Universität Bremen

Fachbereich 3: Mathematik und Informatik

Bachelorarbeit

Der Algorithmus von Christofides zur Approximation des metrischen TSP

The algorithm of Christofides for the approximation of the metric TSP

Autor: Fabian Hafer

1. Gutachterin: Dr. Sabine Kuske
2. Gutachterin: Prof. Dr. Ute Bormann

12. März 2021

E R K L Ä R U N G

Ich versichere, den Bachelor-Report oder den von mir zu verantwortenden Teil einer Gruppenarbeit*) ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

*) Bei einer Gruppenarbeit muss die individuelle Leistung deutlich abgrenzbar und bewertbar sein und den Anforderungen entsprechen.

Bremen, den _____

_____
(Unterschrift)

Inhaltsverzeichnis

1 Einleitung

Das Problem des Handlungsreisenden ist allgegenwärtig.

Es tritt zum Beispiel auf, wenn ein Vertreter (oder Handlungsreisender, daher der Name) Termine in mehreren Städten wahrnehmen und schließlich zu seinem Ausgangspunkt zurückkehren muss, wobei er jede Stadt, außer der Startstadt, nur einmal besuchen darf. Es ist ihm dabei daran gelegen, dass diese Rundreise oder Tour möglichst kurz ist.

Das Problem ist nun allerdings, dass die Berechnung einer optimalen Tour mit unter sehr aufwändig ist. Sie ist sogar so aufwändig, dass bisher noch nicht bekannt ist, ob man dieses Problem überhaupt effizient lösen kann.

Hier kommen Approximationsalgorithmen ins Spiel. Diese liefern ein Ergebnis zwar um mehrere Größenordnungen schneller, jedoch ist dieses Ergebnis nicht unbedingt optimal. Allerdings ist es immernoch gut genug, dass keine allzu großen Umwege gemacht werden.

Es ist also immer notwendig im Einzelfall abzuwägen, ob ein optimales Ergebnis eine längere Berechnungszeit rechtfertigt. Da es sich in diesem Fall um ein NP-vollständiges Problem handelt, lässt sich insbesondere für große Probleminstanzen eine optimale Lösung praktisch nicht in annehmbarer Zeit berechnen.

Diese Arbeit widmet sich dem Algorithmus von Christofides, einem Approximationsalgorithmus für das metrische Problem des Handlungsreisenden, wobei die Länge der konstruierten Tour im schlechtestes Fall die Länge einer optimalen Tour um das 1,5-fache übersteigt.

Ziel ist es dabei, eine Desktop-Applikation zu implementieren, welche die Arbeitsweise dieses Algorithmus schrittweise visualisiert, um den Algorithmus im Rahmen einer Vorlesung zum Thema Graphentheorie vorstellen zu können. Der Algorithmus von Christofides ist insofern für die Lehre interessant, da er einerseits grundlegende Algorithmen der Graphentheorie, wie minimale Spannbäume, perfekte Matchings, Eulerkreise und Hamiltonkreise, miteinander vereint und andererseits das Thema der Approximationsalgorithmen mehr in den Fokus der Lehre bringt. Die Visualisierung soll insbesondere Studierenden dabei helfen, die Schritte des Algorithmus leichter nachvollziehen können. Sie orientiert sich hierbei an einer Sammlung von Algorithmus-Visualisierungen der TU München (vgl. [Mün]), in der dieser Algorithmus noch nicht behandelt wurde.

Hierfür werden in Kapitel 2 zunächst sowohl die graphentheoretischen Grundla-

gen, als auch verschiedene Arten von Problemen in der theoretischen Informatik erläutert. Darauf folgend wird in Kapitel 3 der Algorithmus von Christofides ausgeführt und anhand eines Beispiels erklärt. In Kapitel 4 geht es um die Implementierung des Programms, wobei auf die Architektur und die Visualisierung eingegangen wird. Kapitel 5 befasst sich mit dem Programm aus der Sicht der Nutzers und gibt eine Hilfestellung zur Installation und Verwendung. Schließlich wird in Kapitel 6 die Bachelorarbeit zusammengefasst und ein Ausblick auf potentielle Verbesserungen an dem entwickelten Programm gegeben.

2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen erläutert, welche für Kapitel 3 benötigt werden. Die Definitionen und Aussagen folgen dabei [Kus19], [KN12] und [Hro14].

2.1 Graphen

Zunächst folgen die notwendigen Grundlagen der Graphentheorie.

2.1.1 Graph

Ein *Graph* ist ein Tripel $G = (V, E, att)$ bestehend aus einer endlichen Menge von *Knoten* V, einer endlichen Menge von *Kanten* E und einer *Aufhängung* $att\colon E \to \binom{V}{2}$, welche jede Kante aus E auf eine 2-elementige Menge von Knoten aus V abbildet. Mit $V(G)$ bezeichnet man die Menge der Knoten des Graphen G und mit $E(G)$ die Menge der Kanten in G.

Diese Definition von Graphen erlaubt *parallele Kanten* $e \neq e'$ mit $att(e) = att(e')$. Enthält ein Graph jedoch keine parallelen Kanten, so nennt man diesen *einfach*. Ein einfacher Graph kann auch durch die kürzere Definition $G = (V,E)$ mit $E \subseteq \binom{V}{2}$ definiert werden.

Man kann Graphen grafisch darstellen, indem man Knoten als Kreise und Kanten als Linien zwischen Kreisen visualisiert. Oft werden Knotenbezeichnungen in die Kreise geschrieben. Bezeichner für Kanten werden bei der Darstellung meist ausgelassen, können jedoch neben die Kante geschrieben werden.

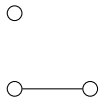
In Abbildung 2.1 sind einige Graphen dargestellt.

2.1.2 Inzidenz und Adjazenz

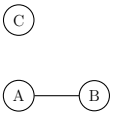
Eine Kante e und ein Knoten v heißen *inzident*, wenn $v \in att(e)$ gilt.

Zwei Kanten e und e' sind *inzident*, wenn sie sich mindestens einen Knoten teilen, also $att(e) \cap att(e') \neq \emptyset$.

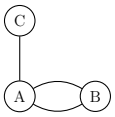
Zwei Knoten v und v' sind *adjazent*, wenn es eine Kante e mit $att(e) = \{v, v'\}$ gibt.



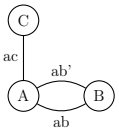
(a) Ein einfacher Graph



(b) Einfacher Graph mit Knotenbezeichnungen



(c) Graph mit parallelen Kanten



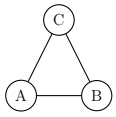
(d) Graph mit parallelen Kanten und Kantenbezeichnungen

Abbildung 2.1: Einige Graphen

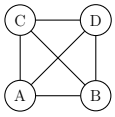
Beispielsweise ist im Graph in Abbildung 2.1b der Knoten C weder mit einer Kante inzident, noch mit einem Knoten adjazent. Die Knoten A und B sind adjazent. Die Kante *ac* ist im Graph aus Abbildung 2.1d inzident zu den beiden parallelen Kanten *ab* und *ab′*.

2.1.3 Vollständiger Graph

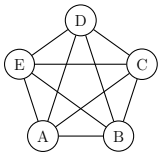
Ein *vollständiger Graph* ist ein einfacher Graph, in dem alle Knoten $v, v' \in V$ mit $v \neq v'$ adjazent sind. Man bezeichnet einen vollständigen Graphen als K_n, wobei $n = |V|$ ist. In Abbildung 2.2 sind die vollständigen Graphen K_n mit $n = 3, 4, 5$ dargestellt.



(a) K_3



(b) K_4



(c) K_5

Abbildung 2.2: Vollständige Graphen mit $n = 3, 4, 5$

2.1.4 Wege und Kreise

Um nun mehr als 2 Knoten oder Kanten miteinander in Beziehung setzen zu können, lassen sich Wege und Kreise definieren.

Weg

Ein *Weg* P von v nach v' in einem Graphen $G = (V, E)$ ist eine alternierende Folge aus Knoten und Kanten $P = v_0 e_1 v_1 e_2 \dots e_n v_n$ mit $v = v_0, v' = v_n, v_i \in V, e_i \in E$

und $att(e_i) = \{v_{i-1}, v_i\}$ für $i = 1, \ldots, n$.

Die Knoten eines Weges P werden auch mit $V(P)$ und die Kanten mit $E(P)$ bezeichnet.

Die *Länge des Weges* P ist die Anzahl der durchlaufenen Kanten $|P| = n$.

Einfacher Weg

Ein *einfacher Weg* ist ein Weg, bei dem alle Knoten paarweise verschieden sind, jeder Knoten also nur einmal vorkommt.

Abbildung 2.3 zeigt einen Graphen mit einem einfachen Weg von A nach C. Die Kanten des Weges sind rot markiert.

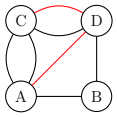


Abbildung 2.3: Einfacher Weg von A nach C

In einfachen Graphen kann jeder Weg $v_0 e_1 \ldots e_n v_n$ kurz als Knotenfolgen $v_0 \ldots v_n$ notiert werden.

Kreis

Ein *Kreis* ist ein Weg, für den gilt, dass Anfang und Ende des Weges gleich sind, also $v_0 = v_n$ gilt. Außerdem muss der Weg mindestens die Länge 1 besitzen, das heißt $n \geq 1$. Weiterhin müssen zwei aufeinanderfolgende Kanten verschieden sein, das heißt, dass für alle $i = 1, \ldots, n-1$ gelten muss, dass $e_i \neq e_{i+1}$ ist. Schließlich müssen die erste und die letzte Kante des Weges ungleich sein, also $e_1 \neq e_n$.

Auch Kreise können in einfachen Graphen als Knotenfolge notiert werden.

Einfacher Kreis

Ein *einfacher Kreis* ist ein Kreis, der zu einem einfachen Weg wird, wenn man eine beliebige Kante entfernt.

Abbildung 2.4 zeigt einen Graph mit einem einfachen Kreis der die Knoten A, B und D durchläuft.

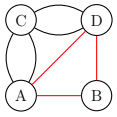


Abbildung 2.4: Einfacher Kreis durch A, B und D

Eulerkreis

Ein *Eulerkreis* ist ein Kreis, der jede Kante eines Graphen genau einmal enthält. Einen Graphen, der einen Eulerkreis besitzt, nennt man *eulersch*.

Im Graph in Abbildung 2.5 gibt es den Eulerkreis Ae_1Be_2De_3Ce_4Ae_5Ce_6De_7A.

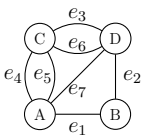


Abbildung 2.5: Eulerkreis Ae_1Be_2De_3Ce_4Ae_5Ce_6De_7A

Hamiltonkreis

Ein *Hamiltonkreis* ist ein Kreis, der jeden Knoten genau einmal enthält.

Der Hamiltonkreis im Graph in Abbildung 2.6 wird durch die roten Kanten dargestellt.

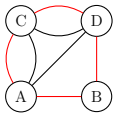


Abbildung 2.6: Hamiltonkreis ABDCA

Ein vollständiger Graph besitzt offensichtlich immer einen Hamiltonkreis. Jede Permutation der Knoten $v_1 \dots v_n$ beschreibt einen einfachen Weg durch die Knoten von K_n. Durch Anhängen von v_1 an das Ende der Permutation, also $v_1 \dots v_n v_1$, ergibt sich ein einfacher Kreis durch die Knoten von K_n und somit auch ein Hamiltonkreis.

2.1.5 Zusammenhang

Ein Graph G ist *zusammenhängend*, falls für alle Knoten v,$w \in V$ ein Weg von v nach w existiert.

Der Graph in Abbildung 2.7a ist zusammenhängend, da es einen Weg von jedem Knoten zu jedem anderen Knoten gibt. Der Graph aus Abbildung 2.7b ist jedoch nicht zusammenhängend, da es zum Beispiel keinen Weg vom Knoten A zum Knoten C gibt.

(a) Zusammenhängender Graph (b) Nicht zusammenhängender Graph

Abbildung 2.7: Graphen mit und ohne Zusammenhang

2.1.6 Knotengrad

Der *Knotengrad* eines Knoten v, kurz als *Grad* bezeichnet, ist die Anzahl der zu v inzidenten Kanten, das heißt $degree_G(v) = |\{e \in E \mid v \in att(e)\}|$. Die Anzahl der Knoten mit ungeradem Grad ist in jedem Graph gerade.

Im Graph aus Abbildung 2.6 sind die Grade der Knoten wie folgt:

$$degree_G(A) = degree_G(C) = degree_G(D) = 4 \text{ und } degree_G(B) = 2$$

Ein Knoten v mit $degree_G(v) = 0$ nennt man *isoliert*.

Ein Graph, der bis auf isolierte Knoten zusammenhängend ist und in dem jeder Knoten einen geraden Grad hat, besitzt einen Eulerkreis.

2.1.7 Kostenfunktion

Eine *Kostenfunktion cost* ist eine Funktion, welche jeder Kante $e \in E$ eines Graphen eine Zahl, in unserem Fall aus den natürlichen Zahlen, zuordnet, das heißt $cost\colon E \to \mathbb{N}$. Diese Zahl wird auch *Kosten*, *Gewicht* oder *Kantengewicht* der Kante genannt, man schreibt sie in grafischen Darstellungen in der Regel neben die Kante. Wenn ein Graph mit einer Kostenfunktion beschrieben wird, spricht man auch von einem *gewichteten Graph*.

Die *Kosten eines Weges* P berechnen sich aus der Summe der Kantengewichte des Weges: $cost(P) = \Sigma_{i=1}^{n} cost(e_i)$.

Die *Gesamtkosten eines Graphen* G berechnen sich aus der Summe der Kosten aller Kanten $cost(G) = \Sigma_{e \in E} cost(e)$.

Metrische Kostenfunktion

Bei vollständigen Graphen kann man auch eine *metrische Kostenfunktion* definieren. Eine metrische Kostenfunktion ist eine Kostenfunktion, welche die folgende Dreiecksungleichung erfüllt:

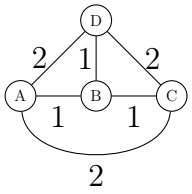
$$cost(\{v, v'\}) \leq cost(\{v, \hat{v}\}) + cost(\{\hat{v}, v'\}) \text{ für alle } v, \hat{v}, v' \in V$$

Das bedeutet, dass die Summe der Kosten der beiden Kanten $\{v, \hat{v}\}$ und $\{\hat{v}, v'\}$ mindestens genauso groß ist, wie die Kosten der Kante $\{v, v'\}$.

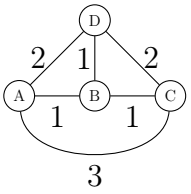
Die Graphen in Abbildung 2.8 zeigen den vollständigen Graphen K_4 mit je einer Kostenfunktion. Die Kostenfunktion beim Graph aus Abbildung 2.8b ist jedoch nicht metrisch, da $cost(\text{AC}) = 3 > cost(\text{AB}) + cost(\text{BC}) = 2$.

2.1.8 Bäume

Ein *Baum* ist ein ungerichteter Graph, welcher keine Kreise enthält und zusammenhängend ist. Ein Baum enthält $|V| - 1$ Kanten.



(a) Graph mit metrischer Kostenfunktion

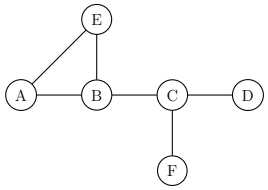


(b) Graph mit nicht metrischer Kostenfunktion

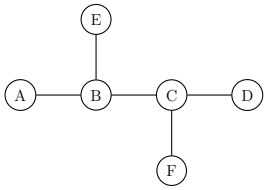
Abbildung 2.8: Graphen mit Kantengewichten

Durch das Verdoppeln aller Kanten eines Baumes erhält man einen eulerschen Graphen.

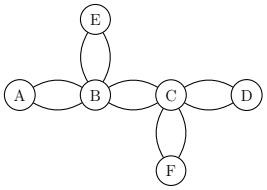
Der Graph in Abbildung 2.9a ist kein Baum, da er einen Kreis durch die Knoten A,B und E enthält. Abbildung 2.9b hingegen zeigt einen zusammenhängenden Graph, der keine Kreise enthält und somit ein Baum ist. Verdoppelt man die Kanten des Graphen aus Abbildung 2.9b so bekommt man den eulerschen Graphen aus Abbildung 2.9c.



(a) Ein Graph



(b) Ein Baum



(c) Eulerscher Graph

Abbildung 2.9: Einige Graphen und Bäume

2.1.9 Teilgraph und Aufspannender Baum

Ein Graph $G' = (V', E', att')$ ist ein *Teilgraph* von $G = (V, E, att)$, wenn $V' \subseteq V$, $E' \subseteq E$ und $att'(e) = att(e)$ für alle $e \in E'$. Man schreibt dann $G' \subseteq G$.

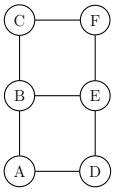
Man bezeichnet G' als *aufspannend*, wenn $V' = V$.

Wenn G' sowohl ein Baum, als auch aufspannend ist, spricht man von einem *aufspannenden Baum* oder *Spannbaum*.

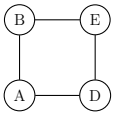
Der Graph in Abbildung 2.10b ist ein Teilgraph vom Graph in Abbildung 2.10a. Dies gilt ebenso für den Graph in Abbildung 2.10c, welcher ein aufspannender Teilgraph von Abbildung 2.10a ist. Beim Graph in Abbildung 2.10d handelt es sich sogar um einen aufspannenden Baum des Graphen aus Abbildung 2.10a.

Minimaler aufspannender Baum

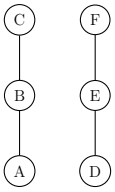
Seien ein Graph G und eine Kostenfunktion $cost : E(G) \to \mathbb{N}$ gegeben. Dann ist ein *minimaler aufspannender Baum* ein aufspannender Baum $T \subseteq G$, welcher minimale Gesamtkosten $cost(T)$ hat.



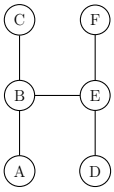
(a) Ein Graph



(b) Teilgraph



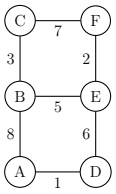
(c) Aufspannender Teilgraph



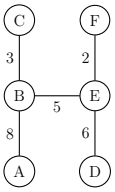
(d) Spannbaum

Abbildung 2.10: Ein Graph mit einigen Teilgraphen

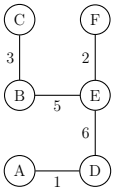
In Abbildung 2.11 sehen wir einen Graphen mit einer Kostenfunktion und zwei seiner Spannbäume. Da die Kosten für den Spannbaum aus Abbildung 2.11b nicht minimal sind, ist dieser kein minimaler aufspannender Baum. Der Graph in Abbildung 2.11c ist jedoch ein minimaler Spannbaum des Graphen aus Abbildung 2.11a.



(a) Ein Graph mit Kostenfunktion



(b) Nicht minimaler Spannbaum von 2.11a



(c) Minimaler Spannbaum von 2.11a

Abbildung 2.11: Graph mit Kostenfunktion und 2 Spannbäumen

2.1.10 Matchings

Ein *Matching* in einem Graphen $G = (V, E, att)$ ist eine Teilmenge $M \subseteq E$ sodass für alle $e, e' \in M$ mit $e \neq e'$ keine Kante aus M mit einer anderen Kante aus M inzident ist, also $att(e) \cap att(e') = \emptyset$.

In Abbildung 2.12a sieht man, in rot markiert, ein Matching im Graphen aus Abbildung 2.11a. Zusätzlich hat der Graph eine Kostenfunktion.

Perfektes Matching

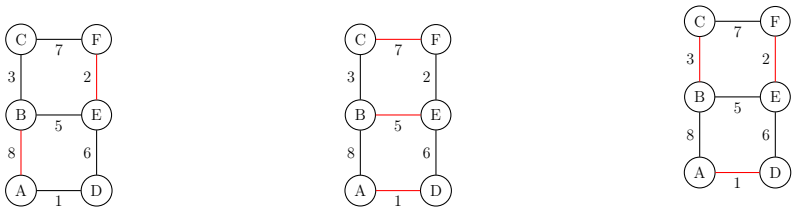
Ein Matching $M \subseteq E(G)$ ist dann ein *perfektes Matching* in $G = (V, E, att)$, wenn zu jedem Knoten aus V eine inzidente Kante in M liegt. Es muss also gelten: $\bigcup_{e \in M} att(e) = V(G)$.

Abbildung 2.12b zeigt ein perfektes Matching für den Graph aus Abbildung 2.11a.

Kostenminimales perfektes Matching

Gegeben sind ein Graph $G = (V, E, att)$, eine Kostenfunktion $cost : E \to \mathbb{N}$ und ein perfektes Matching M in G. M ist dann ein *kostenminimales perfektes Matching*, wenn die Kosten $cost(M) := \Sigma_{e \in M} cost(e)$ des perfekten Matchings minimal sind.

