

Gottfried Wilhelm Leibniz Universität Hannover
Institut für Theoretische Informatik

Eine GUI zur Visualisierung von Pfad-Such-Algorithmen

Bachelorarbeit

Kevin Kießling

Matrikelnr. 3129780

Hannover, den 13. März 2025

Erstprüfer: Prof. Dr. rer. nat. Arne Meier
Zweitprüfer: Prof. Dr. rer. nat. Heribert Vollmer
Betreuer: Vivian Holzapfel

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 13. März 2025

Kevin Kießling

Inhaltsverzeichnis

| | | |
|----------|---|-----------|
| 1 | Einleitung | 1 |
| 2 | Grundlagen | 2 |
| 2.1 | Motivation | 2 |
| 2.2 | Theoretische Grundlagen gerichteter Graphen | 2 |
| 2.3 | Die Datenstruktur Priority Queue | 5 |
| 2.3.1 | Implementierung durch einen Min-Heap | 6 |
| 3 | Der Dijkstra-Algorithmus | 11 |
| 3.1 | Implementierung mithilfe einer Liste | 12 |
| 3.2 | Implementierung mithilfe einer Priority Queue | 13 |
| 3.3 | Implementierung von Lazy Deletion | 16 |
| 4 | Verwendete Bibliotheken und Werkzeuge | 17 |
| 4.1 | Tkinter – Das GUI-Toolkit von Python | 17 |
| 4.2 | NetworkX | 18 |
| 5 | GUI-Aufbau | 19 |
| 5.1 | Main-Frame | 19 |
| 5.2 | Canvas-Frame | 19 |
| 5.2.1 | Canvas | 19 |
| 5.2.2 | Menüleiste | 21 |
| 5.2.3 | Button-Frame | 21 |
| 5.3 | Pseudocode-Frame | 22 |
| 5.3.1 | Pseudocode-Text | 22 |
| 5.3.2 | Distanztabelle | 23 |
| 5.3.3 | Datenstruktur-Visualisierung | 23 |
| 6 | Implementierung | 25 |
| 6.1 | Objektorientierter Ansatz | 25 |
| 6.2 | Dateistruktur | 25 |
| 6.3 | Schrittweise Wiedergabe der Algorithmusausführungen | 26 |
| 6.4 | Animationen der Heap-Operationen | 27 |

| | | |
|----------|---|-----------|
| 6.5 | Graph-Darstellung | 28 |
| 6.6 | Skalierung der GUI | 29 |
| 6.7 | Probleme | 30 |
| 7 | Testen des Programms | 31 |
| 7.1 | Testen der Dijkstra-Algorithmen | 31 |
| 7.2 | Testen der Heap-Operationen | 32 |
| 7.3 | Manuelles Testen der GUI | 32 |
| 8 | Zusammenfassung und Ausblick | 33 |

1 Einleitung

Pfadsuchalgorithmen sind ein wichtiger Bestandteil in vielen alltäglichen Anwendungsbereichen. Sie werden zum Beispiel eingesetzt, um die kürzeste Route in Navigationssystemen zu berechnen, die günstigsten Wege in der Logistik zu planen oder in Videospielen, um Charaktere durch die Spielwelt zu bewegen. Das Ziel solcher Algorithmen ist es, den kürzesten oder effizientesten Pfad zwischen zwei Punkten in einem Graphen zu finden. Die Algorithmen sollen auch bei besonders großen Graphen, wie einem Straßennetz oder einer Spielwelt, in möglichst kurzer Zeit eine Lösung finden. Deshalb sind Effizienz und Skalierbarkeit zwei wichtige Aspekte bei der Implementierung von Pfadsuchalgorithmen. Das Ziel dieser Bachelorarbeit ist die Entwicklung einer grafischen Benutzeroberfläche (GUI) zur Visualisierung von Pfadsuchalgorithmen. Diese soll Studierende dabei unterstützen, die in der Vorlesung Datenstrukturen und Algorithmen vermittelten Algorithmen und Konzepte besser nachzuvollziehen und zu verstehen. Dabei können Nutzer der GUI eigene Graphen erstellen und speichern, bereits erstellte Graphen laden und Schritt für Schritt den Ablauf der Algorithmen nachvollziehen. Dazu wird der Dijkstra-Algorithmus implementiert, da er sich gut als Einführung in das Thema der Pfadsuchalgorithmen eignet.

2 Grundlagen

2.1 Motivation

Um Pfadsuchalgorithmen wie den Dijkstra-Algorithmus zu verstehen, zu implementieren oder die erstellte Benutzeroberfläche zu bedienen, sind grundlegende graphentheoretische Kenntnisse notwendig, die in den folgenden Abschnitten erläutert werden. Ein wichtiger Aspekt bei der Implementierung dieser Algorithmen ist die Wahl der zugrunde liegenden Datenstruktur, da diese die Effizienz maßgeblich beeinflusst. Für den Dijkstra-Algorithmus ist es wichtig, die Knoten anhand ihrer aktuellen Distanz (Priorität) effizient zu verwalten. Die Prioritätswarteschlange (Priority Queue), die in Abschnitt 2.3 definiert wird, ist eine solche Datenstruktur, die Elemente gemäß ihrer Priorität anordnet. Diese Ordnung ermöglicht es, die Knoten mit der geringsten Distanz effizient auszuwählen und zu verarbeiten, was die Laufzeit des Algorithmus verbessert. Die Wahl einer geeigneten Implementierung der Priority Queue hat daher einen wichtigen Einfluss auf die Laufzeiteffizienz des Dijkstra-Algorithmus.

2.2 Theoretische Grundlagen gerichteter Graphen

Das in dieser Arbeit entwickelte Programm basiert auf der Erstellung und Verarbeitung von Graphen. Der folgende Abschnitt führt die grundlegenden Definitionen der Graphentheorie ein, die für das Verständnis dieser Arbeit erforderlich sind.

Definition 2.2.1 ([Cor+09, S. 1168]) *Ein gerichteter Graph $G = (V, E)$ ist definiert durch ein Tupel bestehend aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$.*

Definition 2.2.2 ([Mor13, S. 247]) *Jede Kante $e \in E$ ist definiert als Tupel (i, j) , wobei i der Startknoten und j der Zielknoten der Kante ist.*

Die Kante k von Knoten A nach Knoten B aus Abbildung 2.1 ist definiert als $k = (A, B)$. Der Graph aus Abbildung 2.1 ist vollständig definiert durch die Knotenmenge $V = \{A, B, C, D, E\}$ und die Kantenmenge

$$E = \{(A, C), (A, E), (A, B), (A, D), (B, C), (B, D), (C, D), (D, E)\}.$$

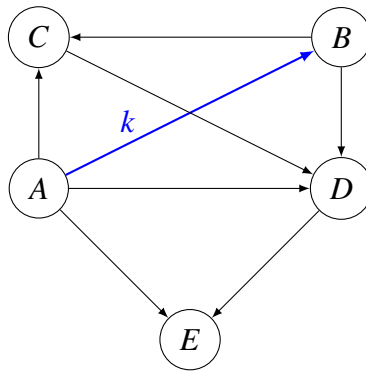


Abbildung 2.1: Beispiel eines gerichteten Graphen

In dieser Bachelorarbeit werden ausschließlich gerichtete Graphen betrachtet, da sich jede ungerichtete Kante (u, v) als zwei gerichtete Kanten (u, v) und (v, u) darstellen lässt [Cor+09, S. 1172].

Definition 2.2.3 ([Cor+09, S. 591]) Ein gewichteter Graph ist definiert durch ein Tupel $G = (V, E)$, bestehend aus der Knotenmenge V und der Kantenmenge E . Zusätzlich wird eine Gewichtsfunktion $\omega: E \rightarrow \mathbb{N}$ definiert, welche jeder Kante $e \in E$ ein positives Gewicht zuordnet.

Die Kante k von Knoten A nach Knoten B aus Abbildung 2.2 ist definiert durch $k = (A, B)$ mit Gewicht $\omega(A, B) = 2$.

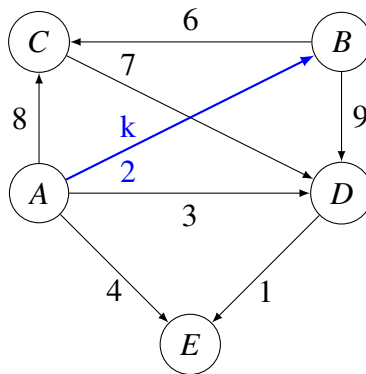


Abbildung 2.2: Beispiel eines gewichteten, gerichteten Graphen

Definition 2.2.4 ([Cor+09, S. 1169]) In einem gerichteten Graphen wird ein Knoten v als adjazent (Nachbar) zu einem Knoten s bezeichnet, wenn eine Kante $(s, v) \in E$ existiert.

Der Knoten B aus Abbildung 2.2 ist zu Knoten A adjazent, da die Kante $(A, B) \in E$ existiert.

Definition 2.2.5 ([Cor+09, S. 590–591]) Eine Adjazenzliste eines gewichteten Graphen $G = (V, E)$ mit einer Gewichtsfunktion $\omega: E \rightarrow \mathbb{N}$ besteht aus einem Array $\text{AdjList}[]$ mit $|V|$

Listen. Für jeden Knoten $s \in V$ gibt es genau eine Liste, die alle Knoten $t \in V$ enthält, für die eine Kante $(s, t) \in E$ existiert. Dabei wird jedes dieser Elemente als Tupel $(t, \omega(s, t))$ gespeichert, wobei t der adjazente Knoten und $\omega(s, t)$ das zugehörige Kantengewicht ist.

Die Adjazenzliste des Graphen aus Abbildung 2.2 ist in Abbildung 2.3 dargestellt. Adjazenzlisten ermöglichen eine kompakte Speicherung von Graphen und sind besonders effizient für Graphen, bei denen die Anzahl der Kanten $|E|$ deutlich kleiner ist als das Quadrat der Anzahl der Knoten $|V|^2$ [Cor+09, S. 589]. Die im Rahmen dieser Arbeit entwickelte GUI nutzt Adjazenzlisten, um den aktuell geladenen Graphen innerhalb der Anwendung zu verwalten. Auch die Import- und Exportfunktionen basieren auf Adjazenzlisten, um Graphen zu speichern und zu laden.

$$\text{AdjList} = \left\{ \begin{array}{l} A : \{(B, 2), (C, 8), (D, 3), (E, 4)\}, \\ B : \{(C, 6), (D, 9)\}, \\ C : \{(D, 7)\}, \\ D : \{(E, 1)\}, \\ E : \{\} \end{array} \right\}$$

Abbildung 2.3: Adjazenzliste des Graphen aus Abbildung 2.2

Definition 2.2.6 ([Cor+09, S. 643]) *Ein Pfad in einem Graphen $G = (V, E)$ ist eine Folge von Knoten*

$$\pi = (v_1, v_2, \dots, v_n),$$

sodass $v_i \in V$ für alle $i = 1, \dots, n$ und $(v_i, v_{i+1}) \in E$ für alle $i = 1, \dots, n - 1$ gilt.

