

BTU Cottbus-Senftenberg

Fachbereich Drahtlose Systeme

Intelligente Pfadsuche

Vergleichende Simulation von Suchverfahren in generierten Maze-Umgebungen

Endprojekt im Rahmen der Vorlesung
Angewandte Modellierung und Systemsimulation

Autor:	Ole Matzky
Matrikelnummer:	5005801
Studiengang:	Künstliche Intelligenz
Semester:	4. Semester
Betreuer:	Dr. Svetlana Meissner
Abgabedatum:	22. Juli 2025

2. Juli 2025

Inhaltsverzeichnis

1	Einleitung	3
1.1	Projektziele	3
2	Maze-Generierung	3
2.1	Randomisierte Tiefensuche	3
2.2	Ungerade Dimensionen und 2er-Schritte	4
2.2.1	Ungerade Dimensionen	4
2.2.2	2er-Schritte	4
2.3	Datenrepräsentation als NumPy-Matrix	5
2.4	Reproduzierbarkeit durch Seed-Kontrolle	5
3	Generator-Pattern und Lazy Evaluation	6
3.1	Python Generatoren	6
3.1.1	Was sind Generatoren?	6
3.1.2	Vergleich mit bekannten Iteratoren	6
3.2	Vorteile der Generator-Nutzung	6
3.2.1	Speichereffizienz	6
3.2.2	Performance-Vorteile	7
4	Grafische Benutzeroberfläche	7
4.1	Tkinter als GUI-Framework	7
4.1.1	Persönliche Erfahrung	7
4.1.2	Community und Dokumentation	7
4.1.3	Widget-Prinzip	7
4.2	Matplotlib-Integration	8
4.2.1	Warum Matplotlib für Animation?	8
4.3	Steuerelemente der Benutzeroberfläche	8
4.3.1	Maze-Parameter	8
4.3.2	Algorithmus-Auswahl	8
4.3.3	Animation-Kontrolle	9
5	Pathfinding-Algorithmen	9
5.1	A* (A-Stern) Algorithmus	9
5.1.1	Funktionsweise	9
5.1.2	Heuristik	9
5.1.3	Eigenschaften	9
5.2	Dijkstra-Algorithmus	9
5.2.1	Funktionsweise	9
5.2.2	Eigenschaften	9
5.3	Greedy Best-First Search	10
5.3.1	Funktionsweise	10
5.3.2	Eigenschaften	10
5.4	Algorithmus-Vergleich	10
5.4.1	Praktische Anwendungsbereiche	10

6	Implementierungsdetails	10
6.1	Architektur-Übersicht	10
6.2	Performance-Optimierungen	11
6.2.1	Memory Management	11
6.2.2	Animation-Optimierung	11
7	Fazit	11
7.1	Erreichte Ziele	11
7.2	Erkenntnisse	11
7.2.1	Algorithmische Erkenntnisse	11
7.2.2	Technische Erkenntnisse	11
7.3	Mögliche Erweiterungen	12
A	Quellcode	13
B	Systemanforderungen	13

1 Einleitung

Die Pfadsuche in komplexen Umgebungen ist ein fundamentales Problem der Informatik mit weitreichenden Anwendungen in der Robotik, Spieleentwicklung und Navigationssystemen. Dieses Projekt implementiert eine interaktive Visualisierung verschiedener Pathfinding-Algorithmen in zufällig generierten Labyrinthen.

Das entwickelte System ermöglicht es, drei klassische Suchalgorithmen – A*, Dijkstra und Greedy Best-First Search – in ihrer Funktionsweise zu vergleichen und deren charakteristische Eigenschaften durch animierte Visualisierungen zu verstehen.

