**Erstellung von Labyrinthen, Analyse von Labyrinthenstruktur**

Von Maria Lyoteva und Tsvetan Stanchev

**Ziel des Projekts:**  
Das Hauptziel unseres Projekts bestand darin, verschiedene Algorithmen zur Erzeugung von Labyrinthen zu implementieren und die Labyrinthe, die mit diesen Algorithmen erstellt werden, zu analysieren. Wir haben acht Algorithmen implementiert und eine einfache GUI (Graphical User Interface) erstellt.

**Aufbau des Projekts:**Unser Projekt besteht hauptsächlich aus drei Klassen: „cell“, „grid“ und „visualization“. Objekte der „cell“-Klasse stellen einzelne Zellen (Knoten) im Labyrinth dar. Für jede Zelle speichern wir Informationen über ihre Nachbarn und über alle Zellen, mit denen diese Zelle in Verbindung steht. In der „grid“-Klassefüllen wir eine zweidimensionale Liste mit „cell“-Objekten. Wir hinfügen dann die Nachbarn jeder Zelle und erhalten wir so ein „Labyrinth“, in dem es keine Verbindungen gibt (voller Mauern). Die acht labyrintherzeugende Algorithmen sind Funktionen in dieser „grid“-Klasse. Jede dieser Funktionen fügt Verbindungen zwischen den Zellen hinzu und dann gibt uns das fertige Labyrinth zurück. In dieser Klasse steht auch die Funktion zum Lösen der Algorithmen (mit Tiefensuche). Die ganze GUI zur Visualisierung und Animation der Aufbau und Lösen der Labyrinthe steht in der „visualization“-Klasse.

**Arbeitsverteilung**Wir haben die Arbeit zwischen uns im Voraus aufgeteilt. Maria Lyoteva hat die “cell” und “grid“ Klassen erstellt und auch vier der Algorithmen (Aldous-Broder, Prim's, Wilson's und DFS) implementiert. Tsvetan Stanchev hat die „visualization“ Klasse gemacht, die anderen vier Algorithmen implementiert (Binary Tree, Sidewinder, Kruskal's, Hunt and kill) und auch die Funktion zum Lösen implementiert. Wir haben gemeinsam den Bericht und die Präsentation gemacht.

**Algorithmen**Wir haben acht Algorithmen zur Erzeugung von Labyrinthen implementiert und jedes baut ein „perfektes“ Labyrinth. Das bedeutet, dass diesen Labyrinthen für je zwei ihrer Zellen genau eine Lösung haben. Jetzt schauen wir genauer jedes Algorithmus an:

1. **Aldous-Broder Algorithmus**Er wird von David Aldous und Andrei Broder entwickelt. Mit diesem Algorithmus können wir ein Labyrinth generieren, der viele kürze Wege (deadends) hat, aber mäßig schwierig zum Lösen ist. Der größte Nachteil dieses Algorithmus ist seine zufällige Zeitkomplexität. Er funktioniert so:  
   1. Wähle eine Startzelle vom Grid und markiere sie als „aktuell“  
   2. Wähle einen zufälligen Nachbarn von „aktuell“  
   3. Wenn dieser Nachbar noch nicht besucht wurde, verbinde die aktuelle Zelle mit dem Nachbarn  
   4. Wenn noch nicht alle Zellen besucht wurden, setze „aktuell“ = Neighbor und gehe zurück zu 2.
2. **Binary Tree Algorithmus**Er wurde 1960 von mehreren Forschern unabhängig voneinander entdeckt.Dieser Algorithmus ist sehr zeiteffizient. Er hat lineare Aufwand, seine Zeitkomplexität ist O(c), wo „c“ die Anzahl der Zelle im Labyrinth ist. Sein größter Nachteil ist, dass das erzeugte Labyrinth relativ leicht zu lösen ist (Lösungspfad oft neben Diagonal). Um dieser Algorithmus zu implementieren, müssen wir zwei Richtungen wählen (oben, unten, links, rechts). In unserer Implementierung haben wir „oben“ und „links“ gewählt. Er funktioniert so:  
   Wir iterieren durch alle Zelle des Grids in Ordnung der Indizes (00, 01, 02, 03, ...)  
   1. Wenn die aktuelle Zelle Nachbarn in beide Richtungen hat, wählen wir zufällig eine  
   2. Wir verbinden die aktuelle Zelle und den Nachbarn in der gewählten Richtung und dann gehen wir in die nächste Zelle
3. **Sidewinder Algorithmus**Dieser Algorithmus wurde 1982 von Richard Korf entwickelt. Er hat viel Gemeinsames mit dem Binary Tree Algorithmus. Er hat auch Zeitkomplexität O(c) und wir müssen wieder zwei Richtungen für die Generation des Labyrinths wählen. Der Hauptunterschied besteht darin, dass das generierte Labyrinth ein bisschen schwieriger zu lösen ist. In unserer Implementierung haben wir „unten“ und

