Brainfire: LaTeX-Dokumentation

BANDO

19. September 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.

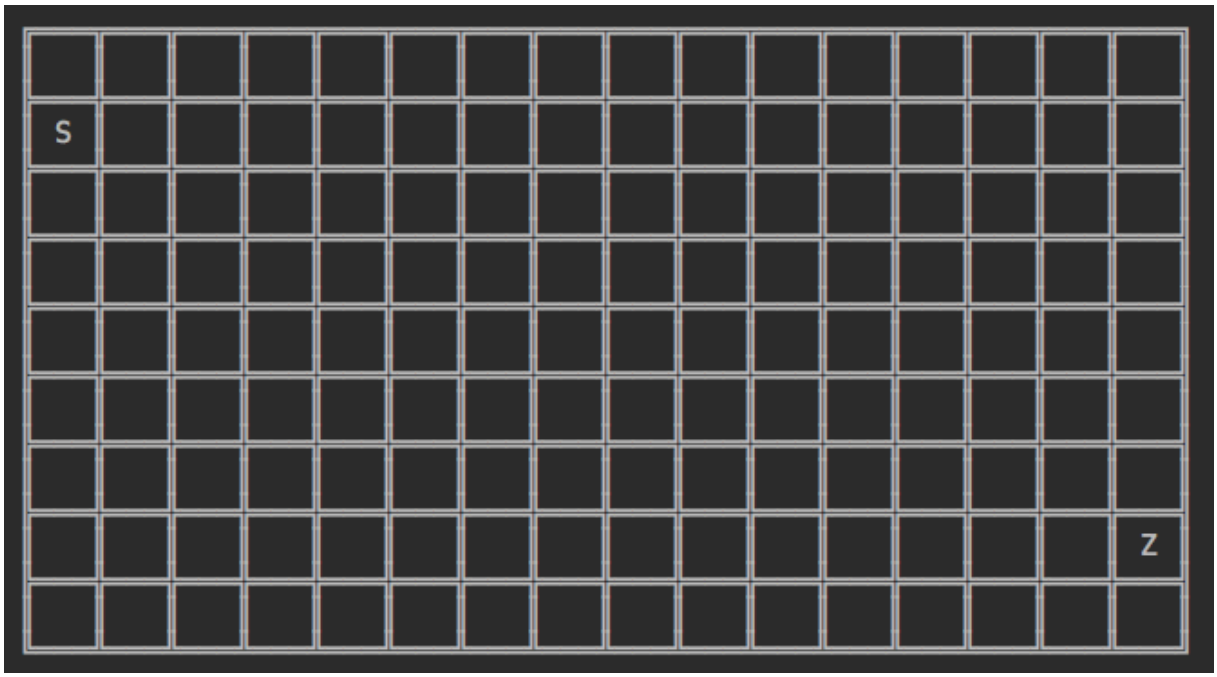

Abbildung 1: Position von S und Z in einem 16 mal 9 Raum

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

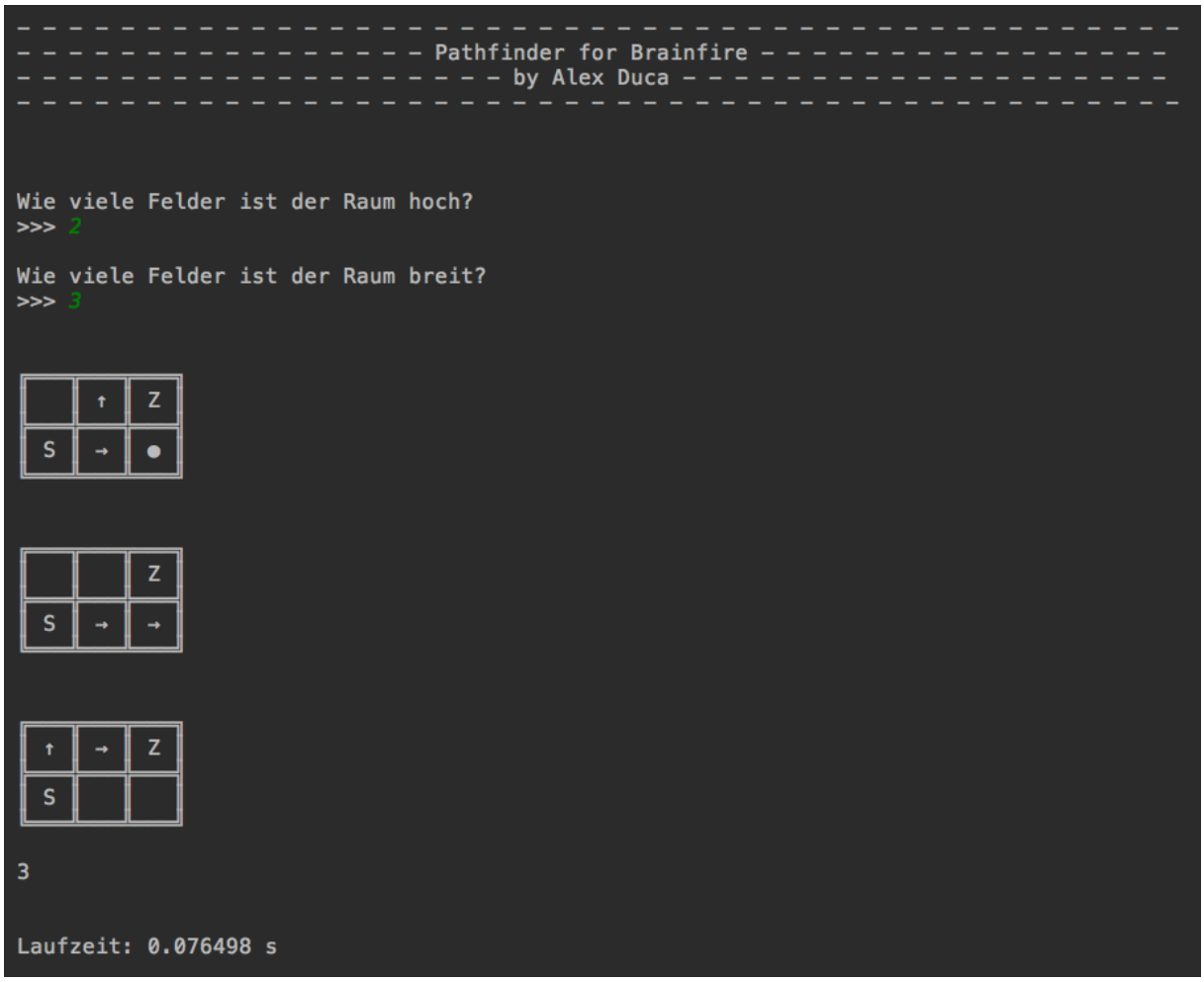

Abbildung 3: Menge aller Lösungen in einem 2x3-Raum

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
```

2 Spray 'n' Pray

2.1 Grundidee

Im Vordergrund steht die Entwicklung eines sehr effizienten Algorithmus, der in kürzester Zeit bestimmt, von welcher Tür welche andere Tür erreicht werden kann. Das restliche Verfahren wird in zwei Teilschritte unterteilt:

- **Spray**: Es werden zufällig Steine auf dem Boden verteilt. Die Wahrscheinlichkeit, dass auf einer Fläche ein Stein liegt.
- **Pray**: Berechne mit dem Algorithmus die existenten Pfade und hoffe, dass der Raum brauchbar ist.

Auf diese Weise wird eine Riesenzahl N an Räumen erstellt und auf ihre Eigenschaften überprüft. Das daraus entstandene Repertoire an Räumen bildet die Auswahl für die Erstellung des Dungeons.

2.2 Umsetzung

Als Lösungsalgorithmus wird die Breitensuche verwendet, die über die Laufzeitkomplexität $O(n^2)$ verfügt, wobei die Problemgröße n die Anzahl der Knoten in dem Suchbaum beschreibt. Da die Knoten durch die Felder des zu untersuchenden Raums bestimmt sind, hat der gesamte Lösungsalgorithmus eine Laufzeit von $O((a \cdot b)^2)$, wobei a, b die Seitenlängen des untersuchten Raums beschreiben. Bei einem quadratischen Raum der Seitenlänge n entspricht das Laufzeitverhalten also $O(n^4)$. Im direkten Vergleich zum Backtracking-Algorithmus mit $O(2^n)$ ist der Breitensuche-Algorithmus ein großer Erfolg.