Ein exemplarischer Pfad von Knoten A nach Knoten E aus Abbildung 2.2 ist der Pfad $p = (A, B, D, E)$.

Definition 2.2.7 ([Cor+09, S. 643]) *Die Kosten eines Pfades π werden berechnet als:*

$$\text{Kosten}(\pi) = \sum_{i=1}^{n-1} \omega(v_i, v_{i+1}),$$

wobei $\omega(v_i, v_{i+1})$ das Gewicht der Kante (v_i, v_{i+1}) darstellt.

Für den Pfad p ergibt sich daher:

$$\text{Kosten}(p) = \omega(A, B) + \omega(B, D) + \omega(D, E) = 2 + 9 + 1 = 12.$$

Definition 2.2.8 ([Cor+09, S. 643]) *Ein Pfad von einem Knoten X zu einem Knoten Y wird als kürzester Pfad bezeichnet, wenn er unter allen möglichen Pfaden von X nach Y die geringsten Kosten aufweist, wobei die Kosten nach Definition 2.2.7 berechnet werden.*

Im gewichteten, gerichteten Graphen aus Abbildung 2.2 existieren zwei kürzeste Pfade zwischen Knoten A und E :

1. Der Pfad $z = (A, D, E)$ mit

$$\text{Kosten}(z) = \omega(A, D) + \omega(D, E) = 3 + 1 = 4.$$

2. Der Pfad $h = (A, E)$ mit

$$\text{Kosten}(h) = \omega(A, E) = 4.$$

2.3 Die Datenstruktur Priority Queue

Eine Priority Queue ist eine Datenstruktur, die Elemente anhand ihrer Priorität verwaltet. Der folgende Abschnitt definiert die für diese Arbeit relevanten Eigenschaften einer Priority Queue sowie deren Implementierung und basiert, sofern nicht anders angegeben, auf Kapitel B.5.3 und Kapitel 6 aus dem Standardwerk Introduction to Algorithms, 3rd Edition [Cor+09]. Die Definition der Priority Queue und der Pseudocode der Heap-Operationen wurden dabei in angepasster Form dargestellt.

Definition 2.3.1 ([OW12, S. 28]) *Ein abstrakter Datentyp (ADT) ist ein Datentyp, der aus einer Menge von Objekten und den darauf definierten Operationen besteht.*

Definition 2.3.2 *Eine Priority Queue ist ein ADT, der eine Menge von Objekten des ADT Item verwaltet. Jedes Item e besteht aus einem Schlüssel $\text{key}(e)$ und einem Inhaltsobjekt $\text{obj}(e)$. Der Schlüssel $\text{key}(e)$ ist ein Wert $k \in S$, wobei S eine geordnete Menge darstellt, die durch die Ordnungsrelation $\leq: S \times S$ definiert ist. Dieser Schlüssel $\text{key}(e)$ beschreibt die Priorität des Items e . Je kleiner der Wert $\text{key}(e)$, desto höher ist die Priorität des Items. Das Inhaltsobjekt $\text{obj}(e)$ kann beliebig sein, muss jedoch innerhalb der Datenstruktur einheitlich sein.*

Auf dem ADT Priority Queue sind typischerweise die folgenden Operationen definiert:

1. `insert` (Item e): Fügt Item e der Priority Queue hinzu.
2. `remove` (Item e): Entfernt Item e aus der Priority Queue.
3. `empty()`: Gibt `true` zurück, falls die Priority Queue leer ist.
4. `removeMin()`: gibt das Element mit der höchsten Priorität zurück und entfernt es aus der Priority Queue.

Je nach Implementierung können weitere Operationen hinzugefügt oder bestimmte Operationen nicht berücksichtigt werden.

2.3.1 Implementierung durch einen Min-Heap

Die Implementierung des Dijkstra-Algorithmus, wie in Kapitel 3 beschrieben, verwendet eine Priority Queue, um die Menge der noch zu bearbeitenden Knoten effizient zu verwalten. Dabei wird die Priority Queue durch einen Min-Heap realisiert. Dies ermöglicht eine effiziente Auswahl und Aktualisierung der Knoten während der Ausführung des Algorithmus.

Definition 2.3.1.1 Ein Binärbaum ist eine rekursiv definierte Struktur, die auf einer endlichen Menge von Knoten basiert und entweder leer ist (keine Knoten enthält) oder aus einem Wurzelknoten sowie zwei Teilbäumen besteht. Diese Teilbäume werden als linker und rechter Teilbaum bezeichnet. Wenn der linke Teilbaum nicht leer ist, ist die Wurzel des linken Teilbaums das linke Kind des Wurzelknotens des gesamten Baums. Entsprechend wird die Wurzel des rechten Teilbaums als rechtes Kind des Wurzelknotens bezeichnet.

Definition 2.3.1.2 Ein Binärbaum T wird als vollständiger Binärbaum bezeichnet, wenn die folgenden Bedingungen erfüllt sind:

1. Alle Ebenen i mit $0 \leq i < h - 1$, wobei h die Höhe des Baumes ist, sind vollständig aufgefüllt. Das bedeutet, jede Ebene i enthält genau 2^i Knoten.
2. In der letzten Ebene $h - 1$ sind die Blätter (Knoten ohne Kinder) linksbündig angeordnet.

Die Höhe des Baumes h ist definiert als die Anzahl der Ebenen in T , wobei die Wurzel in Ebene 0 liegt.

Definition 2.3.1.3 Ein Min-Heap ist ein Binärbaum, dessen Knoten die Objekte des ADT Item repräsentieren. Diese Knoten sind durch eine Ordnungsrelation geordnet, sodass die folgenden zwei Bedingungen erfüllt sind:

1. Heap-Eigenschaft: Für jeden Knoten v muss gelten, dass $key(v) \leq key(u)$, wenn Knoten u ein Kind von Knoten v ist.
2. Der Binärbaum ist vollständig.

Ein effizienter Ansatz, Heaps zu repräsentieren, ist es, die Knoten ebenenweise in einem Array zu speichern. Die Kindknoten eines Knotens v an Position k befinden sich an der Position $2k$ (linker Kindknoten) und $2k + 1$ (rechter Kindknoten). Der Elternknoten von v befindet sich an Position $\lfloor \frac{k}{2} \rfloor$. Der Index 0 des Arrays wird bei dieser Implementierung nicht belegt. Die Abbildung 2.4 zeigt den Min-Heap H , und die Tabelle 2.1 zeigt die zugehörige Arraydarstellung. An dem Arrayindex 1 befindet sich das Objekt mit der höchsten Priorität. Die in den Knoten gespeicherten Objekte sind des Typs ADT Item und die Inhaltsobjekte dieser Objekte sind für diese Implementierung nicht relevant.

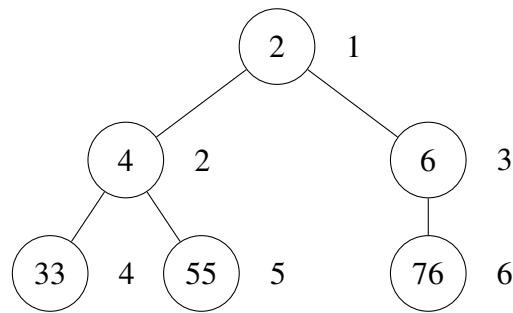


Abbildung 2.4: Min-Heap H

| Array[] | Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|-----------|-------|-----------|---|---|---|----|----|----|
| Priorität | | $-\infty$ | 2 | 4 | 6 | 33 | 55 | 76 |

Tabelle 2.1: Arraydarstellung des Min-Heaps H

Das Einfügen eines neuen Elements in einen Min-Heap mit n Elementen erfolgt über das Hinzufügen des neuen Elements an Position $n + 1$ des Arrays, wobei angenommen wird, dass ausreichend Speicherplatz vorhanden ist. Nach dem Hinzufügen eines neuen Elements wird der `Heapify-Up-Algorithmus` durchgeführt, um die Min-Heap-Eigenschaft zu reparieren, falls diese durch das Einfügen des neuen Elements verletzt wurde. Der Algorithmus 1 zeigt den Pseudocode des `Heapify-Up-Algorithmus`. Die Abbildung 2.5 zeigt die Baumdarstellung des Min-Heaps nach dem Einfügen eines Objektes mit $key = 5$. Dieses Objekt wird in der untersten Zeile des Baumes linksbündig positioniert und im Array an Index $n + 1 = 7$ gespeichert (siehe Tabelle 2.2), und die Funktion `Heapify-Up($H, 7$)` wird aufgerufen. Durch das Einfügen an dieser Stelle wurde die Min-Heap-Eigenschaft $key(j) \leq key(i)$ verletzt (Zeile 4), da der Knoten an Index i mit $key = 5$ ein Kind des Knotens an Index j mit $key = 6$ ist. Der `Heapify-Up-Algorithmus` tauscht in diesem Fall die Knoten an den Indizes i und j (siehe Tabelle 2.3). Die Abbildung 2.6 zeigt die Baumdarstellung des Min-Heaps nach dem Vertauschen dieser Knoten. Nach dem Tausch wird rekursiv `Heapify-Up($H, 3$)` aufgerufen, und in diesem Durchlauf wird die Bedingung $key(j) \leq key(i)$ nicht verletzt. Der `Heapify-Up-Algorithmus` endet, und die Heap-Eigenschaft ist repariert.

Algorithm 1 Heapify-Up

```

1: Input: Heap  $H$  und Index  $i$ 
2: if  $i > 1$  then
3:    $j \leftarrow \lfloor i/2 \rfloor$ 
4:   if  $H[i].getKey() < H[j].getKey()$  then
5:     Vertausche  $H[i]$  und  $H[j]$ 
6:     Heapify-Up( $H, j$ )
7:   end if
8: end if

```

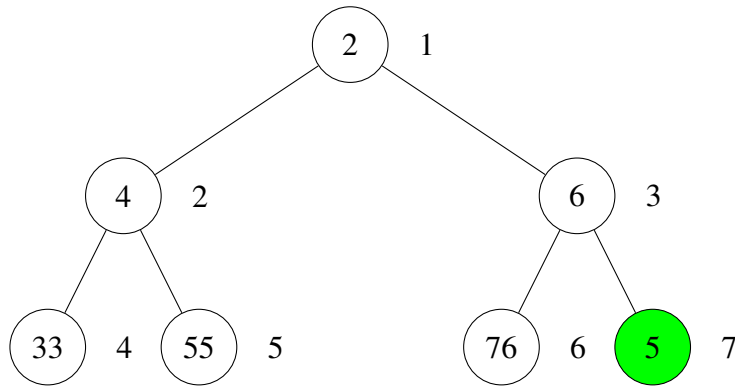


Abbildung 2.5: Min-Heap H , nach Einfügen von Objekt mit Priorität 5 und vor Heapify-Up

| | | | | | | | | |
|---------------|-----------|---|---|---|----|----|----|---|
| Array[] Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Priorität | $-\infty$ | 2 | 4 | 6 | 33 | 55 | 76 | 5 |

