HAW HAMBURG

BACHELORARBEIT

Analyse und exemplarische Implementation eines klassischen Approximationsalgorithmus für das NP-vollständige Problem Vertex Cover

vorgelegt am 05. September 2025

Avery Laura Lönker

Erstprüfer: Prof. Dr. Edmund Weitz
Zweitprüfer: Jörg Balzer

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Department Medientechnik
Finkenau 35
22081 Hamburg

Zusammenfassung

Das Ziel der vorliegenden Arbeit ist die Analyse und exemplarische Implementation des im Jahr 1981 von Reuven Bar-Yehuda und Shimon Even veröffentlichten Approximationsalgorithmus für das Problem Vertex Cover. Einleitend wird das gewichtete Vertex-Cover-Problem vorgestellt, das im Verlauf der Arbeit wiederholt aufgegriffen und schließlich als NP-vollständig nachgewiesen wird. Im ersten Teil der Arbeit werden grundlegenden Konzepte der NP-Vollständigkeit aufgearbeitet, formal definiert und mit Hilfe von Beispielen veranschaulicht. Zudem wird das für den Algorithmus benötigte Hintergrundwissen aus Bereichen der Approximationsalgorithmen und der linearen Programmierung erläutert. Die anschließende Analyse des Algorithmus erfolgt anhand einer ausführlichen Beschreibung des Ablaufs, einer exemplarischen Python-Implementierung inklusive Laufzeittests und einer detaillierten Ausarbeitung der Beweise. Abschließend wird die historische Bedeutung der Entwicklung der ersten Primal-Dual-Methode für einen Approximationsalgorithmus durch Bar-Yehuda und Even dargestellt.

Das Verfahren wurde in der Programmiersprache Python implementiert, und die lineare Zeitkomplexität konnte mit einer Laufzeitanalyse bestätigt werden. Die Analyse ergab zudem, dass eine Python-Implementation des Verfahrens besondere Schritte erfordert, um das vorgegebene lineare Laufzeitverhalten zu gewährleisten. Eine schrittweise Erklärung konnte die Beweise übersichtlich wiedergeben und die Primal-Dual-Approximations-Methode hervorheben. Diese wurde von Bar-Yehuda und Even zum ersten Mal für die iterative Konstruktion einer approximierten Lösung eines NP-vollständigen Problems verwendet und ermöglichte seitdem den Entwurf von vielen weiteren Approximationsalgorithmen für NP-vollständige Probleme.

Abstract

The goal of this thesis is to analyse and implement the approximation algorithm for the vertex cover problem introduced in 1981 by Reuven Bar-Yehuda and Shimon Even. The thesis begins with an introduction to the weighted vertex cover problem, which will play a recurring role in later chapters and will be demonstrated to be NP-complete. In chapter 2, the fundamental concepts of NP-completeness will be established and explained in detail, accompanied by multiple illustrative examples. Additional concepts from approximation algorithms and linear programming, essential for the algorithm, will be presented in this chapter. The subsequent study of the algorithm is broken down into three parts: Firstly, the algorithm process will be explained step by step. Secondly, an exemplary Python implementation including runtime tests will be given. Lastly, the underlying proof of the algorithm will be elaborated in detail.

The last chapter will conclude by presenting the historical significance of the development of the first primal-dual method for an approximation algorithm by Bar-Yehuda and Even.

The algorithm was implemented in Python and shown to achieve linear runtime, verified through multiple runtime tests. The study further revealed that ensuring linear runtime in a Python implementation of the method requires special steps to be taken. A systematic explanation provided the necessary proof in a clear and illustrative manner and highlighted the use of the primal-dual approximation method. Originally developed by Bar-Yehuda and Even for the iterative construction of approximate solutions to NP-complete problems, this method has since enabled the design of numerous other approximation algorithms for such problems.

Inhaltsverzeichnis

Abbildungsverzeichnis

Programmverzeichnis

Algorithmusverzeichnis

1 Einleitung

Im Jahr 1981 veröffentlichten die israelischen Mathematiker Reuven Bar-Yehuda und Shimon Even einen Algorithmus für das gewichtete Vertex-Cover-Problem[1] und ermöglichten auf diese Weise viele Fortschritte im Bereich der Optimierungs- und Approximationsalgorithmen (Bar-Yehuda & Even, 1981).

Das Ziel dieser Arbeit ist, den Approximationsalgorithmus von Bar-Yehuda und Even vorzustellen, zu implementieren und zu analysieren. Insbesondere werden anschauliche Beispiele und ausführliche Erklärungen entwickelt, die es auch Personen, die sich noch nicht ausführlich mit der NP-Vollständigkeit beschäftigt haben, ermöglichen soll, dem Inhalt zu folgen. Trotzdem wird für das Verständnis dieser Arbeit Wissen vorausgesetzt, welches typischerweise in den ersten Semestern eines Informatikgrundstudiums oder -nebenfachs vermittelt wird, und beispielsweise mit dem Abschluss der ersten vier Semester des Bachelorstudiengang „Media Systems“ an der Hochschule für Angewandte Wissenschaften Hamburg erworben werden kann.

In Abschnitt 1.1 wird das gewichtete Vertex-Cover-Problem anhand eines Anwendungsbeispiels vorgestellt, das als Leitbeispiel für Erklärungen wiederholt aufgegriffen wird. Anschließend werden in Kapitel 2 die theoretischen Grundlagen und wichtige Begriffe aus den Bereichen der NP-Vollständigkeit und der Approximationsalgorithmen eingeführt und anschaulich erklärt. Zusätzlich wird gezeigt, dass das gewichtete Vertex-Cover-Problem NP-vollständig ist. In Kapitel 3 folgt der Hauptteil dieser Arbeit, in dem der Algorithmus von Bar-Yehuda und Even schrittweise vorgestellt wird. Insbesondere soll der Ablauf detailliert beschrieben, eine Python-Implementation des Verfahrens inklusive Laufzeittests vorgestellt und die für das Verfahren benötigten Beweise anschaulich erklärt werden. Die Arbeit wird in Kapitel 4 mit einem Fazit abgerundet, in dem auch die geschichtliche Relevanz der ersten Realisierung der Primal-Dual-Methode für einen Approximationsalgorithmus durch Bar-Yehuda und Even hervorgehoben wird.

[1] Deutsch: Knotenüberdeckungsproblem. Für eine einheitliche Benennung wird im Rest der Arbeit nur noch der englische Begriff Vertex Cover und dessen Abkürzung VC verwendet.

1.1 Ein Leitbeispiel

Das Hauptproblem, mit dem sich Bar-Yehuda und Even in ihrer Publikation beschäftigt haben, ist das gewichtete Vertex-Cover-Problem (Bar-Yehuda & Even, 1981). Für ein ausführliches Verständnis dieses Sachverhalts folgt nun ein konkretes Beispiel mit Alltagsbezug, welches in den folgenden Kapiteln dieser Arbeit wiederholt aufgegriffen wird.

Man stelle sich ein Kommunikationsnetzwerk eines großen Unternehmens vor. Insgesamt beinhaltet es 100 Komponenten, die in unübersichtlicher Weise miteinander verbunden wurden. Netzwerke lassen sich gut als Graphen modellieren und ein kleiner, vom Hauptteil des Kommunikationsnetzwerks abgetrennter Teil ist in Abbildung 1.1 zu sehen.

