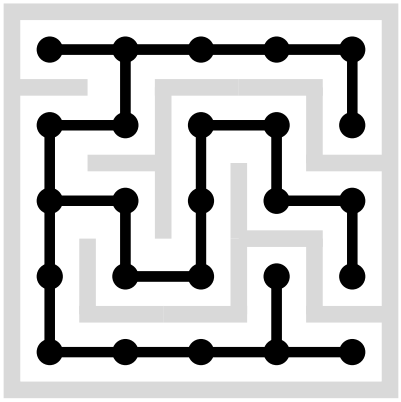
Technische Universität Berlin **SoSe 2024**
Fakultät II, Institut für Mathematik
Sekretariat MA 6–2, Dorothea Kiefer-Hoeft
Prof. Dr. Max Klimm
Dr. Tobias Hofmann, Martin Knaack, Marcel Wack

Programmierprojekt Computerorientierte Mathematik II

Abgabe: Durch Vorstellung bis zum 12.07.2024

Hinweis: Das Programmierprojekt wird in Dreiergruppen bearbeitet und bei einem Tutor oder Tutorin vorgestellt. Termine für die Vorstellung werden über ISIS bekannt gegeben.



Ein Labyrinth als Spannbaum eines 5x5 Gitters

Maze Generation

Einführung

Labyrinthe können mathematisch als Graphen aufgefasst werden. Als Knotenmenge dient das Gitter der durch Wände getrennten Zellen. Die Kantenmenge besteht aus den Nachbarschaftsbeziehungen der Zellen. D. h. zwei Zellen sind benachbart, wenn man von einer Zelle zur anderen gelangen kann, ohne eine Wand zu durchqueren. (Einfache) Labyrinthe sind zusammenhängende Graphen, d.h. es gibt einen Weg von jeder Zelle zu jeder anderen Zelle. Außerdem sind sie zyklenfrei. Labyrinthe sind also Bäume und können mit Hilfe von Algorithmen zur Erzeugung von minimalen Spannbäumen erzeugt werden. Auch die Lösung eines Labyrinths ist durch diese Charakterisierung einfach. Es muss nur ein Weg von einem Startknoten zu einem Zielknoten gefunden werden.

Übersicht

In diesem Programmierprojekt sollen beliebige Strukturen und Funktionen zur Erzeugung, Visualisierung und Lösung von Labyrinthen in Julia implementiert werden. Dazu schreiben Sie eine randomisierte Variante der Tiefensuche. Außerdem sollen Sie die rechte Hand Regel zur Erstellung eines Lösungsweges in Julia realisieren, der Lösungsweg wird dann zusammen mit dem Labyrinth im Terminal visualisiert.

Task 0: Erstellen Sie ein `Julia`-Projekt

Um zu beginnen, erstellen Sie ein neues `Julia`-Projekt mit dem Namen `MazeGeneration`. Der Projektorder sollte die folgende Struktur haben:

Directory Structure

```
julia> run(`tree`);
.
├── Manifest.toml
├── Project.toml
└── src
    ├── core.jl
    ├── MazeGeneration.jl
    ├── solver.jl
    └── visualize.jl
```

Die Dateien in `src` müssen nicht mit dem Beispiel übereinstimmen. Nur die Datei `MazeGeneration.jl` muss vorhanden sein, um alle Funktionen und `structs` zu exportieren, die in dieser Aufgabe benötigt werden.

Task 1: Das `struct` Node

Zunächst benötigen Sie eine Datenstruktur, um die Knoten des Graphen darzustellen. Dazu erstellen Sie eine `struct` Node, die genaue Struktur können Sie selbst wählen. Sie müssen nur darauf achten, dass die Knoten dieses Graphen die Positionen der Zellen des Gitters repräsentieren und **nicht** die Ecken des dazu dualen Graphen der Wände. Zusätzlich sollte eine Funktion `neighbors` implementiert werden, die die Nachbarn eines Knotens zurückgibt. Die Funktion sollte (mindestens) für die Signatur `neighbors(node::Node)` definiert werden.

> **Hinweis:** Ich empfehle, die `struct` in einer separaten Datei zu definieren und diese Datei nur in `src/MazeGeneration.jl` zu importieren.

Task 2: Das `struct` Maze

Nachdem Sie nun eine Möglichkeit haben, die Knoten des Graphen darzustellen, können Sie die Kernstruktur des Labyrinths erstellen. Diese dient als Container für alle Informationen, die zur Generierung und Visualisierung des Labyrinths benötigt werden. Erstellen Sie eine `struct` Maze mit den folgenden Feldern:

`nodes::Matrix{Node}`: Die Knoten der Graphendarstellung abgespeichert in einem Array.