Tabelle 2.2: Arraydarstellung des Min-Heaps H nach Einfügen und vor Heapify-Up

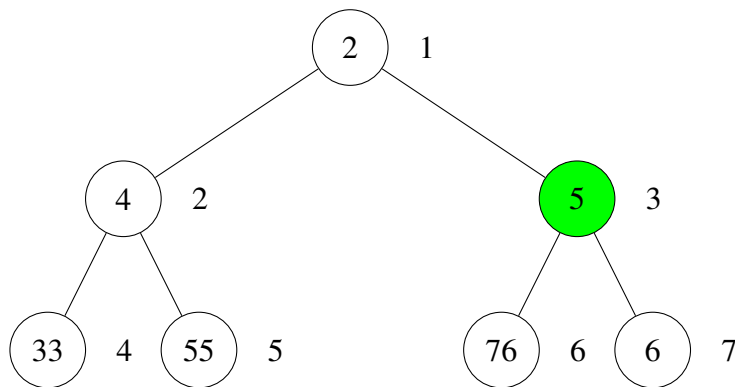


Abbildung 2.6: Min-Heap H , nach Heapify-Up

| | | | | | | | | |
|---------------|-----------|---|---|---|----|----|----|---|
| Array[] Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Priorität | $-\infty$ | 2 | 4 | 5 | 33 | 55 | 76 | 6 |

Tabelle 2.3: Arraydarstellung des Min-Heaps H nach Heapify-Up

Da der Heap in diesem Kontext zur Implementierung einer Priority Queue dient, ist das Ausgeben und anschließende Löschen des Wurzelements an Index 1 für viele Anwendungsfälle relevant, da dieses Element die höchste Priorität besitzt. Die Min-Heap-Operation `extractMin()` ist eine Kombination aus den Min-Heap-Operationen `findMin()` und `delete(H, 1)`, und sie dient dazu, das Minimum des Heaps H zu finden, auszugeben und anschließend zu löschen. Nach dem Löschen eines Elements wird das Element an dem

letzten Index n des Heaps an eine freie Stelle verschoben. Im Fall von `extractMin()` wird das Element von Position n an Position 1 verschoben, was die Heap-Eigenschaft verletzen kann. Der Algorithmus 2 zeigt den Pseudocode des `Heapify-Down-Algorithmus`, der diese Eigenschaft repariert. Abbildung 2.7 zeigt den Zustand des Min-Heaps vor dem Aufruf von `extractMin()`. Durch den Aufruf von `extractMin()` wird die Wurzel mit $key = 2$ gelöscht, das Element an Position 7 auf Position 1 verschoben und `Heapify-Down($H, 1$)` aufgerufen. Die Abbildung 2.8 zeigt den Heap nach der Löschung des Minimums. Im ersten Durchlauf des Algorithmus tritt die `if`-Bedingung aus Zeile 5 ein, berechnet die Indizes der Kindknoten des an Index i stehenden Knotens und setzt j auf den Index des Kindknotens mit dem kleineren key . Im ersten Durchlauf ist $i = 1$, und der Kindknoten mit dem kleinsten key ist an Index $j = 2$ gespeichert. Der key des Knotens an Index j ist kleiner als der key des Knotens an Index i , sodass die `if`-Anweisung in Zeile 11 erfüllt wird und die Knoten an Index i und j getauscht werden. Die Abbildung 2.9 zeigt den Heap nach dem Tausch der beiden Knoten. Im nächsten Schritt wird rekursiv `Heapify-Down($H, 2$)` aufgerufen. Der Algorithmus stellt erneut fest, dass einer der Kindknoten (Index j) des Knotens an Index i einen kleineren key besitzt, und tauscht die Knoten an den Indizes i und j . Die Abbildung 2.10 zeigt den Heap nach dem Aufruf von `Heapify-Down($H, 2$)`. Anschließend wird rekursiv `Heapify-Down($H, 4$)` aufgerufen. Dieser Aufruf des Algorithmus wird direkt durch die `if`-Anweisung in Zeile 3 beendet, da $2 \cdot i = 8 > n$ ist. Damit ist der Algorithmus abgeschlossen und die Heap-Eigenschaft ist repariert.

Algorithm 2 Heapify-Down

```

1: Input: Ein Heap  $H$  und ein Index  $i$ 
2:  $n \leftarrow \text{Length}(H) - 1$ 
3: if  $2 \cdot i > n$  then
4:   return
5: else if  $2 \cdot i < n$  then
6:    $left \leftarrow 2 \cdot i, right \leftarrow 2 \cdot i + 1$ 
7:    $j \leftarrow$  Index des kleineren Key-Werts von  $H[left]$  und  $H[right]$ 
8: else
9:    $j \leftarrow 2 \cdot i$ 
10: end if
11: if  $H[j].getKey() < H[i].getKey()$  then
12:   Vertausche  $H[i]$  und  $H[j]$ 
13:   Heapify-Down( $H, j$ )
14: end if

```

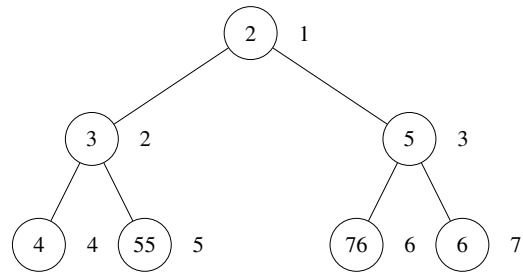


Abbildung 2.7: Min-Heap H vor `extractMin()`

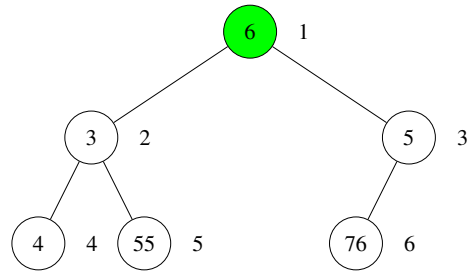


Abbildung 2.8: Min-Heap H nach `extractMin()` und vor `Heapify-Down()`

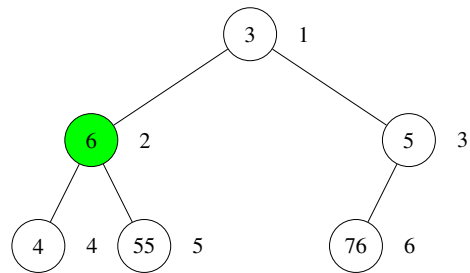


Abbildung 2.9: Min-Heap H nach `Heapify-Down(H,1)`

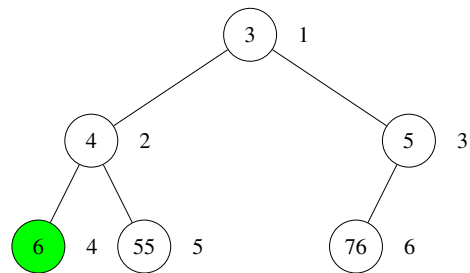


Abbildung 2.10: Min-Heap H nach `Heapify-Down(H,2)`

3 Der Dijkstra-Algorithmus

Der Dijkstra-Algorithmus wurde 1956 von E. W. Dijkstra entwickelt und 1959 erstmals formal beschrieben [Dij59]. Der Algorithmus berechnet ausgehend von einem Startknoten s die kürzesten Pfade zu allen Knoten $v \in V$ in einem gewichteten Graphen $G = (V, E)$ mit Gewichtsfunktion $\omega: E \rightarrow \mathbb{N}$. Die Laufzeiten der Heap-Operationen, die in diesem Kapitel verwendet werden, basieren auf den Angaben im Standardwerk Introduction to Algorithms, 3rd Edition [Cor+09, S. 153].

Die Idee hinter dem Dijkstra-Algorithmus besteht darin, die kürzesten Pfade schrittweise zu berechnen, indem in jedem Schritt der Knoten mit der momentan minimalen bekannten Distanz gewählt wird. Zu Beginn setzt der Algorithmus die Distanz zum Startknoten $d(s) = 0$, während die Distanzen zu allen anderen Knoten $v \in V$ auf $d(v) = \infty$ gesetzt werden. Eine weitere Datenstruktur $discovered[v]$ speichert, ob ein Knoten bereits betrachtet wurde, wobei anfangs für alle Knoten $discovered[v] = false$ gilt. Anschließend betrachtet der Algorithmus schrittweise den Knoten $u \in V$ mit $discovered[u] = false$ und der kleinsten Distanz $d(u)$, setzt $discovered[u] = true$ und überprüft, ob es von diesem Knoten aus kürzere Pfade zu seinen Nachbarknoten $w \in V$ gibt. Dabei werden nur Nachbarknoten überprüft, für die $discovered[w] = false$ gilt, und falls ein kürzerer Pfad gefunden wird, wird die Distanz $d(w)$ angepasst.

Definition 3.0.1 ([Cor+09, S. 414]) *Ein Algorithmus wird als **greedy** bezeichnet, wenn er in jedem Schritt die aus aktueller Sicht beste Wahl trifft. Das heißt, er trifft eine lokal optimale Entscheidung in der Hoffnung, dass diese zu einer global optimalen Lösung führt.*

Der Dijkstra-Algorithmus folgt diesem Prinzip, da er in jedem Schritt den Knoten mit der geringsten bekannten Distanz auswählt. Er liefert immer eine optimale Lösung für kürzeste Wege in Graphen mit nicht-negativen Kantengewichten. Dies liegt daran, dass Knoten in der Reihenfolge ihrer minimalen Distanz verarbeitet werden und bereits besuchte Knoten nicht mehr aktualisiert werden. Dadurch ist sichergestellt, dass jeder Knoten zum frühestmöglichen Zeitpunkt mit der kürzesten Distanz erreicht wird [KT06, S. 138-140]. Die zugrunde liegende Datenstruktur ist für den Dijkstra-Algorithmus flexibel. Der Abschnitt 3.1 analysiert den Algorithmus unter Verwendung einer Liste, während Abschnitt 3.2 den Algorithmus unter Verwendung einer Priority Queue zur Verwaltung der noch zu bearbeitenden Knoten hinsichtlich ihrer Laufzeiten untersucht.

