Brainfire: LaTeX-Dokumentation

BANDO

26. August 2019

Inhaltsverzeichnis

1 Pathfinder-Generierung

1.1 Grundidee

Betrachte die zwei Punkt S (für "Start") und Z (für "Ziel") in einem Raum. Da der Raum in seiner Ausbreitung beschränkt ist, existieren von S zu Z - bei gewünschter Platzierung der Steine - auch nur eine endliche Anzahl an denkbarer Lösungspfade. Ein Algorithmus soll all diese Pfade bestimmen, sortieren und speichern. Die für einen Pfad notwendigen Steine sollen ebenfalls gespeichert werden, aber auch die Menge aller Felder, die nicht mit Steinen belegt werden dürfen.

Wann immer ein Raum generiert werden soll, sodass mindestens eine Lösung von S nach Z existiert, so ist es hinreichend einen Pfad der gespeicherten Menge zufällig auszuwählen und die entsprechenden Felder mit Steinen zu belegen bzw. freizuhalten. Mögliche Laufzeit-Probleme sind auf diese Weise auf einen Zeitpunkt vor Beginn des Spiels verschoben worden. Das eigentliche Spiel bedarf auch nicht eines Algorithmus, der die Existenz einer Lösung sicherstellt.

tl;dr: Die Eigenschaft, dass mindestens ein Lösungspfad von S nach Z in einem Raum existiert, wird durch eine zuvor untersuchte und langfristig gespeicherte Menge definiert.

1.2 Umsetzbarkeit

Um die Idee auf Umsetzbarkeit zu überprüfen, wurde ein Backtracking-Algorithmus geschrieben, der strukturiert alle Lösungspfade von S nach Z in einem beliebig großen Raum bestimmt. Obwohl die Punkte S und Z frei wählbar sind, wurden sie wie im Skript festgelegt: S ist ein Feld unter der links oberen Ecke und Z ist ein Feld über der rechts unteren Ecke.

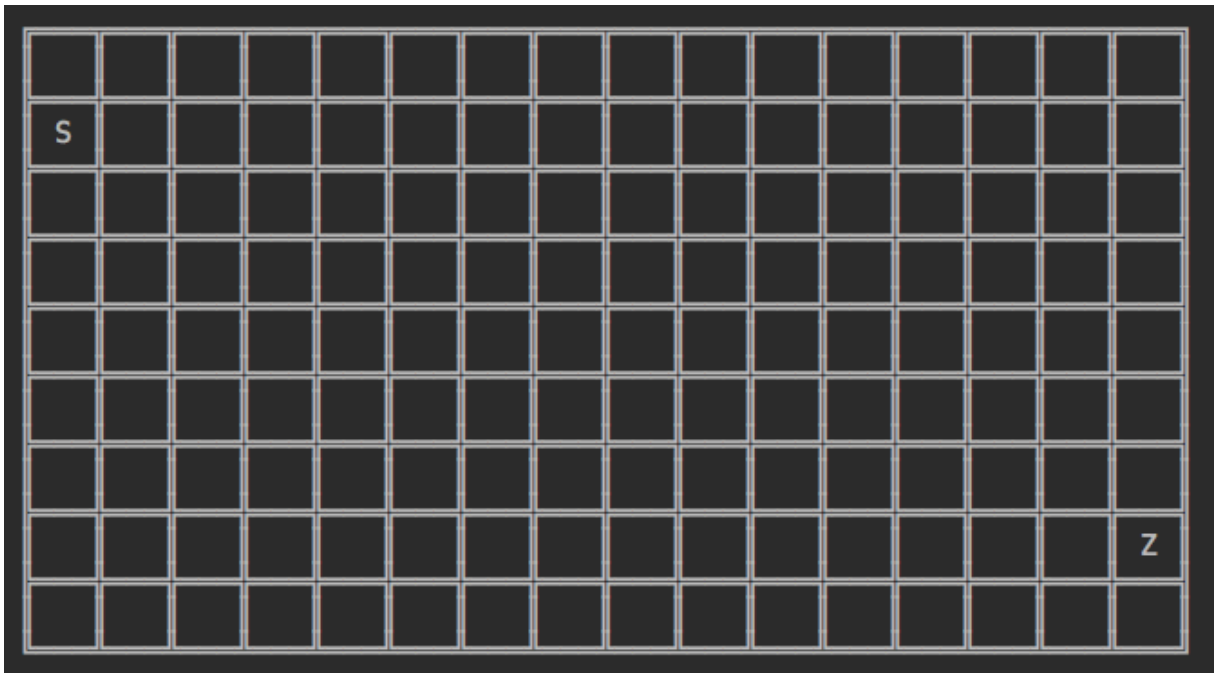

Abbildung 1: Position von S und Z in einem 16 mal 9 Raum

Der Algorithmus lieferte die Menge aller Lösungspfade und insbesondere auch die Anzahl der Lösungen. Folgende Tabelle hält die Menge aller Lösungen abhängig von der Dimension des Raums fest:

	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1
2	1	2	3	5	8	13	21	34
3	1	3	7	17	43	106	241	561
4	1	5	16	52	175	606	2113	7379
5	1	8	40	179	848	4522	22268	?
6	1	14	102	664	4300	32828	?	?
7	1	22	252	2462	24160	246222	?	?
8	1	36	605	8761	?	?	?	?