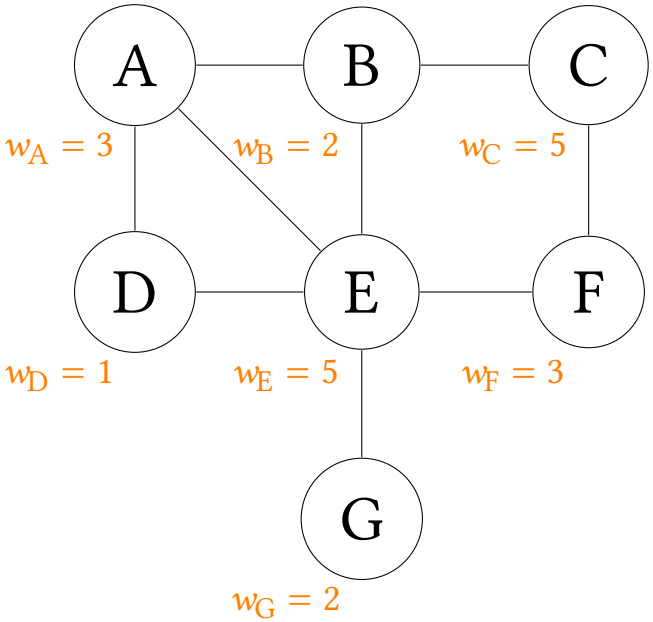

Abbildung 1.1: Gewichteter Graph des kleinen Teilnetzwerks [eigene Darstellung]

Die Knoten des abgebildeten Graphen repräsentieren die Endgeräte, Hubs und Switches, während die Kanten die dazwischenliegenden Datenverbindungen abbilden. Eine fiktive Person namens Vera ist eine der Netzwerkadministratoren für dieses Unternehmen und hat sich die Aufgabe zugeteilt, Überwachungsgeräte an den einzelnen Endpunkten zu installieren. Diese Einheiten sollen das gesamte Netzwerk überwachen, aber können jeweils nur direkt anliegende Verbindungen abhören. Eine Einheit kostet viel Geld und je nach vorliegender Hardware kann eine unterschiedlich hohe Summe für die Installation an einer bestimmten Netzwerkkomponente anfallen. In Abbildung 1.1 sind die verschiedenen Kosten für jeden Knotenpunkt in orangener Farbe aufgetragen; in der Graphentheorie werden sie auch als *Gewichte* bezeichnet. Ausschlaggebend für den Erfolg der Sicherheitsstrategie ist, dass nach der Installation ausnahmslos jede Verbindung überwacht werden muss. Eine Knotenmenge,

die diese Bedingung erfüllt, bezeichnet man – entsprechend dem Namen des Problems – als *Vertex Cover*. Ein mögliches Vertex Cover des Teilnetzwerks mit einem Gesamtgewicht von 13 könnte aus den Knoten A, B, E und F gebildet werden und ist in Abbildung 1.2 eingezeichnet.

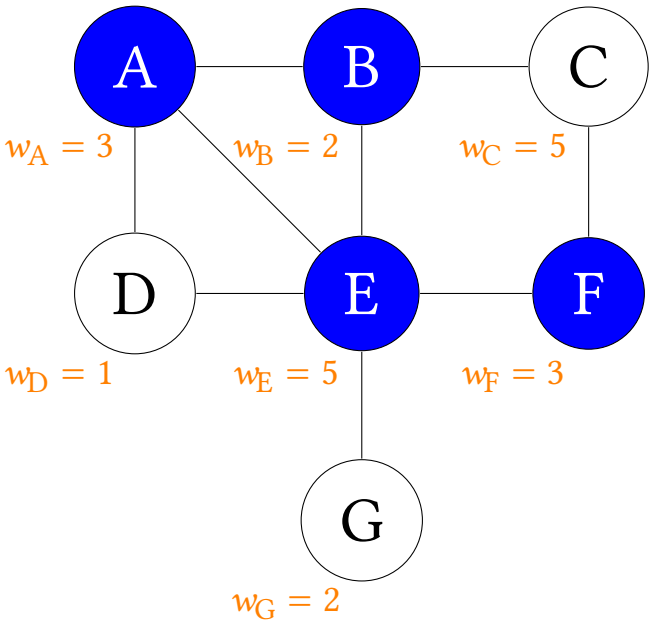

Abbildung 1.2: Die Knotenmenge {A, B, E, F} bildet ein Vertex Cover [eigene Darstellung].

Eine Möglichkeit, ein Vertex Cover für das gesamte Netzwerk zu erhalten, wäre die Installation einer Überwachungseinheit an jeder Schnittstelle, da auf diese Weise jede Verbindung von beiden Seiten abgehört wird. Ein derartiger Ansatz wäre jedoch mit hohen Kosten verbunden und würde zu einem merklich ineffizienten Betrieb der Überwachungssysteme führen. Stattdessen möchte Vera zuerst berechnen, wie eine optimale Verteilung von Einheiten für ihr Netzwerk aussieht, sodass nur eine minimale Anzahl von Geräten benötigt wird.

Als erster Lösungsansatz kommt ihr der Gedanke, alle möglichen Verteilungen von Überwachungseinheiten durchzugehen, um die optimale Lösung ausfindig zu machen. Diese Methode wird auch als *Brute-Force* bezeichnet und ist in der Regel äußerst ineffizient. Auch für Veras Problem trifft dies zu, denn insgesamt gibt es 2^{100} Kombinationen, eine einzigartige Menge aus den 100 Knoten zu bilden. Das sind etwa $1.27 \cdot 10^{30}$ Möglichkeiten, bei denen Vera überprüfen müsste, ob ein neues optimales Vertex Cover gefunden werden konnte. Diese außerordentlich große Anzahl kommt dadurch zustande, dass jeder Knoten die Anzahl der möglichen Knotenmengen verdoppelt, weil an dem jeweiligen Endpunkt entweder ein Gerät installiert werden kann oder nicht. Dies resultiert in einem exponentiellen Wachstum der Möglichkeiten, welches in Abbildung 1.3 dargestellt ist. Die orangefarbenen Abzweigungen nach links zeigen, wie jeder neue Knoten zu allen bisherigen Mengen hinzugefügt werden kann, während die blauen Abzweigungen nach rechts eine vorherige Menge unverändert

übernehmen. Dadurch entsteht für jede mögliche Knotenmenge eines Graphen der Größe n, jeweils zwei neue Kombinationen in einem Graphen der Größe $n + 1$.

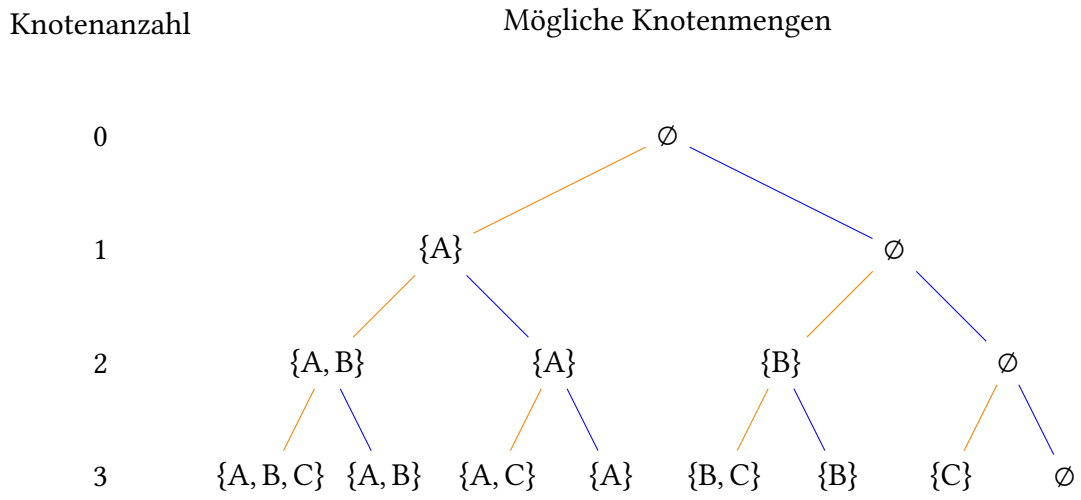

Abbildung 1.3: Exponentieller Anstieg möglicher Knotenmengen bei steigender Knotenanzahl [eigene Darstellung]

Selbst wenn Vera einen Computer verwenden würde, der eine Milliarde Kombinationen in einer Sekunde ausprobieren könnte, bräuchte dieser ca. 40 Billionen Jahre,[2] um eine optimale Antwort zu garantieren. Moderne Computer haben in den letzten Jahrzehnten eine erstaunliche Geschwindigkeit erreicht, aber diese Größe ist auch für sie noch unerreichbar.[3] Vera bleibt also nichts anderes übrig als nach einer besseren Lösung zu suchen. Nach einer kurzen Recherche findet sie heraus, dass ihr Problem allgemein unter dem Namen Vertex Cover bekannt ist und zu den NP-vollständigen Problem gehört.