3.1 Implementierung mithilfe einer Liste

In diesem Abschnitt wird die Implementierung des Dijkstra-Algorithmus mit einer Liste zur Verwaltung der noch nicht bearbeiteten Knoten analysiert. Der Algorithmus 3 zeigt den zugehörigen Pseudocode der Implementierung. Im Folgenden wird die Laufzeit der Implementierung des Dijkstra-Algorithmus untersucht, da, wie eingangs erläutert, Effizienz und Skalierbarkeit zwei wichtige Aspekte von Pfadsuchalgorithmen darstellen. Der Rechenaufwand wird in Abhängigkeit von der Eingabe anhand der Landau-Notation (O -Notation) abgeschätzt [Cor+09, S. 47]. Der Algorithmus lässt sich in zwei Abschnitte unterteilen: die Initialisierung von Zeile 2 bis 12 und die Berechnung der Distanzen von Zeile 13 bis 22. Dabei bezeichnet n die Anzahl der Knoten im Graphen. Die Initialisierung besteht zunächst aus zwei `foreach`-Schleifen mit je $O(n)$ -Laufzeit, da die Anzahl der Schleifendurchläufe direkt von der Eingabe n abhängt und die Operationen innerhalb der Schleifen Zuweisungen mit einer Laufzeit von $O(1)$ sind. Die dritte `foreach`-Schleife in Zeile 10 hat ebenfalls eine Laufzeit von $O(n)$, da das Einfügen eines Elements am Ende einer Liste eine Laufzeit von $O(1)$ hat und diese Operation insgesamt n -mal durchgeführt wird. Die Zuweisung in Zeile 8 und die Listenerstellung in Zeile 9 haben $O(1)$ -Laufzeit. Die Gesamtlaufzeit des Initialisierungsteils ergibt sich zu

$$L(n) = 3 \cdot O(n) + 2 \cdot O(1).$$

Da bei der Landau-Notation nur die dominanten Terme berücksichtigt werden und konstante Faktoren sowie Terme, die nicht von der Eingabemenge abhängen, vernachlässigt werden, ergibt sich für die Laufzeit des Initialisierungsabschnitts

$$L(n) = O(n).$$

Der zweite Teil des Algorithmus besteht aus einer `while`-Schleife, die über alle Knoten iteriert und eine Laufzeit von $O(n)$ aufweist. In jedem Schritt der `while`-Schleife wird die gesamte Liste nach dem Element u mit der kleinsten Distanz $d(u)$ durchsucht, was ebenfalls die Laufzeit $O(n)$ hat. Dies führt zu einer Laufzeit von $O(n^2)$. Das Entfernen des Elements mit der kleinsten Distanz $d(u)$ aus der Liste erfolgt in $O(1)$, da die Position des Elements bereits in der vorherigen Zeile ermittelt wurde. Zeile 16 enthält eine einfache Zuweisung, die ebenfalls eine Laufzeit von $O(1)$ hat. Da diese beiden Operationen n -mal durchgeführt werden, ergibt sich je eine Laufzeit von $O(n)$. Die `foreach`-Schleife in Zeile 17 iteriert maximal über alle Kanten und hat eine Laufzeit von $O(m)$, wobei m die Anzahl der Kanten ist. Die `if`-Bedingung in Zeile 18 und die Zuweisung in Zeile 19 haben eine Laufzeit von je $O(1)$ und werden maximal m -mal durchgeführt, was je eine Laufzeit von $O(m)$ ergibt. Der zweite Teil des Algorithmus hat demnach eine Laufzeit von

$$L(n, m) = O(n^2) + 2 \cdot O(n) + 2 \cdot O(m).$$

Da $m \in O(n^2)$ sowie $n \in O(n^2)$ gilt und konstante Terme vernachlässigbar sind, ergibt sich für diesen Teil des Algorithmus eine Laufzeit von

$$L(n) = O(n^2).$$

Durch Zusammenfügen beider Teillaufzeiten ergibt sich eine Gesamtlaufzeit von

$$L(n) = O(n^2) + O(n).$$

Da $O(n)$ hier vernachlässigbar ist, ergibt sich für die Gesamtlaufzeit der Implementierung des Dijkstra-Algorithmus mit einer Liste:

$$L(n) = O(n^2).$$

Algorithm 3 Implementierung Dijkstra-Algorithmus mithilfe einer Liste

```

1: Dijkstra(Gerichteter Graph  $G = (V, E)$ , Gewichtsfunktion  $\omega : E \rightarrow \mathbb{N}$ , Startknoten
    $s \in V$ )
2: for each  $v \in V$  do ▷  $O(n)$ 
3:   discovered[ $v$ ] ← false ▷  $O(1)$ 
4: end for
5: for each  $v \in V$  do ▷  $O(n)$ 
6:    $d(v) \leftarrow \infty$  ▷  $O(1)$ 
7: end for
8:  $d(s) \leftarrow 0$  ▷  $O(1)$ 
9:  $L \leftarrow []$  ▷  $O(1)$ 
10: for each  $v \in V$  do ▷  $O(n)$ 
11:    $L.add(v)$  ▷  $O(1)$ 
12: end for
13: while not  $L.empty()$  do ▷  $O(n)$ 
14:   wähle  $u \in L$  mit kleinstem Wert  $d[u]$  ▷  $O(n)$ 
15:    $L.remove(u)$  ▷  $O(1)$ 
16:   discovered[ $u$ ] ← true ▷  $O(1)$ 
17:   for each  $(u, v) \in E$  do ▷  $O(m)$ 
18:     if not discovered[ $v$ ] then ▷  $O(1)$ 
19:        $d[v] \leftarrow \min(d[v], d[u] + \omega(u, v))$  ▷  $O(1)$ 
20:     end if
21:   end for
22: end while

```

3.2 Implementierung mithilfe einer Priority Queue

Die Effizienz der Implementierung aus dem vorherigen Abschnitt lässt sich durch die Verwendung einer Priority Queue statt einer Liste steigern. Der Algorithmus 4 zeigt den Pseudocode

einer Implementierung des Dijkstra-Algorithmus, die eine Priority Queue anstelle einer Liste verwendet und mithilfe eines Min-Heaps realisiert wird. Um die Verbesserung hervorzuheben, werden erneut die Laufzeiten des Algorithmus in Abhängigkeit von der Eingabegröße mithilfe der Landau-Notation [Cor+09, S. 47] angegeben. Die Zeilen 2 bis 8 umfassen den Initialisierungsteil dieser Implementierung und enthalten zunächst eine `foreach`-Schleife. Diese durchläuft jeden der n Knoten und führt dabei zwei Zuweisungen mit Laufzeit $O(1)$ aus. Diese `foreach`-Schleife hat somit eine Laufzeit von $O(n)$. Die Zuweisung in Zeile 6 und die Erstellung des leeren Heaps in Zeile 7 haben eine Laufzeit von je $O(1)$. Das Einfügen in den Heap in Zeile 8 hat eine Laufzeit von $O(\log n)$. Die Gesamtlaufzeit des Initialisierungsteils beträgt

$$L(n) = O(n) + 2 \cdot O(1) + O(\log n).$$

Die Laufzeiten $O(1)$ sowie $O(\log n)$ sind vernachlässigbar, sodass die Gesamtlaufzeit des ersten Abschnitts $O(n)$ beträgt. Der zweite Teil des Algorithmus besteht aus einer `while`-Schleife, die höchstens n -mal durchlaufen wird. Die Funktion `extractMin()` aus Zeile 10 hat eine Laufzeit von $O(\log n)$ und wird maximal n -mal aufgerufen. Dies ergibt eine Laufzeit von $O(n \cdot \log n)$. In Zeile 11 erfolgt eine einfache Zuweisung mit Laufzeit $O(1)$, die n -mal ausgeführt wird, sodass sich eine Laufzeit von $O(n)$ ergibt. Die Gesamtlaufzeit für den zweiten Teil beträgt somit $O(n \cdot \log n)$. Der dritte Abschnitt des Algorithmus umfasst die `foreach`-Schleife von Zeile 12 bis 18, die maximal m -mal durchlaufen wird, wobei m die Anzahl der Kanten des Eingabegraphen ist.

- Die Überprüfung, ob ein Knoten v bereits besucht wurde (Zeile 13), sowie die Auswertung der `if`-Bedingung (Zeile 14) haben jeweils die Laufzeit $O(1)$.
- Das Suchen einer Position im Heap (Zeile 15) und das Löschen eines bestimmten Elements (Zeile 16) haben jeweils eine Laufzeit von $O(\log n)$.
- Die Zuweisung in Zeile 17 hat eine Laufzeit von $O(1)$.
- Das Einfügen in den Heap in Zeile 18 erfolgt in $O(\log n)$.

Da diese drei Heap-Operationen, eine Zuweisung und zwei `if`-Bedingungen maximal m -mal ausgeführt werden, ergibt sich für den dritten Algorithmusabschnitt die Laufzeit

$$L(n, m) = O(m) \cdot (3 \cdot O(\log n) + 3 \cdot O(1)).$$

Nach Vernachlässigung der konstanten Faktoren und der Terme, die nicht von den Eingabegrößen abhängen, ergibt sich eine Laufzeit $O(m \cdot \log n)$. Die Zusammenführung der 3 Teillaufzeiten ergibt eine Laufzeit von

$$L(n, m) = O(n) + O(n \cdot \log n) + O(m \cdot \log n).$$

Nach Vernachlässigung der weniger dominanten Terme und konstanten Faktoren beträgt die Gesamtlaufzeit für die Implementierung des Dijkstra-Algorithmus mit einem Heap

$$L(n, m) = O((n + m) \cdot \log n).$$

Algorithm 4 Implementierung Dijkstra-Algorithmus mithilfe einer Priority Queue

```

1: DijkstraH(Gerichteter Graph  $G = (V, E)$ , Gewichtsfunktion  $\omega : E \rightarrow \mathbb{N}$ , Startknoten
    $s \in V$ )
2: for each  $v \in V$  do ▷  $O(n)$ 
3:   discovered[ $v$ ] ← false ▷  $O(1)$ 
4:    $d[v] \leftarrow \infty$  ▷  $O(1)$ 
5: end for
6:  $d[s] \leftarrow 0$  ▷  $O(1)$ 
7:  $H \leftarrow \emptyset$  ▷  $O(1)$ 
8:  $H.insert((s, d[s]))$  ▷  $O(\log n)$ 
9: while  $H.length() > 0$  do ▷  $O(n)$ 
10:   $u \leftarrow H.extractMin()$  ▷  $O(\log n)$ 
11:  discovered[ $u$ ] ← true ▷  $O(1)$ 
12:  for each  $(u, v) \in E$  do ▷  $O(m)$ 
13:    if not discovered[ $v$ ] then ▷  $O(1)$ 
14:      if  $d[v] > d[u] + \omega(u, v)$  then ▷  $O(1)$ 
15:        bestimme Position  $i$  von  $v$  in  $H$  ▷  $O(\log n)$ 
16:         $H.delete(i)$  ▷  $O(\log n)$ 
17:         $d[v] \leftarrow d[u] + \omega(u, v)$  ▷  $O(1)$ 
18:         $H.insert((v, d[v]))$  ▷  $O(\log n)$ 
19:      end if
20:    end if
21:  end for
22: end while

```

Die Tabelle 3.1 fasst die vorherigen Ergebnisse zusammen und zeigt die Laufzeitunterschiede zwischen den Dijkstra-Algorithmus-Implementierungen mit einer Liste und einer Priority Queue.

| Algorithmus Schritt | Dijkstra mit Liste | Dijkstra mit Priority Queue (Min-Heap) |
|---|--------------------|--|
| Initialisierung | $O(n)$ | $O(n)$ |
| Auswahl der Knoten und Berechnung der Distanzen | $O(n^2)$ | $O((n + m) \cdot \log n)$ |
| Gesamtlaufzeit | $O(n^2)$ | $O((n + m) \cdot \log n)$ |

Tabelle 3.1: Vergleich der Laufzeiten des Dijkstra-Algorithmus: Liste vs. Priority Queue