1.1 Projektziele

- Implementierung eines Maze-Generators basierend auf randomisierter Tiefensuche
- Entwicklung einer benutzerfreundlichen grafischen Oberfläche
- Vergleichende Analyse verschiedener Pathfinding-Algorithmen
- Bereitstellung von Exportfunktionalität für Animationen

2 Maze-Generierung

2.1 Randomisierte Tiefensuche

Die Generierung der Labyrinth erfolgt mittels einer randomisierten Tiefensuche (Randomized Depth-First Search). Dieser Algorithmus erzeugt garantiert ein *perfektes Labyrinth*, das folgende Eigenschaften aufweist:

- **Zusammenhängend:** Jede freie Zelle ist von jeder anderen freien Zelle aus erreichbar
- **Azyklisch:** Es existiert genau ein Pfad zwischen zwei beliebigen Punkten
- **Minimal:** Das Labyrinth enthält keine redundanten Verbindungen

Algorithm 1 Randomisierte Tiefensuche für Maze-Generierung

```
1: Initialisiere Grid mit Wänden
2: Wähle zufällige Startposition (ungerade Koordinaten)
3: Markiere Startposition als Pfad
4: stack = [Startposition]
5: while stack nicht leer do
6:   current = stack.top()
7:   neighbors = unbesuchte Nachbarn von current (2 Schritte entfernt)
8:   if neighbors existieren then
9:     next = zufälliger Nachbar aus neighbors
10:    Entferne Wand zwischen current und next
11:    Markiere next als besucht
12:    stack.push(next)
13:   else
14:     stack.pop()
15:   end if
16: end while
```

2.2 Ungerade Dimensionen und 2er-Schritte

Die Verwendung ungerader Dimensionen und 2er-Schritte in der Tiefensuche ist essenziell für die korrekte Funktionsweise des Algorithmus:

2.2.1 Ungerade Dimensionen

- Garantieren, dass Start- und Endpunkte auf gültigen Pfadpositionen liegen
- Vermeiden Randprobleme bei der Wandentfernung
- Stellen sicher, dass das resultierende Gitter die erforderliche Struktur aufweist

2.2.2 2er-Schritte

- **Wanderhaltung:** Zwischen zwei Pfadzellen muss immer eine Wand liegen
- **Gitterstruktur:** Pfadzellen liegen nur auf ungeraden Koordinaten (1,1), (1,3), (3,1), etc.
- **Konnektivität:** Beim Überbrücken einer Wand werden genau zwei Schritte benötigt

Beispiel 5x5 Grid:

```
W W W W W    # W = Wand, P = Pfad
W P W P W
W W W W W
W P W P W
W W W W W
```

Nach Verbindung von (1,1) zu (1,3):

```
W W W W W
W P P P W    # Wand bei (1,2) entfernt
W W W W W
W P W P W
W W W W W
```

Abbildung 1: Gitterstruktur und Wandentfernung

2.3 Datenrepräsentation als NumPy-Matrix

Das Labyrinth wird als zweidimensionale NumPy-Matrix gespeichert, wobei jeder Zellenwert eine spezifische Bedeutung hat:

Wert	Farbe	Bedeutung
0	Weiß	Freier Pfad
1	Schwarz	Wand
2	Grün	Startposition
3	Rot	Zielposition
4	Gelb	Offene Knoten (in Queue)
5	Hellblau	Aktueller Pfad
6	Blau	Finaler optimaler Pfad

Tabelle 1: Farbkodierung der Maze-Zellen

2.4 Reproduzierbarkeit durch Seed-Kontrolle

Die Übergabe eines `numpy.random.Generator`-Objekts gewährleistet:

- **Reproduzierbarkeit:** Identische Seeds erzeugen identische Labyrinthe
- **Testbarkeit:** Algorithmen können unter gleichen Bedingungen verglichen werden
- **Debugging:** Problematische Fälle können gezielt reproduziert werden
- **Wissenschaftlichkeit:** Experimente sind wiederholbar und verifizierbar

```
1 from numpy.random import Generator, PCG64
2
3 # Deterministischer Generator
```

```
4 seed = 42
5 rng = Generator(PCG64(seed))
6
7 # Erzeugt immer das gleiche Labyrinth
8 maze = generator.generate_maze_grid(width, height, rng)
```

Listing 1: Beispiel für deterministische Maze-Generierung

3 Generator-Pattern und Lazy Evaluation

3.1 Python Generatoren

Ein zentrales Designelement der Implementierung ist die extensive Nutzung von Python-Generatoren durch das `yield`-Keyword. Generatoren sind eine spezielle Art von Iteratoren, die Werte on-demand erzeugen.