[2] Das ist ca. das zweitausendfache des aktuellen Alters unseres Universums.

[3] Einer Statistik vom Juni 2025 zufolge kann der aktuell leistungsstärkste Computer ca. 1.7 Trillionen FLOPS (Gleitkomma-Rechenoperationen pro Sekunde) erreichen (Statista Research Department, 2025). Dass das für die Brute-Force Variante von Veras Problem immer noch nicht ausreichend ist, kann man an dieser Stelle selbst nachrechnen.

2 Grundlegende Konzepte

Um die Konzepte der NP-Vollständigkeit zu verstehen und eine mögliche Lösung für Veras „schweres“ Problem zu finden, muss zunächst geklärt werden, was „schwer“ in diesem Zusammenhang bedeutet und wie solche Probleme von „leichteren“ Aufgaben unterschieden werden können. In den folgenden Unterkapiteln werden daher wichtige Begriffe definiert, die für die Lösung des gewichteten Vertex-Cover-Problems relevant sind. Die Hauptquelle dieses Kapitels ist das Buch „Computers and Intractability: A Guide to the Theory of NP-Completeness“ von Michael R. Garey und David S. Johnson (Garey & Johnson, 1979). Soweit nicht anders gekennzeichnet, stammen sämtliche Definitionen und Erkenntnisse, die in Abschnitt 2.1, Abschnitt 2.2, Abschnitt 2.3, Abschnitt 2.4 und Abschnitt 2.5 vorgestellt werden, aus diesem Buch.

2.1 Komplexitätsklassen und Kodierungsschemata

Die Komplexitätstheorie untersucht Probleme hinsichtlich ihrer Laufzeit und ordnet sie entsprechenden *Komplexitätsklassen* zu. Für diese Klassifikation werden jedoch keine genauen Zeitbeträge betrachtet, sondern es wird das Laufzeitverhalten eines Verfahrens in Abhängigkeit von der Eingabegröße analysiert. Erhält ein Verfahren zwei Eingaben von unterschiedlicher Länge, so benötigt die umfangreichere in der Regel eine längere Bearbeitungszeit, da möglicherweise mehr Informationen oder Kombinationsmöglichkeiten beachtet werden müssen. Auch Veras Beispielproblem teilt diese Eigenschaft, denn das kleine, aus sieben Knoten bestehende, Teilnetzwerk bietet nur 2^7 mögliche Knotenmengen, die man bereits im Kopf allesamt durchprobieren könnte, während das Lösen des gesamten Netzwerks einen unvorstellbaren Zeitaufwand benötigen würde (siehe Abschnitt 1.1). Wie rapide diese Zeitbeträge mit steigender Eingabegröße zunehmen, kann mit der *Landau-Schreibweise* ausgedrückt werden. Mit dem Landau-Symbol $\mathcal{O}$ lässt sich beschreiben, dass ein Laufzeitverhalten nicht deutlich schneller wächst als ein angegebener Term. Eine Laufzeit von $\mathcal{O}(n)$ bedeutet beispielsweise, dass die benötigte Zeit des Algorithmus mit größer werdenden Eingaben maximal linear ansteigt. Man sagt in diesem Fall auch die Laufzeit ist von der Ordnung n (Weitz, 2021, S. 487). Der Ausdruck innerhalb der Landau-Schreibweise bildet dabei in der Regel nur den schnellsten steigenden Term der tatsächlichen Steigung ab, wobei konstante

Faktoren ignoriert werden. Wenn ein Laufzeitverhalten mit den Funktionen $f(n) = n + 15$ oder $g(n) = 100n$ beschrieben werden kann, wäre es daher in beiden Fällen von der Ordnung n.

Der für diese Arbeit relevante Teil der Komplexitätstheorie unterscheidet insbesondere zwischen Algorithmen mit *polynomieller Laufzeit* und solchen mit *exponentieller Laufzeit*. Denn eine Laufzeit von exponentieller Ordnung verursacht grundsätzlich bereits bei sehr wenigen Erhöhungen der Eingabelänge einen extremen Anstieg der Zeitbeträge. Das hat zur Folge, dass ein Algorithmus mit exponentiellem Laufzeitverhalten Eingaben ab einer bestimmten Größe nicht mehr in praktikabler Zeit verarbeiten kann.

Als polynomielle Laufzeit bezeichnet man ein Laufzeitverhalten, das nicht schneller als ein beliebiges Polynom wächst. Eine Laufzeit, die proportional zu n^2 ansteigt, könnte auf diese Weise beschrieben werden. Exponentielle Laufzeit wächst hingegen schneller als jedes Polynom. Ein mögliches Beispiel hierfür ist die Funktion $f(n) = 2^n$, da kein Polynom langfristig ein höheres Wachstum erreichen kann. In Abbildung 2.1 sind die beiden beispielhaften Laufzeitverhalten eingezeichnet und die Unterschiede in der Größenordnung ihrer Steigungen sind deutlich erkennbar.[1]

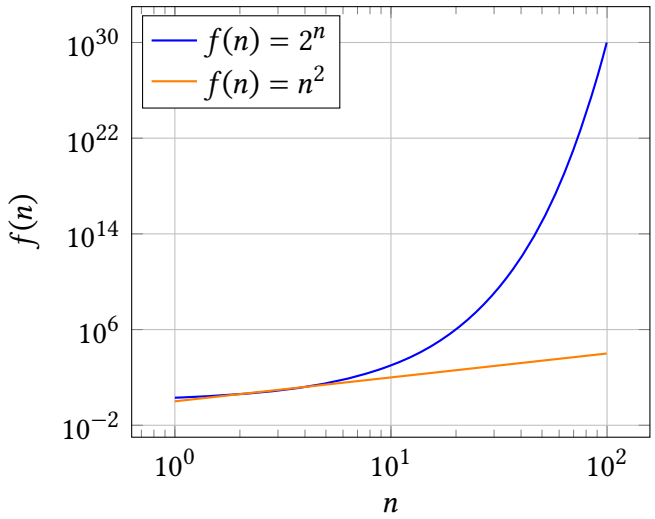

Abbildung 2.1: 2^n und n^2 im Vergleich [eigene Darstellung nach Garey und Johnson, 1979, S. 7]

Manchem könnte es als befremdlich vorkommen, das Laufzeitverhalten von der Eingabegröße abhängig zu machen, obwohl Informationen in vielen verschiedenen Schreibweisen an einen Algorithmus weitergegeben werden können. Bereits die Kodierung einer Zahl – zum Beispiel die 42 – lässt sich in Eingaben verschiedener Länge notieren. Sie könnte dezimal als „42“, binär als „101010“, ausgeschrieben als „Zweiundvierzig“ oder unär als „1111111111 1111111111

[1] Es sei an dieser Stelle auf die logarithmischen Achsen hingewiesen.