Mit Hilfe von Multiprocessing, ein Verfahren, das alle Computer-Kerne gleichzeitig belastet, lässt sich der Algorithmus zusätzlich um einen Linearfaktor beschleunigen.

2.3 Wahl der Parameter

Eine noch zu beantwortende Frage ist die Wahl des Parameters, der beschreibt, wie wahrscheinlich es ist, dass auf einem Feld ein Stein liegt: P(Stein). Mit Hilfe einer empirischen Untersuchung, die durch die folgende Tabelle festgehalten wurde, wird eine Antwort auf dieses Problem gegeben:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0%	10000	0	0	0	0	0	0	0	0	0	0	0	0
10%	1334	656	364	2886	702	220	2356	191	19	1050	0	0	222
20%	879	460	788	2004	935	442	2412	358	21	1391	0	0	310
30%	1962	1790	1898	1259	1086	368	902	258	11	391	0	0	75
40%	5901	2326	987	324	274	52	83	26	4	21	0	0	2
50%	9134	692	138	21	13	0	2	0	0	0	0	0	0
60%	9891	82	25	1	1	0	0	0	0	0	0	0	0
70%	9999	1	0	0	0	0	0	0	0	0	0	0	0
80%	10000	0	0	0	0	0	0	0	0	0	0	0	0
90%	10000	0	0	0	0	0	0	0	0	0	0	0	0
100%	10000	0	0	0	0	0	0	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9	10	11	12
0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
10%	13%	7%	4%	29%	7%	2%	24%	2%	0%	11%	0%	0%	2%
20%	9%	5%	8%	20%	9%	4%	24%	4%	0%	14%	0%	0%	3%
30%	20%	18%	19%	13%	11%	4%	9%	3%	0%	4%	0%	0%	1%
40%	59%	23%	10%	3%	3%	1%	1%	0%	0%	0%	0%	0%	0%
50%	91%	7%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
60%	99%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
70%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
80%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
90%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
100%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