Abbildung 2.12c zeigt ein perfektes Matching, welches, wie man leicht sehen kann, kostenminimal ist.



(a) Matching (b) Perfektes Matching (c) Kostenminimales perfektes Matching

Abbildung 2.12: Einige Matchings

2.2 Entscheidungsprobleme

Ein *Entscheidungsproblem* bezeichnet eine Fragestellung, welche nur mit JA oder mit NEIN beantwortet werden kann. Ein Entscheidungsproblem besitzt eine Eingabe IN, die man auch als *Instanz* des Problems bezeichnet. Somit kann man Entscheidungsprobleme als eine Abbildung $D\colon IN \to \{\text{JA, NEIN}\}$ auffassen.

Nicht jedes Entscheidungsproblem, das man formulieren kann, ist *entscheidbar*, das heißt algorithmisch lösbar. Ein Beispiel dafür ist das sogenannte Halteproblem.

Ein bekanntes entscheidbares Entscheidungsproblem mit Bezug auf Graphen ist das Entscheidungsproblem `EULERKREIS`.

`EULERKREIS`
Instanz: Ein Graph $G = (V, E)$.
Ausgabe: JA, wenn es einen Kreis in G gibt, der jede Kante genau einmal enthält.
NEIN, sonst.

Das Problem `EULERKREIS` lässt sich mit Hilfe der Beobachtung aus 2.1.6, dass ein bis auf isolierte Knoten zusammenhängender Graph, dessen Knoten alle einen geraden Grad haben, leicht lösen, da man sowohl die Knotengrade als auch den Zusammenhang effizient bestimmen kann. Wie man einen Eulerkreis schließlich findet, wurde 1873 bereits von [HW73] beschrieben.

Dass `EULERKREIS` effizient lösbar ist, meint dabei, dass es sich mit *polynomiellen Aufwand* lösen lässt. Dies wiederum bedeutet, dass die Dauer für die Lösung

des Problems polynomiell zur Größe der Eingabe ist. Die Klasse der Entscheidungsprobleme, die mit polynomiellem Aufwand deterministisch gelöst werden können, bezeichnet man als P. Diese Klasse gilt als effizient lösbar, da polynomielles Wachstum hinreichend klein ist. Dies gilt natürlich nur für Polynome mit kleinem Grad, da ein Aufwand von $O(n^{1000000})$ offensichtlich nicht praktikabel ist. Hromkovič sagt in [Hro03] dazu, dass für die meisten Algorithmen, die ein Entscheidungsproblem in polynomiellem Aufwand mit beliebigem Grad lösen, im Laufe der Zeit neue Algorithmen gefunden wurden, die maximal den Aufwand $O(n^6)$ haben, meistens jedoch $O(n^3)$.

Eine weitere wichtige Klasse ist die Klasse der Entscheidungsprobleme, welche nichtdeterministisch mit polynomiellem Aufwand gelöst werden können, *NP*. Es gilt $P \subseteq NP$, jedoch ist bislang noch nicht bekannt, ob auch $NP \subseteq P$ gilt. Diese Frage ist deswegen von Relevanz, da viele interessante Entscheidungsprobleme, für die es jedoch bislang noch keine effizienten Algorithmen gibt, in der Klasse NP liegen. Zudem gibt es außerdem *NP-vollständige* Entscheidungsprobleme. Diese liegen in der Klasse NP und sind mindestens so schwer zu lösen, wie alle anderen Probleme in NP.

Ein solches Entscheidungsproblem ist `HAMILTONKREIS` oder kurz `HAM`.

`HAMILTONKREIS (HAM)`
Instanz: Ein Graph $G = (V, E)$.
Ausgabe: JA, wenn es einen einfachen Kreis der Länge $|V|$ in G gibt.
NEIN, sonst.

Die NP-vollständigkeit und somit auch die Schwere dieses Problems haben Garey und Johnson in [GJ79] gezeigt.

2.3 Optimierungsprobleme

Neben den Entscheidungsproblemen gibt es auch eine andere interessante Art von Problemen: *Optimierungsprobleme*.

Um ein Optimierungsproblem zu beschreiben, benötigt man:

- Eine Eingabe IN wie bei Entscheidungsproblemen.
- Eine Menge von *Lösungskandidaten* $L(x)$ für jedes $x \in IN$
- Eine Kostenfunktion $COST\colon L(x) \times IN \to \mathbb{N}$, welche jedem Lösungskandidaten $l \in L(x)$ für jedes $x \in IN$ eine natürliche Zahl zuordnet.
- Ein Ziel $GOAL = \{\text{Minimum}, \text{Maximum}\}$.

Ein Optimierungsproblem mit $GOAL = \text{Minimum}$ wird *Minimierungsproblem*, mit $GOAL = \text{Maximum}$ wird es *Maximierungsproblem* genannt.

Ein einfaches Optimierungsproblem haben wir bereits in Unterabschnitt 2.1.9 gesehen: `MINIMALER SPANNBAUM`. Formal handelt es sich dabei um das folgende Minimierungsproblem:

MINIMALER SPANNBAUM

Instanz:	Ein zusammenhängender gewichteter Graph $G = (V, E)$.
Kandidaten:	$L(G)$ ist die Menge aller Spannbäume von G.
Kosten:	$COST(l, G) = cost(l)$ mit $l \in L(G)$. [1]
Ziel:	Minimum.
Ausgabe:	Ein Spannbaum $T \subseteq G$ mit minimalen Kosten $cost(T)$.

Dieses Problem ist deterministisch mit polynomiellem Aufwand lösbar, wie zum Beispiel der Algorithmus von Kruskal [Kru56] zeigt.

Zwei weitere Optimierungsprobleme, welche allerdings schwer zu lösen sind, sind einerseits das TRAVELING SALESPERSON PROBLEM, kurz TSP und andererseits das damit verwandte METRISCHE TRAVELING SALESPERSON PROBLEM, kurz Δ-TSP.

TRAVELING SALESPERSON PROBLEM (TSP)

Instanz:	Ein vollständiger gewichteter Graph G.
Kandidaten:	$L(G)$ ist die Menge aller Hamiltonkreise in G.
Kosten:	$COST(l, G) = cost(l)$ mit $l \in L(G)$.
Ziel:	Minimum.
Ausgabe:	Hamiltonkreis $l = v_1 v_2 \dots v_n v_1$ in G mit minimalen Kosten $cost(l)$.

METRISCHES TRAVELING SALESPERSON PROBLEM (Δ-TSP)

Instanz:	Ein vollständiger gewichteter Graph G, dessen Kostenfunktion die Dreiecksungleichung wie in Unterabschnitt 2.1.7 erfüllt.
Kandidaten:	$L(G)$ ist die Menge aller Hamiltonkreise in G.
Kosten:	$COST(l, G) = cost(l)$ mit $l \in L(G)$.
Ziel:	Minimum.
Ausgabe:	Hamiltonkreis $l = v_1 v_2 \dots v_n v_1$ in G mit minimalen Kosten $cost(l)$.

Jedes Optimierungsproblem lässt sich lösen, indem man das zugehörige Entscheidungsproblem endlich oft aufruft, die gesuchten Kosten dabei immer kleiner beziehungsweise größer werden lässt. Zum Beispiel formuliert das zu TSP gehörende Entscheidungsproblem die Frage, ob es in einem vollständigen gewichteten Graphen einen Hamiltonkreis gibt, dessen Kosten k nicht übersteigen. Hierbei sind sowohl der vollständig gewichtete Graph, als auch die Kosten k, Teil der Eingabe.

Es lässt sich leicht ein deterministischer Algorithmus für TSP und Δ-TSP überlegen. Hierfür zählt man einfach alle Hamiltonkreise des Eingabegraphen auf und ermittelt für jeden dieser Hamiltonkreise seine Kosten. Letzteres hat einen Aufwand von $\mathcal{O}(|V|)$. Das Problem ist jedoch das Aufzählen aller Hamiltonkreise in einem vollständigen Graphen.

Um alle Hamiltonkreise aufzuzählen, sucht man sich zunächst einen beliebigen Knoten $v \in V$ aus. Danach bildet man alle Permutationen π über

[1] *COST* ist die Kostenfunktion des Optimierungsproblems und *cost* die Kostenfunktion des Graphen G (vgl. Unterabschnitt 2.1.7).

$v_i \in V \backslash \{v\}$ und fügt den Knoten v sowohl an den Anfang als auch an das Ende von π. Da jeder Hamiltonkreis nun doppelt vorkommen wird, da ein Kreis vorwärts und rückwärts gleich ist, ergeben sich nun $\frac{(|V|-1)!}{2}$ Hamiltonkreise. Tabelle 2.1 zeigt, wie viele Hamiltonkreise die vollständigen Graphen K_5, K_{10}, K_{15} und K_{20} haben. Man sieht schnell, dass durch das exponentielle Wachstum der Anzahl der Hamiltonkreise dieser deterministische Algorithmus sehr weit davon entfernt ist, polynomiellen Aufwand zu haben.

n	$\frac{(n-1)!}{2}$
5	12
10	181440
15	43589145600
20	60822550204416000

Tabelle 2.1: Anzahl der Hamiltonkreise für K_5, K_{10}, K_{15} und K_{20}

2.4 Approximationsalgorithmen

Zum Lösen von Optimierungsproblemen gibt es neben den exponentiellen deterministischen Algorithmen, die eine optimale Lösung berechnen, wie beispielsweise dem Aufzählen der Hamiltonkreise beim `TSP` und Δ-`TSP`, auch solche Algorithmen, welche zwar kein optimales Ergebnis, jedoch ein um einen konstanten Faktor schlechteres Ergebnis liefern. Der Vorteil dieser Algorithmen ist eine, meist polynomielle, Laufzeit. Somit kann man Optimierungsprobleme schneller lösen, verzichtet dabei aber auf eine optimale Lösung. Solche Algorithmen nennt man *Approximationsalgorithmen.*

Seien O ein Optimierungsproblem, x eine Instanz von O, $Opt(x)$ die Kosten einer optimale Lösung des Optimierungsproblems, A ein Approximationsalgorithmus und $A(x)$ die Kosten für das Ergebnis des Approximationsalgorithmus. Setzt man die Kosten des Approximationsalgorithmus ins Verhältnis mit den Kosten der optimalen Lösung, so erhält man das Approximationsverhältnis $r = \frac{A(x)}{Opt(x)}$ bei Minimierungsproblemen beziehungsweise $r = \frac{Opt(x)}{A(x)}$ bei Maximierungsproblemen. Wenn ein Approximationsalgorithmus für jede Instanz einen Wert für $r \leq c$ liefert, spricht man von einem c-Approximationsalgorithmus. c bezeichnet dabei die *Gütegarantie* des Algorithmus.

Ein Beispiel für einen Approximationsalgorithmus ist die *MST-Heuristik.* Die MST-Heuristik ist ein 2-Approximationsalgorithmus für Δ-`TSP`, liefert also Lösungen für das Δ-`TSP`, die im schlimmsten Fall um den Faktor 2 schlechter sind als eine optimale Lösung.

Der Algorithmus erstellt dabei zunächst einen minimalen Spannbaum für den vollständigen Graphen und konstruiert durch Verdoppeln der Kanten des Spannbaumes einen eulerschen Graphen, dessen Eulertour, nach Entfernen der mehrfach vorkommenden Knoten, einen Hamiltonkreis ergibt.

3 Der Algorithmus von Christofides

Zur Approximation des Δ-TSP wurde bereits die MST-Heuristik erwähnt, welche eine Gütegarantie von 2 hat. Fokus dieser Arbeit ist allerdings der *Algorithmus von Christofides*. Im Grunde basiert dieser auf der MST-Heuristik, aber mit Hilfe eines kostenminimalen perfekten Matchings konnte Christofides die Gütegarantie auf 1,5 verbessern.

Der Algorithmus konstruiert aus dem vollständigen Graphen zunächst einen minimal aufspannenden Baum. Für die Knoten, die im Spannbaum einen ungeraden Grad haben, wird dann im vollständigen Graphen ein kostenminimales perfektes Matching gesucht. Wenn dieses Matching gefunden wurde, werden die Kanten des Matchings dem Spannbaum hinzugefügt, wodurch ein Eulerscher Graph entsteht. In diesem wird schließlich ein Eulerkreis konstruiert, welcher durch Weglassen der mehrfach vorkommenden Knoten einen Hamiltonkreis im ursprünglichen vollständigen Graphen bildet.

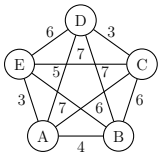


Abbildung 3.1: Vollständiger Graph G mit metrischer Kantengewichtung

Die einzelnen Schritte werden am Beispiel vom Graphen G in Abbildung 3.1 im Folgenden genauer erläutert.

3.1 Konstruktion eines minimalen Spannbaums

Im ersten Schritt des Algorithmus wird für den vollständigen Graphen $G = (V, E, att)$[1] ein minimaler Spannbaum T konstruiert.

$$T = (V, E', att') \text{ mit } att'(e) = att(e) \text{ für } e \in E'$$

[1] Da im Laufe des Algorithmus parallele Kanten entstehen können, wird hier der vollständigen Graphen *mit* Aufhängung definiert, obwohl dies für einen vollständigen Graphen nicht nötig wäre, da vollständige Graphen stets einfach sind und somit keine parallelen Kanten besitzen.

Hierfür bieten sich diverse Algorithmen an, [KN12] nennt hierfür den Algorithmen von Prim, den Algorithmus von Fredman & Tarjan, den Algorithmus von Borůvka, eine Kombination der Algorithmen von Prim und Borůvka und schließlich den Algorithmus von Kruskal. Da alle diese Algorithmen polynomiellen Aufwand haben, sind alle für die Konstruktion des minimalen Spannbaumes geeignet.

Im Folgenden verwenden wir den Algorithmus von Kruskal [Kru56], da dieser vergleichweise einfach zu erklären und implementieren ist. Der Algorithmus folgt dabei wenigen einfachen Anweisungen, die als Pseudocode wie folgt lauten:

```
1  Kruskal(G, c) // G muss zusammenhängend sein
2  Start
3      Sortiere die Kanten E(G) aufsteigend nach ihren Kosten
4      E(T) := ∅
5      Für jede Kante e aus E(G):
6          Wenn (V(G), E(T) ∪ {e}) kreisfrei ist, dann:
7              E(T) := E(T) ∪ {e}
8      Gib den Spannbaum (V(G), E(T)) zurück
9  Ende
```

Listing 3.1: Algorithmus von Kruskal

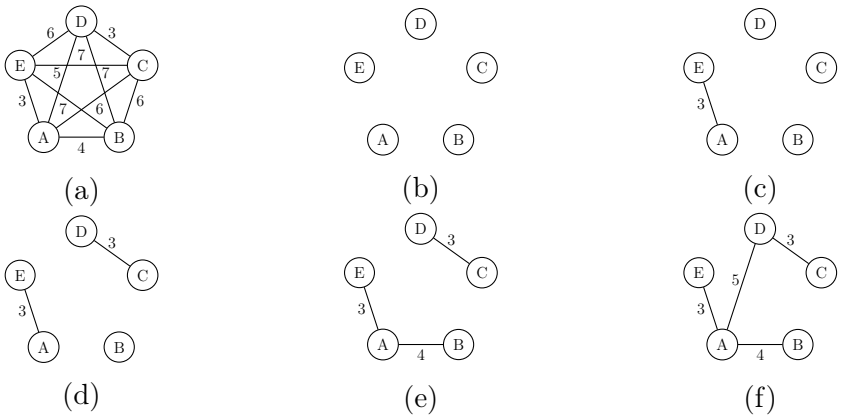


Abbildung 3.2: Konstruktion von T mit dem Algorithmus von Kruskal

Abbildung 3.2 zeigt die Konstruktion eines minimalen Spannbaumes T aus dem vollständigen Graphen G mit Hilfe des Algorithmus von Kruskal. Abbildung 3.2a zeigt dabei den vollständigen Graphen G, welcher der Ausgangspunkt der Konstruktion ist.

Nach aufsteigendem Sortieren der Kanten E des Graphen G ergibt sich die folgende Reihenfolge für die Betrachtung der Knoten:

$$\{A, E\},\{C, D\},\{A, B\},\{A, D\},\{A, C\},\{B, C\},\{D, E\},\{B, D\},\{B, E\},\{C, E\}$$

Nun wird zunächst in Abbildung 3.2b der Graph $T = (V, \emptyset)$ initialisiert. In Abbildung 3.2c wird dem Graphen T die erste Kante $\{A, E\}$ aus der sortierten Liste der Kanten hinzugefügt. Dies wird in den Abbildungen 3.2c, 3.2d und 3.2e wiederholt, da jeweils kein Kreis entsteht. Schließlich wird in Abbildung 3.2f noch die

Kante $\{A, D\}$ hinzugefügt. Nun sieht man bereits, dass ein minimaler Spannbaum entstanden ist und der Algorithmus keine weiteren Kanten hinzufügen würde, da jede weitere der verbliebenen Kanten einen Kreis in T bilden würde. Der minimale Spannbaum T aus Abbildung 3.2f würde vom Algorithmus nun zurückgegeben werden.

Der Aufwand für den Algorithmus von Kruskal ist laut [KN12] $\mathcal{O}(|E| \log |E|)$, lässt sich durch geschickte Implementierung jedoch verbessern.

3.2 Konstruktion eines kostenminimalen perfekten Matchings

Nachdem der minimale Spannbaum T erstellt wurde, betrachtet man die Knoten in T, welche einen ungeraden Grad haben. In Abbildung 3.2f sind dies die Knoten A, B, C und E. Für diese Knoten findet man nun ein kostenminimales perfektes Matching $M \subseteq E$.

Ein Algorithmus zum Finden eines kostenminimalen perfekten Matchings ist in [Coo+98] aufgeführt, welcher von Edmonds in [Edm65] das erste Mal beschrieben wurde.

Das Besondere an diesem Algorithmus ist, dass in seinem Verlauf die Knoten von Kreisen ungerader Länge zu *Blüten* zusammengefasst werden. Das Problem wird durch diese Zusammenfassung von Knoten sukzessive auf kleinere Graphen reduziert.

Da dieser Algorithmus jedoch sehr komplex ist, sei an dieser Stelle für die Erläuterung des Algorithmus auf [Coo+98] verwiesen. Der Aufwand ist bei geschickter Implementierung $\mathcal{O}(|V|^2|E|)$.

(a) (b)

Abbildung 3.3: Finden des kostenminimalen perfekten Matchings

Abbildung 3.3a zeigt den vollständigen Teilgraphen von G, der die Knoten aus T enthält, die einen geraden Grad in T haben. Die Kanten und Knoten, die sich nicht in diesem Teilgraph befinden sind ausgegraut. In diesem Teilgraph wird nun ein kostenminimales perfektes Matching gesucht, welches in Abbildung 3.3b abgebildet ist.

3.3 Konstruktion eines Eulerschen Graphen

Mit Hilfe des Matchings M und des minimalen Spannbaums T lässt sich nun der Eulersche Graph H konstruieren:

$$H = (V, E(T) \uplus M, att'') \text{ mit } att''(e) = \begin{cases} att(e), & \text{für } e \in M \\ att'(e), & \text{für } e \in E(T) \end{cases}$$

Dass H eulersch ist, wird unter der Berücksichtigung ersichtlich, dass die Knoten, für die M gefunden wurde, alle einen ungeraden Grad in T hatten. Da in H nun jeder dieser Knoten inzident zu einer weiteren Kante aus M ist, erhöht sich der Knotengrad jeweils um 1, wodurch dieser gerade wird. Da die Knoten mit geradem Grad mit keiner zusätzlichen Kante aus M inzident sind, sind nun alle Knoten in H gradgradig. Da H außerdem zusammenhängend ist, folgt aus der Beobachtung in 2.1.6, dass H eulersch ist.

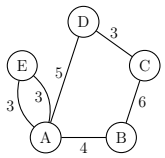


Abbildung 3.4: Eulerscher Graph H

Abbildung 3.4 zeigt nun den eulerschen Graph H, der durch die beschriebene Kombination des minimalen Spannbaumes T aus Abbildung 3.2f mit dem kostenminimalen Matching M aus Abbildung 3.3b entsteht.

3.4 Konstruktion eines Hamiltonkreises

Nun lässt sich im Eulerschen Graphen H zunächst ein Eulerkreis konstruieren. Hierfür kann man zum Beispiel den Algorithmus von Hierholzer (siehe [HW73] und Listing 3.2) verwenden. Er basiert darauf, dass man in einem eulerschen Graphen einen beliebigen Kreis findet, um dann ausgehend von den Knoten des Kreises, welche noch unbenutzte Kanten haben, weitere Kreise zu finden, bis schließlich ein Eulerkreis entstanden ist. Er hat laut [KN12] einen Aufwand von $\mathcal{O}(|V| + |E|)$.

