Berufsakademie Sachsen
Staatliche Studienakademie Leipzig

Prim Algorithmus zur Berechnung eines minimalen Spannbaumes

Hausarbeit
zur Erlangung des akademischen Grades eines
Bachelor of Science (B. Sc.)
in der Studienrichtung Informatik

Eingereicht von: Simon Nikolaidis
[redacted]

Gutachter/Dozent: Prof. Dr. habil. Holger Perlt

Leipzig, den 06.03.2023

Inhaltsverzeichnis

1 Einleitung 3

2 Algorithmus von Prim 3

2.1 Minimaler Spannbaum 3

2.2 Funktionsweise 4

2.3 Laufzeitkomplexität 5

2.4 Unterschied zum Kruskal Algorithmus 6

3 Anwendung 6

3.1 Voraussetzungen 6

3.2 Anwendungsfelder 7

4 Einsatz in einem Beispielprogramm 7

4.1 Nutzung des Programms 7

4.2 Erläuterung der Funktionsweise 8

5 Zusammenfassung 10

Quellenverzeichnis 11

Abkürzungsverzeichnis 11

Selbstständigkeitserklärung 12

1 Einleitung

Diese Arbeit befasst sich mit dem Prim Algorithmus. Das zugrundeliegende Thema ist in den Bereich Algorithmen und Datenstrukturen, genauer in der Graphentheorie einzuordnen. Zuerst sollen die relevanten Grundlagen erläutert werden, welche notwendig sind, um den Algorithmus verstehen zu können. Anschließend soll der Algorithmus selbst sowie seine Eigenschaften und Voraussetzungen zur Anwendung beschrieben werden. Zum Schluss soll der Algorithmus dann noch in einem praktischen Beispielprogramm implementiert werden, mit welchem ein korrekter MST zu einem vorgegebenen Graphen der Form Knoten1 Knoten2 Gewicht12 erzeugt werden.

2 Algorithmus von Prim

Der Algorithmus von Prim wird verwendet, um einen minimalen Spannbaum (engl.: „minimum spanning tree“ – MST) für einen endlichen, zusammenhängenden, kantengewichteten und ungerichteten Graphen zu ermitteln.

Der Algorithmus wurde erstmals im Jahr 1930 von dem tschechischen Mathematiker Vojtěch Jarník entwickelt [1] und später von dem amerikanischen Mathematiker und Informatiker Robert C. Prim im Jahr 1957 [2] und dem niederländischen Informatiker Edsger W. Dijkstra im Jahr 1959 [3] wiederentdeckt und neu veröffentlicht. Aus diesem Grund findet man den Algorithmus ebenfalls unter den Bezeichnungen: „Algorithmus von Jarnik, Prim und Dijkstra“ oder „Prim-Dijkstra-Algorithmus“, sowie „Jarnik’s algorithm“ oder „DJP algorithm“ im englisch sprachigen Raum.

2.1 Minimaler Spannbaum

Ein aufspannender Baum oder auch **Spannbaum** eines zusammenhängenden ungerichteten Graphens G = (V, E) ist ein Teilgraph T = (V, E‘) von G, der ein Baum ist und alle Knoten von G enthält. Es handelt sich also um einen Baum T, welcher den Graphen G „aufspannt“. [4]

Unter einem **Baum** versteht man eine spezielle Form eines Graphens, der zusammenhängend ist und keine geschlossenen Pfade beziehungsweise Kreise enthält. Es gelten also folgende Bedingungen [5]:

- Durch das Hinzufügen einer Kante in einem Baum, die zwei Knoten verbindet, entsteht ein Zyklus.
- Wird eine Kante aus einem Baum entfernt, so entstehen zwei separate Teilbäume.

Gegeben sei ein ungerichteter Graph G = (V, E) mit einer Kantengewichtung w. Das Gewicht des Graphens w(G) wird definiert als die Summe der Gewichte aller Kanten im Graphen. Ein

Spannbaum T = (V, E‘) von G heißt **minimaler Spannbaum** von G, wenn sein Gewicht minimal ist. Anders ausgedrückt, für alle anderen Spannbäume T‘ von G gilt: Das Gewicht von T‘ ist größer oder gleich dem Gewicht von T ($w(T') \geq w(T)$). [4] Das bedeutet auch, dass es für einen Graphen mehrere minimale Spannbäume geben kann.