1111111111 1111111111 11“ kodiert werden und in jedem Fall eine unterschiedliche Länge verursachen. Eine Schreibweise, in der alle für ein Problem wichtigen Informationen einheitlich kodiert werden können, nennt man ein *Kodierungsschema* (engl. encoding scheme). Eine Kodierung für das klassische Vertex-Cover-Problem[2] könnte die Information eines Graphens beispielsweise in eine Adjazenzmatrix oder eine Nachbarschaftsliste übertragen. Ein Vergleich verschiedener Kodierungsschemata wird in dieser Arbeit jedoch nicht behandelt, da bewiesen werden kann, dass das Verwenden von verschiedenen Kodierungsschemata sich in der Regel nicht merkbar auf das Laufzeitzeitverhaltens eines Algorithmus auswirkt: Solange eine Eingabe mit einem „nachvollziehbaren“ Kodierungsschema enkodiert wurde, unterscheidet sich ihre Länge maximal polynomiell von der Länge derselben Eingabe bei Verwendung eines anderen „nachvollziehbaren“ Kodierungsschemata; das Laufzeitverhalten eines Verfahrens variiert durch polynomielle Längenunterschiede der Eingabe ebenfalls höchstens um einem polynomiellen Faktor und wie sich später herausstellen wird, ist dieser polynomielle Unterschied für Vertex Cover vernachlässigbar. „Nachvollziehbar“ bedeutet in diesem Zusammenhang, dass unnötige Füller in der Kodierung vermieden werden müssen und für die Repräsentation von Zahlen, nur ein Stellenwertsystem der Basis Zwei oder höher verwendet werden darf (Garey & Johnson, 1979, S. 10). Die unäre und die ausgeschriebene Schreibweise der Zahl 42 sind aus diesem Grund nicht erlaubt.

2.2 Entscheidungsprobleme

Bevor mit dem Definieren einzelner Komplexitätsklassen begonnen werden kann, muss zuerst noch ein wichtiger Grundbegriff erläutert werden. Aus Gründen ihrer Praktikabilität, werden meistens sogenannte *Entscheidungsprobleme* für Definitionen im Bereich der Komplexitätstheorie verwendet. Das gewichtete Vertex-Cover-Problem wurde in Abschnitt 1.1 als Optimierungsproblem eingeführt, kann aber auch in einer leicht veränderten Variante als Entscheidungsproblem formuliert werden.

Ein Entscheidungsproblem Π besteht aus zwei Teilen: Eine unendliche Menge von *Instanzen* D_Π, die jeweils genaue Angaben für die zu lösende Aufgabe beinhalten, und eine *generische Frage*, die nur mit „Ja“ oder „Nein“ beantwortet und auf alle Instanzen angewandt werden kann. Die generische Frage der alternativen Variante für das gewichtete Vertex-Cover-Problem könnte beispielsweise folgendermaßen formuliert werden:

Besitzt der Graph $G = (V, E)$, der aus den Knoten V und Kanten E und den Gewichten w_v für alle $v \in V$ besteht, eine Knotenmenge $I \subseteq V$, sodass jede Kante in E mindestens einen Endpunkt in I hat und das Gesamtgewicht von I maximal B ist?

[2] Damit ist das ungewichtete Vertex-Cover-Problem gemeint, das dieselbe Zielsetzung wie das gewichtete hat, aber statt Gewichten die Anzahl der Knoten in einem Vertex Cover minimieren soll.

Eine Instanz des gewichteten Vertex-Cover-Entscheidungsproblems muss alle für die Frage relevanten Informationen zur Verfügung stellen. Dies umfasst einen vollständigen gewichteten Graphen und eine zusätzliche Grenze B. Betrachtet man nun das kleine Teilnetzwerk aus Veras Beispiel (siehe Abbildung 1.1), stellt man fest, dass es als Instanz noch nicht vollständig ist. Es fehlt die obere Schranke B, die das Maximalgewicht, das ein gefundenes Vertex Covers haben darf, festlegt. Wird dieser Schranke ein konkreter Wert – beispielsweise die 12 – zugeordnet, bildet sie zusammen mit dem Graphen eine Instanz des Entscheidungsproblems gewichtetes Vertex Cover, die nur mit „Ja“ oder „Nein“ beantwortet werden kann:

Ein Graph $G = (V, E)$ *mit Knoten* $V = \{A, B, C, D, E, F, G\}$ *, Kanten* $E = \{AB, AD, AE, BC, BE, CF, DE, EF, EG\}$ *und Gewichten* $(3, 2, 5, 1, 5, 3, 2)$ *und eine Grenze* $B = 12$.

Jede Kombination eines beliebigen Graphens mit einer beliebigen Grenze bildet eine neue Instanz. Trägt man alle dadurch möglichen Instanzen zusammen, erhält man die unendliche Menge $D_{\Pi_{VC}}$, die zusammen mit der oben bestimmten generischen Frage das gewichtete Vertex-Cover-Entscheidungsproblem konstituiert. Da eine Menge D_{Π} unmöglich formuliert werden kann, nutzt man häufig eine *generische Instanz* um ein Problem zu beschreiben. Diese gibt an, welche Informationen jede Instanz aus D_{Π} liefern muss. Die generische Instanz für das gewichtete Vertex-Cover-Problem sieht wie folgt aus:

Ein Graph $G = (V, E)$ *mit Gewichten* w_v *für alle* $v \in V$ *und eine Grenze* B.

Eine alternative Schreibweise von Entscheidungsproblemen tauscht die generische Frage mit einer Menge von *Ja-Instanzen* J_{Π} aus. J_{Π} kann man sich als eine unendliche Menge vorstellen, die alle Instanzen beinhaltet, die ein „Ja“ als Resultat auf die generischen Frage liefern.

Das gewichtete Vertex-Cover-Entscheidungsproblem wird in formalen Definitionen als Stellvertreter des verwandten Optimierungsproblems fungieren, weil sich die für das Lösen einer gegebenen Instanz nötigen Laufzeiten der beiden Varianten nur polynomiell voneinander unterscheiden. Umgangssprachlich könnte man sagen, beide Probleme sind etwa gleich schwer. Findet man für eine Instanz des Entscheidungsproblems ein Vertex Cover, dessen Gewicht unter der gegebenen Grenze liegt, kann man durch iteratives Anpassen der Schranke das optimale Vertex Cover ausfindig machen. Wie viele Iterationen dafür benötigt werden, ist durch die Anzahl der Knoten beschränkt, weil ein optimales Vertex Covers maximal alle Knoten beinhalten kann.[3] Da die Eingabelänge unter anderem auch von der Anzahl der Knoten abhängig ist, übersteigt die Gesamtlaufzeit aller benötigten Wiederholungen die Laufzeit einer einzelnen Iteration nur um einen polynomiellen Faktor.

[3] Verwendet man einen effizienten Suchalgorithmus, wie zum Beispiel die binäre Suche, lässt sich das optimale Vertex Cover eines Graphens mit 100 Knoten bereits durch siebenfaches Lösen des Entscheidungsproblems bestimmen.

2.3 Die Komplexitätsklasse P

Die erste Komplexitätsklasse, die in diesem Kapitel vorgestellt wird, ist die Komplexitätsklasse P. Das P steht für Polynomialzeit und benennt eine Klasse von Problemen, die algorithmisch in polynomieller Zeit gelöst werden können. Für eine formale Definition dieser Klasse müssen in den folgenden Abschnitten noch ein paar Konzepte spezifiziert werden. Aus einem für diese Arbeit vorausgesetzten Grundkurs der Informatik (wie beispielsweise die Vorlesung „Mathematische Methoden der Informatik“ des Media Systems Studiengangs an der HAW) sollten die Konzepte von deterministischen Turingmaschinen und formalen Sprachen bereits bekannt sein. Im folgenden Abschnitt werden sie daher ohne genaue Erklärungen formuliert.