Abbildung 2: Menge aller Lösungen in Abhängigkeit von Breite und Höhe des Raums

Das Fazit, das aus dieser Betrachtung gezogen werden kann, ist, dass alle Raumgrößen im grauen Bereich der Tabelle nicht denkbar für diesen Generierungsansatz sind. Im besten Fall ist dieser Algorithmus für maximal 7x7 große Räume nutzbar.

1.3 Beispiel

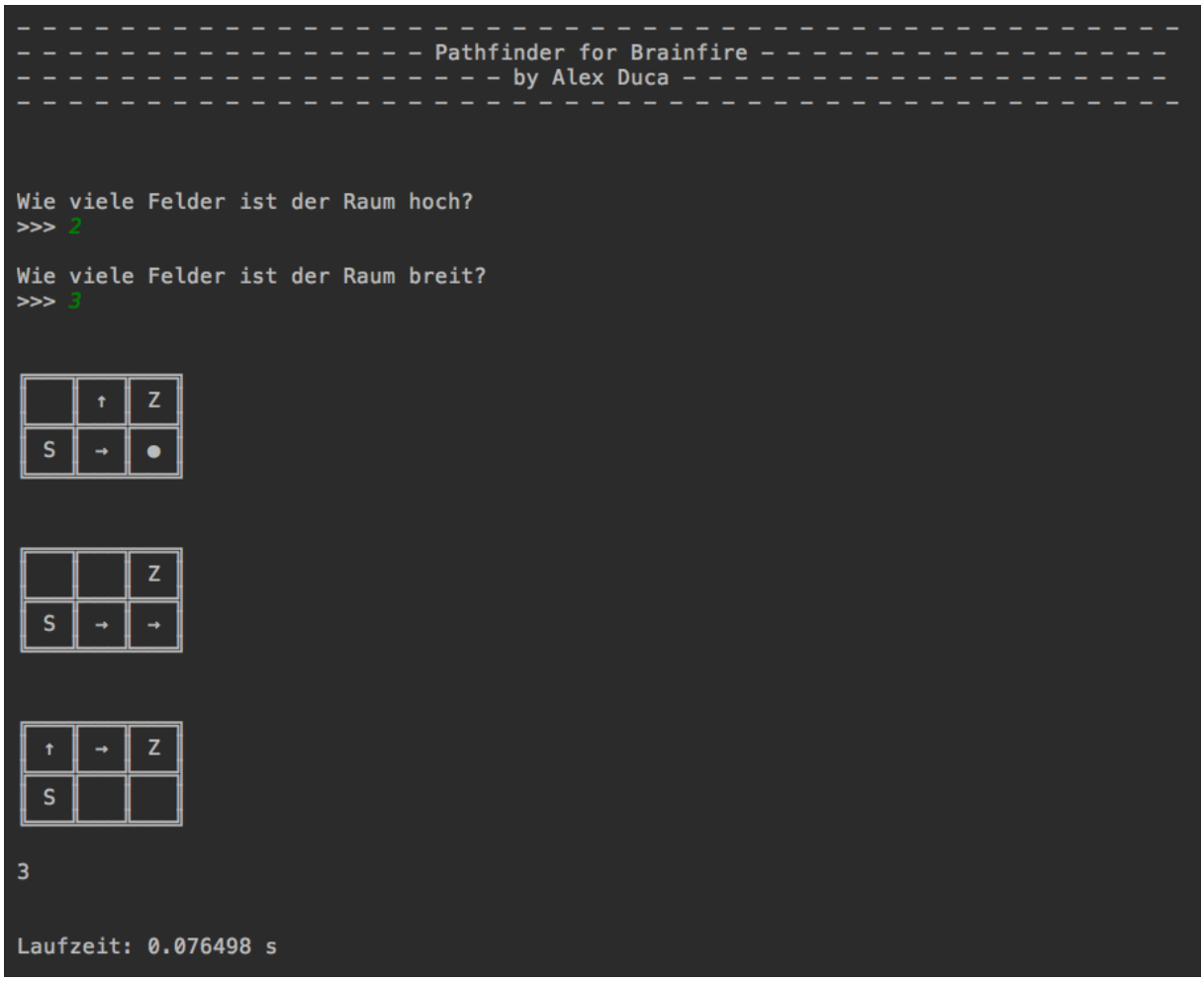

Abbildung 3: Menge aller Lösungen in einem 2x3-Raum

1.4 Quellcodeausschnitt

Methode, die das Backtracken in Gang setzt.

```
1       alleRaeume = []
        Richtungen = [ (1,0), (0,1), (-1,0), (0,-1) ]
3
        def startBacktracking(Matrix):
5
          # Globale Informationen
7         global alleRaeume
          alleRaeume = []
9         backupKopieMatrix = deepcopy(Matrix)

11        # Probiere in alle Richtungen
          for richtung in Richtungen:
13
            neuePos = startPos
15          Matrix = deepcopy(backupKopieMatrix)

17          while True:

19            altePos = neuePos

21            # Probiere einen Schritt nach vorne
```

```
            neuePos = ( altePos[0] + richtung[0], altePos[1] + richtung[1] )

            # Wenn du gegen eine Wand stoesst
            if not inRange(neuePos):
              # Wenn diese Wand nicht gleich am Anfang ist, dann backtracke
              if altePos != startPos:
                backtracking(Matrix, altePos, richtung)
                # Wir sind dann auch fertig fuer die Richtung
                break

            # Lege einen Pfeil auf dem Boden in aktuelle Laufrichtung
            Matrix[neuePos[0]][neuePos[1]] = pfeil(richtung)


            # Versuche einen Stein vor die neue Position zu legen und starte Backtracking
            stein = ( neuePos[0] + richtung[0], neuePos[1] + richtung[1] )
            if inRange(stein):

              Matrix[ stein[0] ][ stein[1] ] = "O"
              backtracking(Matrix, neuePos, richtung)

              # Nach erfolgreichem Steinelegen, nimm ihn wieder weg und lauf weiter in die Richtung
              Matrix[ stein[0] ][ stein[1] ] = "␣"


        for Raum in alleRaeume:
        printMatrixRaum(Raum)
        print(len(alleRaeume))
```

Methode, die beim Backtracking rekursiv aufgerufen wird.

```
