

Brainfire: L^AT_EX-Dokumentation

BANDO

28. August 2019

Inhaltsverzeichnis

1	Pathfinder-Generierung	1
1.1	Grundidee	1
1.2	Umsetzbarkeit	1
1.3	Beispiel	3
1.4	Quellcodeausschnitt	3
2	Gestaltung des Hintergrundmusik	6
2.1	Inspirationsquellen	6
2.1.1	The Legend of Zelda: Ocarina of Time	6
2.1.2	The Legend of Zelda: Majora's Mask	6
2.1.3	Pokemon HeartGold/SoulSilver	6
2.1.4	Super Mario Galaxy	6

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.

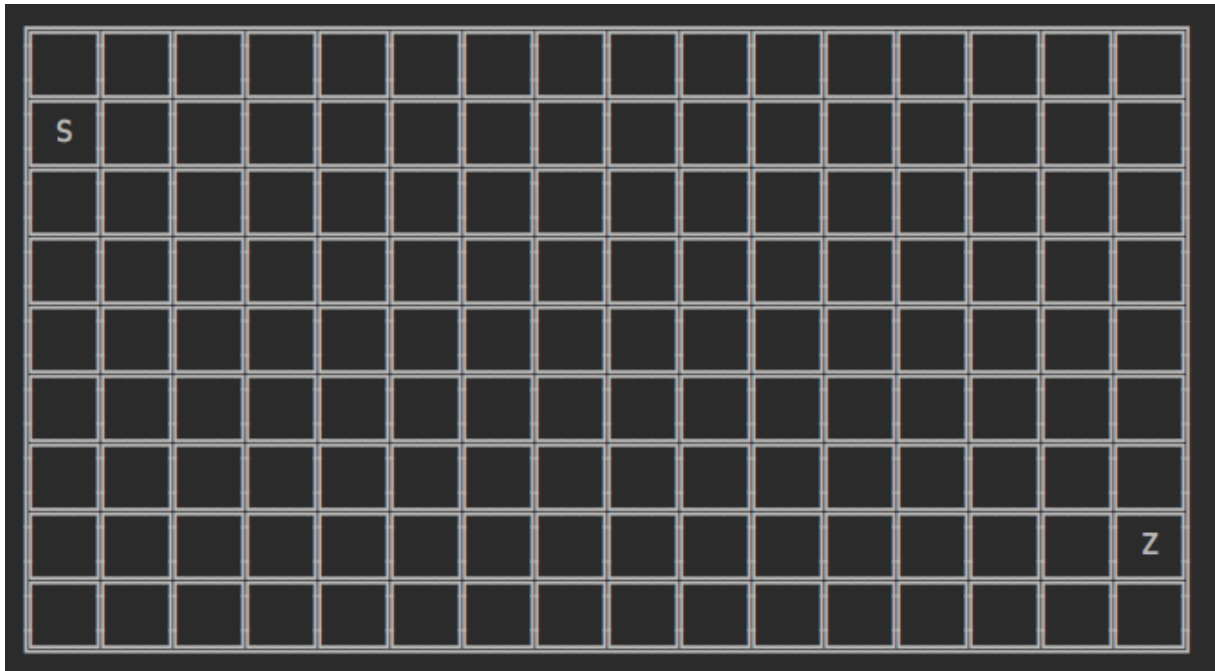


Abbildung 1: Position von S und Z in einem 16 mal 9 Raum

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	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

```

-----
----- Pathfinder for Brainfire -----
----- by Alex Duca -----
-----

Wie viele Felder ist der Raum hoch?
>>> 2

Wie viele Felder ist der Raum breit?
>>> 3



|   |   |   |
|---|---|---|
|   | ↑ | Z |
| S | → | ● |



|   |   |   |
|---|---|---|
|   |   | Z |
| S | → | → |



|   |   |   |
|---|---|---|
| ↑ | → | Z |
| S |   |   |



3

Laufzeit: 0.076498 s