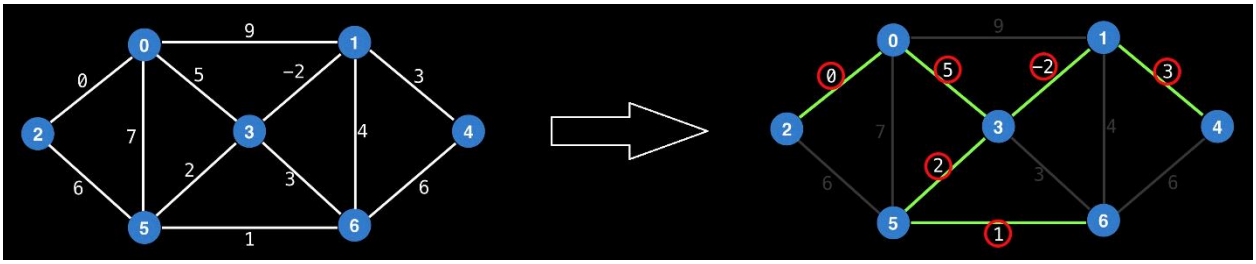


Abbildung 1: Graph mit einem dazugehörigen MST [6]

2.2 Funktionsweise

Gegeben ist ein Graph G = (V, E, w):

- Der Spannbaum T beginnt mit einem einzelnen Knoten v_i: T = {v_i}
- Solange T weniger Knoten als V enthält, wird die Kante e ∈ E mit dem geringsten Gewicht (w) gesucht, die T mit G \ T verbindet, und diese Kante wird zu T hinzugefügt. [5]

Man kann hierbei zwischen einer „lazy“ und einer „eager“ Version des Algorithmus unterscheiden.

Lazy Implementierung:

Für diese Art der Implementierung wird eine min Priority Queue (PQ) verwendet. Die Priority Queue wird als Puffer verwendet, um Objekte in einer bestimmten Reihenfolge zwischenzuspeichern, bevor sie weiterverarbeitet werden. Eine solche Priority Queue kann man sich wie eine Liste vorstellen, in welcher Werte gespeichert werden können. Beim Entnehmen der Werte aus dieser Queue, werden diese entsprechend ihrer Priorität ausgegeben. Das bedeutet, das Element mit der höchsten Priorität wird zuerst entnommen. In der Regel wird eine Priority Queue mit einem Heap implementiert. *(Heap: Binärbaum, bei dem jeder Knoten entweder größer/gleich oder kleiner/gleich seiner Kinder ist.)*

Im Fall des Algorithmus von Prim, wird die PQ verwendet, um die Kanten basierend auf ihrer minimalen Gewichtung zu sortieren. Dadurch kann ermittelt werden, welcher Knoten als nächstes besucht wird und welche Kante genutzt wird, um dorthin zu gelangen.

Zu Beginn des Algorithmus wird ein beliebiger Knoten s ausgewählt und als besucht markiert. Anschließend wird über alle Kanten von s iteriert und jede dieser Kanten wird zur Priority Queue hinzugefügt.

Solange die PQ nicht leer ist und noch kein MST gebildet wurde – d.h. solange der MST weniger

Knoten als der ursprüngliche Graph enthält – wird die nächst „leichtere“ Kante aus der Warteschlange entnommen. Zeigt diese Kante auf einen bereits besuchten Knoten, wird sie übersprungen und eine Neue aus der PQ entnommen. Anderenfalls wird der aktuelle Knoten als besucht markiert und die aktuell ausgewählte Kante zum MST hinzugefügt.
Abschließend werden alle Kanten des neu ausgewählten Knotens zur PQ hinzugefügt, außer jenen Kanten, die zu bereits besuchten Knoten führen. [6]

Eager Implementierung:

Der Grundaufbau ist gleich der Lazy Version. Jedoch wird bei dieser Art der Implementierung die Priority Queue durch eine min Indexed Priority Queue (IPQ) der Größe V ersetzt. Diese IPQ beinhaltet Knoten-Kanten Paare (V, E) und sortiert diese wieder basierend auf der Gewichtung der Kante. Es wird also jeder Knoten gemeinsam mit der aktuell minimal gewichteten, auf ihn zeigenden Kante gespeichert.
Hierbei wird anders als bei der vorher verwendeten PQ nicht jede ausgehende Kante des Knotens zu der IPQ hinzugefügt. Stattdessen wird zunächst geprüft, ob der Zielknoten bereits in der IPQ ist. Falls nicht, wird er zusammen mit der Kante hinzugefügt. Wenn der Knoten bereits vorhanden ist, wird überprüft, ob es bereits eine „günstigere“ auf ihn zeigende Kante gibt. Wenn nicht, dann wird seine bisherige Kante durch die aktuelle, „leichtere“ Kante ersetzt. Die IPQ wird also immer wieder aktualisiert anstatt erweitert, wodurch ihre Größe deutlich kleiner bleibt, als die der PQ. [7]

2.3 Laufzeitkomplexität

Die Zeitkomplexität des Algorithmus von Prim hängt von den verwendeten Datenstrukturen für den Graphen sowie von der Sortierung der Kanten nach ihrem Gewicht ab. Also beispielsweise der zugrundeliegenden Logik der Priority Queue.
Die im Punkt 2.2 beschriebene Lazy Implementierung hat eine Komplexität von O(E*log(E)). Durch die Anwendung der Eager Variante kann die Komplexität auf O(E*log(V)) reduziert werden, da in der IPQ höchstens V (Knoten, Kanten) Paare sein können, wodurch die Aktualisierungs- und Entnahme-Operationen darauf mit O(log(V)) beschrieben werden können. [6][7]
Hier die Zeitkomplexität für einige weitere typische Kombinationen [8]:

Datenstruktur – minimales Kantengewicht	**Zeitkomplexität**
Adjazenzmatrix und Suche (ohne PQ)	$O(\|V\|^2)$
Adjazenzmatrix und Min Heap, Adjazenzliste und binärer Heap	$O(\|E\| \log(\|V\|))$
Adjazenzliste und Fibonacci Heap	$O(\|E\| + \|V\| \log(\|V\|))$

2.4 Unterschied zum Kruskal Algorithmus

Auch beim Kruskal Algorithmus handelt es sich um einen Greedy-Algorithmus zur Erzeugung eines MST. Der entscheidende Unterschied zwischen den Beiden liegt darin, wie sie entscheiden, welche Kanten zum MST hinzugefügt werden sollen. Beim Kruskal Algorithmus werden zunächst alle Kanten des Graphen betrachtet und nach Gewicht sortiert. Dann wird die leichteste Kante ausgewählt, die nicht zu einem Zyklus im bisherigen MST führt. Dieser Schritt wird wiederholt, bis alle Knoten des Graphen mit dem MST verbunden sind. Anders als beim Prim Algorithmus, welcher nur die von den bisher besuchten Knoten erreichbaren Kanten betrachtet und nach Kosten sortiert. Das heißt, beim Kruskal-Algorithmus können während des Erstellens mehrere Teilgraphen entstehen, während es beim Prim Algorithmus nur einen Teilgraphen gibt, welcher sich vergrößert. [5]

3 Anwendung

3.1 Voraussetzungen

Es gibt mehrere Anforderungen an einen Graphen, damit auf ihn der Algorithmus von Prim angewendet werden kann. Der Graph muss [5]:

1. endlich sein: Das heißt er besteht aus einer endlichen Menge von Knoten und einer endlichen Menge von Kanten. $G = (V, E)$, $E \subseteq V \times V$
2. ungerichtet sein: Der Graph besteht aus ungerichteten Kanten. Das bedeutet, die Kanten zwischen den Knoten haben keine Richtung und können in beide Richtungen genutzt werden.
3. kantengewichtet sein: Jeder Kante eines gewichteten Graphens wird eine reelle Zahl als Kantengewicht zugeordnet. Das ermöglicht es, die Kanten miteinander zu vergleichen und das Gesamtgewicht des Graphens/MSTs zu bestimmen. $G = (V, E, w)$
4. zusammenhängend sein: Für jedes beliebe Knotenpaar des Graphens, muss es einen Weg geben um von einem zum anderen Knoten zu gelangen. Jeder Knoten ist also von jedem beliebigen Knoten aus erreichbar.