Eine *formale Sprache* L wird über einem *Alphabet* Σ gebildet, welches alle in der Sprache vorkommenden Symbole beinhaltet. Σ^* umfasst alle „Wörter“, die mit diesem Alphabet gebildet werden könnten. Ist eine Eingabe binär kodiert, ist das Alphabet $\Sigma = \{0, 1\}$ und $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, ...\}$.[4] Eine Sprache L ist eine Teilmenge von Σ^* und beinhaltet alle durch sie erlaubten Wörter. Eine mögliche über dem Alphabet $\{0, 1\}$ gebildete Sprache wäre beispielsweise $\{01, 101, 1110, 0101100\}$. Sie beinhaltet vier erlaubte Wörter; „01“ ist eines von ihnen. Die formale Sprache $L[\Pi, e]$ kann für ein Entscheidungsproblem Π mit einem festgelegten Kodierungsschema e mit

$$L[\Pi, e] = \left\{ x \in \Sigma^* \ : \begin{array}{l} \Sigma \text{ ist das Alphabet, welches von } e \text{ benutzt wird und} \\ x \text{ ist die Kodierung der Instanz } I \in J_\Pi \text{ unter } e \end{array} \right\}$$

definiert werden. Die in dieser Sprache vorkommenden Wörter sind mit e kodierte Instanzen des Problems Π, die alle in der Menge J_Π enthalten sind und folglich von der generischen Frage des Problems mit „Ja“ beantwortet werden. In simpler Sprache ausgedrückt, beinhaltet $L[\Pi, e]$ alle kodierten „Ja-Instanzen“ eines Problems.

Es folgt der Aufbau einer *deterministischen Einband-Turingmaschine* (DTM). In diesem Kapitel wird sie für die Definition der Polynomialzeit verwendet werden, um präzise Formulierungen zu gewährleisten. Gedanklich lässt sich eine DTM jedoch durch jedes etablierte Computermodell ersetzen, da sie im Bereich der polynomiellen Zeitkomplexität äquivalent zu fast allen modernen Computern ist.[5] Eine DTM besteht aus einem endlichen *Steuerungsautomaten* (engl. Finite State Control), einem *Lese-Schreibkopf* und einem *Band* mit unendlich vielen Feldern in beiden Richtungen, die mit ganzen Zahlen beschriftet werden. In Abbildung 2.2 ist eine beispielhafte Visualisierung zu sehen. Ein auf der DTM laufendes Programm M agiert, indem es zwischen verschiedenen Zuständen wechselt und dabei bestimmte Operationen auf dem Band ausübt. Sobald das Programm in einem der beiden Endzustände q_{Ja} oder q_{Nein}

[4] ϵ ist das leere Wort.

[5] Seltene Ausnahmen – zum Beispiel Quantencomputer – existieren, sollen aber in dieser Arbeit nicht weiter beschrieben werden.

landet, endet es und gibt die jeweilige Antwort zurück. Ist das Resultat für eine gegebene Eingabe x „Ja“, spricht man allgemein davon, dass M mit dem Eingabe-Alphabet Σ die Eingabe $x \in \Sigma^*$ *akzeptiert.*

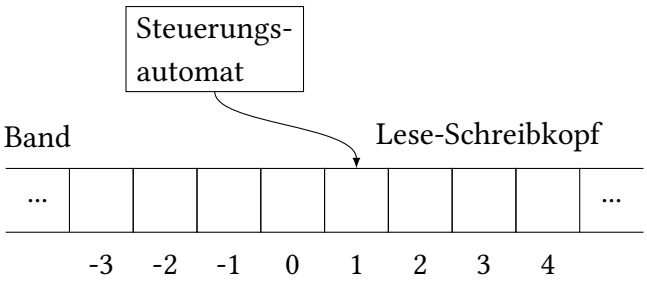

Abbildung 2.2: Aufbau einer deterministischen Einband-Turingmaschine (DTM) [eigene Darstellung nach Garey und Johnson, 1979, S. 23]

Die Sprache L_M kann mit

$$L_M = \{x \in \Sigma^* \ : \ M \text{ akzeptiert } x\}$$

definiert werden. Sie beinhaltet alle Eingaben, die von M akzeptiert werden und man sagt dazu, dass L_M durch das Programm M *anerkannt* wird. Ein Programm M *löst* ein Entscheidungsproblem Π unter einem gegebenen Kodierungsschema e, genau dann, wenn es für jede Eingabe nach einer endlichen Anzahl von Schritten zum Halten kommt und $L_M = L[\Pi, e]$. Einfacher ausgedrückt, M löst Π, wenn es für jede Instanz die richtige Lösung der generischen Frage von Π ausgibt und dabei nie in einer Endlosschleife landet. Die formale Definition einer Zeitkomplexitätsfunktion $T_M : \mathbb{Z}^+ \to \mathbb{Z}^+$, die für eine gegebene Eingabenlänge die von M benötigte Zeit wiedergibt, ist nun möglich und lautet

$$T_M(n) = \max \left\{ m \ : \ \begin{array}{c} \text{es gibt eine Eingabe } x \in \Sigma^*, \text{ mit } |x| = n, \text{ sodass} \\ \text{die Ausführung von } M \text{ die Zeit } m \text{ benötigt} \end{array} \right\}.$$

M ist ein *Polynomialzeit-DTM-Programm*, genau dann, wenn ein Polynom p existiert, so dass für alle $n \in \mathbb{Z}^+$ gilt $T_M(n) \leq p(n)$. Einfacher ausgedrückt, ein Programm hat eine polynomielle Zeitkomplexität, wenn dessen Laufzeit nicht schneller wächst, als ein mögliches Polynom. Mit dieser Information lässt sich die Komplexitätsklasse P ebenfalls mit

$$\mathrm{P} = \{L \ : \ \text{Es gibt ein Polynomialzeit-DTM-Programm } M, \text{ für das gilt } L = L_M\}$$

genau bestimmen. Ein Problem Π gehört unter dem Kodierungsschema e zu P, wenn $L[\Pi, e] \in \mathrm{P}$, das heißt, wenn ein in Polynomialzeit laufendes Programm M das Problem Π unter e löst. Da in Abschnitt 2.1 bereits festgestellt werden konnte, dass verschiedene „vernünftige“ Kodierungsschemata höchstens eine Veränderung der Zeitkomplexität um einen polynomiellen Faktor hervorrufen, kann das Schema e in dieser Bezeichnung unter der Annahme, dass ein

solches „vernünftiges“ Schema vorliegt, weggelassen werden. Dies ermöglicht die Aussage $\Pi \in P$, also ein Problem Π liegt in P.

Wie in Abschnitt 1.1 bereits festgestellt wurde, benötigt ein Brute-Force-Verfahren für Vertex Cover mindestens eine Laufzeit der Ordnung 2^n. Dies deutet darauf hin, dass das Problem nicht in P liegt, ist aber als Beweis noch nicht ausreichend. Ob man zum aktuellen Zeitpunkt überhaupt beweisen kann, dass das gewichtete Vertex-Cover-Problem nicht in P liegt, wird im Abschnitt 2.5 dieser Arbeit erläutert werden.

2.4 Die Komplexitätsklasse NP

Ein häufiges Missverständnis in der Komplexitätstheorie ist, dass die Klasse NP ein Gegenstück zur Klasse P darstellt und eine Abkürzung für „Nicht-Polynomiell“ ist. Beide Behauptungen sind falsch und sollten für ein bestmögliches Verständnis verworfen werden. Denn die Klasse NP steht für „Nichtdeterministisch Polynomiell“ und ist keineswegs disjunkt von der Klasse P, sondern beinhaltet diese sogar. Der Erklärungsstruktur von Garey und Johnson (Garey & Johnson, 1979) folgend, wird die Klasse NP zunächst nur informell an einem Beispiel beschrieben, um die nachfolgenden Definitionen verständlicher zu machen.

Es wird erneut das Netzwerk aus Veras Problem betrachtet, welches aus 100 Knoten besteht. Ein Arbeitskollege von Vera hat sich das Problem ebenfalls angeschaut und behauptet eine mögliche Lösung für ihre Instanz des gewichteten Vertex-Cover-Problems gefunden zu haben. Er stellt ihr eine Menge von Knoten vor, die zusammen alle Kanten des Netzwerks abdecken und in ihrem Budget liegen sollen. Um zu überprüfen, ob die von ihm gefundene Knotenmenge tatsächlich alle ihrer Vorgaben erfüllt, möchte Vera nun ein Programm schreiben. Sie entscheidet sich dazu, einen universellen Validierungsalgorithmus zu programmieren und überlegt sich vorher das Laufzeitverhalten der einzelnen Schritte, damit sie nicht versehentlich auf ein nahezu endlos laufendes Programm warten muss. Ihr Programm inklusive der Angaben über die Zeitkomplexität ist in Codeblock 2.1 dokumentiert.