```

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

```

```

23     neuePos = ( altePos[0] + richtung[0], altePos[1] + richtung[1] )
24
25     # Wenn du gegen eine Wand stoest
26     if not inRange(neuePos):
27         # Wenn diese Wand nicht gleich am Anfang ist, dann backtracke
28         if altePos != startPos:
29             backtracking(Matrix, altePos, richtung)
30             # Wir sind dann auch fertig fuer die Richtung
31             break
32
33     # Lege einen Pfeil auf dem Boden in aktuelle Laufrichtung
34     Matrix[neuePos[0]][neuePos[1]] = pfeil(richtung)
35
36     # Versuche einen Stein vor die neue Position zu legen und starte Backtracking
37     stein = ( neuePos[0] + richtung[0], neuePos[1] + richtung[1] )
38     if inRange(stein):
39
40         Matrix[ stein[0] ][ stein[1] ] = "0"
41         backtracking(Matrix, neuePos, richtung)
42
43         # Nach erfolgreichem Steinelegen, nimm ihn wieder weg und lauf weiter in die Richtung
44         Matrix[ stein[0] ][ stein[1] ] = "┐"
45
46
47     for Raum in alleRaeume:
48         printMatrixRaum(Raum)
49     print(len(alleRaeume))

```

Methode, die beim Backtracking rekursiv aufgerufen wird.

```

1     def backtracking(Matrix, eigenePos, letzteRichtung):
2
3         # Globale Informationen
4         global alleRaeume
5         backupKopieMatrix = deepcopy(Matrix)
6
7         # Sind wir schon am Ziel?
8         if eigenePos == endPos:
9             backupKopieMatrix[endPos[0]][endPos[1]] = "Z"
10             alleRaeume.append(backupKopieMatrix)
11
12         else:
13             # lauf nur links oder rechts
14             Senkrechten = senkrechte(letzteRichtung)
15             for richtung in Senkrechten:
16
17                 neuePos = eigenePos
18                 Matrix = deepcopy(backupKopieMatrix)
19
20                 while True:
21
22                     altePos = neuePos
23
24                     # Probiere einen Schritt nach vorne
25                     neuePos = (altePos[0] + richtung[0], altePos[1] + richtung[1])
26
27                     # Ist da eine Wand?
28                     if not inRange(neuePos):
29                         # Wir koennen hier aufhoeren
30                         break
31
32                     # Liegt da ein Stein?
33                     elif Matrix[ neuePos[0] ][ neuePos[1] ] == "0":
34                         # Wir koennen hier aufhoeren
35                         break
36
37                     # Ist der Boden NICHT mit einem Pfeil markiert?
38                     elif Matrix[ neuePos[0] ][ neuePos[1] ] == "┐":
39
40                         # 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
47 # Ist dort, wo der Stein liegen sollte, schon eine Wand?
    if not inRange(stein):
49         # Wenn Ja, dann kann man von der Wand aus abbiegen
            backtracking(Matrix, neuePos, richtung)

51
53 # Wenn Nein, liegt dort schon ein Stein?
    elif Matrix[ stein[0] ][ stein[1] ] == "0":
55         # Wenn Ja, dann kann man von dem Stein aus abbiegen
            backtracking(Matrix, neuePos, richtung)

57
59 # Wenn Nein, ist die Flaeche frei fuer ein Stein?
    elif Matrix[ stein[0] ][ stein[1] ] == "□":
61         # Platziere einen Stein und biege ab!
            Matrix[stein[0]][stein[1]] = "0"
            backtracking(Matrix, neuePos, richtung)
63         # Und lege ihn danach wieder weg
            Matrix[stein[0]][stein[1]] = "□"

```

2 Gestaltung des Hintergrundmusik

2.1 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.

2.1.1 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)

2.1.2 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 angepasst werden muss, indem von Goronen-Link vereinzelt Eisschichten aus der Säule raus geschlagen werden. (bit.ly/2Zx7Xj5)

2.1.3 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)

2.1.4 Super Mario Galaxy

“Ice Mountain“ (bit.ly/348siys) ist ein Musikstück, das in der “Freeze Flame 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)