Der Algorithmus von Hierholzer bekommt einen Eulerschen Graphen und einen Startknoten als Eingabe. Abbildung 3.5 zeigt wie der Algorithmus, aufgerufen mit dem Knoten B als Startknoten, nach und nach Kanten hinzufügt und Kreise findet. Der erste gefundene Kreis ist in Abbildung 3.5e rot gekennzeichnet und durchläuft die Knotenfolge BCDAB. Da dieser Kreis noch kein Eulerkreis ist, wird in diesem Kreis ein Knoten gesucht, der noch Kanten hat, die nicht benutzt wurden. Dies ist der Knoten A. Somit wird nun der zweite Kreis in Abbildung 3.5g gefunden, der die Knotenfolge AEA durchläuft. Dieser Kreis ist blau gekennzeichnet. Der zweite Kreis wird nun in den ersten eingesetzt. Damit erhalten wir den Eulerkreis, der die Knoten in der Reihenfolge BCDAEAB durchläuft.

```
1  Hierholzer (G, v_0) // G muss eulersch sein
2  Start
3      C := v_0
4      v_curr := v_0
5      Solange C kein Eulerkreis ist, Wiederhole
6          C_sub := Finde_Kreis(G, v_0)
7          Ersetze das erste Vorkommen von v_curr in C durch C_sub
8          v_curr := Finde_Knoten_mit_unbenutzten_Kanten(G, C)
9      Gib C zurück
10 Ende
11
12 Finde_Kreis(G, v)
13 Start
14     C := v
15     v_next := v
16     Wiederhole
17         Sei e eine zu v_next inzidente Kante aus E(G) mit e = {v_next, v'}
18         Entferne e aus E(G)
19         C := Cev' // Anhängen von ev' an das Ende von C
20         v_next := v'
21     Solange, bis v = v_next
22     Gib C zurück
23 Ende
24
25 Finde_Knoten_mit_unbenutzten_Kanten(G, C)
26 Start
27     Sei L die Liste der Knoten des Kreises C
28     Für jeden Knoten v in L:
29         Wenn v noch unbenutzte Kanten in G hat:
30             Gib v zurück
31     Gib NULL zurück
32 Ende
```

Listing 3.2: Algorithmus von Hierholzer zum Finden eines Eulerkreises

Mit Hilfe dieses Eulerkreises lässt sich nun ein Hamiltonkreis in G konstruieren, indem man die Knoten des Eulerkreises durchgeht und Knoten, die bereits vorkamen, auslässt. Dies ist in Listing 3.3 skizziert. Dadurch ergibt sich eine Knotenfolge, welche in G den gesuchten Hamiltonkreis bildet. Wenn man sich bei der Konstruktion schließlich noch die Kosten der Kanten merkt, erhält man direkt auch die Kosten für den gesamten Hamiltonkreis.

Abbildung 3.6a zeigt den Eulerkreis aus Abbildung 3.5g im ursprünglichen vollständigen Graphen. Durch die Konstruktion mit Listing 3.3 entsteht nun aus dem Graph in Abbildung 3.6a der Hamiltonkreis BCDAEB, der in Abbildung 3.6b zu sehen ist. Die Kosten dieser Rundreise betragen 24.

Nun stellt sich die Frage, ob die gefundene Rundreise BCDAEB gut genug ist. Davon kann man sich bei einem solch kleinen Beispiel noch von Hand selber überzeugen, indem man sich alle Rundreisen vor Augen führt. Diese sind in Abbildung 3.7 dargestellt. Man sieht einerseits, dass der gefundene Hamiltonkreis aus Abbildung 3.6b und der Graph aus Abbildung 3.7k übereinstimmen und an-

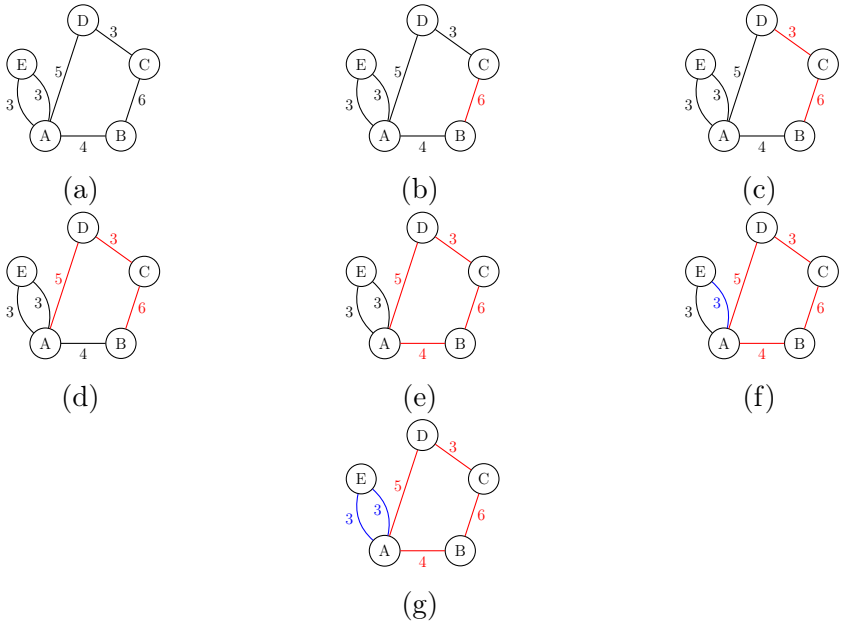


Abbildung 3.5: Konstruktion des Eulerkreises

```
1  Hamilton_aus_Euler(C) // C ist ein Eulerkreis
2  Start
3      Sei H eine leere Liste
4      Sei L die Liste der Knoten des Eulerkreises C
5      Für jeden Knoten v in L:
6          Wenn v nicht in H ist:
7              Hänge den Knoten v hinten an die Liste H
8      Füge den ersten Knoten von H noch einmal hinten an die Liste H an
9      Gib H zurück
10 Ende
```

Listing 3.3: Transformation eines Eulerkreises zu einem Hamiltonkreis

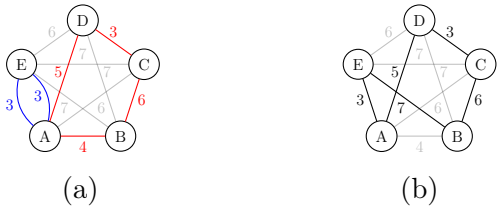
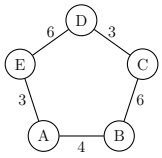


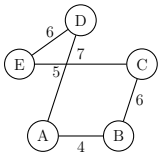
Abbildung 3.6: Konstruktion des Hamiltonkreises in G

dererseits, dass dies nicht der optimale Hamiltonkreis ist. Die optimale Lösung wäre der Hamiltonkreis aus Abbildung 3.7a gewesen, mit den Kosten 22.

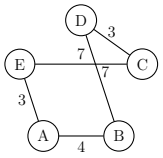
Nun kann noch überprüft werden, ob das Ergebnis der Gütegarantie genügt. Das Beispiel ergibt ein Approximationsverhältnis von $r = \frac{A(x)}{Opt(x)} = \frac{24}{22} \approx 1{,}1 < 1{,}5$ und erfüllt somit die Gütegarantie des Algorithmus.



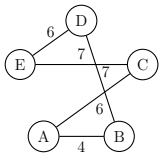
(a) Kosten: 22



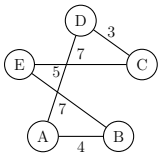
(b) Kosten: 28



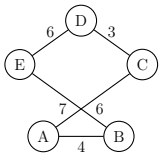
(c) Kosten: 24



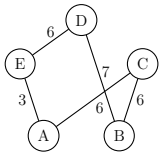
(d) Kosten: 30



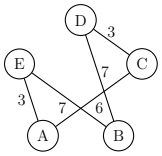
(e) Kosten: 26



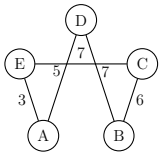
(f) Kosten: 26



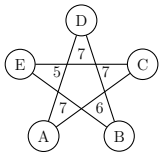
(g) Kosten: 28



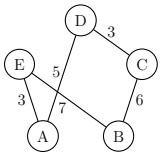
(h) Kosten: 26



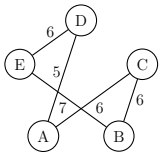
(i) Kosten: 28



(j) Kosten: 32



(k) Kosten: 24



(l) Kosten: 30

Abbildung 3.7: Alle Hamiltonkreise für den Graphen aus Abbildung 3.1

3.5 Korrektheit, Aufwand und Approximationsgüte

Neben der reinen Beschreibung des Algorithmus von Christofides, wird in [KN12] auch dessen Korrektheit, Aufwand und Approximationsgüte angegeben.

Die Korrektheit des Algorithmus ergibt sich dadurch, dass für alle gültigen Eingaben des Δ-TSP ein Hamiltonkreis gefunden wird.

Der Gesamtaufwand des Algorithmus von Christofides ist gleich dem Maximum der Teilaufwände der von ihm verwendeten Algorithmen, da diese stets nacheinander ausgeführt werden. Dieses Maximum findet sich in Abschnitt 3.2 im Algorithmus zum Finden eines kostenminimalen perfekten Matchings, welcher einen Aufwand von $\mathcal{O}(n^3)$ besitzt. Somit besitzt der Algorithmus von Christofides ebenfalls einen Aufwand von $\mathcal{O}(n^3)$.

Der Beweis für die Approximationsgüte von 1,5 wird ebenfalls in [KN12] beschrieben.

4 Implementierung

In diesem Kapitel wird die Implementierung beschrieben. Zunächst wird auf die verwendeten Technologien eingegangen. Danach wird die Architektur der Implementierung beschrieben, wobei der Fokus nicht auf dem Quellcode selber liegen wird, sondern auf den Konzepten, welche bei der Implementierung angewandt wurden. Schließlich werden noch die erfolgten Qualitätssicherungmaßnahmen diskutiert.

Das Ziel der Implementierung war es, ein Programm zu entwickeln, mit dem man den Algorithmus von Christofides darstellen und jeden Schritt erklären kann.

4.1 Technologien

Bei der Implementierung wurde die Programmiersprache *Python* in Version 3.8 verwendet. Python ist eine Skriptsprache, bei der die Quelltexte meist leicht lesbar sind, da viele Syntax-Strukturen, wie zum Beispiel geschweifte Klammern für öffnende/schließende Blöcke, welche unter anderem in Programmiersprachen der C-Familie zu finden sind, nicht vorkommen. Die Strukturierung von Blöcken geschieht vollkommen über die Einrückung der einzelnen Anweisungen. Außerdem ist Python *stark dynamisch typisiert*, das heißt, dass der Typ eines Objektes erst zu Laufzeit des Programms feststeht (dynamisch) und sich der Typ eines Objektes ohne explizite Typumwandlung nicht ändert (stark).

Die grafische Benutzerschnittstelle, kurz *GUI*, wurde mit dem Python-Paket *PySide2* erstellt, welches die offizielle Python-Implementierung der GUI-Bibliothek *Qt5* ist. Ein Vorteil dieses Frameworks ist die Plattformunabhängigkeit, sodass das Programm sowohl für Linux als auch für Windows gleichzeitig entwickelt werden konnte.

Während der Entwicklung wurden außerdem die folgenden Python-Tools verwendet:

- *pip*, zum Nachinstallieren von Python-Paketen.
- *unittest*, ein Framework zum Schreiben von Tests.
- *coverage*, ein auf unittest aufbauendes Framework, um die Testabdeckung messen zu können.

- *pylint*, ein Tool zur statischen Codeanalyse, um Fehlern vorzubeugen und Code Standards einzuhalten.
- *cx_freeze*, ein Framework, um das Programm mit samt der verwendeten Bibliotheken zu einer Distribution zusammenzupacken.

Außerdem wurde das Dateiformat *JSON* verwendet, um die Graphen zu speichern. Die Verwendung von JSON hat den Vorteil, dass es kein binäres Dateiformat ist, und man somit die Möglichkeit hat, die gespeicherten Graphen nachträglich direkt in der JSON-Datei zu bearbeiten oder gar direkt neue Graphen als JSON-Datei zu erstellen.

Während der Implementierung wurden außerdem noch das Versionskontrollsystem *git* und die vom Fachbereich 3 zur Verfügung gestellte DevOps Platform *GitLab* verwendet.

4.2 Architektur

Die Architektur der implementierten Software greift auf Ideen des Konzeptes *Model-View-Controller*, kurz *MVC*, zurück, welches aus den drei Rollen *Model*, *View* und *Controller* besteht.

Nach [Fow02] sind die Aufgaben dieser Rollen die Folgenden:

Das *Model* modelliert die Objekte der Anwendungsdomäne, ohne diese darzustellen oder Interaktionen anzubieten.

Der *View* ist die Ausgabeschnittstelle, die das Model darstellt. Dies kann beispielsweise eine Ausgabe in einem Kommandozeileninterpreter oder aber auch ein Applikationsfenster sein.

Schließlich dient der *Controller* als Eingabeschnittstelle. Er bekommt die Eingaben des Benutzers und verändert anhand der Eingaben das Model in geeigneter Weise. Danach meldet der Controller dem View, dass sich dieser aktualisieren soll.

Wichtig sind hierbei die Trennung von Model und der Darstellung, sowie die Trennung von Ein- und Ausgabe. Letztere wird jedoch in vielen modernen GUI-Frameworks nicht korrekt umgesetzt, so auch bei der benutzten GUI-Bibliothek Qt5. Da View und Controller in Qt5 miteinander vereint sind, bezieht sich die Aussage der Verwendung des MVC Konzeptes auf die strenge Trennung vom Model und dessen Darstellung.

Abbildung 4.1 zeigt nun das Paketdiagramm der implementierten Software. Sie besteht aus dem Oberpaket `christofides`, welches die Unterpakete `model`, `view`, `algorithms` und `util` enthält.

Hierbei entspricht das Paket `model` dem *Model* des *MVC* Konzeptes und das Paket `view` vereint *View* und *Controller* von *MVC*.

Das Paket `algorithm` enthält die benötigten Graph-Algorithmen.

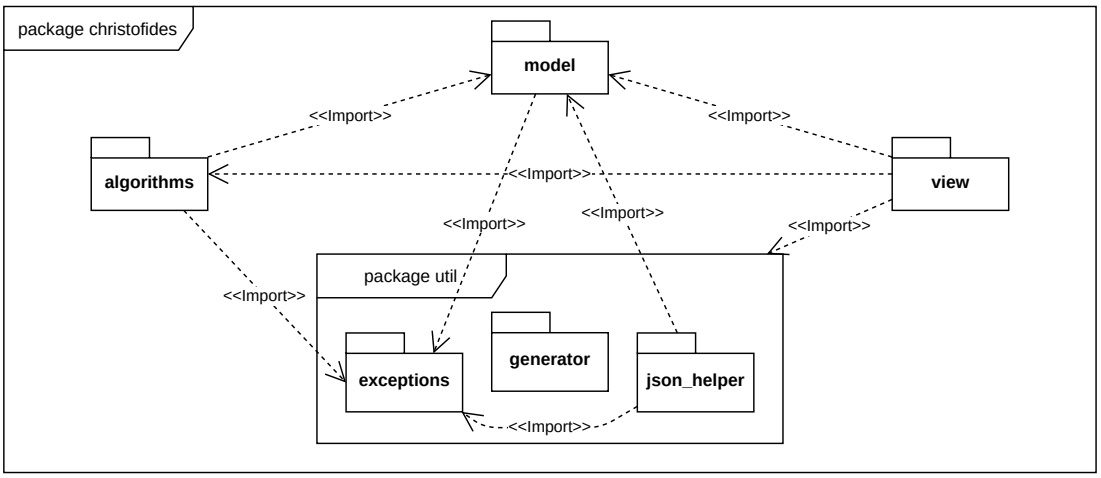


Abbildung 4.1: Paketdiagramm

Das Paket `util` enthält schließlich noch alles Nützliche, welches in den anderen Paketen keinen Platz gefunden hat.

Abbildung 4.1 zeigt außerdem die Zusammenhänge zwischen den Paketen. So sieht man leicht, dass `model` und `view` getrennt sind, wie es das *MVC* Konzeptes vorsieht, da `view` Zugriff auf `model` hat, jedoch nicht anders herum. Außerdem sieht man, dass das `util` Paket durch alle Pakete benutzt wird.

Im Folgenden wird detailliert auf die einzelnen Pakete eingegangen und deren Zweck erläutert.

4.2.1 Das Paket `model`

Die Beziehungen, welche die Klassen des Pakets `model` untereinander haben, sind im Klassendiagramm in Abbildung 4.2 aufgeführt. Aus Gründen der Lesbarkeit wurde an dieser Stelle auf die Darstellung der Attribute und Methoden verzichtet. Diese sind in Anhang A abgebildet.

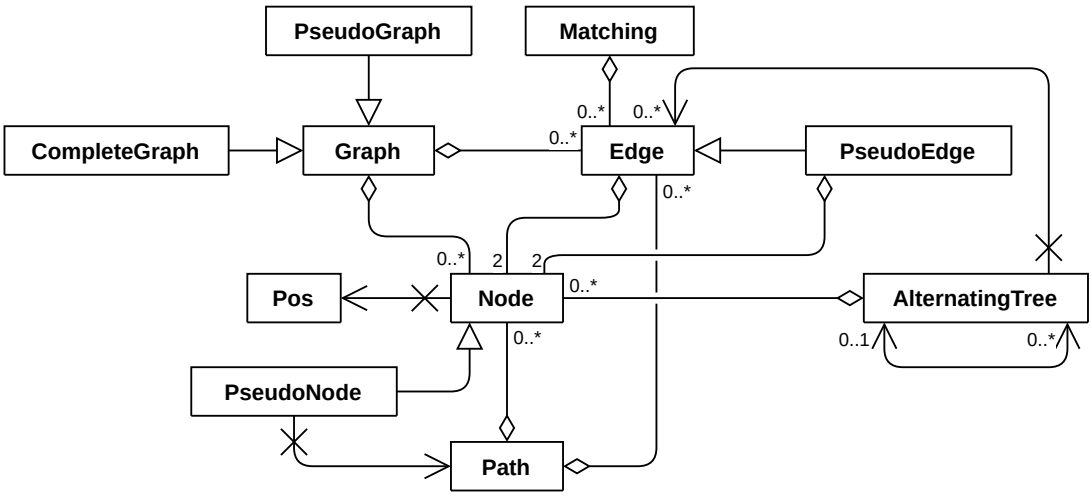


Abbildung 4.2: Klassendiagramm des Paketes Model

Alle Klassen besitzen einen Konstruktor `__init__()`, eine Methode `__str__()`, um Objekte als Text darstellen zu können, eine Methode `__repr__()`, welche die Objekte ebenfalls als Text darstellt, jedoch den Fokus auf eine technische Repräsentation legt, sowie eine Methode `__eq__()`, um Objekte miteinander vergleichen zu können.

Einige Klassen implementieren außerdem noch manche der folgenden Methoden, welche dies sind, kann den Klassen aus Anhang A entnommen werden:

`__hash__()` um Objekte hashen zu können, `__lt__()` um den Kleiner-als-Operator $<$ nutzen zu können, `__contains__()`, um prüfen zu können, ob ein Objekt ein anderes enthält und `__getitem__()` um wie bei Listen mit Hilfe eines Index auf das Objekt zuzugreifen.

Pos

Die Klasse `Pos` stellt eine Position in einem zweidimensionalen Raum dar und besitzt dafür die Attribute `x` und `y`.

Um leichter mit einem Pos Objekt umgehen zu können, gibt es die Methode `to_tuple()`, welche die x- und y-Koordinate in einem Tupel (x, y) zurückgibt.

Schließlich gibt es noch die Methode `distance()`, welche ein anderes `Pos` Objekt erhält und die Distanz zwischen diesen beiden berechnet und zurückgibt. Hierbei wird die euklidische Distanz der beiden Punkte berechnet, wobei das Ergebnis aufgerundet wird. Dies ist nötig, da das Ergebnis der Formel für die euklidische Distanz eine Gleitkommazahl ist, welche als `float` repräsentiert wird. Da dieser Datentyp eine gewisse Ungenauigkeit besitzt, kann es passieren, dass die Dreiecksungleichung, welche in Unterabschnitt 2.1.7 beschrieben wird, nicht erfüllt ist. Um dies zu umgehen, wird das Ergebnis der Distanzberechnung stets aufgerundet und somit als `int` zurückgegeben.

Node

Die Klasse `Node` stellt einen Knoten dar, wie er in einem Graph verwendet werden kann. Er besitzt ein `Pos` Objekt `pos`, welches seine Postion darstellt und ein `label` vom Typ `str`, um den Knoten von anderen Knoten unterscheiden zu können.