Codeblock 2.1: Veras Validierungsalgorithmus

```
1 def verify_vertex_cover(possible_vertex_cover, weights, edges, bound):
2   for edge in edges:                                                    # O(n)
3     if(edge[0] in possible_vertex_cover or edge[1] in possible_vertex_cover): # O(1)
4       continue                                                          # O(1)
5     else:
6       return False                                                      # O(1)
7   if(sum(weights.get(v) for v in possible_vertex_cover) <= bound):      # O(m)
8     return True                                                       # O(1)
```

```
9   else:
10    return False                                                    # O(1)
```

Für ihr Programm hat sie die Kanten als Python-Tuple kodiert, deren Elemente die beiden anliegenden Knoten referenzieren. Der Hauptteil ihres Codes besteht aus einer Schleife von Zeilen 2 bis 6, die über alle Kanten iteriert und prüft, ob sie mindestens einen Endpunkt in dem möglichen Vertex Cover haben. Solange diese zu überprüfende Knotenmenge in Python als Set-Objekt[6] deklariert wurde, benötigt die Überprüfung nur eine konstante Laufzeit. Daraus resultiert das lineare Laufzeitverhalten $\mathcal{O}(n)$ mit $n = |\text{edges}|$ für die gesamte Schleife. Abschließend prüft das Programm in Zeile 7, ob das Gesamtgewicht der Knotenmenge unter der Kostengrenze der Instanz liegt. Da das zu überprüfende Vertex Cover maximal alle Knoten des Netzwerks beinhalten kann, fällt dafür im Worst-Case eine Laufzeit von $\mathcal{O}(m)$ mit $m = |\text{vertices}|$ an. Insgesamt benötigt der Algorithmus zum Überprüfen der Lösung folglich maximal eine polynomielle Laufzeit von $\mathcal{O}(n+m)$. Die Aufgabe, für einen gewichteten Graphen zu prüfen, ob eine gefundene Knotenmenge ein Vertex Cover unter der Kostengrenze bildet, liegt somit in P. Viele weitere ineffiziente Probleme neben Vertex Cover teilen die Eigenschaft, dass eine mögliche Lösung nur in polynomieller Zeit überprüfbar ist. Sie gehören zu der Klasse NP, die sich – umgangssprachlich beschrieben – über diese Eigenschaft definiert.

Leider ist die Lösung von Veras Kollegen fehlerhaft, denn einige Kanten sind ungedeckt geblieben. Doch Vera ist mittlerweile ein neuer Gedanke gekommen: Wenn sie einen Supercomputer hätte, der zu Beginn mit einer guten Schätzung eine richtige Lösung erraten könnte, so könnte dieser das Problem mit einem Durchlauf ihres Programms in polynomieller Zeit lösen. Dieser Supercomputer ist ein reines Gedankenexperiment und kann nicht wirklich existieren,[7] er unterstützt aber eine Definition der Klasse NP. Das formale Gegenstück zu diesem Supercomputer ist eine *nichtdeterministische Einband Turing Maschine* (NDTM). Eine NDTM ist sehr ähnlich zu der bereits in Abschnitt 2.3 beschriebenen DTM aufgebaut und beinhaltet nur ein zusätzliches Modul, das in Abbildung 2.3 blau markiert ist.

Dieser Teil kann als *Rate-Modul* bezeichnet werden und ist für die unrealistische Schätzung zu Beginn zuständig. Ein NDTM Algorithmus M besteht folglich aus zwei Teilen. Im ersten Teil wird das neue Modul verwenden, um neben die Eingabe x einen beliebigen weiteren String auf das Band zu schreiben. Die neue Zeichenfolge darf eine beliebige Länge haben und durch jedes auf dem Band erlaubte Symbol gebildet werden. Die einzige Einschränkung ist, dass die neue Zeichenfolge auf dem Band links neben die Eingabe geschrieben wird und diese daher nicht überschreiben kann. Im zweiten Teil von M werden zuvor festgelegte Schritte – genau

[6] Ein Set-Objekt ist das Python-Äquivalent einer Menge.

[7] Ein solcher Supercomputer sollte nicht mit Quantencomputern verwechselt werden, da diese nicht in der Lage sind Lösungen zu „raten“. Ein Quantencomputer kann zwar mit mehreren Lösungen parallel arbeiten, das Ausgeben einer optimalen Lösung eines NP-vollständigen Problems benötigt aktuell aber dennoch einen exponentiellen Zeitaufwand (Grover, 1996).

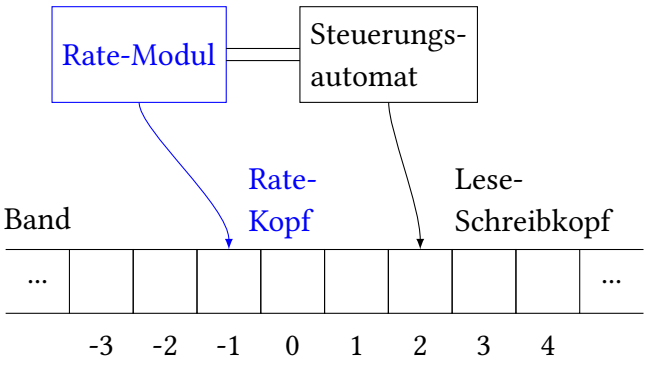

Abbildung 2.3: Aufbau einer nichtdeterministischen Einband-Turingmaschine (NDTM) [eigene Darstellung nach Garey und Johnson, 1979, S. 30]

wie nach den Regeln einer DTM – ausgeführt. Für eine möglichst simple Beschreibung der folgenden Konzepte, wird der zweite Teil von M, in dem die Eingabe x und der geratene String in die Berechnungen des Programms einfließen, als *Verarbeitung* bezeichnet. Dadurch, dass eine beliebige Zeichenfolge von beliebiger Länge von dem Rate-Modul hinzugefügt werden kann, hat ein NDTM-Algorithmus M eine unendliche Menge von möglichen Verarbeitungen für eine gegebene Eingabe x – jeweils eine für jeden möglichen hinzugefügten String. Eine Verarbeitung wird als *akzeptierende Verarbeitung* bezeichnet, wenn sie zuletzt im Endzustand q_{Ja} hält. Alle Verarbeitungen, die nicht in diesem Zustand halten – dazu zählen auch solche die in einer Dauerschleife landen – werden *nicht-akzeptierende Verarbeitungen* genannt. M *akzeptiert* x, wenn mindestens eine dieser unendlichen Zahl von Verarbeitungen eine akzeptierende Verarbeitung ist. Einfacher ausgedrückt: Wenn für die Eingabe einer Instanz (durch x) ein möglicher String gefunden werden kann, mit dem das vorliegende Problem gelöst wird, dann akzeptiert M die Eingabe dieser Instanz. Dabei ist nicht relevant wie viele Zeichen ein solcher String benötigen würde. Wieder kann eine Sprache, die M *anerkennt*, durch

$$L_M = \{x \in \Sigma^* \ : \ M \text{ akzeptiert } x\}$$

definiert werden. L_M kann erneut dadurch beschrieben werden, dass sie alle Eingaben von Instanzen enthält, die von M akzeptiert werden.

