1063	0.487s
labyrinthe10.txt	12	0.004s
labyrinthe11.txt	257	0.302s

Abbildung 4: ohne Gruben

Eingabedatei	Anweisungen	Laufzeit
labyrinthe0.txt	8	0.001s
labyrinthe1.txt	31	0.005s
labyrinthe2.txt	65	0.036s
labyrinthe3.txt	164	0.113s
labyrinthe4.txt	14558	5.234s
labyrinthe5.txt	1360	0.636s
labyrinthe6.txt	1844	0.788s
labyrinthe7.txt	Unlösbar	0s
labyrinthe8.txt	487	0.248s
labyrinthe9.txt	1063	0.451s
labyrinthe10.txt	14	0.014s
labyrinthe11.txt	278	0.379s

Abbildung 5: mit Gruben

5 Quelltext

```
//funktion um zu ermitteln, wo die Person nach einer Bewegung landet
function move(width, height, labyrinth, direction, position, forward = true) {
 let changes = [[0, -1], [1, 0], [0, 1], [-1, 0]]
 //falls in die Richtung gegangen werden darf und die Position nicht das Ziel ist
 if (labyrinth[position] & (1 << direction) && (position != height * width - 1 || !forward)) {
  //bewege die Person
  position += changes[direction][0] + changes[direction][1] * width
  //falls dort eine Grube ist, gehe zum Start
  if (labyrinth[position] & 16) {
   if (!forward) {
    position -= changes[direction][0] + changes[direction][1] * width
   } else {
    position = 0
   }
  }
 }
 return position
}
//funktion um alle in einer Anweisung erreichbaren Positionen zu ermitteln
function reachablePositions(width, height, labyrinths, positions) {
 let reachedPositions = []
 //für jede Richtung
 for (let direction = 0; direction < 4; direction++) {
  let movedTo = []
  let changed = false
  //für jedes Labyrinth, bewege die Person
  for (let j = 0; j < labyrinths.length; j++) {
   let newPosistion = move(width, height, labyrinths[j], direction, positions[j])
   movedTo.push(newPosistion)
   if (newPosistion != positions[j]) {
    changed = true
   }
  }
  //überprüfe, ob die neue Positionenkombination verschieden zu eingegebenen ist
  if (changed) {
   reachedPositions.push(movedTo)
  }
 }
 return reachedPositions
```

```
}
//funktion um die Distanzen in einem Labyrinth zu erfassen
function singleBFS(width, height, labyrinth) {
 let distances = Array(labyrinth.length).fill(Number.POSITIVE_INFINITY)
 let distance = 0
 let oldPositions = [width * height - 1]
 distances[oldPositions] = distance
 //solange noch nicht alles erfasst wurde
 while (oldPositions.length > 0) {
  distance++
  let newPositions = []
  //erfasse alle Positionen dieser Distanz
  for (let i = 0; i < oldPositions.length; i++) {
   for (let direction = 0; direction < 4; direction++) {
    let position = move(width, height, labyrinth, direction, oldPositions[i], false)
    if (distances[position] > distance) {
     distances[position] = distance
     newPositions.push(position)
    }
   }
  }
  oldPositions = newPositions
 }
 return distances
}
//funktion um mehrere Labyrinthe auf einmal zu lösen
function multipleBFS(width, height, labyrinths, distances, bound) {
 //heuristik um zu bestimmen, wie gut eine Position ist
 let heuristic = (heurPos) => {
  let sum = 0
  for (let i = 0; i < labyrinths.length; i++) {
   sum += distances[i][heurPos[i]]
  }
  return sum
 }
 let ways = {}
 let archive = []
 let oldPositions = [Array(labyrinths.length).fill(0)]
 ways[oldPositions[0] + ""] = oldPositions[0]
 let goal = Array(labyrinths.length).fill(width * height - 1) + ""
 //solange das Ziel nicht erreicht wurde
 while (!ways[goal]) {
  //eliminiere die schlechtesten Positionen
  oldPositions.sort((a, b) => heuristic(a) - heuristic(b))
  while (oldPositions.length > bound) {
   archive.push(oldPositions.pop())
  }
  //gehe alle Positionen durch, und ermittle neue erreichbare Positionen
  let newPositions = []
  for (let i = 0; i < oldPositions.length; i++) {
   let reachable = reachablePositions(width, height, labyrinths, oldPositions[i])
   for (let j = 0; j < reachable.length; j++) {
    let poition = reachable[j]
    if (!ways[poition + ""]) {
     ways[poition + ""] = oldPositions[i]
     newPositions.push(poition)
    }
   }
```

```
  }
  oldPositions = newPositions
  //falls nicht genügend Positionen zur Auswahl stehen, füge elimierte Positionen hinzu
  if (!oldPositions.length && archive.length) {
   oldPositions.push(archive.pop())
  }
 }
 let position = Array(labyrinths.length).fill(width * height - 1)
 let way = []
 let repeat = true
 //solange das Backtracking nicht am Start angekommen ist
 while (repeat) {
  //speichere wo du warst
  way.push(position)
  //gehe den Weg weiter zurück
  position = ways[position + ""]
  repeat = false
  //überprüfe, ob die Startposition erreicht wurde
  for (let i = 0; i < labyrinths.length; i++) {
   repeat |= position[i]
  }
 }
 way.push(position)
 return way.reverse()
}
//funktion um ein annäherndes optimales Ergebnis zu ermitteln
function main(width, height, labyrinths, bound) {
 let distances = []
 //berechne die Distanzen innerhalb der einzelnen Labyrinthe
 for (let i = 0; i < labyrinths.length; i++) {
  distances.push(singleBFS(width, height, labyrinths[i]))
  if (distances[i][0] == Number.POSITIVE_INFINITY) {
   return [`Nicht Lösbar, da Labyrinth ${i + 1} nicht Lösbar ist`]
  }
 }
 //berechne das Ergebnis
 return ["", multipleBFS(width, height, labyrinths, distances, bound)]
}
```