Außerdem bietet die Klasse noch eine Methode `move()` an, um dem Knoten eine neue Position zuweisen zu können.

Edge

Eine Kante wird durch die Klasse `Edge` dargestellt. Sie besitzt eine `id_` zum identifizieren der Kante und ein Tupel `nodes`, welches die Knoten, die mit der Kante inzident sind, enthält.

Die Klasse besitzt außerdem noch die Methoden `get_other()`, um für einen mit der Kante inzidenten Knoten den jeweils anderen Knoten der Kante zurückzugeben, die Methode `get_length()`, um die Länge einer Kante zu ermitteln und die

Methode `is_parallel()`, welche für eine übergebene Kante überprüft, ob diese zu den gleichen Knoten inzident ist.

Graph

Ein Graph wird durch die Klasse `Graph` dargestellt. Dieser speichert seine Knoten und Kanten ähnlich einer Inzidenzmatrix als ein Dictionary `graph`, welches die Knoten des Graphen auf die Menge der zum Knoten inzidenten Kanten abbildet. Jede Kante ist in dieser Struktur also doppelt vorhanden, für jeden inzidenten Knoten einmal.

Die Klasse besitzt die Methoden `add_edge()` und `add_node()` für das Hinzufügen und die Methoden `remove_edge()` und `remove_node()` für das Entfernen von Knoten und Kanten. Die Methode `cost()` summiert die Kosten aller im Graph befindlichen Kanten und gibt diese zurück. Die Methoden `get_edges()` und `get_nodes()` dienen dazu, die Knoten beziehungsweise Kanten als eine unveränderliche Menge zu erhalten. Die Methode `get_edges_for_nodes()` kann aufgerufen werden, um die Menge der Kanten zu erhalten, die inzident zu den übergeben Knoten sind. Die Methode `edge_length_for_nodes()` kann die Länge der Kanten zwischen zwei Knoten ermitteln, sofern es mindestens eine Kante zwischen den beiden Knoten gibt. Die Methode `has_parallels()` prüft, ob es im Graphen parallele Kanten gibt, also ob der Graph einfach ist oder nicht. Die Methode `is_cycle_free()` nutzt Tiefensuche um zu überprüfen, ob es im Graph Kreise gibt oder nicht. Schließlich hat die Klasse Graph noch die Methode `move_node()`, welche einem Knoten eine neue Position zuweist.

CompleteGraph

Die Klasse `CompleteGraph` ist eine Unterklasse der Klasse `Graph` und repräsentiert vollständige Graphen.

Sie besitzt die gleichen Methoden wie die Klasse `Graph` mit Ausnahme der Methoden `add_edge()` und `remove_edge()`. Diese wurden so überschrieben, dass sie nicht aufrufbar sind, da die in einem vollständigen Graph vorhandenen Kanten stets abhängig von dessen Knoten sind. Methoden, die das Hinzufügen oder Löschen von Kanten erlauben, würden dem widersprechen.

Path

Die Klasse `Path` repräsentiert einen Weg. Hierfür besitzt die Klasse das Attribut `path`, welches eine Liste ist, die Knoten und Kanten eines Weges in alternierender Reihenfolge beinhaltet, wobei das erste und das letzte Element der Liste immer ein Knoten sein müssen.

Die Klasse `Path` hat die Methoden `add_to_path()`, um eine Kante und einen Knoten an den Weg anzuhängen und die Methode `extend()`, um den Weg durch einen anderen Weg zu verlängern.

Zudem besitzt die Klasse die Methode `insert_cycle_at()`, um einen Knoten des Weges durch den `path` eines weiteren `Path` Objektes, welches einen Kreis

durch diesen Knoten darstellt, zu ersetzen. Dies lässt sich mit dem Graph G in Abbildung 4.3 verdeutlichen. Zu dem Weg Ae_1Be_2C ließe sich der Knoten B durch den Kreis Be_3De_4B ersetzen, sodass man den Weg $Ae_1Be_3De_4Be_2C$ erhält.

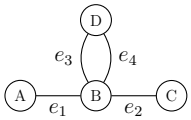


Abbildung 4.3: Graph G

Die Methoden `is_cycle()` und `is_eulerian_cycle()` prüfen, ob es sich bei dem Weg um einen Kreis beziehungsweise einen Eulerkreis handelt.

Die Methoden `get_nodes()` und `get_edges()` dienen dazu nur die Knoten oder Kanten in ihrer Reihenfolge im Weg zu erhalten. Außerdem besitzt die Klasse die Methoden `get_start()` und `get_end()`, welche den ersten beziehungsweise letzten Knoten des Weges zurückgeben und die Methode `index()`, welche die erste Position eines Knoten oder einer Kante im Weg zurückgibt.

Die Methode `get_length()` gibt die Länge des Weges zurück, also die Anzahl der durchlaufenen Kanten.

Die Methode `get_cycle_with_other_start()` dient dazu, den Startpunkt des Weges, sollte dieser ein Kreis sein, auf einen anderen im Weg befindlichen Knoten zu setzen und diesen neuen Weg zurückzugeben

Schließlich besitzt die Klasse `Path` noch die Methode `get_subpath()`, welche dazu dient nur einen Teil des Weges mit einer bestimmten Länge zu erhalten.

PseudoNode

Die Klasse `PseudoNode` wird nur in dem Algorithmus zum Finden von minimalen perfekten Matchings benötigt. Sie repräsentiert die Knoten, die entstehen, wenn die Knoten von Kreisen ungerader Länge zusammengefasst werden. Sie ist eine Unterklasse der Klasse `Node`.

Neben den Attributen der Klasse `Node` besitzt ein `PseudoNode` außerdem das Attribut `cycle`, welches den Kreis repräsentiert, aus dem der Pseudoknoten erstellt wurde.

Die Klasse `PseudoNode` besitzt die Methoden `get_nodes()` und `get_edges()`, um die Knoten beziehungsweise Kanten des Kreises des Pseudoknoten zu erhalten. Da es passieren kann, dass der Kreis eines Pseudoknotens ebenfalls einen Pseudoknoten enthält, gibt es außerdem noch die Methode `get_all_nodes()`, welche rekursiv alle Knoten ermittelt, die durch den Pseudoknoten zusammengefasst wurden, und diese zurückgibt.

PseudoEdge

Die Klasse `PseudoEdge` ist eine Unterklasse der Klasse `Edge` und dient dazu eine Kante zu repräsentieren, welche inzident zu einem Pseudoknoten ist und wird

ebenso nur für den Algorithmus zum Finden von minimalen perfekten Matchings benötigt.

Bei Erstellen eines `PseudoNode` kann man nun die Kanten, die inzident zu einem Knoten des Pseudoknoten waren, durch eine `PseudoEdge` ersetzen. Damit die Information, mit welchen Knoten die Pseudokante ursprünglich inzident war, erhalten bleibt, hat die Klasse `PseudoEdge` das Attribut `original_nodes`, ein Tupel, welches die Knoten der ursprünglichen Kante enthält.

Außerdem hat die Klasse `PseudoEdge` noch die statische Funktion `get_original_nodes()`, welche für ein `PseudoEdge` Objekt deren Attribut `original_nodes` liefert, für ein `Edge` Objekt jedoch dessen `nodes` Attribut zurückgibt.

PseudoGraph

Wie auch die Klassen `PseudoNode` und `PseudoEdge` wird auch die Klasse `PseudoGraph` nur für den Algorithmus zum Finden von minimalen perfekten Matchings gebraucht. Diese Klasse ist eine Unterklasse der Klasse `Graph`, bietet einerseits zusätzlich die Möglichkeit Kreise ungerader Länge zu `PseudoNode` Objekten zusammenzufassen, beziehungsweise dieses wieder rückgängig zu machen, und andererseits haben Knoten dieses Graphen eine Bewertung.

Dafür hat die Klasse `PseudoGraph` ein Attribut `node_rating`, welches ein Dictionary ist, das einen Knoten oder Pseudoknoten auf eine Zahl abbildet. Da die Kosten einer `PseudoEdge` nicht sinnvoll über den Abstand der Knoten repräsentiert werden kann, gibt es außerdem ein Dictionary `edge_cost`, welches die Kanten beziehungsweise Pseudokanten auf deren Kosten abbildet. Schließlich hat die Klasse noch das Attribut `original_edges`, damit die ursprünglichen Kanten des Graphen nach dem Expandieren eines Pseudoknoten wiederhergestellt werden können.

Zusätzlich zu den von der Klasse `Graph` geerbten Methoden besitzt die Klasse `PseudoGraph` noch einige weitere Methoden. Hierzu gehören die Methoden `get_edge_cost()` und `get_node_rating()`, welche aufgerufen werden können, um die Kosten einer Kante beziehungsweise die Bewertung eines Knotens zu bekommen. Außerdem hat die Klasse `PseudoGraph` noch die Methode `get_reduced_edge_cost()`, welche die Kosten zurückgibt, die um die Summe der Knotenbewertungen der mit der Kante inzidenten Knoten reduziert wurden. Desweiteren besitzt die Klasse `PseudoGraph` die Methode `get_equality_edge()`, welche alle Kanten zurückgibt, bei denen die reduzierten Kosten 0 sind und die Methode `get_rated_nodes()`, die alle Knoten zurückgibt, welche eine Bewertung haben.

Zusätzlich besitzt die Klasse noch die Methode `get_replacement_edge()`, welche für ein Knotenpaar die Kante oder Pseudokante zurückgibt, die diejenige Kante repräsentiert, die inzident zu den beiden Knoten ist. Darüberhinaus implementiert die Klasse die Methode `get_pseudonode_for_node()`, welche für einen Knoten den Pseudoknoten zurückgibt, in dem dieser Knoten zur Zeit enthalten ist.

Um zu prüfen, ob ein `PseudoGraph` zur Zeit Pseudoknoten besitzt, stellt die Klasse die Methode `has_pseudonode()` zur Verfügung.

Schließlich hat die Klasse noch die Methode `update_node_rating()`, welche die Bewertung eines Knoten auf einen anderen Wert setzt und die Methoden `shrink_to_pseudonode()` und `expand_pseudonode()`, mit denen ein ungerader Kreis zu einem Pseudoknoten zusammengefasst werden kann oder ein solcher wieder entpackt werden kann.

Matching

Die Klasse `Matching` repräsentiert ein Matching, wie es in Unterabschnitt 2.1.10 beschrieben ist.

Hierfür besitzt die Klasse ein Attribut `edges`, in welcher die im Matching befindlichen Kanten enthalten sind.

Zusätzlich besitzt die Klasse die Methode `add_edge()`, um dem Matching eine Kante hinzuzufügen und die Methode `get_matched_nodes()`, welche alle Knoten zurückgibt, die eine zu sich inzidente Kante im Matching haben. Außerdem bietet die Klasse die Methoden `is_node_m_covered()` und `is_perfect_matching()` an, um zu prüfen, ob es eine Kante im Matching gibt, die inzident zu einem bestimmten Knoten ist, beziehungsweise ob das Matching für einen gegebenen Graphen perfekt ist.

Schließlich gibt es noch die Methode `symmetric_difference_update()`, welche das aktuelle Matching durch die symmetrische Differenz mit einer Menge von Kanten ersetzt. Um dieses Ersetzen zu ermöglichen gibt es außerdem noch die Methode `replace_with_matching()`.

AlternatingTree

Die Klasse `AlternatingTree` implementiert eine Datenstruktur, in der ein Baum erstellt wird, welcher einen Knoten als Wurzel enthält und einen anderen `AlternatingTree` als Vorgänger beziehungsweise mehrere andere alternierende Bäume als Kinder haben kann, wobei zwei Bäume stets durch eine zu ihren Wurzelknoten inzidente Kante verbunden sind. Diese Datenstruktur wird während des Algorithmus für minimale perfekte Matchings benötigt. Der Baum ist alternierend, weil die Kanten auf dem Weg von der Wurzel zu einem Blatt des Baumes abwechselnd im und nicht im aktuellen Matching des Algorithmus sind.

Hierfür hat die Klasse das Attribut `root`, in welchem ein Knoten gespeichert ist, das Attribut `parent`, welches den Vorgängerbaum darstellt, sowie das Attribut `children`. Dieses ist ein Dictionary, welches die Kanten, die die Wurzelknoten der Eltern-Kind Paare verbinden, auf die Kind-Bäume abbildet. Die Wurzel eines gesamten Baumes hat keinen Vorgängerbaum.

Um die Wurzel der gesamten Struktur zu ermitteln, besitzt die Klasse `AlternatingTree` die Methode `get_topmost_parent()`.

Die Methode `find_node()` gibt den Baum zurück, dessen `root` Attribut gleich dem übergebenen Knoten ist. `get_path_from_root_to_node()` gibt ein `Path` Objekt zurück, welches den Weg vom aktuellen Baum zu dem Kind-Baum darstellt, dessen `root` Attribut gleich dem übergebenen Knoten ist. Die Methode `get_all_paths()` gibt eine Liste aller Wege von der Wurzel des Baumes zu seinen Blättern zurück und die Methode `find_nearest_common_tree()` findet für zwei Knoten die im Baum sind, den Baum, der am weitesten von der Wurzel entfernt ist, aber beide Bäume als Kind hat. Sollten diese beiden Bäume auf einem Pfad von der Wurzel zu einem Blatt liegen, wird der Baum zurückgegeben, der sich näher an der Wurzel befindet.

Um Kind-Bäume hinzufügen zu können gibt es die Methoden `add_child_tree()` und `add_path()`. Erstere erstellt einen neuen Baum und fügt diesen dem Aktuellen hinzu, verbunden durch eine Kante. Letztere erstellt alle Kind-Bäume so, dass der übergebene Weg einen Teilweg von der Wurzel des Baumes zu einem Blatt darstellt.

Weiterhin gibt es noch die Methode `remove_edge()`, welche die Kante zu einem Kind-Baum löscht, wodurch sich der Kind-Baum und dessen Kind-Bäume nicht mehr in dem aktuellen Baum befinden.

Die Methode `clear_tree()` dient lediglich dazu, dass ein Baum komplett zurückgesetzt werden kann, ohne dass eine äußere Referenz auf das Objekt verloren geht.

Schließlich gibt es noch die Methoden `get_B()`, die alle Knoten zurückgibt, die eine gerade Tiefe im Baum haben und die Methode `get_A()`, die alle Knoten mit ungerader Tiefe im Baum zurückgibt.

4.2.2 Das Paket `util`

Im Paket `util` finden sich drei Module, `Exceptions`, `generator` und `json_helper`, welche Klassen beinhalten, die keinen Platz in den anderen Paketen gefunden haben, da sie an verschiedenen Stellen benötigt werden.

exceptions

Dieses Modul enthält mehrere Exception Klassen. Diese sind `CycleException`, `FinishedAlgorithmException` und `NoPerfectMatchingException`, welche direkt von der Klasse `Exception` erben. Außerdem gibt es noch die Klassen `CompleteGraphDecoderError` und `NotEnoughNodesError`, welche von der Klasse `ValueError` erben. All diese Exceptions dienen dazu anzuzeigen, dass eine bestimmte Ausnahmesituation aufgetreten ist. Da alle diese Klassen keinerlei Methoden, Funktion oder Konstruktoren überschreiben, können sie genauso verwendet werden wie ihre entsprechenden Oberklassen.

generator

Das Modul `generator` enthält die Klasse `LabelGenerator`, welche dabei hilft, die vom Nutzer erstellten Knoten fortlaufend zu benennen. Dabei ist `LabelGenerator` eine Generatorklasse, was durch das Schlüsselwort `yield` gekennzeichnet ist.

Eine Instanz dieser Klasse liefert durch Aufruf der Methode `__next__()` das nächste Element des Iterators. Dies ist das nächste Vorkommen des Schlüsselwortes `yield`. Dieses dient einerseits wie das Schlüsselwort `return` dazu einen Wert zurückzugeben. Andererseits wird der Zustand der Methode so gespeichert, dass diese beim nächsten Aufrufen von `__next__()` so fortfahren kann, als wäre die Methode nie verlassen worden.

Beim `LabelGenerator` ist dies nun nützlich, da vorher nicht bekannt ist, wieviele Label erstellt werden sollen. Somit kann der Generator immer wenn benötigt das nächste Label liefern, ohne sich alle bereits erstellten Labels merken zu müssen.

Der Generator zählt die Buchstaben des Alphabets von `A-Z`, dann von `AA-ZZ`, und so weiter, auf.

json_helper

Das Modul `json_helper` enthält die Klassen `CompleteGraphEncoder` und `CompleteGraphDecoder`.

Die Klasse `CompleteGraphEncoder` dient dazu, eine Instanz der Klasse `CompleteGraph` in JSON zu kodieren, um diesen speichern zu können.

```
1  {
2      "CompleteGraph": {
3          "Nodes": {
4              "<NodeLabel>": {
5                  "x": <x-Value>,
6                  "y": <y-Value>
7              }
8          }
9      }
10 }
```

Listing 4.1: JSON Format eines vollständigen Graphen

Wie ein solcher Graph aufgebaut ist, ist in Listing 4.1 zu sehen. Ein leerer Graph würde die Zeilen 4 – 7 des Listings einfach weglassen. Ansonsten würde `<NodeLabel>` durch das Label des Knotens ersetzt, sowie `<x-Value>` und `<y-Value>` durch die x- beziehungsweise y-Koordinate der `Pos` des Knotens.

Die Klasse `CompleteGraphDecoder` dient dazu einen JSON-kodierten vollständigen Graphen zu einem `CompleteGraph` Objekt zu dekodieren.

Somit ist es einerseits möglich, vollständige Graphen, welche mit Hilfe des `CompleteGraphEncoder` als JSON gespeichert wurden, wieder einzulesen, andererseits kann man natürlich die Graphen direkt als JSON-Datei nach dem Format aus Listing 4.1 erstellen und dann laden.

4.2.3 Das Paket `algorithms`

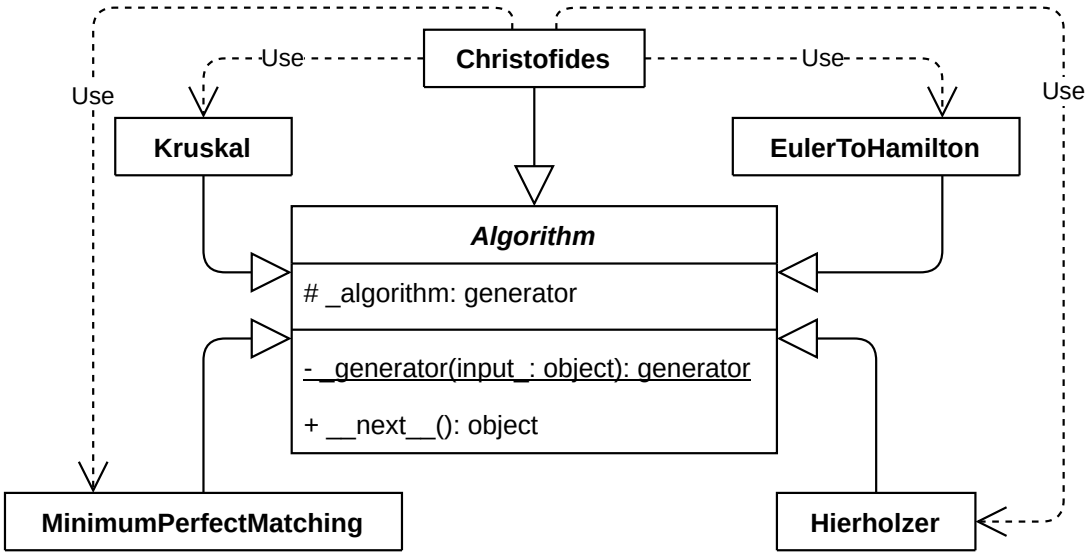


Abbildung 4.4: Klassendiagramm des Paketes Algorithms

Das Paket `algorithms` beinhaltet sowohl die Klasse für den Algorithmus von Christofides, als auch die Klassen der Graphalgorithmen, die durch den Algorithmus von Christofides verwendet werden. Da die Algorithmen bereits in Kapitel 3 erläutert wurden, beziehungsweise im Falle des Algorithmus für minimale perfekte Matchings auf entsprechende Literatur verwiesen wurde ([Coo+98]), und deren Implementierungen sich sehr stark an diesen Erläuterungen orientieren, werde ich auf die konkreten Implementierungen nicht eingehen.

Algorithm

Die Klasse `Algorithm` ist eine abstrakte Klasse, von der alle implementierten Graphalgorithmen erben. Somit setzt diese Klasse die Schnittstelle aller Algorithmen fest. Da die Algorithmen in dem entwickleten Programm nicht nur für eine Eingabe die entsprechende Ausgabe liefern sollen, sonder auch die Zwischenschritte dargestellt werden sollen, ist die Klasse `Algorithm` eine Generatorklasse. Dies ermöglicht mehrere Ergebnisse nacheinander liefern zu können.