Abbildung 4: Betrachtet wird ein 16x16-Feld. Die y-Achse beschreibt P(Stein), die x-Achse beschreibt die Anzahl der Tür-Tupel (A,B), für die gilt, dass von Tür A aus Tür B erreichbar ist.

Einen etwas detaillierteren Eindruck gibt die folgende Tabelle, die den Bereich P(Stein) $\in [12\%, 22\%]$ betrachtet:

	0	1	2	3	4	5	6	7	8	9	10	11	12
12%	1070	459	387	2697	707	271	2535	266	19	1329	0	0	260
13%	994	427	434	2518	674	271	2817	297	14	1297	0	0	257
14%	940	365	438	2492	673	301	2673	301	17	1461	0	0	339
15%	860	389	472	2304	674	330	2802	296	17	1488	0	0	368
16%	854	346	528	2279	729	334	2656	313	17	1575	0	0	369
17%	875	379	532	2219	752	379	2649	332	21	1524	0	0	338
18%	850	341	645	2218	803	395	2560	361	11	1477	0	0	339
19%	862	364	715	2118	831	402	2530	352	24	1491	0	0	311
20%	879	460	788	2004	935	442	2412	358	21	1391	0	0	310
21%	881	501	951	1939	1001	456	2239	385	23	1342	0	0	282
22%	949	523	992	1851	1040	476	2187	424	25	1230	0	0	303

	0	1	2	3	4	5	6	7	8	9	10	11	12
12%	11%	5%	4%	27%	7%	3%	25%	3%	0%	13%	0%	0%	3%
13%	10%	4%	4%	25%	7%	3%	28%	3%	0%	13%	0%	0%	3%
14%	9%	4%	4%	25%	7%	3%	27%	3%	0%	15%	0%	0%	3%
15%	9%	4%	5%	23%	7%	3%	28%	3%	0%	15%	0%	0%	4%
16%	9%	3%	5%	23%	7%	3%	27%	3%	0%	16%	0%	0%	4%
17%	9%	4%	5%	22%	8%	4%	26%	3%	0%	15%	0%	0%	3%
18%	9%	3%	6%	22%	8%	4%	26%	4%	0%	15%	0%	0%	3%
19%	9%	4%	7%	21%	8%	4%	25%	4%	0%	15%	0%	0%	3%
20%	9%	5%	8%	20%	9%	4%	24%	4%	0%	14%	0%	0%	3%
21%	9%	5%	10%	19%	10%	5%	22%	4%	0%	13%	0%	0%	3%
22%	9%	5%	10%	19%	10%	5%	22%	4%	0%	12%	0%	0%	3%

Abbildung 5: Betrachtet wird ein 16x16-Feld. Die y-Achse beschreibt P(Stein), die x-Achse beschreibt die Anzahl der Tür-Tupel (A,B), für die gilt, dass von Tür A aus Tür B erreichbar ist.