    def backtracking(Matrix, eigenePos, letzteRichtung):

    # Globale Informationen
    global alleRaeume
    backupKopieMatrix = deepcopy(Matrix)

    # Sind wir schon am Ziel?
    if eigenePos == endPos:
      backupKopieMatrix[endPos[0]][endPos[1]] = "Z"
      alleRaeume.append(backupKopieMatrix)

    else:
      # lauf nur links oder rechts
      Senkrechten = senkrechte(letzteRichtung)
      for richtung in Senkrechten:

        neuePos = eigenePos
        Matrix = deepcopy(backupKopieMatrix)

        while True:

          altePos = neuePos

          # Probiere einen Schritt nach vorne
          neuePos = (altePos[0] + richtung[0], altePos[1] + richtung[1])

          # Ist da eine Wand?
          if not inRange(neuePos):
            # Wir koennen hier aufhoeren
            break

          # Liegt da ein Stein?
          elif Matrix[ neuePos[0] ][ neuePos[1] ] == "O":
            # Wir koennen hier aufhoeren
            break

          # Ist der Boden NICHT mit einem Pfeil markiert?
          elif Matrix[ neuePos[0] ][ neuePos[1] ] == "␣":

            # Markiere den Boden mit einem Pfeil
```

```
41              Matrix[neuePos[0]][neuePos[1]] = pfeil(richtung)

43              # Potentielle Abbiegung gefunden!
                stein = (neuePos[0] + richtung[0], neuePos[1] + richtung[1])
45
                # Ist dort, wo der Stein liegen sollte, schon eine Wand?
47              if not inRange(stein):
                  # Wenn Ja, dann kann man von der Wand aus abbiegen
49                backtracking(Matrix, neuePos, richtung)

51              # Wenn Nein, liegt dort schon ein Stein?
                elif Matrix[ stein[0] ][ stein[1] ] == "O":
53                # Wenn Ja, dann kann man von dem Stein aus abbiegen
                  backtracking(Matrix, neuePos, richtung)
55
                # Wenn Nein, ist die Flaeche frei fuer ein Stein?
57              elif Matrix[ stein[0] ][ stein[1] ] == "␣":
                  # Platziere einen Stein und biege ab!
59                Matrix[stein[0]][stein[1]] = "O"
                  backtracking(Matrix, neuePos, richtung)
61                # Und lege ihn danach wieder weg
                  Matrix[stein[0]][stein[1]] = "␣"
63
```