Dafür hat ein `Algorithm` Objekt eine statische Funktion `_generator()`, in welcher die Logik des Algorithmus implementiert werden soll, und ein Attribut `_algorithm`, welches die Generatorfunktion verwaltet. Die Eingabe des Algorithmus wird durch den Konstruktor an die `_generator()` Funktion weitergegeben.

Durch einen Aufruf von `__next()__` auf das `Algorithm` Objekt ist es dann möglich, den nächsten Schritt zu erhalten. Das Ende eines Algorithmus ist dann erreicht, wenn es kein weiteres `yield` Statement mehr gibt. In diesem Fall wird automatisch durch den Python-Interpreter eine `StopIteration` Exception ausgelöst.

Da `Algorithm` eine abstrakte Klasse ist, wird in der Funktion `_generator()` ein `NotImplementedError` ausgelöst.

Erwähnenswert ist noch, dass die implementierten Algorithmen nicht nur die Zustände des Algorithmus selbst zurückgeben, sondern auch einen Text, welcher den aktuellen Schritt des Algorithmus erläutert, sowie bei Bedarf eine Abbildung von Knoten oder Kanten auf Farben, um diese Elemente in einer Visualisierung hervorheben zu können.

4.2.4 Das Paket `view`

Das Paket `view` beinhaltet, wie bereits erwähnt, sowohl `View` als auch `Controller` des *MVC* Konzeptes.

MainWindow

Die Klasse `MainWindow` stellt das Hauptfenster der Applikation dar. Sie beinhaltet ein `GraphWidget`, um den Graphen darzustellen und zu zeichnen, ein `QTextEdit` Widget, um Text zur Erklärung darzustellen, eine Statuszeile, um kontextabhängige Daten, wie die aktuelle Position des Mauszeigers oder den aktuellen Modus des GraphWidget, darzustellen, zwei Knöpfe, um im Algorithmus vor- oder rückwärts zu gehen, und eine Menüzeile.

In dieser Klasse finden sich die Methoden, welche durch die Schaltflächen *New*, *Open*, *Save*, *Quit*, *Toggle Mode* sowie *Help* aufgerufen werden.

Das einzige Event, das durch die Klasse selber implementiert wird, ist das `resizeEvent()`. Dieses sorgt dafür, dass bei einer Größenänderung des Fensters das `GraphWidget` und das `QTextEdit` Widget ihre Größe anpassen.

GraphWidget

Die Klasse `GraphWidget` ermöglicht es, dass Graphen dargestellt und bearbeitet werden können. Hierfür erbt die Klasse von der Klasse `QGraphicsView`. Sie besitzt das Attribut `input_graph`, welches den vom Nutzer erstellten Graph repräsentiert und das Attribut `graph`, welches den zu zeichnenden Graph darstellt. Je nach aktuellen Modus des `GraphWidget` können `input_graph` und `graph` gleich oder unterschiedlich sein. Die beiden Modi, *Algorithm* und *Modify*, werden durch die beiden Klassen `AlgorithmScene` und `GraphModifyScene` implementiert, welche noch erläutert werden.

Zum Zeichnen eines Graphen besitzt die Klasse die Methode `drawGraph()`, welche zunächst die Methode `_drawEdges()` und danach die Methode `_drawNodes()` ausführt. Die Reihenfolge ist in sofern wichtig, da so die gezeichneten Knoten die Kanten überdecken und es so wirkt, als würden die Kanten bis zum Rand eines Knoten gehen und nicht bis zu dessen Mittelpunkt.

Zusätzlich gibt es die Methoden `getNodeForLabel()` und `getNodesForLine()`, um die `Node` beziehungsweise `Edge` Objekte mit Hilfe der gezeichneten Objekte zu erhalten.

Die Klasse hat außerdem die Klasse `toggleMode()`, welche dazu dient, den Modus zu ändern und die entsprechende Szenen-Klasse zu initialisieren.

Schließlich implementiert auch diese Klasse das `resizeEvent()`, um im Falle einer Änderung der Fenstergeometrie die Größe anpassen zu können.

GraphScene

Die Klasse `GraphScene` ist eine abstrakte Klasse, welches vorgibt, dass eine abgeleitete Klasse die drei Events `mousePressEvent`, `mouseMoveEvent` und `mouseReleaseEvent` besitzt.

Hierbei besitzt das `mouseMoveEvent` bereits eine Implementierung, welche einerseits dazu dient, die aktuelle Position des Mauszeigers auf der Szene in der Statusleiste des `MainWindow` darzustellen, andererseits aber auch ermöglicht, Informationen über Elemente des zur Zeit dargestellten Graphen anzuzeigen, sollte der Mauszeiger sich über diesen befinden.

ModifyScene

Die Klasse `ModifyScene` dient der Erstellung und Bearbeitung eines Graphen, ist eine Unterklasse der `GraphScene` und stellt den *Modify* Modus des `GraphWidget` dar.

Hierfür wurden die Events der `GraphScene` so überschrieben, dass bei einem Linksklick mit der Maus auf eine leere Fläche der Szene ein neuer Knoten entsteht und bei einem Rechtsklick mit der Maus auf einen Knoten dieser gelöscht wird. Außerdem wurde das Verschieben eines Knoten durch *Drag-and-drop* realisiert.

AlgorithmScene

Die Klasse `AlgorithmScene` ist ebenfalls eine Unterklasse von `GraphScene` und dient der Visualisierung eines Algorithmus. Sie stellt den *Algorithm* Modus des `GraphWidget` dar.

Diese Szene bekommt im Konstruktor einen `algorithm_generator` übergeben, in diesem Fall ein Objekt der Klasse `Christofides` aus dem Paket `algorithms`, und speichert sich zunächst alle Rückgabewerte, die der Generator liefert, in der Liste `graphs`. Dadurch, dass diese Liste alle Zwischenschritte des Algorithmus beinhaltet, wird ermöglicht, dass man durch die Schritte des Algorithmus beliebig vor und zurück navigieren kann. Realisiert wird dies mit Hilfe der Methode `navigate_algorithm_steps()`, indem der Index des aktuellen Elements der Liste `graphs` gespeichert und, je nachdem in welche Richtung navigiert werden soll, dieser Index inkrementiert oder dekrementiert wird. Schließlich ändert diese Methode das `graph` Attribut des `GraphWidget` und ruft dessen `drawGraph()` Methode auf, um den aktuellen Schritt des Algorithmus zu zeichnen.

Diese Klasse überschreibt das `mousePressEvent` so, dass ein Linksklick auf die Zeichenfläche den nächsten und ein Rechtsklick den vorherigen Schritt des Algorithmus anzeigt.

Zusätzlich implementiert diese Klasse auch das `keyPressEvent`, sodass das Vor- und Zurückgehen auch durch Drücken der linken beziehungsweise rechten Pfeiltaste auf der Tastatur möglich ist, wobei das Drücken der rechten Pfeiltaste den nächsten und das Drücken der linken Pfeiltaste den vorherigen Schritt anzeigt.

4.3 Visualisierung

Die GUI wird durch das Paket `view` erstellt, wobei sich deren Aufbau an den Algorithmus-Visualisierungen der TU München (vgl. [Mün]) orientiert. Wie auch bei der Implementierung der TU München wird während der schrittweisen Visualisierung des Algorithmus stets ein Text angezeigt, welcher beschreibt, was im aktuellen Schritt geschehen ist.

Da es jedoch, insbesondere in großen Graphen, schwer zu erkennen ist, was sich zwischen zwei Schritten geändert hat, werden gelegentlich Knoten und Kanten farblich hervorgehoben. In Abbildung 4.5 ist in einigen Beispielen zu sehen, wie die verschiedenen Hervorhebung aussehen.

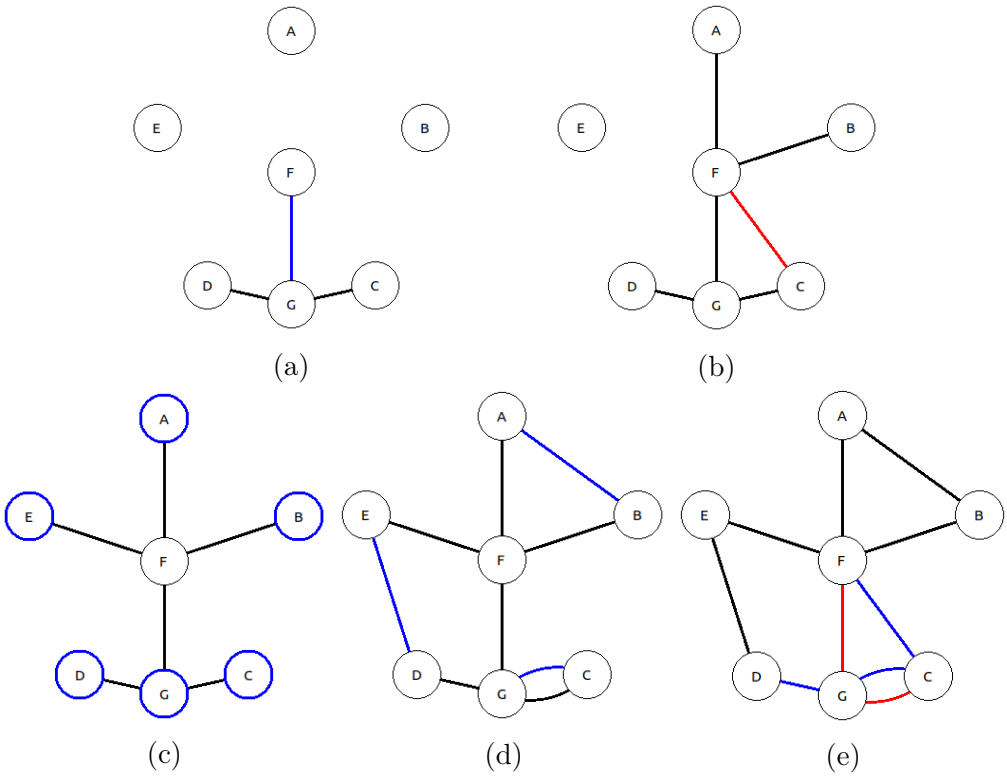


Abbildung 4.5: Hervorhebungen während des Algorithmus

Das hierbei verwendete Farbschema sieht vor, dass blaue Kanten immer solche Kanten zeigen, die zu dem Graphen hinzugefügt werden, wohingegen rote Kanten zeigen, dass diese Kanten ausgelassen werden. Wenn Knoten hervorgehoben werden, so sind diese blau gekennzeichnet.

In Abbildung 4.5a ist nun ein Zwischenschritt bei der Konstruktion des minimalen Spannbaumes zu sehen, wobei gerade die Kante $\{F,G\}$ zum Spannbaum

hinzugefügt wird. Abbildung 4.5b zeigt ebenfalls die Konstruktion des minimalen Spannbaumes, hier wird jedoch die Kante $\{C,F\}$ rot gezeichnet, weil durch deren Hinzufügen ein Kreis entstünde und sie somit nicht zum Spannbaum hinzugefügt wird.

Eine Hervorhebung von Knoten ist in Abbildung 4.5c zu sehen. Hier sind die Knoten hervorgehoben, die im Spannbaum einen ungeraden Grad haben und somit die Knoten darstellen, die für das Finden des kostenminimalen perfekten Matchings relevant sind.

Abbildung 4.5d zeigt nun wie die Kanten des kostenminimalen perfekten Matchings zum Spannbaum hinzugefügt werden.

Schließlich werden in Abbildung 4.5e sowohl manche Kanten blau, als auch manche Kanten rot hervorgehoben. In diesem Schritt wird gerade der Hamiltonkreis konstruiert. Hier stellen die blauen Kanten die Kanten dar, die zum Hamiltonkreis hinzugefügt werden und die roten Kanten diejenigen Kanten, die im Eulerkreis vorhanden waren, jedoch nicht zum Hamiltonkreis hinzugefügt werden dürfen, da die Knoten bereits besucht wurden.

Diese farblichen Hervorhebungen helfen einem Nutzer nun dabei, den aktuellen Schritt besser nachvollziehen zu können. Dies kann man in Abbildung 4.6 sehen, wo für einen Graphen gerade der minimale Spannbaum erstellt wird.

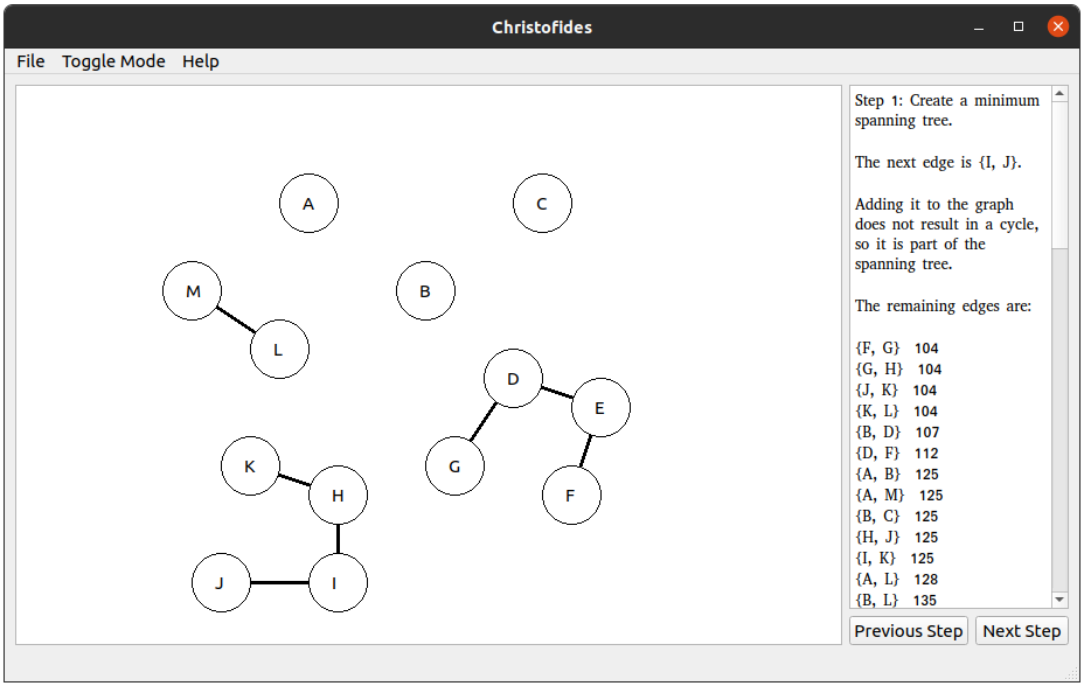


Abbildung 4.6: Hinzufügen der Kante $\{I,J\}$ zum Spannbaum ohne Hervorhebung

Hier sieht man, dass der Text auf der rechten Seite zwar beschreibt, dass die Kante $\{I,J\}$ gerade hinzugefügt wird, jedoch muss diese erst gesucht werden, was insbesondere bei großen Graphen lange dauern kann.

Im Gegensatz dazu zeigt Abbildung 4.7 den gleichen Graphen, hebt die hinzugefügte Kante jedoch hervor, wodurch diese direkt lokalisiert werden kann.

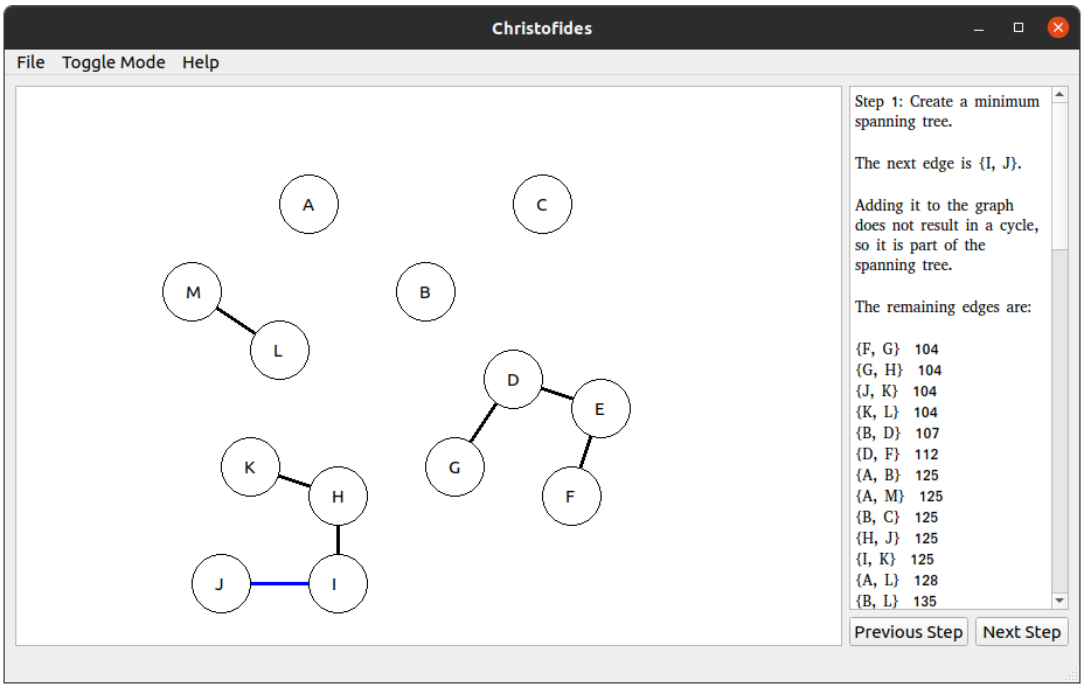


Abbildung 4.7: Hinzufügen der Kante $\{I,J\}$ zum Spannbaum mit Hervorhebung

4.4 Verifikation und Validierung

Zur Sicherung der Qualität der Software wurden verschiedene Arten der Verifikation und Validierung eingesetzt. Dazu gehören die statische Code-Analyse, Unittests, welche die einzelnen Klassen beziehungsweise Module testen und Integrationstests, welche das Zusammenspiel verschiedener Module bis hin zum gesamten Programm testen.

4.4.1 Statische Code-Analyse

Zur Durchführung der statischen Code-Analyse wurde das Tool *pylint* verwendet. *pylint* hilft bei der Entwicklung dabei, Programmierfehler zu vermeiden, gewisse Coding-Standards einzuhalten und sogenannte *Code Smells* aufzudecken. Außerdem dient es dazu, Stellen im Quelltext zu erkennen, in denen sich duplizierter Code befindet.

Durch Integration von *pylint* in den Entwicklungsprozess, beispielsweise durch Einbinden in die Entwicklungsumgebung, erreicht man, dass jede Änderung am Quelltext direkt durch die statische Code-Analyse überprüft wird. Somit erhält man direkt Feedback für den gerade geschriebenen Code und kann diesen, wenn notwendig, direkt so verändern, dass *pylint* keine Fünde mehr liefert.

Hilfreich ist dies deshalb, da Python keine kompilierte Programmiersprache ist.

Dadurch kann es passieren, dass Fehler, wie zum Beispiel der Zugriff auf eine noch nicht existierende Variable, erst bei der Ausführung des Programms auffallen. Durch eine statische Code-Analyse wird solchen Fehlern vorgebeugt. Außerdem fällt insbesondere syntaktisch inkorrekter Code sofort auf.

Da *pylint* außerdem in der Lage ist, duplizierten Code zu erkennen, kann man auch Fehler, welche zum Beispiel durch *Kopieren und Einfügen* geschehen sind, leicht finden und beheben. Dadurch, dass duplizierter Code somit von vornherein sichtbar wird, hilft *pylint* dabei, diesen direkt zu vermeiden.

4.4.2 Unittests

Neben der statischen Analyse wurden auch Unittests geschrieben, welche die Module automatisiert testen. Hierfür wurde das python Modul *unittest* verwendet. Zusätzlich wurde bei der Ausführung der Unittests auch stets die Testabdeckung mit dem Tool *coverage* gemessen.

Mit dem Befehl `coverage run -source christofides -branch -m unittest -q` lassen sich die Unittests ausführen und deren Testabdeckung ermitteln, mit dem Befehl `coverage report -skip-empty` lässt sich das Ergebnis dann anzeigen. Das Ergebnis ist in Anhang B abgebildet.

Wie man an der Testabdeckung sehen kann, sind insbesondere die Pakete `model`, `algorithms` und `util` nahezu vollständig von Tests abgedeckt. Dies bedeutet zwar nicht, dass man dort keinerlei Fehler mehr erwarten kann, jedoch wurden die meisten Anweisungen im Laufe der Tests zumindest einmal ausgeführt, ohne dass es zu Fehlern kam.