3.3 Implementierung von Lazy Deletion

Im Algorithmus 4 muss bei jeder Anpassung der Distanz die Position des Knotens im Heap bestimmt werden. Dieser wird dann gelöscht und mit der aktualisierten Distanz neu eingefügt. Im Gegensatz dazu entfernt der Algorithmus mit **Lazy Deletion** (siehe Algorithmus 5) keine Elemente mehr aus dem Heap, mit Ausnahme der Wurzel, sondern prüft für jeden Knoten, der per `extractMin()` aus dem Heap entfernt wird und als nächstes betrachtet werden soll, ob dieser bereits zuvor betrachtet wurde. Dies vereinfacht die Implementierung und reduziert Fehlerquellen, da Knoten nicht an beliebigen Positionen entfernt werden müssen. Einige Heap-Module, wie `heapq`, unterstützen das direkte Löschen von Knoten an beliebigen Positionen nicht, sodass eigene Methoden zur Löschung implementiert werden müssen. Solche Implementierungen können fehleranfällig sein, da etwa Knoten versehentlich zu früh entfernt werden könnten, was zu einer fehlerhaften Berechnung der kürzesten Pfade führen kann. Wird erkannt, dass der Knoten bereits betrachtet wurde, wird der aktuelle Schleifenaufruf direkt abgebrochen und der nächste Knoten wird per `extractMin()` aus dem Heap entfernt. Die Laufzeit wird durch Lazy Deletion nicht verbessert, da durch die Operation `insert()` immer noch ein $O(\log n)$ -Term in der `foreach`-Schleife verbleibt, die alle Ausgangskanten prüft. Die Laufzeit bleibt daher

$$L(n, m) = O((n + m) \cdot \log n).$$

Algorithm 5 Dijkstra-Algorithmus mit einer Priority Queue inkl. Lazy Deletion

```

1: DijkstraH(Gerichteter Graph  $G = (V, E)$ , Gewichtsfunktion  $w : E \rightarrow \mathbb{N}$ , Startknoten
    $s \in V$ )
2: for each  $v \in V$  do ▷  $O(n)$ 
3:   discovered[v] ← false, d[v] ← ∞ ▷  $O(1), O(1)$ 
4: end for
5:  $d[s] ← 0$ , Erstelle einen leeren Heap  $H$ ,  $H.insert((s, d[s]))$  ▷  $O(1), O(1), O(\log n)$ 
6: while  $H.length() > 0$  do ▷  $O(n)$ 
7:    $u ← H.extractMin()$  ▷  $O(\log n)$ 
8:   if discovered[u] then continue ▷  $O(1)$ 
9:   else
10:    discovered[u] ← true ▷  $O(1)$ 
11:   end if
12:   for each  $(u, v) \in E$  do ▷  $O(m)$ 
13:     if not discovered[v] then ▷  $O(1)$ 
14:       if  $d[v] > d[u] + w(u, v)$  then ▷  $O(1)$ 
15:          $d[v] ← d[u] + w(u, v)$  ▷  $O(1)$ 
16:          $H.insert((v, d[v]))$  ▷  $O(\log n)$ 
17:       end if
18:     end if
19:   end for
20: end while

```

4 Verwendete Bibliotheken und Werkzeuge

4.1 Tkinter – Das GUI-Toolkit von Python

Die im Rahmen dieser Arbeit entwickelte GUI basiert auf der Python-Version 3.13 und nutzt das standardmäßige GUI-Toolkit Tkinter [Fou24]. Die Vor- und Nachteile von TKinter basieren auf den Informationen von der Website [Fou25b].

Vorteile von Tkinter:

1. **Integriert in Python und plattformübergreifend** – Funktioniert ohne Anpassungen und zusätzliche Installationen auf Windows, macOS und Linux.
2. **Keine Lizenzprobleme** – Tkinter ist Teil der standardmäßigen Python-Distribution und kann daher ohne Einschränkungen genutzt werden.
3. **Einfach zu lernen und anzuwenden** – Grafische Oberflächen lassen sich schnell erstellen, ohne tiefgehende Kenntnisse in der GUI-Programmierung zu benötigen.

Nachteile von Tkinter:

1. **Begrenzte Gestaltungsmöglichkeiten** – Das Design wirkt veraltet und ist weniger anpassbar als Frameworks wie PyQt [Com24].
2. **Kein offizieller Benutzeroberflächen-Designer** – Es gibt keine offizielle Anwendung zum Entwerfen von Benutzeroberflächen, was die Entwicklung für komplexe GUIs erschwert.

Tkinter wurde gewählt, da es eine schnelle und einfache Möglichkeit bietet, die Benutzeroberfläche zu entwickeln, ohne externe Abhängigkeiten durch zusätzliche Frameworks einzuführen. Obwohl das Fehlen eines offiziellen GUI-Designers die Entwicklung komplexer GUIs erschwert, stellt dies für das Projekt kein Problem dar, da die Funktionalität im Vordergrund steht.

4.2 NetworkX

NetworkX [Dev24] ist eine Python-Bibliothek zur Erstellung, Visualisierung und Analyse von Graphen. In dieser Arbeit wird NetworkX zum Testen der Dijkstra-Algorithmen verwendet. Mit `nx.DiGraph()` lässt sich ein gerichteter Graph erstellen, dem Knoten per `add_node()` und Kanten per `add_edge()` hinzugefügt werden können. Die Methode `single_source_dijkstra_path_length(G, source_node)` berechnet die kürzesten Distanzen vom Startknoten `source_node` zu allen anderen Knoten des Graphen `G` mithilfe des Dijkstra-Algorithmus. Um die Korrektheit der drei implementierten Varianten des Dijkstra-Algorithmus zu überprüfen, werden deren berechnete Distanzen mit den von NetworkX berechneten Distanzen verglichen. Eine detaillierte Beschreibung der Testmethoden und -programme erfolgt in Kapitel 7.1.

5 GUI-Aufbau

Dieser Abschnitt beschreibt die im Rahmen dieser Arbeit entwickelte GUI. Sie besteht aus drei Hauptbereichen, die als Tkinter-Frames implementiert sind: dem **Main-Frame**, dem **Canvas-Frame** und dem **Pseudocode-Frame**. Diese werden in den folgenden Unterabschnitten detailliert erläutert.

5.1 Main-Frame

Der **Main-Frame** bildet das Hauptfenster, das beim Start des Programms geöffnet wird, und ist für das Layout verantwortlich. Mithilfe des Tkinter-Grid-Layouts werden der **Canvas-Frame**, der der Nutzerinteraktion dient, und der **Pseudocode-Frame**, der zur Visualisierung der Algorithmen verwendet wird, angeordnet.

5.2 Canvas-Frame

Der **Canvas-Frame** dient der visuellen Darstellung und Interaktion mit den Graphen. Dieser Frame ermöglicht die Interaktion des Nutzers mit dem Programm und besteht aus einer Menüleiste, einem Button-Frame und einem Canvas.

5.2.1 Canvas

Der **Canvas** (siehe Abbildung 5.1) stellt den aktuell geladenen Graphen dar, visualisiert die Algorithmisschritte farblich und ermöglicht es dem Nutzer, interaktiv Graphen zu erstellen. Auf dem Canvas sind die folgenden Interaktionen möglich:

- **Linksklick**: Erstellt einen Knoten an der Mausposition.
- **Doppelter Linksklick** auf einen Knoten: Setzt den angeklickten Knoten als Startknoten.
- **Rechtsklick** auf einen Knoten: Beginnt die Kantenerstellung.
- **Erneuter Rechtsklick** auf einen anderen Knoten: Erstellt eine Kante zwischen den beiden Knoten.

- **Rechtsklick** auf eine Kante: Öffnet ein Fenster, in dem der Nutzer das Gewicht der angeklickten Kante ändern kann.
- **Mittlerer Mausklick** oder **STRG + Linksklick** auf einen Knoten oder eine Kante: Löscht den Knoten oder die Kante an der Mausposition.
- **Linksklick halten** auf einen Knoten: Der Knoten folgt der Maus und kann an eine neue Position auf dem Canvas verschoben werden.

Knoten

Knoten werden als Ovale dargestellt, die mit der Tkinter-Methode `canvas.create_oval()` auf dem Canvas erzeugt werden. Die Knotennamen werden innerhalb des Ovals mithilfe von `canvas.create_text()` dargestellt. Optional kann unter dem Knotennamen die Distanz vom Startknoten angezeigt werden.

Kanten

Kanten bestehen aus zwei Linien, die mit der Tkinter-Methode `canvas.create_line()` auf dem Canvas gezeichnet werden. Eine Linie verläuft vom Ausgangspunkt der Kante bis knapp vor die Mitte. Das Gewicht der Kante wird mit der Methode `canvas.create_text()` an der Mitte zwischen Ausgangs- und Endpunkt dargestellt. Die zweite Linie beginnt leicht versetzt hinter der Mitte und verläuft bis zum Endpunkt der Kante. Um die Richtung der Kante anzuzeigen, wird am Ende der zweiten Linie ein Pfeil durch den `arrowshape`-Parameter hinzugefügt.

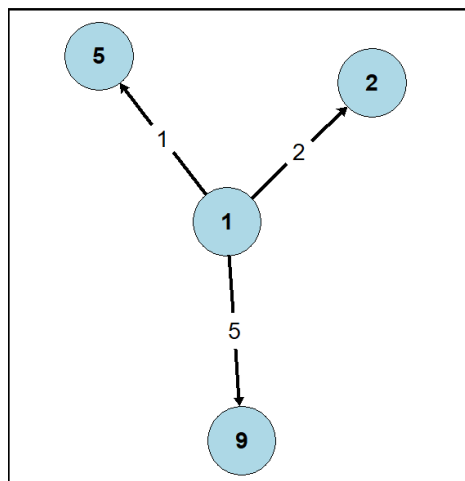


Abbildung 5.1: Canvas mit 4 Knoten und 3 Kanten

5.2.2 Menüleiste

Die **Menüleiste** (siehe Abbildung 5.2) besteht aus einem Tkinter-Menu mit vier Drop-down-Menüs, die jeweils durch die Tkinter-Methode `menu.add_cascade()` erzeugt werden:

- **Optionen:** Ermöglicht das Beenden des Programms sowie den Zugriff auf das Einstellungsfenster.
- **Graph Funktionen:** Bietet die folgenden Funktionen zur Verwaltung von Graphen:
 - **Lade Standardgraph:** Lädt einen als Standardgraph gespeicherten Graphen.
 - **Lösche Graph:** Löscht den aktuell geladenen Graphen.
 - **Importiere Graph:** Öffnet einen Dateidialog, über den der Nutzer einen Graphen im JSON-Format importieren kann.
 - **Exportiere Graph:** Öffnet einen Dateidialog, über den der Nutzer einen Graphen im JSON-Format exportieren kann.
- **Algorithmen:** Auswahl zwischen drei Varianten des Dijkstra-Algorithmus:
 - **Dijkstra mit Liste:** Verwendet eine Liste zur Verwaltung der noch zu bearbeitenden Knoten (siehe Abschnitt 3.1).
 - **Dijkstra mit Priority Queue:** Verwendet eine Priority Queue zur Verwaltung der noch zu bearbeitenden Knoten (siehe Abschnitt 3.2).
 - **Dijkstra mit Priority Queue (Lazy Deletion):** Erweitert die Dijkstra-Variante mit Priority Queue durch die Implementierung von Lazy Deletion (siehe Abschnitt 3.3).
- **Hilfe:** Öffnet ein Hilfsfenster, das die grundlegenden Funktionen des Programms erklärt.