Um den Prim Algorithmus anzuwenden, ist es ebenfalls notwendig, einen Startknoten auszuwählen, von dem aus der Algorithmus beginnt, den minimalen Spannbaum des Graphen zu konstruieren. Ohne einen Startknoten kann der Algorithmus nicht ausgeführt werden.
Der Algorithmus von Prim eignet sich dabei vor allem für dichte Graphen. Also Graphen, bei denen die Anzahl der Kanten sehr nah an der Anzahl der insgesamt möglichen Kanten ist. In solchen Fällen ist er dem Kruskal Algorithmus in der Regel überlegen. [9]

3.2 Anwendungsfelder

An dieser Stelle ist vorerst zu bemerken, dass ein MST und damit auch der Prim Algorithmus nicht dazu verwendet werden können, um den kürzesten Weg zwischen zwei Punkten zu finden. Obwohl ein MST den kleinsten Baum mit allen Knoten eines Graphens darstellt, garantiert dies nicht, dass der kürzeste Weg zwischen zwei beliebigen Knoten des Graphens durch die Kanten des MSTs verläuft. Vielmehr wird er zum Beispiel für das Konstruieren von Netzwerken genutzt.
Dazu zählen beispielsweise Telefon- oder Computernetzwerke. Also allgemein in Fällen, wo es sinnvoll oder notwendig ist, einen Baum zu verwenden. Bei einem Telefonnetzwerk könnte man anderenfalls eine Leitung kappen und Geld sparen, während die Verbindung immer noch gesichert wäre. [5]
Weitere Anwendungsfelder sind zum Beispiel: Clusteranalyse, Elektrizitätsnetze, das Problem des Handlungsreisenden, Gas- und Wasser-Versorgungsnetzwerke, Kognitionswissenschaft sowie Straßen- und Schienennetze zur Verbindung aller Städte. [9]

4 Einsatz in einem Beispielprogramm

4.1 Nutzung des Programms

Die Ausführung des im folgenden Kapitel beschriebenen Java-Programms setzt voraus, dass auf dem PC eine Java (JRE) Version 17 oder höher installiert ist. Sollte das noch nicht der Fall sein, kann hier eine aktuelle JDK-Version heruntergeladen werden: https://www.oracle.com/de/java/technologies/downloads/.
Das Programm kann durch das Doppelklicken auf die Datei primMST.exe gestartet werden. Sollte ein Antivirenprogramm, wie der Windows Defender oder ähnliches, den Start verhindert, so ist „trotzdem ausführen“ auszuwählen. Eventuell ist vorher ein Klick auf „weitere Informationen“ notwendig.
Eine alternative Möglichkeit, um das Programm zu starten, besteht darin, die Konsole zu nutzen. Hierzu muss man sich mit der Konsole im Verzeichnis der Datei befinden und anschließend folgendes eingeben: „java -jar primMST.jar“.
Anschließend wird dem Nutzer das Format der Textdatei erläutert, in welcher der zu benutzende Graph an das Programm übergeben werden muss. Durch das drücken der „Enter“-Taste öffnet sich ein JFileChooser, in dem der Nutzer die Datei auswählen muss. In unserem Fall handelt es sich dabei um die Datei MST_Prim1.txt.
Sollte es dabei zu Problemen kommen, kann der FileChooser ohne die Auswahl einer Datei geschlossen werden, woraufhin das Programm in dem Verzeichnis, in welchem es sich befindet, nach einer Datei namens „MST_Prim1.txt“ sucht und diese, wenn vorhanden, nutzt.

4.2 Erläuterung der Funktionsweise

Es gibt verschiedene Wege diesen Algorithmus zu implementieren, welche vor allem von der Datenstruktur zur Verarbeitung des Graphens sowie der Methode zur Sortierung der Kanten abhängen. In dieser Arbeit wurde versucht, zur Implementierung des Algorithmus, einen objektorientierten und damit möglichst anschaulichen Ansatz zu wählen. Dafür wurden vorerst einige Datenstrukturen implementiert, welche nun vorgestellt werden.