Die Tests aus dem Paket `view` wurden zwar auch mit dem Framework *unittest* erstellt, sind aber im eigentlichen Sinne keine reinen Unittests mehr, sondern viel mehr Integrationstests, da die bestehenden Abhängigkeiten zu den anderen Paketen nicht *gemockt*, also durch *dumme* Stellvertreterobjekte ersetzt wurden, sondern die tatsächlichen Objekte verwendet wurden.

4.4.3 Integrationstests

Wie bereits erwähnt, stellen die Tests für das Paket `view` Integrationstests dar. Wie man bereits an der Testabdeckung in Anhang B sehen kann, fallen die automatischen Tests für dieses Paket nicht so umfangreich aus, wie es bei den Unittests der Fall ist.

Dies ist dem geschuldet, dass das Testen der grafischen Objekte nicht ohne weiteres mit dem *unittest* Framework möglich ist und eine Einarbeitung in das Testframework *QTest* der verwendeten GUI-Bibliothek *PySide2* notwendig gewesen wäre, hierfür jedoch die Zeit fehlte.

Daher beschränken sich die Tests für das Paket `view` hauptsächlich auf die Klasse `MainWindow`. Hierbei wurde einerseits getestet, dass gewisse Objekte korrekt erstellt werden, andererseits wurde getestet, dass die Methoden, welche durch das Betätigen der Schaltflächen ausgeführt werden, korrekt arbeiten.

Sowohl die Klasse `GraphWidget` als auch die beiden Klassen `GraphModifyScene` und `AlgorithmScene` wurden somit nicht automatisiert getestet.

Um sicherzustellen, dass die Applikation dennoch korrekt funktioniert, wurde ein Testprotokoll erstellt, welches in Anhang C zu sehen ist. Diese Tests stellen somit auch die Validierung des Programmes dar.

5 Installation und Verwendung

Dieses Kapitel hilft bei der Installation des entwickelten Programms und beschreibt, wie dieses zu verwenden ist.

5.1 Installation

Auf der beiliegenden CD befinden sich im Verzeichnis `installer` drei Dateien zum Installieren des entwickelten Programms. Wie genau die Installation funktioniert, ist im Folgenden erläutert.

5.1.1 Installation auf Linux

Für die Benutzung auf einem Linux Betriebssystem muss die Bibliothek *glibc* in Version 2.23 oder höher installiert sein.

Zum Installieren wird die Datei `Christofides-1.4.3.tgz` verwendet, es handelt sich hierbei um ein komprimiertes Tar-Archiv. Dieses kann mit dem Befehl `tar xfz Christofides-1.4.3.tgz` installiert werden.

Darin befindet sich der Ordner `Christofides`, in dem sich alle zum Programm gehörenden Dateien befinden, und eine Datei `install.sh`, welches die lokale Installation ein wenig vereinfacht.

Das Programm kann jedoch auch ohne die Verwendung des Installationsskriptes gestartet werden, indem die ausführbare Datei `Christofides/Christofides` direkt ausgeführt wird.

Wenn das Installationsskript ausgeführt wird, so wird der Ordner Christofides unter `~/.local/share/` abgelegt. Außerdem wird eine symbolische Verknüpfung `~/.local/bin/Christofides` angelegt, welche auf die im Ordner Christofides befindliche ausführbare Datei zeigt. Schließlich wird noch die Datei `~/.local/share/applications/Christofides.desktop` angelegt, damit das Programm als Applikation gefunden werden kann.

Sollte sich das Verzeichnis `~/.local/bin` in der `PATH` Variable befinden, so lässt sich das Programm aus einer Shell mit dem Befehl `Christofides` starten, hierbei ist das aktuelle Verzeichnis irrelevant. Je nach Desktop Umgebung lässt sich das Programm durch die `.desktop` Datei auch über die Benutzeroberfläche starten.

Sollte das Starten der Applikation nicht sofort möglich sein, liegt das daran, dass `~/.local/bin` noch nicht in der `PATH` Variable zu finden ist. In der Grundkonfiguration vieler moderner Linux Distributionen, wie zum Beispiel Ubuntu, Fedora oder Manjaro, wird das Verzeichnis jedoch beim Anmelden in die `PATH` Variable mit aufgenommen, sofern das Verzeichnis existiert. Ein Abmelden und erneutes Anmelden sollte also genügen. Alternativ kann man die `PATH` Variable natürlich auch manuell anpassen.

5.1.2 Installation auf Windows

Für die Installation auf einer 64-Bit Version von Windows 8.1 beziehungsweise Windows 10 wird die Datei `Christofides-1.4.3-amd64.msi` verwendet. Bei einer Installation unter Windows 8.1 muss allerdings die Laufzeit Bibliothek *Microsoft Visual C++ Redistributable für Visual Studio 2015* ggf. nachinstalliert werden, welche von Microsoft direkt bezogen werden kann.

Durch Doppelklicken der Datei öffnet sich ein Windows Installer. Hier kann nun ein individueller Installationspfad angegeben werden, standardgemäß lautet der Pfad `%LOCALAPPDATA%\Programs\Christofides`. Dieser Pfad kann beliebig angepasst werden, jedoch muss darauf geachtet werden, dass das Ziel keine Administratorrechte benötigt.

Klickt man nun auf *Next*, wird die Installation gestartet.

Schließlich beendet man den Installer durch Drücken von *Finish*.

Nun befinden sich die Programmdaten unter dem ausgewählten Pfad. Außerdem wurde im Windows-Startmenü eine Verknüpfung erstellt, sodass man das Programm über das Startmenü öffnen kann. Hierzu klickt man unten links auf das Windows-Logo. Unter *C* befindet sich nun die Verknüpfung für das Program *Christofides*. Durch einen Rechtsklick auf diese Verknüpfung ist es außerdem möglich, eine weitere Verknüpfung auf dem Desktop erstellen zu lassen.

5.1.3 Installation mit pip

Als Letztes gibt es noch die Möglichkeit, das Programm unabhängig vom Betriebssystem zu installieren. Hierfür ist lediglich ein installiertes Python in Version 3.8 oder höher notwendig.

Mit dem Befehl `pip install Christofides-1.4.3-py3-none-any.whl` wird das Programm und dessen Abhängigkeiten zu anderen Paketen installiert. Danach lässt sich das Programm direkt von der Kommandozeile mit dem Befehl `christofides` starten.

5.2 Verwendung

Bei Programmstart öffnet sich das Hauptfenster der Applikation. Diese hat zwei Modi, einen, um vollständige Graphen zu erstellen, und einen, um den Algorith-

mus von Christofides anhand des Graphen demonstriert zu bekommen. Zu Beginn befindet man sich immer im Bearbeitungsmodus.

Abbildung 5.1 zeigt das Hauptfensters des Programmes. Der große weiße Bereich auf der linken Seite, der mit der 1 markiert ist, ist die Zeichenfläche, auf der die Graphen gezeichnet und angezeigt werden können. Daneben befindet sich ein weiteres weißes Feld, gekennzeichnet mit einer 2. Dies ist das Textfeld, in welchem während des Algorithmus die aktuellen Schritte erläutert werden.

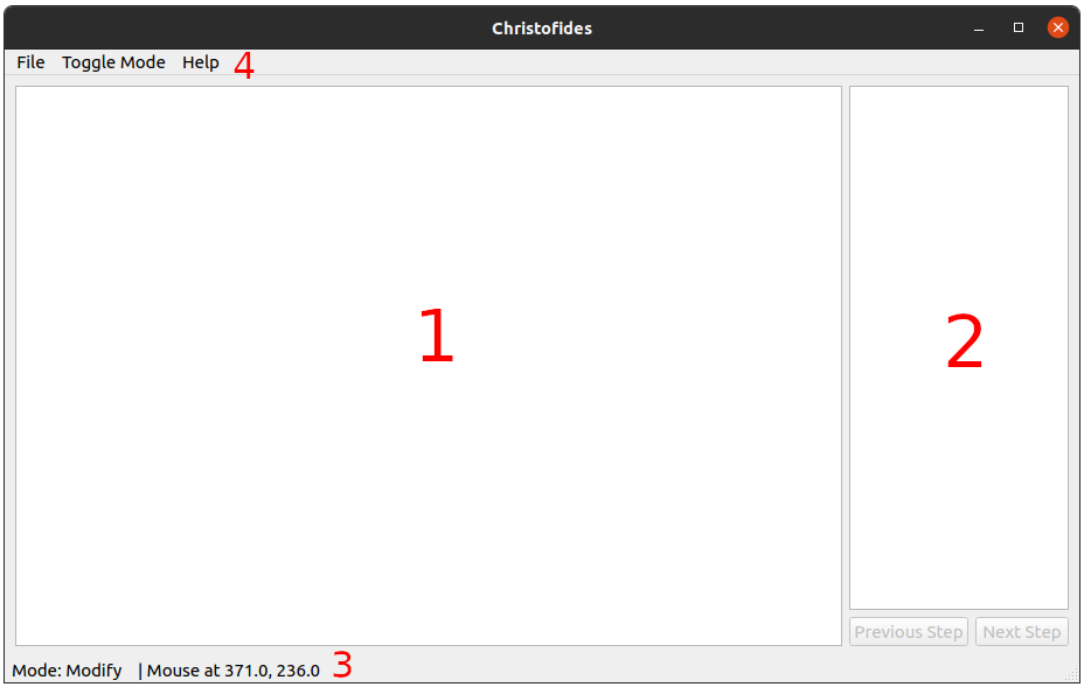


Abbildung 5.1: Applikationsfenster

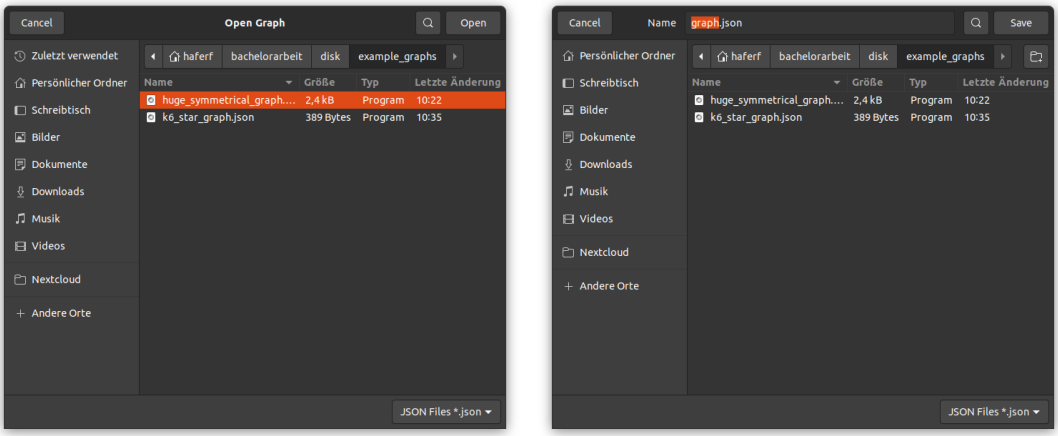
Am unteren Rand des Fensters befindet sich die Statusleiste, in Abbildung 5.1 mit einer 3 gekennzeichnet, welche den aktuellen Modus sowie Position des Mauszeigers auf der Zeichenfläche anzeigt. Sollte sich unterhalb des Mauszeigers ein Teil des Graphen befinden, so wird anstelle der Mausposition erklärt, was sich unterhalb des Mauszeigers befindet.

Schließlich befindet sich am oberen Rand die Menüleiste, sie ist in Abbildung 5.1 mit der 4 gekennzeichnet. Hier befindet sich einerseits das `File` Menü, andererseits die Schaltflächen zum Umschalten des Modus und Aufrufen der Hilfe.

Innerhalb des `File` Menüs gibt es Schaltflächen zum Zurücksetzen des Programms (`New`), zum Laden eines bereits gespeicherten Graphen vom Dateisystem (`Open`), zum Speichern des aktuellen Graphen in einer JSON-Datei auf dem Dateisystem (`Save`) und zum Schließen des Programmes (`Quit`).

Beim Laden beziehungsweise Speichern eines Graphen öffnet sich ein nativer Dateidialog, in welchem man die zu ladende Datei vom Dateisystem auswählen/den Speicherort auf dem Dateisystem und den Namen der zu speichernden Datei auswählen kann. Wie diese unter Ubuntu 20.04 aussehen, ist in Abbildung 5.2 zu

sehen.



(a) Öffnen Dialog (b) Speichern Dialog

Abbildung 5.2: Dialoge zum Öffnen/Speichern eines Graphen

Beim Drücken der Schaltfläche `Help` öffnet sich das Fenster, welches in Abbildung 5.3 zu sehen ist, in dem die Maus- und Tastatursteuerung angezeigt werden.

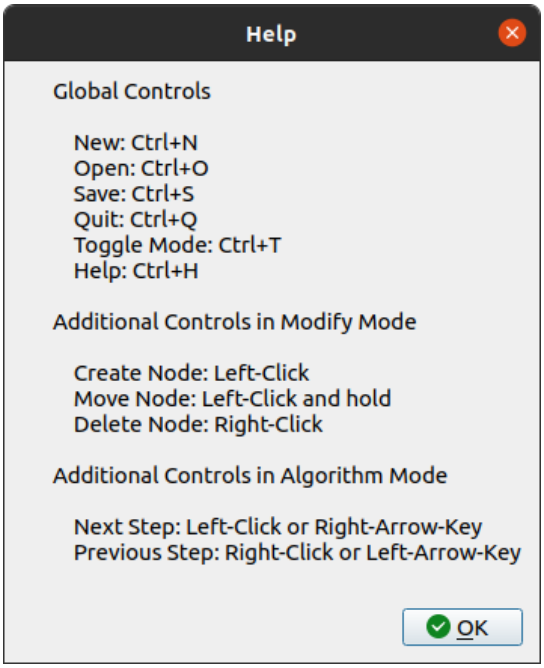


Abbildung 5.3: Hilfefenster

Bearbeitungsmodus

Im Bearbeitungsmodus kann durch einen Linksklick auf die Zeichenfläche ein Knoten erstellt werden, durch einen Rechtsklick auf einen Knoten wird dieser entfernt. Da es sich bei der Eingabe für den Algorithmus um einen vollständigen

Graph handelt, ist es nicht nötig, selber die Kanten zu erstellen. Stattdessen werden die Kanten gezeichnet, sobald ein neuer Knoten erstellt wird, beziehungsweise gelöscht, wenn ein Knoten gelöscht wird.

Außerdem können die Knoten verschoben werden, indem man einen Knoten mit der linken Maustaste anklickt und die Maustaste gedrückt hält und anschließend die Maus bewegt. Hierbei wird sowohl die neue Position angezeigt, der Knoten folgt dabei dem Mauszeiger, als auch die alte Position des Knotens. Lässt man nun die Maustaste los, wird der Knoten an der entsprechenden Stelle abgelegt, der alte Knoten gelöscht und die Kanten neu gezeichnet.

Schließlich lässt sich der Bearbeitungsmodus durch Klicken auf die Schaltfläche *Toggle Mode* verlassen, sofern mindestens zwei Knoten vorhanden sind.

Algorithmusmodus

Im Algorithmusmodus lässt sich der Graph nun nicht mehr bearbeiten. Hier wird nun der Algorithmus von Christofides mit dem erstellten Graphen als Eingabe demonstriert. Im Textfeld auf der rechten Seite wird dabei immer der aktuelle Schritt erklärt.

Zum nächsten Schritt des Algorithmus gelangt man durch Klicken auf die Schaltfläche *Next Step*, beziehungsweise durch einen Linkskick auf die Zeichenfläche oder das Drücken der rechten Pfeiltaste auf der Tastatur. Zum vorherigen Schritt gelangt man mit Hilfe der Schaltfläche *Previous Step*, beziehungsweise durch Rechtsklick auf die Zeichenfläche oder durch Drücken der linken Pfeiltaste auf der Tastatur.

Zum Bearbeitungsmodus gelangt man wieder, indem man die Schaltfläche *Toggle Mode* betätigt. Hierbei wird das Textfeld geleert und der Eingabegraph wiederhergestellt.

6 Zusammenfassung und Ausblick

Das Ziel dieser Bachelorarbeit war die Implementierung eines Programmes, welches den Algorithmus von Christofides anschaulich und verständlich visualisiert und erklärt. Vorbild hierfür waren die Algorithmus-Visualisierungen der TU München (vgl. [Mün]).

Hierfür wurden zunächst die notwendigen Grundladen aus der Graphentheorie erklärt. Darauf aufbauend wurden dann Entscheidungsprobleme wie `EULERKREIS` und Optimierungsprobleme wie das `TSP` beschrieben und anhand des Optimierungsproblems Δ-`TSP` der Begriff des Approximationsalgorithmus veranschaulicht.

Anschließend wurde der Algorithmus von Christofides schrittweise und anhand eines Beispiels erklärt und die von ihm verwendeten Algorithmen dargestellt. Leider fehlte jedoch die Zeit, den recht komplizierten Algorithmus für minimale perfekte Matchings auszuführen.

Danach wurde die Architektur des entwickelten Programmes vorgestellt und erläutert und schließlich folgte noch eine Anleitung für die Installation und Verwendung des Programmes.

Die vorliegende Implementierung ermöglicht es dem Nutzer zwar vollständige Graphen zu erstellen und den Algorithmus von Christofides daran zu visualisieren, dennoch gibt es einige Ideen, wie das Programm erweitert werden könnte.

Derzeit ist die Menge der möglichen Eingabegraphen insofern beschränkt, dass die Kantengewichte stets dem euklidischen Abstand zwischen den Knoten entsprechen. Durch ein manuelles Eingeben der Kantengewichte wäre es möglich, noch mehr Graphen darzustellen, auch wenn dies für den Nutzer großen Aufwand bei großen Graphen darstellen würde. Der Vorteil wäre jedoch, dass der Nutzer die volle Kontrolle über die Graphen behält. Außerdem wäre es hilfreich, da man das Überschneiden von Kanten durch das Verschieben der Knoten verhindern könnte, ohne dass sich die Kantengewichte und somit der Verlauf des Algorithmus ändern.

Weiterhin könnte die Implementierung der Algorithmen auf ihre Laufzeit hin analysiert werden, um potentiell große Graphen besser verarbeiten zu können.

Da das Programm dem Nutzer einerseits bereits ermöglicht, vollständige Graphen zu erstellen, und andererseits neben dem Algorithmus von Christofides bereits die Algorithmen von Kruskal und Hierholzer sowie der Algorithmus zum Finden von minimalen perfekten Matchings implementiert sind, ist auch eine Erweiterung

denkbar, andere Graphalgorithmen zu visualisieren. Hierfür müsste die Bearbeitung von allgemeinen Graphen möglich sein, insbesondere das manuelle Erstellen von Kanten. Ein Optionsmenü wäre dann denkbar, in welchem man den zu visualisierenden Algorithmus auswählen könnte.

Generell wäre es denkbar, ein Optionsmenü zu implementieren, damit ein Nutzer sich das Programm anpassen kann wie er möchte. Hierfür wären die Farben der Hervorhebungen, die Größe der Knoten, die Dicke der Kanten oder die Sprache des Programmes potentiell einstellbare Parameter.

Um die User Experience zu verbessen wäre es zudem denkbar, den Dateinamen eines geöffneten Graphen in die Titelleiste des Fensters zu schreiben und Änderungen am Graphen zu kennzeichnen. In einem solchen Fall wäre auch ein zusätzlicher Dialog hilfreich, der beim Speichern des Graphen, Öffnen beziehungsweise Neuerstellen eines weiteren Graphen, oder dem Schließen der Applikation, angezeigt wird, damit ein potentiell ungespeicherter Graph nicht so einfach verloren geht.

Bevor das Programm jedoch mit Features erweitert würde, wäre es notwendig eine Nutzerstudie durchzuführen, in der zunächst die aktuelle User Experience evaluiert wird. Darauf folgend kann dann mit Hilfe einer Nutzerbefragung untersucht werden, welche Features zukünftig gewünscht sind. Probanden für eine solche Studie beziehungsweise Nutzerbefragung könnten beispielsweise Dozierende und Studierende in einem Modul sein, welches das Thema Graphentheorie behandelt.

Literatur

[Coo+98] William J. Cook u. a. „Optimal Matchings“. In: *Combinatorial Optimization*. John Wiley & Sons, Ltd, 1998. Kap. 5, S. 127–198. ISBN: 9781118033142. DOI: 10.1002/9781118033142.ch5. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118033142.ch5. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118033142.ch5.

[Edm65] Jack Edmonds. „Paths, Trees, and Flowers“. In: *Canadian Journal of Mathematics* 17 (1965), S. 449–467. DOI: 10.4153/CJM-1965-045-4.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. 1st edition. 560. Addison-Wesley Professional, 2002. ISBN: 0321127420 and 9780321127426. URL: https://learning.oreilly.com/library/view/-/0321127420/?ar.