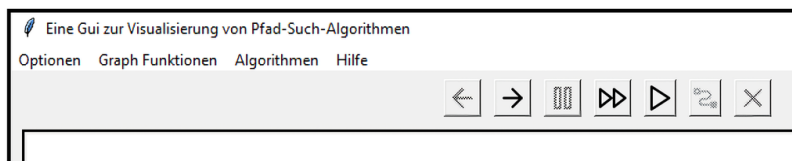


Abbildung 5.2: Menüleiste

5.2.3 Button-Frame

Der **Button-Frame** (siehe Abbildung 5.3) enthält sieben Buttons zur Steuerung der Algorithmusausführung. Die folgende Aufzählung beschreibt ihre Funktionen, wobei die tatsächlichen Schaltflächen nur Symbole enthalten:

- **1 Schritt zurück:** Springt einen Schritt in der schrittweisen Ausführung des Algorithmus zurück.
- **1 Schritt vor:** Springt einen Schritt in der schrittweisen Ausführung des Algorithmus nach vorne.
- **Pausieren:** Hält die automatische Ausführung an.
- **Vorspulen:** Startet die automatische Ausführung der Algorithmusschritte.
- **Algorithmus starten:** Startet den ausgewählten Algorithmus.
- **Kürzeste Pfade:** Öffnet ein Fenster mit Buttons zur Visualisierung der kürzesten Pfade.
- **Abbrechen:** Stoppt die laufende Algorithmuswiedergabe und setzt den Graphen zurück.

Falls kein Startknoten gesetzt wurde, fordert das Programm den Nutzer bei den Buttons **Vorspulen**, **1 Schritt vor** und **Algorithmus Starten** auf, einen Startknoten zu wählen.



Abbildung 5.3: Button-Frame

Die Icons für alle Buttons, außer **Kürzeste Pfade**, stammen von Feather Icons [Ico25], während das Icon für **Kürzeste Pfade** von Google Fonts [Fon25] bezogen wurde.

5.3 Pseudocode-Frame

Der **Pseudocode-Frame** ist ein nicht interaktiver Bereich zur Visualisierung der einzelnen Algorithmusschritte und der entsprechenden Ergebnisse. Er besteht aus einem Pseudocode-Text, einer Distanztabelle und einer grafischen Darstellung der verwendeten Datenstruktur während der Ausführung der Dijkstra-Algorithmen.

5.3.1 Pseudocode-Text

Der Pseudocode wird durch das Tkinter-Widget `Text` mit dem Status `DISABLED` dargestellt (siehe Abbildung 5.4). Er basiert auf der Vorlesung Datenstrukturen und Algorithmen im Wintersemester 2024/25 und hebt den jeweils aktuellen Schritt während der Visualisierung des Dijkstra-Algorithmus farblich hervor.

```

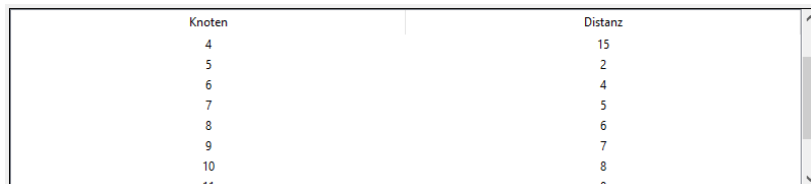
1:DijkstraH(Gerichteter Graph  $G = (V, E)$ ,
  Gewichtsfunktion  $\omega : E \rightarrow \mathbb{N}$ , Startknoten  $s \in V$ ):
2:  foreach  $v \in V$  do discovered[v]  $\leftarrow$  false,  $d[v] \leftarrow \infty$ 
3:   $d[s] \leftarrow 0$ , Heap  $H \leftarrow \emptyset$ ,  $H.insert((s, d[s]))$ 
4:  while  $H.length() > 0$  do
5:     $u \leftarrow H.extractMin()$ 
6:    if discovered[u] then continue else
      discovered[u]  $\leftarrow$  true
7:    forall  $(u, v) \in E$  do
8:      if not discovered[v] then
9:        if  $d[v] > d[u] + \omega(u, v)$  then
10:          $d[v] \leftarrow d[u] + \omega(u, v)$ 
11:          $H.insert((v, d[v]))$ 

```

Abbildung 5.4: Pseudocode-Text von Dijkstra mit Priority Queue (Lazy Deletion)

5.3.2 Distanztabelle

Die Distanztabelle wird durch das Tkinter-Widget Treeview (siehe Abbildung 5.5) dargestellt und zeigt die Distanzen vom Startknoten zu den jeweiligen Knoten.



| Knoten | Distanz |
|--------|---------|
| 4 | 15 |
| 5 | 2 |
| 6 | 4 |
| 7 | 5 |
| 8 | 6 |
| 9 | 7 |
| 10 | 8 |
| 11 | 9 |

Abbildung 5.5: Distanztabelle

5.3.3 Datenstruktur-Visualisierung

Die Visualisierung der Datenstruktur erfolgt über einen Tkinter-Canvas, der die Datenstruktur der aktuell geladenen Implementierung darstellt. Falls eine Liste genutzt wird, werden die Elemente als Tkinter-Rectangles mit der Methode `canvas.create_rectangle()` dargestellt (siehe Abbildung 5.6). Der Text innerhalb jedes Listenelements gibt den jeweiligen Knoten und seinen zugehörigen Distanzwert an. Im Fall der Verwendung einer Priority Queue wird die Datenstruktur als Binärbaum visualisiert, wobei die einzelnen Elemente als Tkinter-Ovals mit der Methode `canvas.create_oval()` dargestellt werden (siehe Abbildung 5.7). Der Text innerhalb jedes Knotens enthält ein Tupel $(v, d[v])$, wobei v den Namen des Knotens und $d[v]$ die Distanz vom Startknoten zu v angibt.

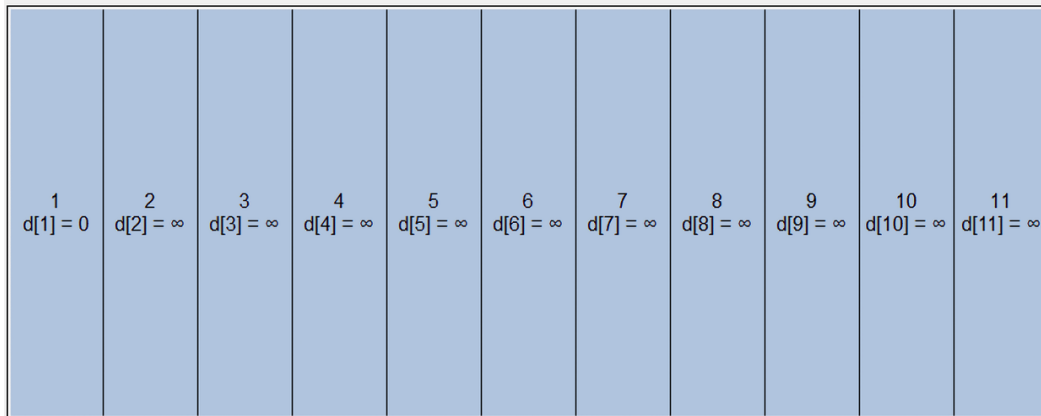


Abbildung 5.6: Datenstruktur Visualisierung für eine Liste

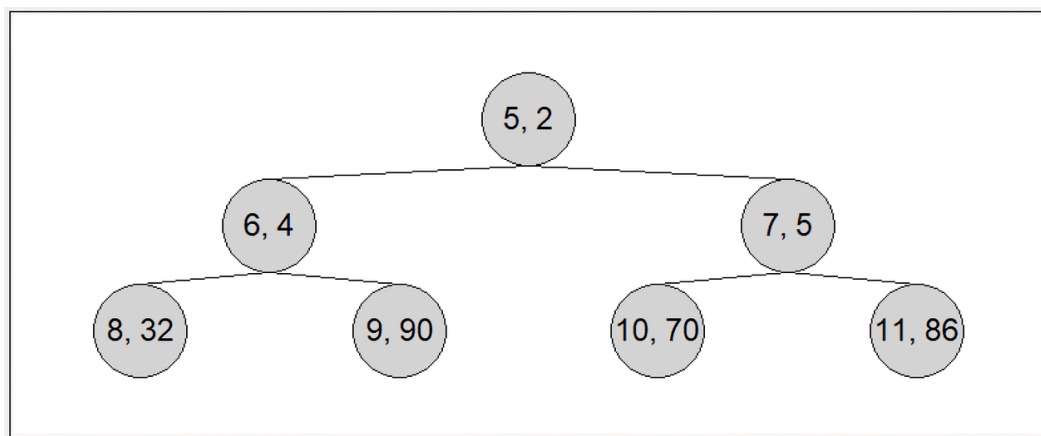


Abbildung 5.7: Datenstruktur Visualisierung für einen Heap

6 Implementierung

Dieses Kapitel beschreibt die wesentlichen Aspekte der Implementierung des Programms. Zunächst wird der gewählte objektorientierte Ansatz erläutert. Anschließend wird die Dateistruktur des Programms vorgestellt, um die Aufteilung der verschiedenen Komponenten und deren Aufgaben zu verdeutlichen. Danach folgt eine Erklärung der wichtigsten Methoden, die für die Umsetzung der GUI implementiert wurden. Abschließend werden während der Implementierung aufgetretene Probleme aufgezeigt.

6.1 Objektorientierter Ansatz

Bei der Implementierung des Programms wurde ein objektorientierter Ansatz gewählt, bei dem Daten und Funktionalitäten in Klassen gebündelt werden [Fou25a]. Die wichtigsten Komponenten des Programms, darunter die Algorithmen, die grafische Visualisierung und die GUI, sind als Klassen implementiert, was es ermöglicht, die Komponenten unabhängig voneinander zu entwickeln. Zudem kann das Programm erweitert werden, indem weitere Algorithmus- und Visualisierungsklassen implementiert und in die `Main_Frame`-Klasse integriert werden. Diese Struktur erleichtert zudem die Wartung, da Änderungen gezielt an einzelnen Klassen vorgenommen werden können.

6.2 Dateistruktur

Das Programm ist in sechs Ordner unterteilt, die eine Trennung der Funktionalitäten ermöglichen:

- **algorithms:** Enthält die Klassen `dijkstra_list`, `dijkstra_Priority_queue` und `dijkstra_Priority_queue_lazy`. Diese Klassen implementieren die drei Varianten des Dijkstra-Algorithmus.
- **graph_visualizer:** Beinhaltet die Klassen `graph_visualizer_dijkstra_list`, `graph_visualizer_dijkstra` und `graph_visualizer_dijkstra_lazy`, die die Visualisierung des aktuellen Graphen übernehmen. Abhängig vom gewählten Algorithmus werden verschiedene Elemente hervorgehoben. Zudem enthält dieser Ordner die Klasse `graph_visualizer_path`, die speziell für die Darstellung der kürzesten Pfade innerhalb des Graphen zuständig ist.