Für die Zeit, die von M benötigt wird, um eine Eingabe x aus L_M zu akzeptieren, wird das Minimum der Schritte aller akzeptierenden Verarbeitungen durch M von x gewertet. Wie diese Wertung verstanden werden kann, ist an einem Beispiel in Abbildung 2.4 veranschaulicht. Hier wurde Veras kleines Teilnetzwerk aus Abbildung 1.1 mit der Kostengrenze 12 als Instanz an den nichtdeterministischen Algorithmus übergeben. Nachdem die Informationen in M eingegeben wurden, beginnt die in orange dargestellte erste Phase, in der das Rate-Modul

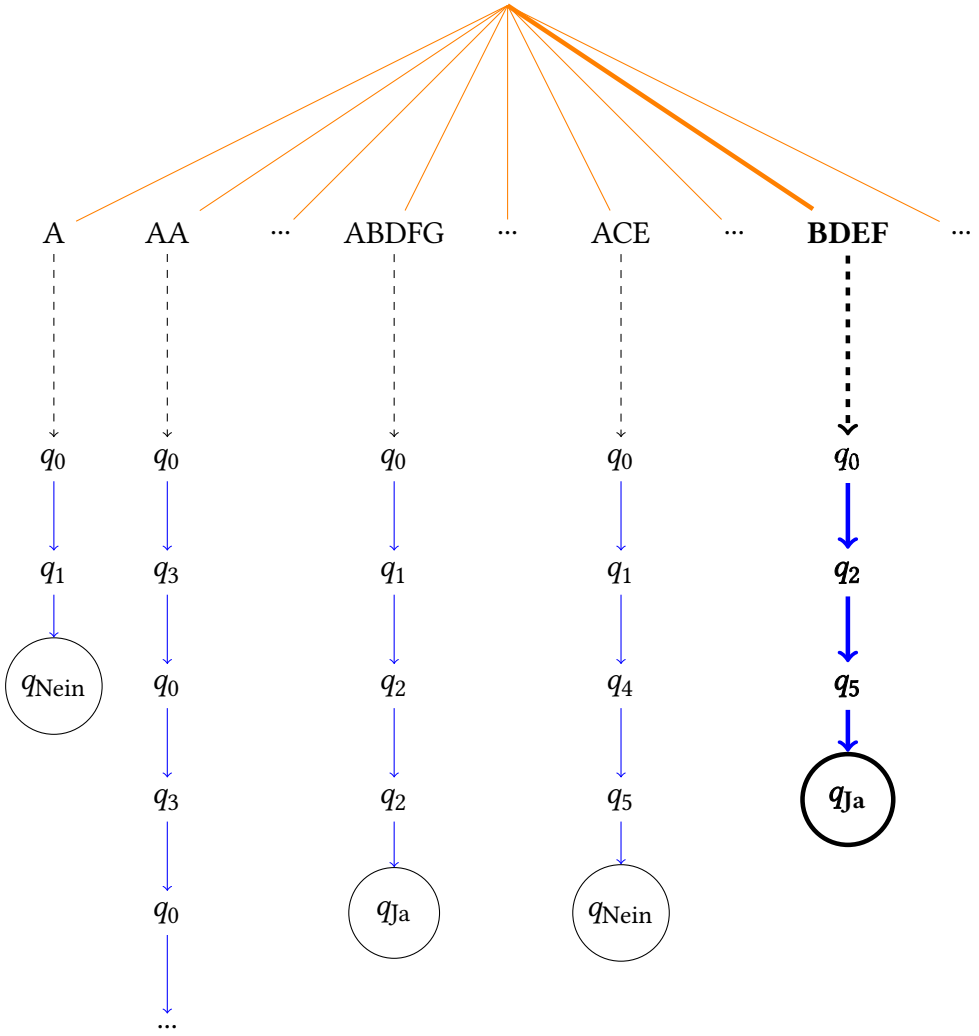

Abbildung 2.4: Visualisierter Ablauf eines beispielhaften NDTM-Algorithmus [eigene Darstellung nach Garey und Johnson, 1979, S. 27 - 32]

jeden möglichen String auf das unendliche Band schreibt.[8] In diesem Schritt spaltet sich der Ablauf des Algorithmus in eine unendliche Zahl von Verarbeitungen – eine für jeden möglichen String. In blau sind die Schritte des darauffolgenden deterministischen zweiten Teils des Programms eingezeichnet.[9] Hier führt die NDTM Rechenoperationen nach den Regeln einer DTM durch und arbeitet dafür in Zuständen. Für diesen Teil könnte man beispielsweise Veras Validierungsalgorithmus aus Codeblock 2.1 verwenden, der einen erratenen String als mögliche Knotenmenge interpretiert und überprüft, ob damit ein Vertex Cover unter der

[8] Dieser String kann nur aus Bandsymbolen gebildet werden. Für ein möglichst anschauliches Beispiel sind sie hier als Buchstaben festgelegt worden, damit ein NDTM-Algorithmus hinzugefügte Strings als Angaben zu Knoten einer Knotenmenge interpretieren kann.

[9] Die beispielhaften Zustandsübergänge in den Verarbeitungen sind frei erfunden, um eine simple Visualisierung zu ermöglichen, und beruhen auf keinem echten Algorithmus.

Kostengrenze gebildet werden kann (vergleiche mit Abbildung 1.1).

Um die Zeit, die M für das Akzeptieren von Veras Instanz benötigt, zu bestimmen, kann man sich die einzelnen Verarbeitungen als parallellaufende Berechnungen vorstellen. Sobald eine akzeptierende Verarbeitung gefunden wurde, endet das Programm. Um die benötigte Dauer zu berechnen, müssen allerdings nicht alle parallel erfolgten Schritte der unendlichen Menge an Verarbeitungen berücksichtigt werden. Stattdessen lässt sich die Zeit durch die Schritte der beiden Teile von M für die eine gefundene Verarbeitung bestimmen. Die in Abbildung 2.4 hervorgehobene Abzweigung zeigt die erste gefundene akzeptierende Verarbeitung des Beispiels und visualisiert auf diese Weise welche Schritte in die Berechnung der Zeit einfließen müssten. Wie und ob weitere Verarbeitungen danach zum Halten kommen, ist nun nicht mehr relevant.

Mit der Definition für die benötigte Zeit eines NDTM-Programms lässt sich eine Zeitkomplexitäts-Funktion $T_M : \mathbb{Z}^+ \to \mathbb{Z}^+$ für M bilden. Sie lautet

$$T_M(n) = \max\left(\{1\} \cup \left\{ m : \begin{array}{c}\text{es gibt eine Eingabe } x \in L_M, \text{ mit } |x| = n, \text{ sodass das} \\ \text{Akzeptieren von } x \text{ durch } M \text{ die Zeit } m \text{ benötigt}\end{array} \right\}\right).$$

Die Laufzeit für eine Eingabe der Länge n wird durch das Maximum von 1 und der von M benötigten Zeit für alle akzeptierte Eingaben dieser Länge gebildet. Folglich wird das Ergebnis der Laufzeitfunktion auf 1 gesetzt, wenn keine Eingaben der Länge n von M akzeptiert werden. Dementsprechend muss die Funktion nicht zwischen endlos laufenden Verarbeitungen und Verarbeitungen, die in q_{Nein} halten, unterscheiden, da im Fall von fehlenden akzeptierenden Verarbeitungen die Zeit von M nicht relevant ist (und nach der gegebenen Definition auch undefiniert ist). M ist zudem ein *Polynomialzeit-NDTM-Programm*, wenn ein Polynom p existiert, so dass für alle $n \in \mathbb{Z}^+$ gilt $T_M(n) \leq p(n)$. Zuletzt lässt sich die Klasse NP formal durch

$$\text{NP} = \{L : \text{Es gibt ein Polynomialzeit-NDTM-Programm } M, \text{ für das gilt } L_M = L\}$$

definieren. Wenn die Sprache $L[\Pi, e]$ eines Problems Π, die alle kodierten „Ja“-Instanzen eines Problems beinhaltet, ein Element von NP ist, liegt Π in NP.[10]