[GJ79] Michael R. Garey und David S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. A series of books in the mathematical sciences. X, 338 S. ; 24 cm : Ill. New York, NY [u.a.]: Freeman, 1979. ISBN: 0716710455 and 0716710447 and 9780716710455 and 9780716710448 and 9780716710448. URL: https://suche.suub.uni-bremen.de/peid=B02315168.

[Hro03] Juraj Hromkovič. *Algorithmics for hard problems : introduction to combinatorial optimization, randomization, approximation, and heuristics*. 2nd ed. Texts in theoretical computer science. XIII, 536 S ; 24 cm : graph. Darst. Berlin [u.a.]: Springer, 2003. ISBN: 3540441344 and 9783540441342. URL: https://dx.doi.org/10.1007/978-3-662-05269-3.

[Hro14] Juraj Hromkovič. *Theoretische Informatik : Formale Sprachen, Berechenbarkeit, Komplexitätstheorie, Algorithmik, Kommunikation und Kryptographie*. 5., überarb. Aufl. Lehrbuch. Online-Ressource (XVIII, 349 S.) : Ill. Wiesbaden: Springer Vieweg, 2014. ISBN: 9783658064334. URL: https://dx.doi.org/10.1007/978-3-658-06433-4.

[HW73] Carl Hierholzer und Chr. Wiener. „Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren“. In: *Mathematische Annalen* 6.1 (März 1873), S. 30–32. ISSN: 1432-1807. DOI: 10.1007/BF01442866. URL: https://doi.org/10.1007/BF01442866.

[KN12] Sven O. Krumke und Hartmut Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. 3. Auflag. Leitfäden der Informatik. 1 Online-

Ressource (X, 431 Seiten). Wiesbaden: Vieweg+Teubner Verlag, 2012. ISBN: 9783834822642. URL: https://dx.doi.org/10.1007/978-3-8348-2264-2.

[Kru56] J. B. Kruskal. „On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem“. In: *Proceedings of the American Mathematical Society*. 7. 1956, S. 48–50. DOI: 10.1090/S0002-9939-1956-0078686-7.

[Kus19] Sabine Kuske. *Algorithmen auf Graphen*. Vorlesungsskript. Universität Bremen, 2019.

[Mün] TU München. *TUM - Mathematik - M9*. URL: https://www-m9.ma.tum.de/Allgemeines/GraphAlgorithmen (besucht am 21. 02. 2021).

Abbildungsverzeichnis

Tabellenverzeichnis

A Klassen vom Paket `model`

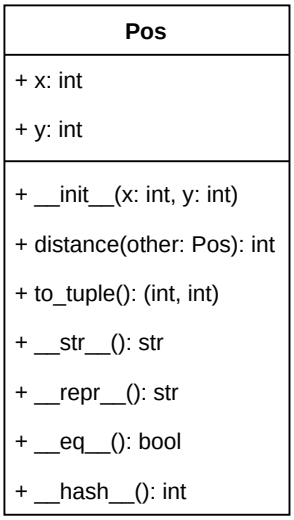


Abbildung A.1: Klasse `pos` des Pakets `model`

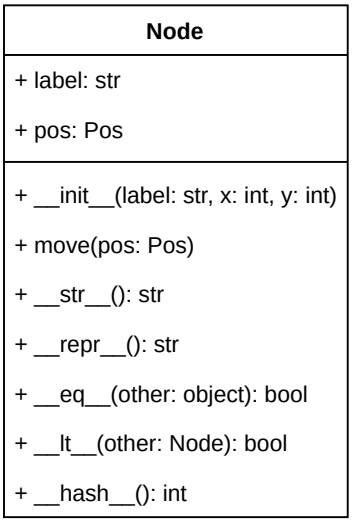


Abbildung A.2: Klasse `node` des Pakets `model`

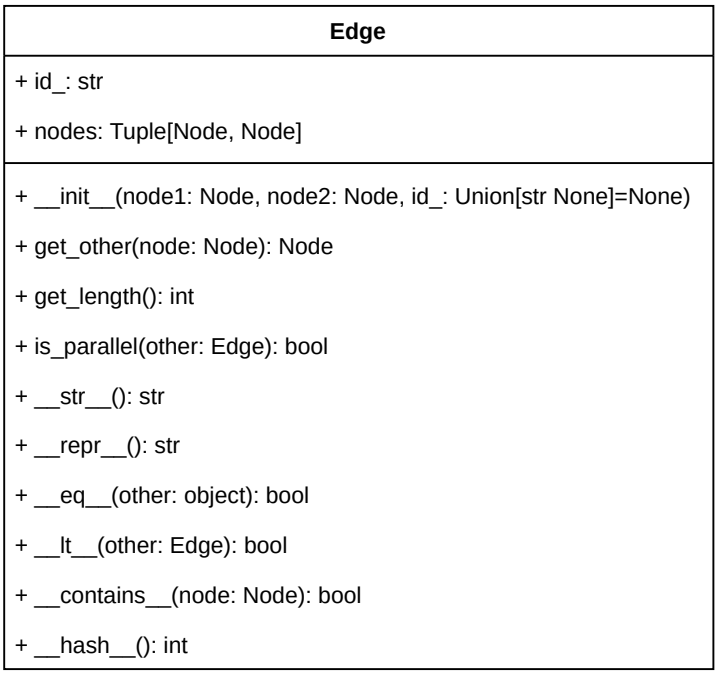


Abbildung A.3: Klasse `edge` des Pakets `model`

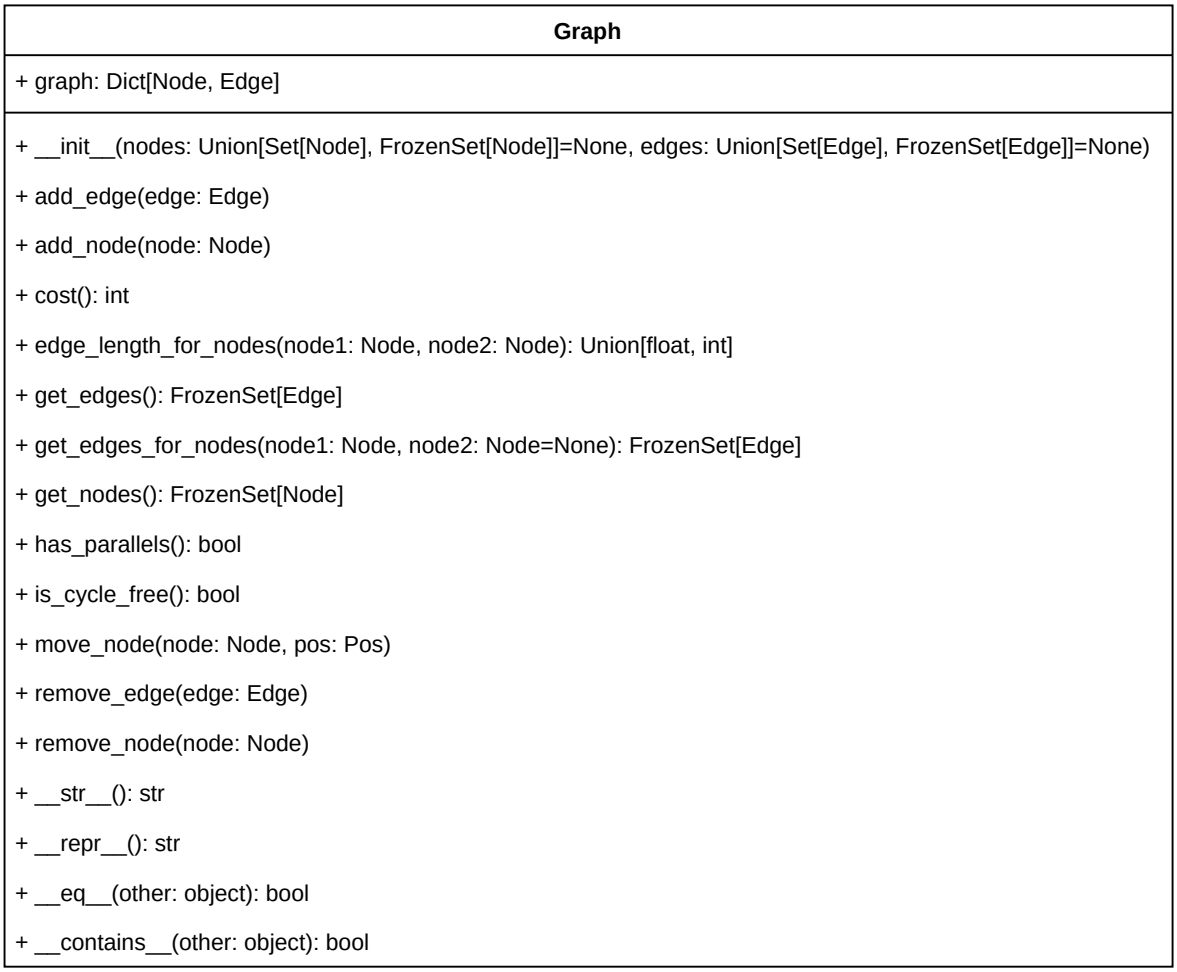


Abbildung A.4: Klasse `graph` des Pakets `model`

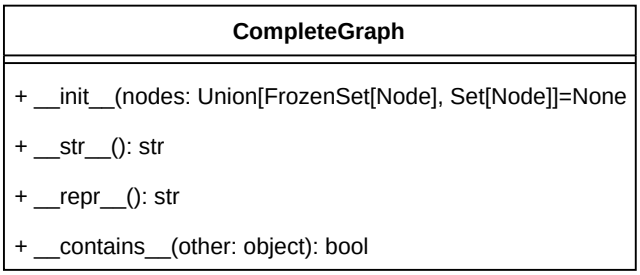


Abbildung A.5: Klasse `CompleteGraph` des Pakets `model`

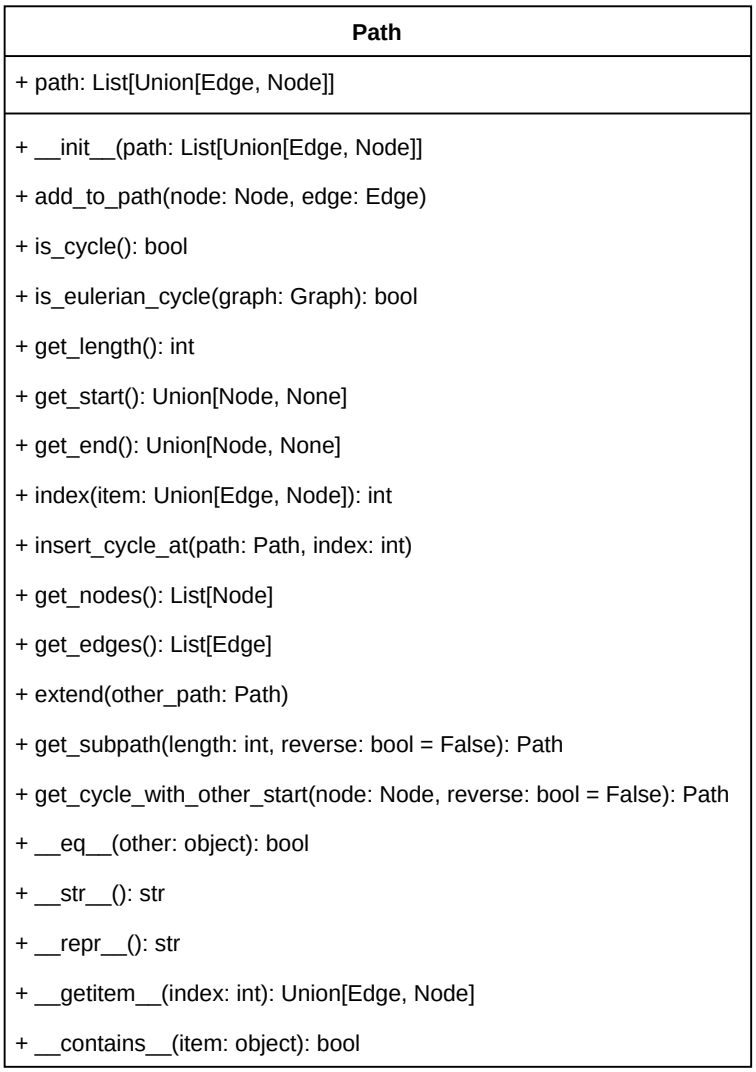


Abbildung A.6: Klasse `path` des Pakets `model`

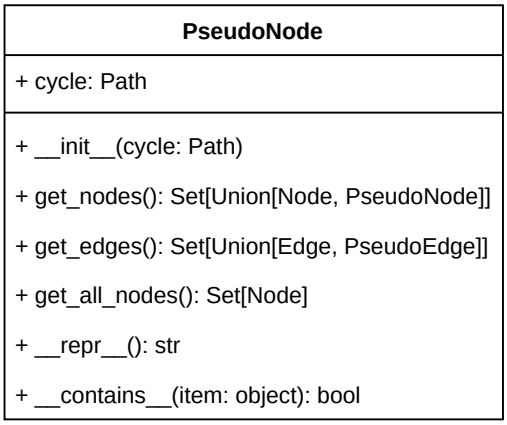


Abbildung A.7: Klasse `PseudoNode` des Pakets `model`

PseudoEdge
+ original_nodes: Tuple[Node, Node]
+ __init__(node1: Union[Node, PseudoNode], node2: Union[Node, PseudoNode, original_nodes: Tuple[Node], id_: Union[str, None]=None) + get_original_nodes(edge: Union[Edge, PseudoEdge]): Tuple[Node, Node] + __repr__(): str

Abbildung A.8: Klasse `PseudoEdge` des Pakets `model`

PseudoGraph
+ original_edges: FrozenSet[Edge] + edge_cost: Dict[Edge, int] + node_rating: Dict[Node, Union[int, float]]
+ __init__(nodes: Union[FrozenSet[Node], Set[Node]]=None, edges: Union[FrozenSet[Node], Set[Node]]=None) + get_edge_cost(edge: Union[Edge, PseudoEdge]): Union[float, int] + get_reduced_edge_cost(edge: Union[Edge, PseudoEdge]): Union[float int] + get_node_rating(node: Union[Node, PseudoNode]): Union[float, int] + get_rated_nodes(): Set[Union[Node, PseudoNode]] + update_node_rating(node: Union[Node PseudoNode], rating[Union[float int]) + shrink_to_pseudonode(cycle: Path): PseudoNode + expand_pseudonode(pseudonode: PseudoNode) + get_equality_edges(): Set[Union[Edge, PseudoEdge]] + has_pseudonode(): Union[PseudoNode, None] + get_replacement_edge(original_nodes: Tuple[Node, Node]): Union[Edge, PseudoEdge, None] + get_pseudonode_for_node(node: Node): Union[PseudoNode, None] + __str__(): str

Abbildung A.9: Klasse `PseudoGraph` des Pakets `model`

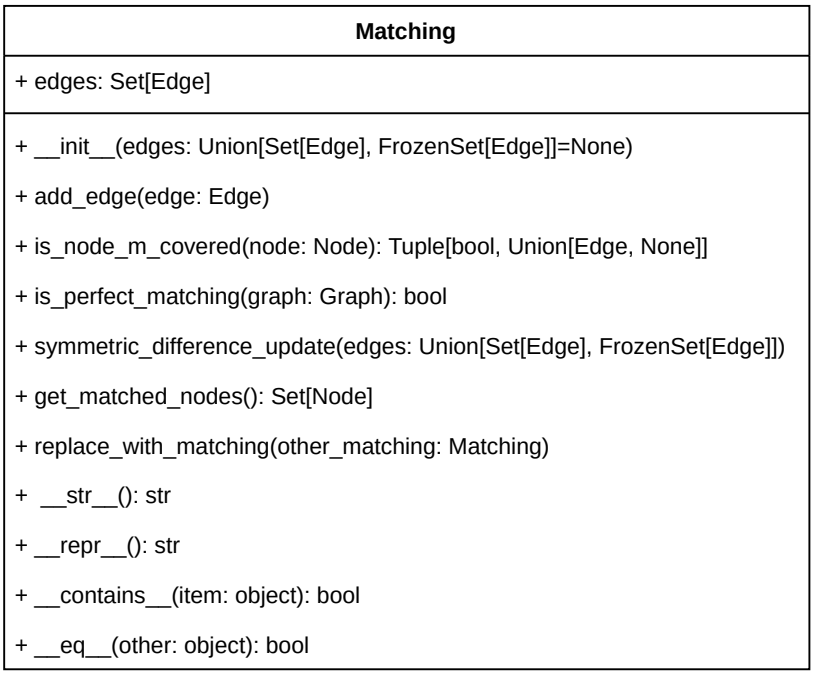


Abbildung A.10: Klasse `Matching` des Pakets `model`

AlternatingTree
+ root: Node + parent: AlternatingTree + children: Dict[Edge, AlternatingTree]
+ __init__(root: Node) + get_B(): Set[Node] + get_A(): Set[Node] + find_node(node: Node): Union[AlternatingTree, None] + add_child_tree(tree: AlternatingTree, edge: Edge) + add_path(path: Path) + remove_edge(edge: Edge): bool + get_path_from_root_to_node(node: Node): Union[Path, None] + find_nearest_common_tree(node1: Node, node2: Node): Union[AlternatingTree, None] + get_all_paths(): List[Path] + clear_tree(new_root: Node) + get_topmost_parent(): AlternatingTree + __contains__(item: object): bool + __str__(): str + __repr__(): str + __eq__(other: object): bool

Abbildung A.11: Klasse `AlternatingTree` des Pakets `model`

B Testabdeckung

Name	Stmts	Miss	Branch	BrPart	Cover
christofides/__init__.py	20	13	6	1	31%
christofides/__main__.py	2	2	0	0	0%
christofides/algorithms/__init__.py	6	0	0	0	100%
christofides/algorithms/algorithm.py	8	1	0	0	88%
christofides/algorithms/christofides.py	84	0	24	4	96%
christofides/algorithms/euler_to_hamilton.py	47	0	12	0	100%
christofides/algorithms/hierholzer.py	44	0	13	0	100%
christofides/algorithms/kruskal.py	28	2	12	1	92%
christofides/algorithms/minimum_perfect_matching.py	241	23	124	16	87%
christofides/model/__init__.py	11	0	0	0	100%
christofides/model/alternating_tree.py	130	1	80	1	99%
christofides/model/complete_graph.py	50	0	24	0	100%
christofides/model/edge.py	46	0	20	0	100%
christofides/model/graph.py	145	0	98	0	100%
christofides/model/matching.py	72	0	44	0	100%
christofides/model/node.py	23	0	6	0	100%
christofides/model/path.py	133	0	92	0	100%
christofides/model/pos.py	23	0	4	0	100%
christofides/model/pseudo_edge.py	20	0	8	0	100%
christofides/model/pseudo_graph.py	113	0	69	1	99%
christofides/model/pseudo_node.py	39	0	12	0	100%
christofides/util/__init__.py	0	0	0	0	100%
christofides/util/exceptions.py	10	0	0	0	100%
christofides/util/generator.py	12	0	4	1	94%
christofides/util/json_helper.py	24	0	12	0	100%
christofides/view/__init__.py	3	0	0	0	100%
christofides/view/graph_widget.py	118	26	28	3	75%
christofides/view/main_window.py	139	6	18	3	94%
christofides/view/scenes.py	108	70	42	1	26%
TOTAL	1699	144	752	32	90%

Tabelle B.1: Testabdeckung

C Testprotokoll

Ausgangszustand	**Aktionen**	**Akzeptanzkriterien**	**Ergebnis**
Hilfemenü			
Bearbeitungs-Modus ist aktiv.	Linksklick auf die Schaltfläche *Help* in der Menüleiste.	Ein weiteres Fenster öffnet sich, welches Hilfe für die Steuerung gibt.	✓
Bearbeitungs-Modus ist aktiv.	Drücken der Tastenkombination *Strg+H* auf der Tastatur.	Ein weiteres Fenster öffnet sich, welches Hilfe für die Steuerung gibt.	✓
Algorithmus-Modus ist aktiv.	Linksklick auf die Schaltfläche *Help* in der Menüleiste.	Ein weiteres Fenster öffnet sich, welches Hilfe für die Steuerung gibt.	✓
Algorithmus-Modus ist aktiv.	Drücken der Tastenkombination *Strg+H* auf der Tastatur.	Ein weiteres Fenster öffnet sich, welches Hilfe für die Steuerung gibt.	✓
Hilfe-Fenster ist offen.	Linksklick auf die Schaltfläche *Ok*.	Das Hilfe-Fenster ist wieder geschlossen.	✓
Hilfe-Fenster ist offen.	Linksklick auf die *Schließen* Schaltfläche des Hilfe-Fensters.	Das Hilfe-Fenster ist wieder geschlossen.	✓
Hilfe-Fenster ist offen.	Drücken der Taste *O* auf der Tastatur.	Das Hilfe-Fenster ist wieder geschlossen.	✓
Applikation schließen			
Bearbeitungs-Modus ist aktiv.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *Quit*.	Die Applikation wurde geschlossen.	✓
Bearbeitungs-Modus ist aktiv.	Linksklick auf die *Schließen* Schaltfläche des Fensters.	Die Applikation wurde geschlossen.	✓
Bearbeitungs-Modus ist aktiv.	Drücken der Tastenkombination *Strg+Q* auf der Tastatur.	Die Applikation wurde geschlossen.	✓
Algorithmus-Modus ist aktiv.	Linksklick auf die *Schließen* Schaltfläche des Fensters.	Die Applikation wurde geschlossen.	✓
Algorithmus-Modus ist aktiv.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *Quit*.	Die Applikation wurde geschlossen.	✓