Die Klasse „Edge“ besteht aus den Attributen: Anfangsknoten, Zielknoten, Gewicht und einem Boolean „used“, welcher die Information speichert, ob die Kante bereits genutzt wurde. Außerdem implementiert die Klasse das Interface „Comparable“ und damit die Methode „compareTo“, was es der später eingesetzten PQ ermöglichen wird, die einzelnen Kanten basierend auf ihrem Gewicht miteinander zu vergleichen.

Die Klasse „Node“ besteht aus einer eindeutigen Nummer („id“) für jeden einzelnen Knoten und einer Liste „edges“, in welcher alle von dem Knoten ausgehenden Kanten gespeichert werden.

Die Klasse „Graph“ ist aufgebaut aus einer Liste von Knoten und einer Liste dazugehöriger Kanten. Aus einem Objekt dieser Klasse soll später der MST erstellt werden.

Dafür gibt es eine Klasse „MST“, welche ebenfalls aus einer Liste von Knoten und einer Liste von Kanten besteht. Außerdem enthält sie das Attribut „cost“, welches am Ende Auskunft über die gesamten Kosten des MSTs geben kann. Diese Klasse enthält unter anderem eine Methode „toString“, welche die wichtigen Eigenschaften des MSTs in Form eines Strings zurückgibt und so die Ausgabe des MSTs auf der Konsole ermöglicht. Alle dieser Klassen besitzen natürlich noch dazugehörige Getter- und Setter-Methoden, falls benötigt.

Das Programm beginnt mit dem Aufruf der "Main"-Methode und ruft dann als erstes die Funktion "getAbsoluteFilePath" auf. In dieser Funktion wird der Benutzer darüber informiert, welches Format die auszuwählende Datei haben sollte und aufgefordert, mithilfe des JFileChoosers eine geeignete Datei auszuwählen. Sobald der Benutzer eine Datei ausgewählt hat, wird der absolute Pfad dieser Datei an die Funktion "readFile" übergeben. In dieser Funktion wird die Datei zeilenweise mithilfe eines BufferedReaders ausgelesen und jede Zeile wird in einen String umgewandelt und in eine ArrayList „fileData“ gespeichert und an die Main-Methode zurückgegeben. Diese Liste wird an die statische Methode „createNodes“ der Knoten-Klasse übergeben, wo eine Liste erzeugt wird, welche alle Knoten des Graphens enthält („nodes“). Mithilfe dieser Liste von Knoten werden 2 „leere“ MST-Objekte erzeugt („mst“ und „eagerMST“). Danach werden die beiden Listen „fileData“ und „nodes“ noch an die statische Methode „createAndAddEdges“ der Kanten-Klasse übergeben. Dort passiert ein wichtiger Schritt für den Algorithmus. Der eigentlich ungerichtete Graph wird hier in einen gerichteten Graphen