- **gui**: Enthält die Klassen `Main_Frame`, `Canvas_Frame` und `Pseudocode_Frame`, die für den Aufbau der GUI zuständig sind (siehe Kapitel 5).
- **save_files**: Dient als Standardverzeichnis für den Import und Export von Graphen.
- **icons**: Enthält die PNG-Dateien der Icons, die für die Buttons verwendet werden (siehe Kapitel 5.2.3).
- **tests**: Enthält die Testprogramme `dijkstra_test` und `heap_test` (siehe Kapitel 7)

Die Datei `pfadsuche` ist der Haupteinstiegspunkt des Programms und wird vom Nutzer zum Starten verwendet. Die Datei `Config` enthält verschiedene Einstellungen im JSON-Format, darunter Farbeinstellungen, die Animationsverzögerung, die Schriftgröße sowie das maximale Kantengewicht für zufällig generierte Gewichte. Diese können über das Einstellungsfenster festgelegt werden und werden beim Programmstart geladen.

6.3 Schrittweise Wiedergabe der Algorithmusausführungen

Das Programm unterstützt eine schrittweise Visualisierung der Algorithmen. Dies geschieht durch die Methode `save_state()` in den Algorithmus-Klassen, die den aktuellen Zustand des Algorithmus in einem `state`-Dictionary speichert und der `steps`-Liste hinzufügt. Durch das Aufrufen von `save_state()` nach jedem Algorithmusschritt wird eine Liste von `steps` erstellt, die den gesamten Dijkstra-Algorithmus abbildet. Diese Liste wird anschließend in der Variable `steps_finished_algorithm` der `Main_Frame`-Klasse gespeichert. Die Variable `current_step` in der `Main_Frame`-Klasse gibt den aktuellen Stand der Wiedergabe an. Der Nutzer kann diesen Stand über Buttons (siehe Kapitel 5.2.3) steuern, die mit den entsprechenden Methoden in der `Main_Frame`-Klasse verbunden sind.

- Die Methode `prev_step()` verringert den Wert von `current_step` um 1.
- Die Methode `next_step()` erhöht den Wert von `current_step` um 1.
- Die Methode `fast_forward()` erhöht `current_step` automatisch in einem vom Nutzer definierten Intervall.

Für die automatische Wiedergabe prüft `fast_forward()`, ob die Variable `fast_forward_paused` auf `false` gesetzt ist. Falls ja, wird die Methode mithilfe der Standard-Python-Methode `after()` erneut aufgerufen. Mit `pause()` kann die Wiedergabe gestoppt werden, indem `fast_forward_paused` auf `true` gesetzt wird. Nach jeder Änderung der Variable `current_step` ruft das Programm die Methode `update_gui()` auf. Diese aktualisiert die Benutzeroberfläche, indem sie die gespeicherten Daten aus `steps_finished_algorithm[current_step]` lädt und darstellt.

6.4 Animationen der Heap-Operationen

In diesem Abschnitt werden die Animationen der Heap-Operationen beschrieben, die in dem Programm implementiert wurden. Diese Animationen ermöglichen eine schrittweise Visualisierung des Entfernens des Minimums, des Einfügens eines neuen Elements sowie des Löschens eines bestimmten Elements.

Entfernen des Minimums

Die Methode `pop_min_animation()` startet die Animation des Entfernens des Minimums aus dem Heap. Der Ablauf der Animation ist wie folgt:

- **Initialisierung:** Die Methode `stop_animation()` wird aufgerufen, um eine mögliche vorherige Animation zu stoppen. Danach wird die Variable `animation_step`, die den aktuellen Stand der Animation speichert, auf 0 gesetzt und die Animationsschritte werden gestartet.
- **Erster Schritt:** Im ersten Schritt wird der Wurzelknoten, der das Minimum des Heaps darstellt, markiert. Dies geschieht durch den Aufruf von `draw_priority_queue()` mit dem Wurzelknoten als optionalem Parameter. Die Animation wird durch das Aufrufen von `canvas.after(1000, run_pop_min_step)` nach einer Verzögerung von einer Sekunde fortgeführt.
- **Zweiter Schritt:** Ein leerer Knoten wird an die Position des Wurzelknotens gesetzt und der Heap wird durch `draw_priority_queue()` neu gezeichnet. Dieser leere Knoten stellt den gelöschten Wurzelknoten dar.
- **Dritter Schritt:** Das letzte Element des Heaps wird an die Stelle des Wurzelknotens verschoben und der Heap wird durch `draw_priority_queue()` erneut gezeichnet.
- **Vierter Schritt:** Der vierte Schritt animiert die `Heapify_Down()`-Operation (siehe Algorithmus 2), bei der der neue Wurzelknoten durch rekursives Vergleichen und Tauschen an seine korrekte Position verschoben wird, sodass die Heap-Eigenschaft wiederhergestellt ist. Nach jeder Vergleichs- und Tauschoperation wird der Heap durch `draw_priority_queue()` erneut gezeichnet.

Einfügen eines neuen Elements

Das Einfügen eines neuen Elements wird durch die Methode `insert_animation()` visualisiert. Der Ablauf ist wie folgt:

- **Initialisierung:** Zuerst werden mögliche laufende Animationen gestoppt und `animation_step` wird auf 0 gesetzt. Anschließend wird die Methode `run_insert_step` aufgerufen, um die Animation zu starten.

- **Erster Schritt:** Im ersten Schritt wird der aktuelle Zustand des Heaps vor dem Einfügen des neuen Elements angezeigt.
- **Zweiter Schritt:** Das neue Element wird an der letzten Position im Heap eingefügt und hervorgehoben.
- **Dritter Schritt:** Nachdem das Element eingefügt wurde, wird die `heapify_up_step`-Methode aufgerufen, um das neue Element an die richtige Position im Heap zu verschieben. Dies wird durch rekursives Vergleichen und Tauschen mit den Elternknoten erreicht (siehe Algorithmus 1). Während jeder Vergleichs- und Tauschoperation wird erneut `draw_priority_queue()` aufgerufen, um den Heap neu zu zeichnen und die Veränderung darzustellen.

Entfernen eines bestimmten Elements

Das Entfernen eines bestimmten Elements und die anschließenden Heapify-Operationen werden durch die Methode `remove_node_heapify_animation()` animiert. Der Ablauf ist wie folgt:

- **Initialisierung:** Die möglicherweise noch laufenden Animationen werden gestoppt, `animation_step` wird auf 0 gesetzt und `run_remove_node_heapify_step()` wird aufgerufen.
- **Erster Schritt:** Das zu entfernende Element wird im Heap hervorgehoben.
- **Zweiter Schritt:** Das zu entfernende Element wird entfernt und durch einen leeren Knoten ersetzt.
- **Dritter Schritt:** Das letzte Elements im Heap ersetzt den leeren Knoten.
- **Vierter Schritt:** In diesem Schritt wird die `Heapify_Down()`-Operation (siehe Algorithmus 2) ausgehend von der ursprünglichen Position des gelöschten Knoten animiert, während der Heap nach jedem Schritt neu gezeichnet wird.

6.5 Graph-Darstellung

Die Darstellung des Graphen erfolgt über die `graph_visualizer`-Klassen. Die `update_gui()`-Methode erstellt Objekte der `graph_visualizer`-Klassen mit den Parametern `node_positions`, der die Koordinaten der einzelnen Knoten speichert, `graph`, der den aktuellen Graphen als Adjazenzliste speichert, und `start_node`, der den aktuellen Startknoten speichert. Anschließend ruft `update_gui()` die jeweilige `draw_graph()`-Methode der `graph_visualizer`-Klassen auf, die, je nach ausgewähltem Algorithmus, den Graphen auf

dem Canvas der `Canvas_Frame`-Klasse zeichnet. Die `draw_graph()`-Methoden iterieren zunächst über alle Knoten in `node_positions` und zeichnen die Knoten an ihren Koordinaten auf dem Canvas. Der Startknoten wird durch ein zusätzliches Oval mit einem um fünf Pixel vergrößerten Radius und einer gestrichelten Linie dargestellt. Dies wird durch den Parameter `dash=(3, 3)` der `canvas.create_oval()`-Methode erreicht. Die Farbe der Knoten wird über den aktuellen Algorithmusschritt bestimmt: Aktuell betrachtete Knoten werden farblich hervorgehoben, während nicht betrachtete Knoten ausgegraut dargestellt werden. Anschließend iteriert die Methode über alle Kanten in der Adjazenzliste `graph` und zeichnet die Kanten auf dem Canvas. Die Kanten und Kantengewichte werden ebenfalls abhängig vom aktuellen Algorithmusschritt eingefärbt und ausgegraut, um die aktuell betrachteten Kanten hervorzuheben.

6.6 Skalierung der GUI

Ein wichtiger Punkt der GUI-Programmierung ist die Skalierung, da sich das Programm an verschiedene Bildschirmgrößen und Seitenverhältnisse anpassen muss. Die Tkinter-Widgets unterstützen standardmäßig das Skalieren durch das Grid-Layout. Allerdings skaliert der Canvas zwar in seiner Größe, jedoch reicht eine einfache Anpassung seiner Abmessungen nicht aus, da die Koordinaten und Größen der darauf gezeichneten Elemente standardmäßig nicht skaliert werden. Dies führt dazu, dass der Graph auf kleinen Bildschirmen über den sichtbaren Bereich des Canvas hinausgeht und die Knoten zu groß dargestellt werden. Auf großen Bildschirmen hingegen wird der Graph nur in einem kleinen Teil des Canvas angezeigt und die Knoten erscheinen zu klein. Um dieses Problem zu lösen, wurde eine Methode `resize_canvas()` implementiert. Diese wird nach jeder Größenänderung des Canvas aufgerufen und berechnet zunächst die Skalierungsfaktoren `scale_x` und `scale_y`, indem die neue Breite bzw. Höhe des Canvas durch die vorherige Breite bzw. Höhe geteilt wird. Anschließend wird die Methode `scale_node_positions(scale_x, scale_y)` aufgerufen, die über alle Knoten des Graphen iteriert und deren neue Positionen berechnet. Die neue X-Position eines Knotens ergibt sich aus der Multiplikation der aktuellen X-Position mit dem Faktor `scale_x`. Entsprechend wird die neue Y-Position durch Multiplikation der aktuellen Y-Position mit dem Faktor `scale_y` bestimmt. Nach der Berechnung der neuen Knotenpositionen wird die Methode `scale_node_size_absolute()` aufgerufen, die sowohl die Knotengröße als auch die Schriftgröße der Knotenbeschriftungen anpasst. Dazu werden zunächst die Skalierungsfaktoren `scale_x` und `scale_y` ermittelt, indem die aktuellen Canvas-Dimensionen durch die ursprünglichen Dimensionen des Canvas geteilt werden. Anschließend wird der Skalierungsfaktor `average_scale` als Durchschnitt von `scale_x` und `scale_y` berechnet. Schließlich werden die Knotengröße und die Schriftgröße durch die Multiplikation ihrer aktuellen Werte mit `average_scale` skaliert. Der Canvas, auf dem die Datenstruktur visualisiert wird, ist

mit einer `on_resize()`-Methode verbunden, die bei jeder Änderung der Seitenverhältnisse den Binärbaum bzw. die Liste neu zeichnet. Die Größe der Knoten und Listenelemente ist dabei abhängig von der Größe des Canvas und der Anzahl der darzustellenden Elemente. Die Schriftgröße des Pseudocodes wird auf eine ähnliche Weise skaliert. Aus Performancegründen wurde jedoch die Skalierung des Pseudocodes auf alle 100 ms begrenzt. Zudem kann die Schriftgröße weiterhin manuell über die Optionen eingestellt werden.