„rechts“ gewählt:  
 Wir iterieren durch alle Zelle des Grids in Ordnung der Indizes (00, 01, 02, 03, ...)  
!Wir werden eine Liste „run“ machen  
1. Append die aktuelle Zelle, wenn wir auf der ersten Zelle der Zeile befinden: ausleere die „run“-Liste  
2. Wähle zufällig eine der beiden Richtungen  
3. Wenn „rechts“ gewählt wird, dann verbinde die aktuelle Zelle mit seinem rechten Nachbar, append aktuelle Zelle zu „run“-Liste, gehe in die nächste Zelle und mache wieder 1.  
4. Wenn „oben“-gewählt, dann wähle zufällige Zelle von der „run“-Liste und verbinde sie mit ihrem Nachbar nach oben, ausleere die Liste und gehe zurück nach 1.

1. **Kruskal’s Algorithmus**Dieser Algorithmus ist nach sein Erfinder Joseph Kruskal gennant. Das generierte Labyrinth ist sehr ähnlich wie bei Aldous-Broder (viele kurze Wege). Er hat aber quasi-lineare Zeitkomplexität (O(log(c)\*c)). Dass heißt ein bisschen zeituneffizienter als die letzten zwei Algorithmen aber zeiteffizienter als Aldous-Broder. Dieser Algorithmus funktioniert ein bisschen verschieden:   
   ! Wir definieren eine Liste mit allen Kanten: „pairs“  
   ! Wir erstellen eine 2-d Liste „sets“, wo in dem Beginn alle Zelle in ihre eigene Sublisten sind: sets = [[cell 00], [cell 01], [cell 02], ...]  
   1. Wähle zufälliges Paar aus „pairs“ und wenn die Zelle un verschiedene Sublisten sind, dann verbinde die Zellen und merge ihre Sublisten  
   2. Wenn „pairs“ mehr als eine Subliste hat gehe zurück zu 1.
2. **Prim’s Algorithmus**Dieser Algorithmus wurde 1930 von dem tschechischen Mathematiker Jarník entwickelt und später von den Informatikern Robert Prim 1957 und Edsger Dijkstra 1959 wiederentdeckt.   
   1. Wir wählen einen beliebigen Knoten aus G (dem Graphen) und fügen ihn zu einer (anfangs leeren) Menge V hinzu.  
   2. Wir wählen die Kante mit dem kleinsten Gewicht aus G, die einen Knoten in V mit einem anderen Knoten nicht in V verbindet. (bei uns das Gewicht ist eine Zufallszahl von eins bis 100)  
   3. Fügen Sie diese Kante zum „minimal spanning tree“ und den anderen Knoten der Kante zu V hinzu.  
   4. Wiederholen Sie die Schritte 2 und 3, bis V jeden Knoten in G enthält.
3. **Wilson’s Algorithmos**Er wurde 1996 von Robin Wilson (britischer Mathematiker und Informatiker) entwickelt. Дas Labyrinth, das dieser Algorithmus erzeugt, sieht ungefähr so aus wie das, das der Aldous-Broder Algorithmus generiert. Er hat aber eine unbefriedige Zeitkomplexität O(c^2). Er funktioniert so:  
   ! definiere eine set „visited“  
   1. Wähle eine zufällige Zelle des Grids und addiere sie zu „visited“  
   2. Wähle eine andere zufällige Zelle des Grids, die nicht in besucht ist, „aktuell“ = diese

Zelle, erstelle eine Pfadliste und füge diese Zelle hinzu, wenn solche Zelle nicht existiert beende den Algorithmus  
3. Wähle einen zufälligen Nachbar von „aktuell“, wenn er nicht in „visited“ steht: append ihn zu Pfadliste, „aktuell“ = diesen Nachbarn, gehe nach 3., wenn er in „visited“ steht, geh nach 4.  
4. verbinde alle Zellen im Pfadliste miteinander, verbinde das letzte Element der Pfadliste mit „aktuell“, markiere alle Zellen aus Pfadiste als „visited“, ausleere die Pfadliste und gehe zurück nach 2.