Algorithmus-Modus ist aktiv.	Drücken der Tastenkombination *Strg+Q* auf der Tastatur.	Die Applikation wurde geschlossen.	✓

Öffnen-Dialog			
Bearbeitungs-Modus ist aktiv. Es gibt keinen Graphen.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *Open*.	Ein Dateiauswahldialogs öffnet sich.	✓
Bearbeitungs-Modus ist aktiv. Es gibt einen Graphen.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *Open*.	Ein Dateiauswahldialogs öffnet sich.	✓
Algorithmus-Modus ist aktiv.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *Open*.	Ein Dateiauswahldialogs öffnet sich.	✓
Bearbeitungs-Modus ist aktiv. Es gibt keinen Graphen.	Drücken der Tastenkombination *Strg+O* auf der Tastatur	Ein Dateiauswahldialogs öffnet sich.	✓
Bearbeitungs-Modus ist aktiv. Es gibt einen Graphen.	Drücken der Tastenkombination *Strg+O* auf der Tastatur	Ein Dateiauswahldialogs öffnet sich.	✓
Algorithmus-Modus ist aktiv.	Drücken der Tastenkombination *Strg+O* auf der Tastatur	Ein Dateiauswahldialogs öffnet sich.	✓
Graph aus JSON Datei öffnen			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer JSON Datei mit richtigem Format.	Der in der Datei spezifizierte Graph ist sichtbar.	✓
Bearbeitungs-Modus ist aktiv, ein Graph vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer JSON Datei mit richtigem Format.	Der in der Datei spezifizierte Graph ist sichtbar, der vorherige Graph ist nicht mehr sichtbar.	✓
Algorithmus-Modus ist aktiv, ein Graph vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer JSON Datei mit richtigem Format.	Der in der Datei spezifizierte Graph ist sichtbar, der vorherige Graph ist nicht mehr sichtbar und die Applikation ist im Bearbeitungs-Modus.	✓
JSON Datei mit falschem Format öffnen			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer JSON Datei mit falschem Format.	Ein neues Fenster öffnet sich und zeigt eine Fehlermeldung an.	✓
Bearbeitungs-Modus ist aktiv, ein Graph vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer JSON Datei mit falschem Format.	Ein neues Fenster öffnet sich und zeigt eine Fehlermeldung an. Nach Schließen der Fehlermeldung ist die Applikation im Ausgangszustand	✓

Algorithmus-Modus ist aktiv, ein Graph vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer JSON Datei mit falschem Format.	Ein neues Fenster öffnet sich und zeigt eine Fehlermeldung an. Nach Schließen der Fehlermeldung ist die Applikation im Ausgangszustand.	✓
Datei öffnen die kein gültiges JSON enthält			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer Datei die kein gültiges JSON enthält.	Ein neues Fenster öffnet sich und zeigt eine Fehlermeldung an.	✓
Bearbeitungs-Modus ist aktiv, ein Graph vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer Datei die kein gültiges JSON enthält.	Ein neues Fenster öffnet sich und zeigt eine Fehlermeldung an. Nach Schließen der Fehlermeldung ist die Applikation im Ausgangszustand	✓
Algorithmus-Modus ist aktiv, ein Graph vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer Datei die kein gültiges JSON enthält.	Ein neues Fenster öffnet sich und zeigt eine Fehlermeldung an. Nach Schließen der Fehlermeldung ist die Applikation im Ausgangszustand.	✓
Graph aus JSON Datei öffnen für die keine Leserechte existieren			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer JSON Datei mit richtigem Format, für die keine Leserechte vorhanden sind.	Ein neues Fenster öffnet sich und zeigt eine Fehlermeldung an.	✓
Bearbeitungs-Modus ist aktiv, ein Graph vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer JSON Datei mit richtigem Format, für die keine Leserechte vorhanden sind.	Ein neues Fenster öffnet sich und zeigt eine Fehlermeldung an. Nach Schließen der Fehlermeldung ist die Applikation im Ausgangszustand	✓
Algorithmus-Modus ist aktiv, ein Graph vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Auswählen einer JSON Datei mit richtigem Format, für die keine Leserechte vorhanden sind. s	Ein neues Fenster öffnet sich und zeigt eine Fehlermeldung an. Nach Schließen der Fehlermeldung ist die Applikation im Ausgangszustand.	✓
Schließen des Dateiauswahldialogss ohne Datei zu öffnen			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Schließen des Dateiauswahldialogs.	Die Applikation befindet sich im Ausgangszustand.	✓
Bearbeitungs-Modus ist aktiv, ein Graph vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Schließen des Dateiauswahldialogs.	Die Applikation befindet sich im Ausgangszustand.	✓

Algorithmus-Modus ist aktiv, ein Graph vorhanden und der Dateiauswahldialogs zum Öffnen ist offen.	Schließen des Dateiauswahldialogs.	Die Applikation befindet sich im Ausgangszustand.	✓

Speichern-Dialog			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *Save*.	Ein Dateiauswahldialog öffnet sich, in dem ein Dateiname angegeben werden kann.	✓
Bearbeitungs-Modus ist aktiv, ein Graph ist vorhanden.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *Save*.	Ein Dateiauswahldialog öffnet sich, in dem ein Dateiname angegeben werden kann.	✓
Algorithmus-Modus ist aktiv.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *Save*.	Ein Dateiauswahldialog öffnet sich, in dem ein Dateiname angegeben werden kann.	✓
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden.	Drücken der Tastenkombination *Strg+S* auf der Tastatur	Ein Dateiauswahldialog öffnet sich, in dem ein Dateiname angegeben werden kann.	✓
Bearbeitungs-Modus ist aktiv, ein Graph ist vorhanden.	Drücken der Tastenkombination *Strg+S* auf der Tastatur	Ein Dateiauswahldialog öffnet sich, in dem ein Dateiname angegeben werden kann.	✓
Algorithmus-Modus ist aktiv.	Drücken der Tastenkombination *Strg+S* auf der Tastatur	Ein Dateiauswahldialog öffnet sich, in dem ein Dateiname angegeben werden kann.	✓
Speichern einer Datei			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden und der Dateiauswahldialog zum Speichern ist offen.	Ein gültiger Dateiname wird angegeben und auf Speichern geklickt.	Die Datei wurde auf dem Dateisystem gespeichert und enthält einen leeren Graphen in JSON kodiert. Die Applikation ist im Ausgangszustand.	✓
Bearbeitungs-Modus ist aktiv, ein Graph ist vorhanden und der Dateiauswahldialog zum Speichern ist offen.	Ein gültiger Dateiname wird angegeben und auf Speichern geklickt.	Die Datei wurde auf dem Dateisystem gespeichert und enthält den Graphen in JSON kodiert. Die Applikation ist im Ausgangszustand.	✓
Algorithmus-Modus ist aktiv und der Dateiauswahldialog zum Speichern ist offen.	Ein gültiger Dateiname wird angegeben und auf Speichern geklickt.	Die Datei wurde auf dem Dateisystem gespeichert und enthält den vollständigen Ausgangsgraphen in JSON kodiert. Die Applikation ist im Ausgangszustand.	✓
Speichern einer Datei in Verzeichnis ohne Schreibrechte			

Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden und der Dateiauswahldialog zum Speichern ist offen.	Ein Verzeichnis wird ausgewählt, für das keine Schreibrechte vorhanden sind, ein gültiger Dateiname wird angegeben und auf Speichern geklickt.	Es öffnet sich ein Fenster mit einer Fehlermeldung. Die Applikation ist im Ausgangszustand.	✓
Bearbeitungs-Modus ist aktiv, ein Graph ist vorhanden und der Dateiauswahldialog zum Speichern ist offen.	Ein Verzeichnis wird ausgewählt, für das keine Schreibrechte vorhanden sind, ein gültiger Dateiname wird angegeben und auf Speichern geklickt.	Es öffnet sich ein Fenster mit einer Fehlermeldung. Die Applikation ist im Ausgangszustand.	✓
Algorithmus-Modus ist aktiv und der Dateiauswahldialog zum Speichern ist offen.	Ein Verzeichnis wird ausgewählt, für das keine Schreibrechte vorhanden sind, ein gültiger Dateiname wird angegeben und auf Speichern geklickt.	Es öffnet sich ein Fenster mit einer Fehlermeldung. Die Applikation ist im Ausgangszustand.	✓
Schließen des Dateiauswahldialogs ohne Datei zu speichern			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden und der Dateiauswahldialog zum Speichern ist offen.	Der Dateiauswahldialog wird geschlossen.	Die Applikation befindet sich im Ausgangszustand, keine Datei wurde gespeichert oder überschrieben.	✓
Bearbeitungs-Modus ist aktiv, ein Graph ist vorhanden und der Dateiauswahldialog zum Speichern ist offen.	Der Dateiauswahldialog wird geschlossen.	Die Applikation befindet sich im Ausgangszustand, keine Datei wurde gespeichert oder überschrieben.	✓
Algorithmus-Modus ist aktiv und der Dateiauswahldialog zum Speichern ist offen.	Der Dateiauswahldialog wird geschlossen.	Die Applikation befindet sich im Ausgangszustand, keine Datei wurde gespeichert oder überschrieben.	✓

Neuen Graphen erstellen			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *New*.	Die Applikation befindet sich im Bearbeitungs-Modus und es ist kein Graph vorhanden. Ein neuer Knoten hat das Label *A*.	✓
Bearbeitungs-Modus ist aktiv, ein Graph ist vorhanden.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *New*.	Die Applikation befindet sich im Bearbeitungs-Modus und es ist kein Graph vorhanden. Ein neuer Knoten hat das Label *A*.	✓

Algorithmus-Modus ist aktiv.	Linksklick auf die Schaltfläche *File*, danach Linksklick auf die Schaltfläche *New*.	Die Applikation befindet sich im Bearbeitungs-Modus und es ist kein Graph vorhanden. Ein neuer Knoten hat das Label *A*.	✓
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden.	Drücken der Tastenkombination *Strg+N* auf der Tastatur.	Die Applikation befindet sich im Bearbeitungs-Modus und es ist kein Graph vorhanden. Ein neuer Knoten hat das Label *A*.	✓
Bearbeitungs-Modus ist aktiv, ein Graph ist vorhanden.	Drücken der Tastenkombination *Strg+N* auf der Tastatur.	Die Applikation befindet sich im Bearbeitungs-Modus und es ist kein Graph vorhanden. Ein neuer Knoten hat das Label *A*.	✓
Algorithmus-Modus ist aktiv.	Drücken der Tastenkombination *Strg+N* auf der Tastatur.	Die Applikation befindet sich im Bearbeitungs-Modus und es ist kein Graph vorhanden. Ein neuer Knoten hat das Label *A*.	✓

Knoten erstellen			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden.	Linksklick auf eine weiße Stelle auf der Zeichenfläche.	Ein Knoten wird erstellt.	✓
Bearbeitungs-Modus ist aktiv, ein Knoten ist vorhanden.	Linksklick auf eine weiße Stelle auf der Zeichenfläche.	Ein zweiter Knoten wird erstellt. Eine Kante verbindet beide Knoten	✓
Bearbeitungs-Modus ist aktiv, ein vollständiger Graph mit mehreren Knoten ist vorhanden.	Linksklick auf eine weiße Stelle auf der Zeichenfläche.	Ein weiterer Knoten wird erstellt. Der Knoten erhält je eine Kante zu jedem anderen Knoten.	✓
Algorithmus-Modus ist aktiv.	Linksklick auf eine weiße Stelle auf der Zeichenfläche.	Es wird kein neuer Knoten erstellt.	✓
Knoten löschen			
Bearbeitungs-Modus ist aktiv, ein Knoten ist vorhanden.	Rechtsklick auf den Knoten.	Der Knoten ist gelöscht und es ist kein weiterer vorhanden.	✓
Bearbeitungs-Modus ist aktiv, ein vollständiger Graph mit mehreren Knoten ist vorhanden.	Rechtsklick auf einen Knoten.	Der Knoten wird gelöscht und alle Kanten die zum Knoten führten sind nicht mehr vorhanden.	✓
Bearbeitungs-Modus ist aktiv, ein vollständiger Graph mit mehreren Knoten ist vorhanden.	Rechtsklick auf die Zeichenfläche.	Kein Knoten wird gelöscht und die Applikation befindet sich im Ausgangszustand.	✓
Algorithmus-Modus ist aktiv.	Rechtsklick auf einen Knoten.	Es wird kein Knoten gelöscht.	✓

Algorithmus-Modus ist aktiv.	Rechtsklick auf die Zeichenfläche.	Es wird kein Knoten gelöscht.	✓
Knoten bewegen			
Bearbeitungs-Modus ist aktiv, ein Knoten ist vorhanden.	Linksklick auf einen Knoten und bei gedrückter Maustaste verschieben. Maustaste an anderer Position als der Ursprünglichen Knotenposition loslassen.	Der Knoten wird dupliziert und folgt dem Mauszeiger. Beim Loslassen wird der Knoten an der alten Position gelöscht und befindet sich an der neuen Position.	✓
Bearbeitungs-Modus ist aktiv, ein vollständiger Graph mit mehreren Knoten ist vorhanden.	Linksklick auf einen Knoten und bei gedrückter Maustaste verschieben. Maustaste an anderer Position als der Ursprünglichen Knotenposition loslassen.	Der Knoten wird dupliziert und folgt dem Mauszeiger. Beim Loslassen wird der Knoten an der alten Position gelöscht und befindet sich an der neuen Position. Alle Kanten die zur vorherigen Position des Knoten gingen, gehen jetzt zur neuen Position des Knoten.	✓
Algorithmus-Modus ist aktiv.	Linksklick auf einen Knoten und bei gedrückter Maustaste verschieben. Maustaste an anderer Position als der Ursprünglichen Knotenposition loslassen.	Es wird kein Knoten bewegt.	✓

Modus wechseln			
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden.	Linksklick auf die Schaltfläche *Toggle Mode* in der Menüleiste.	Ein Fenster mit einer Fehlermeldung öffnet sich. Nach Schließen der Fehlermeldung ist die Applikation im Ausgangszustand.	✓
Bearbeitungs-Modus ist aktiv, ein Graph mit einem Knoten ist vorhanden.	Linksklick auf die Schaltfläche *Toggle Mode* in der Menüleiste.	Ein Fenster mit einer Fehlermeldung öffnet sich. Nach Schließen der Fehlermeldung ist die Applikation im Ausgangszustand.	✓
Bearbeitungs-Modus ist aktiv, ein Graph mit mindestens zwei Knoten ist vorhanden.	Linksklick auf die Schaltfläche *Toggle Mode* in der Menüleiste.	Die Applikation befindet sich im Algorithmus-Modus.	✓
Algorithmus-Modus ist aktiv, der Eingabegraph ist sichtbar.	Linksklick auf die Schaltfläche *Toggle Mode* in der Menüleiste.	Die Applikation befindet sich im Bearbeitungs-Modus und der Eingabegraph ist sichtbar.	✓

Algorithmus-Modus ist aktiv, ein Zwischenschritt des Algorithmus ist sichtbar.	Linksklick auf die Schaltfläche *Toggle Mode* in der Menüleiste.	Die Applikation befindet sich im Bearbeitungs-Modus und der Eingabegraph ist sichtbar.	✓
Bearbeitungs-Modus ist aktiv, kein Graph ist vorhanden.	Drücken der Tastenkombination *Strg+T* auf der Tastatur.	Ein Fenster mit einer Fehlermeldung öffnet sich. Nach Schließen der Fehlermeldung ist die Applikation im Ausgangszustand.	✓
Bearbeitungs-Modus ist aktiv, ein Graph mit einem Knoten ist vorhanden.	Drücken der Tastenkombination *Strg+T* auf der Tastatur.	Ein Fenster mit einer Fehlermeldung öffnet sich. Nach Schließen der Fehlermeldung ist die Applikation im Ausgangszustand.	✓
Bearbeitungs-Modus ist aktiv, ein Graph mit mindestens zwei Knoten ist vorhanden.	Drücken der Tastenkombination *Strg+T* auf der Tastatur.	Die Applikation befindet sich im Algorithmus-Modus.	✓
Algorithmus-Modus ist aktiv, der Eingabegraph ist sichtbar.	Drücken der Tastenkombination *Strg+T* auf der Tastatur.	Die Applikation befindet sich im Bearbeitungs-Modus und der Eingabegraph ist sichtbar.	✓
Algorithmus-Modus ist aktiv, ein Zwischenschritt des Algorithmus ist sichtbar.	Drücken der Tastenkombination *Strg+T* auf der Tastatur.	Die Applikation befindet sich im Bearbeitungs-Modus und der Eingabegraph ist sichtbar.	✓

Im Algorithmus navigieren			
Bearbeitungs-Modus ist aktiv.	Linksklick auf die Zeichenfläche.	Es wird nicht der nächste Schritt des Algorithmus angezeigt.	✓
Bearbeitungs-Modus ist aktiv.	Rechtsklick auf die Zeichenfläche.	Es wird nicht der vorherige Schritt des Algorithmus angezeigt.	✓
Bearbeitungs-Modus ist aktiv.	Drücken der rechten Pfeiltaste auf der Tastatur.	Es wird nicht der nächste Schritt des Algorithmus angezeigt.	✓
Bearbeitungs-Modus ist aktiv.	Drücken der linken Pfeiltaste auf der Tastatur.	Es wird nicht der vorherige Schritt des Algorithmus angezeigt.	✓
Bearbeitungs-Modus ist aktiv.	Klicken auf die Schaltfläche *Next Step*.	Es wird nicht der nächste Schritt des Algorithmus angezeigt.	✓
Bearbeitungs-Modus ist aktiv.	Klicken auf die Schaltfläche *Previous Step*.	Es wird nicht der vorherige Schritt des Algorithmus angezeigt.	✓
Algorithmus-Modus ist aktiv, ein Zwischenschritt ist sichtbar.	Linksklick auf die Zeichenfläche.	Es wird der nächste Schritt des Algorithmus angezeigt.	✓
Algorithmus-Modus ist aktiv, ein Zwischenschritt ist sichtbar.	Rechtsklick auf die Zeichenfläche.	Es wird der vorherige Schritt des Algorithmus angezeigt.	✓

Algorithmus-Modus ist aktiv, ein Zwischenschritt ist sichtbar.	Drücken der rechten Pfeiltaste auf der Tastatur.	Es wird der nächste Schritt des Algorithmus angezeigt.	✓
Algorithmus-Modus ist aktiv, ein Zwischenschritt ist sichtbar.	Drücken der linken Pfeiltaste auf der Tastatur.	Es wird der vorherige Schritt des Algorithmus angezeigt.	✓
Algorithmus-Modus ist aktiv, ein Zwischenschritt ist sichtbar.	Klicken auf die Schaltfläche *Next Step*.	Es wird der nächste Schritt des Algorithmus angezeigt.	✓
Algorithmus-Modus ist aktiv, ein Zwischenschritt ist sichtbar.	Klicken auf die Schaltfläche *Previous Step*.	Es wird der vorherige Schritt des Algorithmus angezeigt.	✓

Tabelle C.1: Testprotokoll

D Beiliegender Datenträger

thesis_hafer.pdf	Diese Bachelorarbeit im PDF Format
example_graphs/	Einige Beispielgraphen als JSON Datei zur Verwendung in der Applikation
installer/	Installationsdateien
Christofides-1.4.3-amd64.msi	MSI Installer für Windows
Christofides-1.4.3-py3-none-any.whl	Python-Wheel zur OS-unabhängigen Installation
Christofides-1.4.3.tgz	Archiv mit Binärdateien für Linux
src/	Verzeichnis der Softwarequellen
christofides/	Quelltexte der Applikation/Paket *christofides*
LICENSE	Lizenz der Applikation
README.rst	Hilfestellung für die Installation der Entwicklungsumgebung
requirements.txt	Liste von Pakete zum Aussetzen einer Entwicklungsumgebung
setup.py	Installationsskript des Paketes *christofides*
tests/	Unittests für die Applikation