`visual::Union{MazeViz, Nothing}`: Die Visualisierung des Graphen.

`path::Union{Vector{Node}, Nothing}`: Die Lösung des Labyrinths.

Hinweis: Überladen Sie Funktionen wie `Base.show` oder auch `Base.getindex` und `Base.setindex!` wenn Sie dies benötigen. Auch eine implementierung eines (inneren) Konstruktors `Maze(height::Int, width::Int)` könnte für weitere Schritte sinnvoll sein.

Task 3: Erzeugung eines randomizierten Spannbaums

Schreiben Sie einen (äußeren) Konstruktor `maze(height::Int, width::Int)` für das `struct` Maze, der ein Labyrinth der Größe `height` x `width` erzeugt. Ihr Algorithmus sollte eine randomisierte Version der Tiefensuche implementieren, um einen Spannbaum des zugrundeliegenden Gitters zu erzeugen. Randomisiert bedeutet hier, dass sowohl der Wurzelknoten als auch die Reihenfolge der Nachbarn zufällig gewählt werden.

Konkret soll der Algorithmus wie folgt funktionieren:

Sie starten an einem zufälligen Knoten und markieren ihn als besucht. Dann wählen Sie zufällig einen Nachbarn, der noch nicht besucht wurde, und entfernen die Wand zwischen den beiden Zellen. Diesen Nachbarn markieren Sie als besucht und fügen ihn zur Liste der besuchten Knoten hinzu. Diesen Schritt wiederholen Sie, bis alle Knoten besucht wurden. Befinden Sie sich in einem Knoten, der keine unbesuchten Nachbarn hat, gehen Sie zurück zum letzten Knoten, der unbesuchte Nachbarn hat. Von dem neuen Knoten aus wiederholen Sie den Schritt, bis alle Knoten besucht wurden.

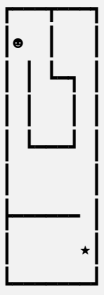
Nachdem Sie ein Labyrinth erstellt haben, können Sie einen zufälligen oder beliebigen Start- und Zielknoten wählen.

Task 4: Visualisierung des Labyrinths

Nachdem Sie nun ein Labyrinth generiert haben, soll es visualisiert werden. Dabei ist es Ihnen überlassen, wie Sie das Labyrinth visualisieren. Auf jeden Fall sollten Sie eine `struct MazeViz` erstellen, die alle Logik und Daten für die Visualisierung enthält. Sie können auch externe Bibliotheken verwenden, um die Visualisierung zu erleichtern. Wenn Sie das Labyrinth ohne externe Bibliotheken visualisieren wollen, können Sie diese auch im Terminal ausgeben. Wenn Sie eine visualisierung implementiert haben, überladen Sie die Funktion `Base.show` für das `struct Maze`, so dass die Visualisierung des Labyrinths ausgegeben wird.

Naive Visualisierung mit Start und Ende

```
julia> maze(4,4)
```



Task 5: Lösung des Labyrinths

Schließlich implementieren Sie eine Funktion `solve(maze::Maze)`, die den Pfad von einem Startknoten zu einem Zielknoten im Labyrinth zurückgibt. Die Funktion sollte den kürzesten Pfad als `Vector{Node}` zurückgeben und im Feld `path` von `maze` speichern. Der Startknoten sollte der erste und der Zielknoten der letzte Eintrag des Vektors sein. Ihre Funktion sollte den Pfad mithilfe der rechten Hand Regel finden. Das bedeutet, dass sie sich immer an der Wand auf der rechten Seite orientiert und nur lokale Informationen über die Nachbarn der aktuellen Zelle verwendet.
Konkreter bedeutet die Regel der rechten Hand, dass sie in eine beliebige Richtung startet und dann zu jedem Zeitpunkt überprüft, ob sich auf ihrer rechten Seite eine Wand befindet. Das heißt, wenn die Wand auf der rechten Seite abzweigt, folgen sie ihr, wenn nicht, gehen sie geradeaus. Wenn sie nicht geradeaus gehen können, drehen sie sich um 90 Grad nach links. Offensichtlich bleibt die Wand immer auf der rechten Seite. Abschließend visualisieren Sie den Lösungsweg zusammen mit dem Labyrinth in der `Base.show` Funktion.

Für die Vorstellung des Projekts sollten Sie folgende Punkte beachten:

- Alle Tasks sollten ausreichend bearbeitet sein.
- Ihr Code sollte gut strukturiert und **kommentiert** sein (vgl. dazu die Übung vom 19.06.2024).
- Sie sollten in der Lage sein, Ihren Code zu erklären und Fragen dazu zu beantworten.
- Schreiben Sie Tests für Ihre Funktionen, um die Funktionalität während der Vorstellung zu demonstrieren.