1. **Hunt and kill Algorithmus**Dieser Algorithmus generiert ein relativ schwerzulösendes Labyrinth mit vielen langen Wegen. Er ist auch sehr zeiteffektiv, mit linearem Aufwand (O(c)). Er funktioniert so:  
   ! definiere set „visited“  
   1. Wähle eine Startzelle, „aktuell“ = diese Zelle, hinfüge „aktuell“ zu „visited“.  
   2. Wenn „aktuell“ noch Nachbarn hat, die nicht in „visited“ stehen, „aktuell“ = wähle zufällig einen dieser Nachbarn und wiederhole 2. Wenn alle in „visited“ stehen, gehe nach 3.  
   3. Iterieren wir durch das Grid (in Ordnung der Indizes) bis wir eine Zelle finden, die nicht in „visited“ steht und auch mindestend einen Nachbar har, der schon besucht ist, „current“ = diese Zelle, gehe nach 2. zurück, wenn es keine solche Zelle existiert, beende den Algorithmus
2. **Tiefensuche (Depth First Search)**Diesen Algorithmus haben wir bereits in den Vorlesungen besprochen. Er erzeugt nicht nur das komplexeste Labyrinth, aber ist auch zeiteffizient (O(c), linearer Aufwand). Er funktioniert so:  
   ! definiere set „visited“  
   1. „Aktuell“ = Wähle eine Startzelle  
   2. Wenn „aktuell“ noch Nachbaren hat, die nicht in „visited“ stehen, setze   
   „aktuell“ = ein zufälliger solcher Nachbar und wiederhole 2., wenn solcher Nachbar nicht existiert, gehe zu 3.  
   3. Finde die letzte Zelle den Nachbar(e) hat, die noch nicht besucht sind. Setze   
   „aktuell“ = diese Zelle. Wenn es keine solche Zelle existiert, beende den Algorithmus.

**Vergleich der Algorithmen**Zum Schluss werden wir die acht Algorithmen bei Zeitkomplexität und bei Komplexität des erstelllendes Labyrinths:  
Zeitkomplexität: DFS = Hunt and kill = Binary Tree = Sidewinder (=O(c)) < Kruskal’s = Prim’s (=O(c\*log(c))) < Wilson’s (=O(c^2)) (! Aldous-Broder hat zufällige Zeitkomplexität)  
Komplexität des Labyrinths: Binary Tree < Sidewinder < Prim’s < Kruskals = Aldous Broder = Wilson’s < Hunt and kill < DFS

**Quellen:**

[**https://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm**](https://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm)

[**https://weblog.jamisbuck.org/2011/2/1/maze-generation-binary-tree-algorithm**](https://weblog.jamisbuck.org/2011/2/1/maze-generation-binary-tree-algorithm)

[**https://weblog.jamisbuck.org/2011/1/24/maze-generation-hunt-and-kill-algorithm**](https://weblog.jamisbuck.org/2011/1/24/maze-generation-hunt-and-kill-algorithm)

[**https://weblog.jamisbuck.org/2011/2/3/maze-generation-sidewinder-algorithm.html**](https://weblog.jamisbuck.org/2011/2/3/maze-generation-sidewinder-algorithm.html)

[**https://weblog.jamisbuck.org/2011/1/20/maze-generation-wilson-s-algorithm**](https://weblog.jamisbuck.org/2011/1/20/maze-generation-wilson-s-algorithm)

[**https://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm**](https://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm)

[**https://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm**](https://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm)

[**https://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking**](https://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking) [**https://en.wikipedia.org/wiki/Maze\_generation\_algorithm**](https://en.wikipedia.org/wiki/Maze_generation_algorithm)

[**https://en.wikipedia.org/wiki/Maze-solving\_algorithm**](https://en.wikipedia.org/wiki/Maze-solving_algorithm)

[**https://www.pygame.org/news**](https://www.pygame.org/news)

[**https://www.youtube.com/watch?v=U3meEXvYFsc**](https://www.youtube.com/watch?v=U3meEXvYFsc)

**Mazes.pdf aus unserem Moodle-Kurs.**