3.1.1 Was sind Generatoren?

Generatoren sind Funktionen, die den Zustand zwischen Aufrufen beibehalten und Werte schrittweise produzieren:

```
1 def simple_generator():
2     print("Start")
3     yield 1
4     print("Zwischen den yields")
5     yield 2
6     print("Ende")
7
8 # Verwendung
9 gen = simple_generator()
10 print(next(gen)) # Output: "Start", dann 1
11 print(next(gen)) # Output: "Zwischen den yields", dann 2
```

Listing 2: Einfaches Generator-Beispiel

3.1.2 Vergleich mit bekannten Iteratoren

Die `range()`-Funktion in Python 3 ist ein klassisches Beispiel für Lazy Evaluation:

```
1 # Erzeugt nicht alle Werte im Speicher
2 large_range = range(1000000) # Sehr wenig Speicherverbrauch
3
4 # Versus Liste (Eager Evaluation)
5 large_list = list(range(1000000)) # Hoher Speicherverbrauch
```

Listing 3: Range als Iterator

3.2 Vorteile der Generator-Nutzung

3.2.1 Speichereffizienz

- **Konstanter Speicherverbrauch:** Nur der aktuelle Frame wird gespeichert

- **Skalierbarkeit:** Funktioniert auch bei sehr großen Labyrinthen
- **Streaming:** Animationsframes werden just-in-time generiert

3.2.2 Performance-Vorteile

- **Lazy Evaluation:** Berechnung nur bei Bedarf
- **Früher Ausstieg:** Animation kann jederzeit gestoppt werden
- **Pipeline-Verarbeitung:** Frames können direkt verarbeitet werden

4 Grafische Benutzeroberfläche

4.1 Tkinter als GUI-Framework

Die Wahl von Tkinter als GUI-Framework basiert auf mehreren Überlegungen:

4.1.1 Persönliche Erfahrung

- Umfangreiche Erfahrung mit Tkinter in verschiedenen Projekten
- Vertrautheit mit Widgets und Layout-Management

4.1.2 Community und Dokumentation

- **Hohe Popularität:** Weit verbreitet in der Python-Community
- **Umfangreiche Dokumentation:** Offizielle Docs und Community-Tutorials
- **Aktive Community:** Schnelle Hilfe bei Problemen
- **Stabilität:** Teil der Python-Standardbibliothek seit Python 1.0

4.1.3 Widget-Prinzip

Tkinter folgt dem bewährten Widget-Prinzip der GUI-Entwicklung:

```
1 class GUI(tk.Tk): # Hauptfenster
2     def __init__(self):
3         # Container-Widgets
4         main_container = tk.Frame(self)
5         maze_config_frame = ttk.LabelFrame(parent_frame, text="Maze
Parameters")
6
7         # Input-Widgets
8         self.width_slider = tk.Scale(maze_config_frame, ...)
9         self.height_slider = tk.Scale(maze_config_frame, ...)
10
11        # Layout-Management
12        maze_config_frame.pack()
13        self.width_slider.grid(row=0, column=0, ...)
```