6.7 Probleme

In diesem Abschnitt werden die während der Implementierung des Programms aufgetretenen Probleme beschrieben.

Die Möglichkeit, bidirektionale Kanten zu erstellen, musste aus Zeitgründen entfernt werden. Stattdessen wird bei der Erstellung einer neuen Kante die bereits bestehende Kante überschrieben. Diese Entscheidung war notwendig, da die entwickelte Logik zur Erkennung von bidirektionalen Kanten nicht korrekt funktionierte, was zu Problemen bei der zuverlässigen Erkennung von Kantenlöschungen sowie bei der Änderung von Kantengewichten führte.

Des Weiteren kann es bei sehr kurzen Kanten zu einer fehlerhaften Darstellung des Kantenpfeils kommen. In solchen Fällen reicht der Platz nicht aus, um die Kante korrekt darzustellen, sodass der Kantenpfeil innerhalb des Knotens mit der falschen Richtung angezeigt wird. Eine direkte Lösung für dieses Problem wurde in dem Zeitraum dieser Arbeit nicht gefunden. Das Problem lässt sich jedoch leicht vermeiden, indem die Knoten weiter auseinandergezogen werden.

Ein weiteres Problem trat im Umgang mit der Konfigurationsdatei `Config` auf. Manuelle Änderungen an dieser Datei können unerwartetes Verhalten verursachen. Die Standardwerte können jedoch jederzeit durch Löschen der Datei `Config` wiederhergestellt werden. Falls beim Programmstart keine `Config`-Datei geladen werden kann, erstellt das Programm automatisch eine neue `Config`-Datei mit den Standardwerten.

7 Testen des Programms

Dieses Kapitel beschreibt die Methoden und Programme, die entwickelt wurden, um das Programm zu testen. Der erste Abschnitt beschreibt das Testen der Dijkstra-Algorithmen, der zweite Abschnitt beschreibt das Testen der Heap-Operationen und der dritte Abschnitt beschreibt das manuelle Testen der GUI. Alle Tests wurden erfolgreich durchgeführt und bestätigen die korrekte Funktionsweise des Programms.

7.1 Testen der Dijkstra-Algorithmen

Das Testen der Dijkstra-Algorithmen erfolgt über das `Dijkstra_test.py`-Programm innerhalb des `tests`-Ordners. Dafür wurden Testgraphen mit verschiedenen Edge-Cases erstellt, darunter:

- Graphen mit nur einem Knoten,
- Graphen mit mehreren Knoten ohne Kanten,
- Graphen, die einen Kreis bilden,
- große Graphen (z. B. mit 100 Knoten und zufällig generierten Kanten).

Für jeden dieser Edge-Cases erstellt die Methode einen `DiGraph` des `NetworkX`-Frameworks, iteriert über alle Knoten und Kanten des Edge-Case-Graphen und fügt diese dem `DiGraph` hinzu. Anschließend berechnet `single_source_dijkstra_path_length(G, source_node)` die Distanzen von allen Knoten des `NetworkX`-Graphen `G` zum gewählten Startknoten `source_node`. Die gleichen Distanzen werden mit den implementierten Dijkstra-Algorithmen berechnet. Abschließend werden die Ergebnisse der eigenen Implementierung mit den Ergebnissen des `NetworkX`-Frameworks verglichen. Die Tests gelten als bestanden, wenn die Ergebnisse übereinstimmen. Eine weitere Möglichkeit, die Dijkstra-Algorithmen zu testen, besteht darin, die Variable `self.enable_tests` innerhalb der `Main-Frame`-Klasse auf `True` zu setzen. Dadurch werden bei jedem Start eines Algorithmus die Distanzen auf die gleiche Weise wie im zuvor beschriebenen Testprogramm `Dijkstra_test.py` überprüft und die Ergebnisse in der Konsole ausgegeben.

7.2 Testen der Heap-Operationen

Für die Visualisierung der Heap-Operationen wurden eigene Heapify-Up- und Heapify-Down-Algorithmen implementiert. Der Grund für die Implementierung eigener Operationen war, dass das in den Dijkstra-Implementierungen genutzte `heapq`-Modul keine schrittweise Durchführung von Heapify-Up oder Heapify-Down ermöglicht, was für die Visualisierung der Priority Queue erforderlich ist. Um die Funktionsweise der implementierten Algorithmen zu überprüfen und sicherzustellen, dass sie korrekt arbeiten, wurde ein Testprogramm `heap_test.py` entwickelt. In diesem Programm werden zwei Heaps erstellt: ein Heap, der mit den eigenen Heapify-Up- und Heapify-Down-Operationen arbeitet, und ein zweiter Heap, der auf dem `heapq`-Modul basiert. Anschließend fügt das Testprogramm mehrere Elemente in beide Heaps ein und löscht mehrfach die Wurzeln beider Heaps. Nach jeder `insert()`- und `pop_min()`-Operation wird überprüft, ob beide Heaps übereinstimmen. Falls dies der Fall ist, dann gelten die `insert()`- und `pop_min()`-Tests als bestanden. Um das Entfernen eines spezifischen Knotens zu testen, wird überprüft, ob nach dem Entfernen sowohl die Heap-Eigenschaft (siehe Kapitel 2.3.1.3) als auch die Vollständigkeit des Binärbaums erhalten bleibt. Der Grund für diesen Ansatz ist, dass es in `heapq` keine direkte Möglichkeit gibt, ein spezifisches Element zu löschen. Der gängige Weg ist, das Element aus der Priority Queue zu entfernen und dann den Heap komplett neu mit `heapq.heapify()` zu erstellen. Um die Löschung zu animieren, wurde jedoch eine andere Implementierung gewählt: Das zu löschende Element wird mit dem letzten Element des Heaps getauscht, dort gelöscht und das getauschte Element wird per `heapify_down()` an die korrekte Stelle verschoben. Die Tests gelten als bestanden, wenn der Heap nach dem Löschen ein valider Min-Heap ist.

7.3 Manuelles Testen der GUI

Die GUI wurde durch manuelle Tests überprüft. Dabei wurden sämtliche Interaktionen auf dem Canvas getestet, einschließlich des Erstellens, Löschens und Verschiebens von Knoten, der Erstellung von Kanten, der Änderung von Kantengewichten sowie der farblichen Visualisierung der Algorithmusschritte. Darüber hinaus wurden die verschiedenen Optionen und Graph-Funktionen getestet, darunter der Import und Export von Graphen, das Löschen eines Graphen sowie das Laden des Standardgraphen.

8 Zusammenfassung und Ausblick

In dieser Bachelorarbeit wurde eine GUI entwickelt, die es den Studierenden ermöglicht, den Ablauf des Dijkstra-Algorithmus mit verschiedenen Datenstrukturen interaktiv nachzuvollziehen. Neben der Visualisierung des Algorithmus werden auch die zugrunde liegenden Datenstrukturen, insbesondere die Operationen des Heaps, anschaulich dargestellt. Die Anwendung bietet den Nutzern die Möglichkeit, eigene Graphen zu erstellen, zu speichern und bereits gespeicherte Graphen zu laden. Durch die schrittweise Darstellung der Algorithmen und Datenstruktur-Operationen wird das Verständnis für deren Funktionsweise und Anwendung gefördert.

Für die Zukunft bieten sich verschiedene Erweiterungsmöglichkeiten. So könnten weitere Pfadsuchalgorithmen, wie der A*-Algorithmus oder der Bellman-Ford-Algorithmus, implementiert werden. Ein weiterer zukünftiger Schritt wäre die Modernisierung der GUI. Dies könnte entweder durch den vollständigen Umstieg auf ein modernes Framework oder durch den Einsatz von Tkinter-Theme-Paketen erfolgen, die es ermöglichen, die GUI visuell ansprechender zu gestalten.

Literatur

- [Com24] Riverbank Computing. *PyQt - Python Bindings for Qt*. letzter Zugriff am: 30. Januar 2025. 2024. URL: <https://www.riverbankcomputing.com/software/pyqt/intro>.
- [Cor+09] Thomas H. Cormen u. a. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: <http://mitpress.mit.edu/books/introduction-algorithms>.
- [Dev24] NetworkX Developers. *NetworkX - Network Analysis in Python*. letzter Zugriff am: 30. Januar 2025. 2024. URL: <https://networkx.org/>.
- [Dij59] Edsger W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische Mathematik 1* (1959), S. 269–271. DOI: 10.1007/BF01386390. URL: <https://doi.org/10.1007/BF01386390>.
- [Fon25] Google Fonts. *Google Fonts Icons*. <https://fonts.google.com/icons>. Zugriff am 21. Februar 2025. 2025.
- [Fou24] Python Software Foundation. *Tkinter - Standard GUI Library for Python*. letzter Zugriff am: 30. Januar 2025. 2024. URL: <https://docs.python.org/3/library/tkinter.html>.
- [Fou25a] Python Software Foundation. *Python Documentation: 9. Classes*. Letzter Zugriff am: 10. Februar 2025. 2025. URL: <https://docs.python.org/3/tutorial/classes.html>.
- [Fou25b] Python Software Foundation. *Tkinter: Python's GUI Toolkit*. Zugriff am 5. März 2025. 2025. URL: <https://www.pythonguis.com/faq/pyqt-vs-tkinter/>.
- [Ico25] Feather Icons. *Feather Icons*. <https://feathericons.com/>. Zugriff am 21. Februar 2025. 2025.
- [KT06] Jon M. Kleinberg und Éva Tardos. *Algorithm design*. Addison-Wesley, 2006. ISBN: 978-0-321-37291-8.
- [Mor13] Pat Morin. *Open Data Structures*. <https://www.aupress.ca/books/120226-open-data-structures/>. letzter Zugriff am: 02. März 2025. 2013.

[OW12] Thomas Ottmann und Peter Widmayer. *Algorithmen und Datenstrukturen*, 5. Auflage. Spektrum Akademischer Verlag, 2012. ISBN: 978-3-8274-2803-5. DOI: 10.1007/978-3-8274-2804-2. URL: <https://doi.org/10.1007/978-3-8274-2804-2>.