Auf Grund der Homogenität dieses kleineren Suchbereichs lassen sich keine signifikante Aussagen über die ideale Wahl des Parameters P(Stein) machen. Der Durchschnittswert 18% ist gut geeignet, um die Wahrscheinlichkeit der Sackgassen-Räume ($x = 0$) möglichst zu minimieren.

3 Gestaltung der Hintergrundmusik

3.1 Inspirationsquellen

Da die Spielmechanik die Fortbewegung auf einer Eisfläche ist, soll Inspiration von Videospielmusikstücken gezogen werden, die ein Kälte-Motiv haben. Im Folgendem findet sich eine solche Auswahl.

3.1.1 The Legend of Zelda: Ocarina of Time

"Ice Cavern" (bit.ly/2L22Kf2) ist eine Eishöhle, in der Link unter anderem riesige Blöcke über einem Eisboden rutschen lässt, um eine Treppe zu seinem Ziel zu bauen. (bit.ly/2LbnzDH)

3.1.2 The Legend of Zelda: Majora's Mask

"Snowhead Temple" (bit.ly/2Uc8sOl) ist ein hoher zugefrorener Turm, zu dessen Spitze Link navigieren muss. In der Mitte des Turms findet sich eine Säule, dessen Höhe anpasst werden muss, indem von Goronen-Link vereinzelte Eisschichten aus der Säule raus geschlagen werden. (bit.ly/2Zx7Xj5)

3.1.3 Pokemon HeartGold/SoulSilver

"Ice Path" (bit.ly/2HwBQtH) ist eine Eishöhle, die mit jener Spielmechanik arbeitet, von der dieses ganze Projekt inspiriert wurde. (bit.ly/30EPnqp)

3.1.4 Super Mario Galaxy

"Ice Mountain" (bit.ly/348siys) ist ein Musikstück, das in der "Freezeflame Galaxy" gespielt wird. Das Spiel führt zu diesem Zeitpunkt eine Bewegungsmechanik ein, mit der Mario über Eis Schlittschuh fahren kann. Auch findet sich in dieser Welt das Power-Up "Iceflower", die Mario für eine begrenzte Zeit die Fähigkeit gibt, Wasseroberflächen einzufrieren, um zusammen mit der Bewegungsmechanik zum Ziel zu navigieren. (bit.ly/2zt42sN)