Listing 4: Widget-Hierarchie in der Anwendung

4.2 Matplotlib-Integration

4.2.1 Warum Matplotlib für Animation?

- **Tkinter-Integration:** Nahtlose Einbettung via FigureCanvasTkAgg
- **Video-Export:** Direkte MP4-Exportfunktionalität
- **Professionelle Visualisierung:** Hochqualitative Grafiken
- **Animation-Framework:** FuncAnimation für flüssige Animationen

```
1 def _setup_matplotlib_canvas(self):
2     # Matplotlib Figure erstellen
3     self.visualization_figure = plt.Figure(figsize=(10, 6))
4
5     # In Tkinter einbetten
6     self.matplotlib_canvas = FigureCanvasTkAgg(
7         self.visualization_figure, master=self
8     )
9     self.matplotlib_canvas.get_tk_widget().grid(row=0, column=1)
10
11    # Animation setup
12    self.current_animation = FuncAnimation(
13        self.visualization_figure,
14        self._update_frame,
15        frames=self.frame_iterator,
16        interval=50, blit=True
17    )
```

Listing 5: Matplotlib in Tkinter einbetten

4.3 Steuerelemente der Benutzeroberfläche

4.3.1 Maze-Parameter

- **Width/Height Slider:** Kontrolle der Labyrinthgröße (30-300 × 20-200)
- **Extra Openings:** Zusätzliche Öffnungen für interessantere Pfade (0-10%)
- **Animation Toggle:** Ein-/Ausschalten der Generierungs-Animation
- **Seed Control:** Deterministische vs. zufällige Generierung

4.3.2 Algorithmus-Auswahl

- **Multi-Selection:** Mehrere Algorithmen gleichzeitig auswählbar
- **Sequential Execution:** Automatische Abarbeitung der gewählten Algorithmen
- **Live Comparison:** Echtzeit-Vergleich der Performance-Metriken

4.3.3 Animation-Kontrolle

- **Delay-Slider:** Geschwindigkeitskontrolle (10-1000ms)
- **Export-Funktion:** MP4-Video-Export mit konfigurierbaren Optionen

5 Pathfinding-Algorithmen

5.1 A* (A-Stern) Algorithmus

A* ist ein informierter Suchalgorithmus, der eine Heuristik verwendet, um die Suche zu leiten.

5.1.1 Funktionsweise

- **Bewertungsfunktion:** $f(n) = g(n) + h(n)$
- **Pfadkosten:** $g(n)$ = tatsächliche Kosten vom Start zu Knoten n
- **Heuristik:** $h(n)$ = geschätzte Kosten von n zum Ziel

5.1.2 Heuristik

Verwendet Manhattan-Distanz: $h(n) = |x_n - x_{goal}| + |y_n - y_{goal}|$

5.1.3 Eigenschaften

- **Optimal:** Findet den kürzesten Pfad (bei zulässiger Heuristik)
- **Vollständig:** Findet eine Lösung, wenn eine existiert
- **Effizient:** Deutlich schneller als uninformierte Suche

5.2 Dijkstra-Algorithmus

Dijkstra ist ein uninformierter Algorithmus, der alle Richtungen gleichmäßig erkundet.

5.2.1 Funktionsweise

- **Bewertungsfunktion:** $f(n) = g(n)$ (nur Pfadkosten)
- **Strategie:** Erkundet Knoten in Reihenfolge der Pfadkosten
- **Garantie:** Findet immer den optimalen Pfad

5.2.2 Eigenschaften

- **Optimal:** Garantiert kürzesten Pfad
- **Vollständig:** Findet Lösung bei Existenz
- **Uninformiert:** Nutzt keine Zielinformation

5.3 Greedy Best-First Search

Ein gieriger Algorithmus, der ausschließlich die Heuristik zur Knotenbewertung nutzt.