umgewandelt, indem aus jeder ungerichtete Kante, 2 in gegensätzliche Richtungen zeigende Kanten erstellt werden. Jede der beiden Kanten wird ihrem Startknoten über die Methode „addEdge“ sowie einer Liste „edges“ hinzugefügt. Die Liste aller Kanten, wird dann an die Main-Methode zurückgegeben. Jeder Knoten hat nun Kenntnis über alle von ihm ausgehenden Kanten. Mit den Listen „nodes“ und „edges“ wird nun schließlich ein Objekt vom Typ Graph erstellt.
Auf dem in einem vorherigen Schritt erstellten „mst“-Objekt wird nachfolgend die Methode „primsAlgorithm“ aufgerufen und der erzeugte Graph als Parameter übergeben.
In dieser Methode ist die „lazy“-Version des Algorithmus von Prim implementiert. Es werden zuerst alle Knoten des übergebenen Graphens in einer Liste „allNodes“ gespeichert und eine leere Liste vom Typ „Node“ namens „visitedNodes“ für alle bereits besuchten Knoten sowie eine Priority Queue vom Typ „Edge“ angelegt. Daraufhin wird der erste Knoten des Graphens aus der Liste „allNodes“ entnommen und in einer Variable „visitingNode“ gespeichert, welche den aktuell besuchten Knoten enthält. Außerdem wird er zur Liste „visitedNodes“ hinzugefügt. Im Anschluss daran wird die Methode „addEdges“ mit der „priorityQueue“ und dem „visitingNode“ aufgerufen. Dort werden alle von dem Knoten ausgehenden Kanten, welche noch nicht benutzt wurden zur PQ hinzugefügt. Danach wird eine while-Schleife mit den Bedingungen gestartet, dass die „priorityQueue“ nicht leer ist und die Anzahl der bereits besuchten Knoten ungleich der Anzahl aller Knoten des Graphens ist. Innerhalb dieser Schleife wird als erstes die aktuell „optimalste“ Kante aus der PQ entnommen („minimalEdge“) und der Zielknoten dieser in der Variablen „visitingNode“ gespeichert und damit zum aktuell besuchten Knoten gemacht. Sollte dieser Knoten bereits in der Liste „visitedNodes“ und damit besucht worden sein, so wird der restliche Schleifenkörper für die Kante übersprungen und zum Anfang zurückgekehrt. Ansonsten wird die ausgewählte Kante zu den Kanten des MSTs über die Methode „addEdge“ hinzugefügt und das Gewicht der Kante zu den Gesamtkosten des MSTs hinzugefügt. Zum Schluss wird dann noch der im aktuell durchlaufene Knoten zu der Liste „visitedNodes“ hinzugefügt und die Methode „addEdges“ mit ihm aufgerufen, wodurch die von ihm ausgehenden Kanten zur PQ hinzugefügt werden. Wenn die Methode vollständig durchlaufen wurde, bedeutet das, dass der erste MST aus dem übergebenen Graphen erzeugt wurde.
Nachdem in der Main-Methode der MST auf der Konsole ausgegeben wurde und alle Kanten der Liste „edges“ nun wieder als unbenutzt markiert wurden, wird über den Methodenaufruf „eagerPrimsAlgorithm“ auf dem Objekt „eagerMST“ ein zweiter MST mit einer optimierten Variante des Algorithmus erzeugt. Hier wird anstatt der Priority Queue eine Indexed Priority Queue verwendet. Eine IPQ grob simulierende Liste wurde dafür mit der Klasse „IPQ“ implementiert. Hierbei handelt es sich um eine Liste vom Typ „NodeWithBestEdge“. Und dabei

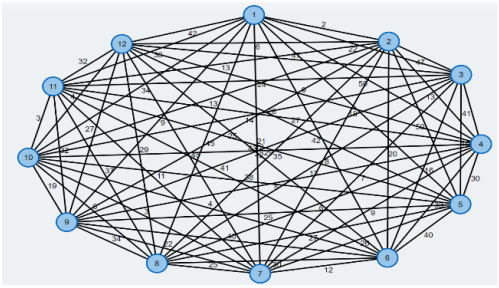
handelt es sich wiederum um eine Art Tupel, aus einem Knoten und einer Kante. In dieser IPQ wird jeder Knoten höchstens einmal zusammen mit der aktuell „leichtesten“ auf ihn zeigenden Kante gespeichert und der Knoten bildet dann außerdem den „Index“ der PQ. Die IPQ enthält folgende Methoden: „insert(node, edge)“ um ein Knoten-Kanten-Paar zur IPQ hinzuzufügen, „contains(node)“ um zu prüfen, ob der übergebene Knoten bereits in der IPQ enthalten ist, „decreaseKey(updatedNode, edge)“ um zu prüfen, ob die übergebene Kante „leichter“ ist als die bisher auf dem Knoten gespeicherte Kante und um diese dann gegebenenfalls zu ersetzen sowie „dequeue“ um das höchst priorisierte Element, also die Kante mit dem geringsten Gewicht, aus der IPQ zu entnehmen.

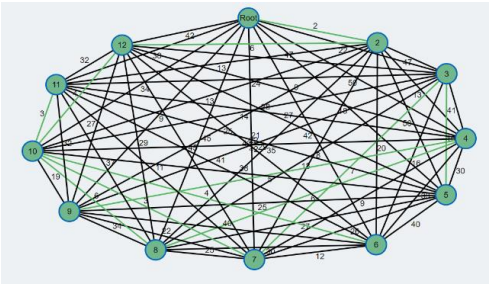
Der Aufbau der Methode „eagerPrimsAlgorithmus“ ist gleich dem der Methode „primsAlgorithmus“. Die einzigen beiden Unterschiede sind, dass die Priority Queue durch die eigen implementierte Datenstruktur IPQ und die Methode „addEdges“ durch die Methode „relaxEdgesAtNode“ ersetzt werden.

In der Methode „relaxEdgesAtNode“ wird mithilfe einer for-Schleife über jede Kante des aktuell besuchten Knotens iteriert. Dabei wird von jeder dieser Kanten der Zielknoten betrachtet. Wenn der Knoten bereits besucht oder die Kante bereits genutzt ist, wird diese Kante übersprungen. Ansonsten wird geprüft, ob die IPQ den Zielknoten bereits enthält. Falls nicht, wird er mit der aktuellen Kante zur IPQ hinzugefügt. Falls doch, wird die Methode „decreaseKey“ mit der aktuellen Kante und ihrem Zielknoten aufgerufen.

Abschließend wird auch dieser MST mit all seinen Kanten und deren Gewichtung sowie seinem Gesamtgewicht ausgegeben.

Das Programm konnte einen folgenden MST für den gegebenen Graphen finden:





5 Zusammenfassung

In dieser Arbeit wurde das Thema Algorithmus von Prim behandelt. Dazu wurden erst die Grundlegenden Thematiken für das Verständnis des Algorithmus, sowie der Algorithmus selbst und seine Anwendung erläutert. Am Ende wurde dann noch ein Beispielprogramm entworfen und beschrieben, welches es ermöglich einen MST für einen beliebigen zusammenhängenden, ungerichteten und gewichteten Graphen zu erzeugen. Eine mögliche Weiterarbeit bestände darin, einen Heap in die entwickelte IPQ einzubauen und das Programm somit weiter zu optimieren.

Quellenverzeichnis

[1] *Jarník, V.* (1930). *"O jistém problému minimálním" [About a certain minimal problem], Práce Moravské Přírodovědecké Společnosti, 6.*

[2] *Prim, R. C. (1957). "Shortest connection networks And some generalizations", Bell System Technical Journal, 36.*

[3] *Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs", Numerische Mathematik, 1.*

[4] *Hochschule Flensburg, Graphenalgorithmen – Minimaler Spannbaum [Online]* – Available at: https://www.inf.hs-flensburg.de/lang/algorithmen/graph/minimum-spanning-tree.htm, Stand 01. März 2023

[5] *Perlt, H. – Algorithmen und Datenstrukturen (Vorlesung / Skript)*

[6] *YoutTube, William Fiset – Prim's Minimum Spanning Tree Algorithm | Graph Theory [Online]* – Available at: https://www.youtube.com/watch?v=jsmMtJpPnhU&t, Stand 02. März 2023

[7] *YoutTube, William Fiset – Eager Prim's Minimum Spanning Tree Algorithm | Graph Theory [Online]* – Available at: https://www.youtube.com/watch?v=xq3ABa-px_g, Stand 02. März 2023

[8] *Javapoint – Prim's algorithm [Online]* – Available at: https://www.javatpoint.com/prim-algorithm, Stand 03. März 2023

[9] *Medium, Mitanshupbhoot – Comparative applications of Prim's and Kruskal's algorithm in real-life scenarios [Online]* – Available at: https://medium.com/@mitanshupbhoot/comparative-applications-of-prims-and-kruskal-s-algorithm-in-real-life-scenarios-4aa0f92c7abc, Stand 04. März 2023

Abkürzungsverzeichnis

MST – Minimum Spanning Tree

PQ – Priority Queue

IPQ – Indexed Priority Queue

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht, noch einer anderen Prüfungsbehörde vorgelegt.

	Leipzig, den 06.03.2023
Nikolaidis, Simon	Ort, Datum