5.3.1 Funktionsweise

- **Bewertungsfunktion:** $f(n) = h(n)$ (nur Heuristik)
- **Strategie:** Wählt immer den Knoten, der dem Ziel am nächsten scheint
- **Gierig:** Trifft lokal optimale Entscheidungen

5.3.2 Eigenschaften

- **Nicht optimal:** Kann suboptimale Pfade finden
- **Schnell:** Sehr direkte Zielannäherung
- **Speichereffizient:** Weniger Knoten in der Queue

5.4 Algorithmus-Vergleich

Eigenschaft	A*	Dijkstra	Greedy BFS
Optimalität	✓	✓	X
Vollständigkeit	✓	✓	X
Zeitkomplexität	$O(b^d)$	$O(V^2)$	$O(b^m)$
Speicherkomplexität	$O(b^d)$	$O(V)$	$O(b^m)$
Heuristik erforderlich	✓	X	✓
Geschwindigkeit	Mittel	Langsam	Schnell

Tabelle 2: Vergleich der Pathfinding-Algorithmen

5.4.1 Praktische Anwendungsbereiche

- **A*:** GPS-Navigation, Spieleentwicklung, Roboterpfadplanung
- **Dijkstra:** Netzwerk-Routing, soziale Netzwerkanalyse, kritische Systeme
- **Greedy BFS:** Echtzeit-Anwendungen, Prototyping, approximative Lösungen

6 Implementierungsdetails

6.1 Architektur-Übersicht

Die Anwendung folgt einer modularen Architektur mit klarer Trennung der Verantwortlichkeiten:

- **GUI-Modul:** Benutzeroberfläche und Ereignisbehandlung

- **Grid-Modul:** Maze-Generierung und -Verwaltung
- **Search-Module:** Implementierung der Pathfinding-Algorithmen

6.2 Performance-Optimierungen

6.2.1 Memory Management

- Lazy Loading durch Generatoren
- Effiziente NumPy-Array-Operationen
- Minimierung von Array-Kopien

6.2.2 Animation-Optimierung

- Blitting für flüssige Maze-Generation
- Adaptive Frame-Rate basierend auf Delay-Einstellung
- Hintergrund-Verarbeitung für Video-Export

7 Fazit

7.1 Erreichte Ziele

Das Projekt erfüllt alle gesetzten Ziele:

- Erfolgreiche Implementierung eines robusten Maze-Generators
- Intuitive und funktionsreiche grafische Benutzeroberfläche
- Vergleichende Visualisierung von drei klassischen Pathfinding-Algorithmen
- Export-Funktionalität für Dokumentation

7.2 Erkenntnisse

7.2.1 Algorithmische Erkenntnisse

- A* bietet den besten Kompromiss zwischen Optimalität und Effizienz
- Dijkstra ist unverzichtbar, wenn absolute Optimalität erforderlich ist
- Greedy BFS eignet sich für Echtzeit-Anwendungen mit Geschwindigkeitspriorität

7.2.2 Technische Erkenntnisse

- Generator-Pattern ermöglicht elegante und speichereffiziente Lösungen
- Tkinter bleibt eine solide Wahl für GUI-Anwendungen
- Matplotlib-Integration erweitert Visualisierungsmöglichkeiten erheblich

7.3 Mögliche Erweiterungen

- Bidirectional A* für noch bessere Performance
- Jump Point Search für Grid-optimierte Suche
- 3D-Maze-Generierung und -Visualisierung
- Verschiedene Maze-Generierungsalgorithmen (Kruskal, Prim)
- Interaktive Hindernis-Platzierung
- Multi-Agent-Pathfinding

A Quellcode

Github Repository: https://github.com/OleMatzky/Modellierung_Endprojekt

- [gui.py](#) ~ 500 LoC
- [grid.py](#) ~ 120 LoC
- [search_base.py](#) ~ 6 LoC
- [astar.py](#) ~ 100 LoC
- [dijkstra.py](#) ~ 100 LoC
- [greedy.py](#) ~ 90 LoC

B Systemanforderungen

verwendete Versionen:

- Python 3.11.4
- NumPy 1.25
- Matplotlib 3.8.3
- Tkinter (normalerweise in Python enthalten)
- FFmpeg (für Video-Export, Version: 2024)