Zusammengefasst, kann man ein Problem zu der Komplexitätsklasse NP zuordnen, wenn für jede Instanz des Problems, die eine akzeptierende Verarbeitung in einem NDTM-Algorithmus ermöglicht, die kürzeste akzeptierende Verarbeitung in polynomieller Zeit ausgeführt werden kann. Für das gewichtete Vertex-Cover-Problem kann diese Eigenschaft durch den Ansatz aus Abbildung 2.4 bestätigt werden, indem die durch die NDTM erratenen Strings als Knotenmengen interpretiert werden. Die im ersten Teil des Programms benötigten Schritte zum Raten einer Knotenmenge sind für jede sinnvolle Schätzung des Rate-Moduls linear von der Anzahl

[10] Diese Formulierung setzt erneut voraus, dass ein vernünftiges Kodierungsschema verwendet wurde.

aller Knoten der Instanz abhängig, da ein mögliches Vertex Cover jeden Knoten maximal einmal beinhalten sollte. Da der Validierungsalgorithmus aus Codeblock 2.1 für jede Instanz des Problems im zweiten Teil des Programms verwendet werden kann, benötigen diese Schritte auch nur dieselbe Laufzeit wie das Programm. Addiert man die polynomiellen Laufzeiten der beiden Teile, die für das Finden einer akzeptierenden Verarbeitung benötigt werden, erhält man ein polynomielles Laufzeitzeitvorhalten für das gesamte NDTM-Programm *M*. *M* kann nun auf jede beliebige Instanz des gewichteten Vertex-Cover-Problems angewandt werden und benötigt dafür nur eine polynomielle Laufzeit. Aus diesem Grund liegt das gewichtete Vertex-Cover-Problem in NP.

Der gesamte Vorgang lässt sich auf die Aussage simplifizieren, dass ein Problem in NP liegt, wenn eine Lösung in polynomieller Zeit verifiziert werden kann. Dies liegt daran, dass jeder von der NDTM hinzugefügte String als mögliche Lösung des jeweiligen Problems interpretiert werden kann. Dass alle Probleme aus P ebenfalls diese Eigenschaft erfüllen, folgt daraus, dass Probleme aus P in Polynomialzeit nicht nur verifiziert, sondern direkt gelöst werden können. Nutzt man einen DTM-Algorithmus *M* um eine Lösung einer Instanz aus $\Pi \in P$ zu finden, könnte man *M* als zweiten Teil des NDTM-Programms verwenden und den vom Rate-Modul geschriebenen String für die Ausführung schlicht ignorieren.

Da nun bekannt ist, dass das „schwere“ gewichtete Vertex-Cover-Problem in NP liegt und $P \subseteq NP$ ist, könnte man vermuten, dass innerhalb von NP eine Grenze zwischen effizient und ineffizient lösbaren Problemen liegt. Allerdings konnte die Existenz einer derartigen Grenze bis heute weder bewiesen noch widerlegt werden. Dieses Problem wird als *P-NP-Problem* bezeichnet und ist eines der sieben Milleniumsproblemen, auf die ein Preisgeld von einer Millionen US-Dollar ausgesetzt wurde (Cook, 2006). Es befasst mit der Frage, ob $P = NP$ oder $P \subset NP$ gilt. Das gewichtete Vertex-Cover-Problem liegt in NP und ist eines der Probleme, bei denen bisher nicht bekannt ist, ob sie zusätzlich auch in P liegen. Für die Arbeit mit solchen Problemen, lässt sich eine weitere Komplexitätsklasse definieren, die die schwersten Probleme aus NP beinhaltet. Probleme dieser Klasse werden als NP-vollständig bezeichnet und das gewichtete Vertex-Cover-Problem ist eines von ihnen.

2.5 Polynomielle Reduktionen und NP-Vollständigkeit

Das P-NP-Problem ist bis heute ungelöst, aber viele Mathematiker gehen mittlerweile davon aus, dass $P \neq NP$ gilt. Solange das Gegenteil unbewiesen bleibt, muss in der Praxis für viele Probleme ohnehin angenommen werden, dass sie in $NP - P$ liegen und nicht in polynomieller Zeit lösbar sind, da ein Beweis des Gegenteils nicht einfacher wäre, als das P-NP-Problem selbst. Natürlich kann aktuell nicht bewiesen werden, dass ein Problem tatsächlich in $NP - P$ liegt. Allerdings kann man $\Pi \in NP - P$ für bestimmte Probleme beweisen, wenn man die

Bedingung P $\neq$ NP voraussetzt. Das Werkzeug, mit dem diese Probleme identifizieren werden können, ist die *polynomielle Reduktion*. Eine polynomiellen Reduktion von einem Problem auf ein anderes ermöglicht Aussagen über die relative Schwierigkeit der beiden Probleme.

Formal definiert, ist die polynomielle Reduktion einer Sprache $L_1 \subseteq \Sigma_1^*$ auf eine weitere Sprache $L_2 \subseteq \Sigma_2^*$ eine Funktion $f : \Sigma_1^* \to \Sigma_2^*$, die die beiden folgenden Bedingungen erfüllt:

1. f kann durch ein Polynomialzeit-DTM-Programm M_f ausgeführt werden.
2. Für alle $x \in \Sigma_1^*$ ist $x \in L_1$, genau dann, wenn $f(x) \in L_2$.

Wenn eine polynomielle Reduktion von L_1 auf L_2 existiert, sagt man, dass L_1 auf L_2 reduziert wird. Mathematisch schreibt man dafür $L_1 \propto L_2$. Die polynomielle Reduktion kann auf Entscheidungsprobleme erweitert werden, indem die Sprache $L[\Pi, e]$ eines Problems Π für die Reduktion eingesetzt wird: Wenn Π_1 und Π_2 Entscheidungsprobleme sind, $L[\Pi_1, e_1] \propto L[\Pi_2, e_2]$ gilt und sinnvolle Kodierungsschemata verwendet wurden, kann man die Formulierung $\Pi_1 \propto \Pi_2$ verwenden. Eine polynomielle Reduktion zwischen Entscheidungsproblemen kann man sich als polynomiellen Algorithmus vorstellen, der alle Instanzen aus Π_1 in Instanzen aus Π_2 umwandelt und dabei jedes Ergebnis beibehält. Ist eine Instanz aus Π_1 in J_{Π_1}, muss sie nach ihrer Umwandlung in J_{Π_2} liegen.[11]

Das Vertex-Cover-Problem kann für ein simples Beispiel einer polynomiellen Reduktion herangezogen werden: Legt man Π_{VC} als klassisches Vertex-Cover-Problem und Π_{GVC} als gewichtetes Vertex-Cover-Problem fest, gilt $\Pi_{\text{VC}} \propto \Pi_{\text{GVC}}$.[12] Die beiden Probleme sind wie folgt definiert:

Vertex Cover (VC)
Generische Instanz: Ein Graph $G = (V, E)$ und eine Grenze B.
Generische Frage: Besitzt G eine Knotenmenge $I \subseteq V$, sodass jede Kante in E mindestens einen Endpunkt in I hat und I maximal von der Größe B ist?

Gewichtetes Vertex Cover (GVC)
Generische Instanz: Ein Graph $G = (V, E)$ mit Gewichten w_v für $\forall v \in \text{V}$ und eine Grenze B.
Generische Frage: Besitzt G eine Knotenmenge $I \subseteq V$, sodass jede Kante in E mindestens einen Endpunkt in I hat und das Gesamtgewicht von I maximal B ist?

Ein Programm M_f, dass die polynomielle Reduktion f von Π_{VC} auf Π_{GVC} ausführt, könnte jedem Knoten eines ungewichteten Graphens G_{VC} das Gewicht 1 zuteilen. Die Grenze B_{VC} kann nun als Kostengrenze B_{GVC} übernommen werden, da das Gewicht eines Vertex Covers im umgewandelten Graphen immer der Anzahl der Knoten entsprechen muss. Wenn ein Vertex

[11] Dies geht aus der zweiten Bedingung hervor, da eine Sprache $L[\Pi, e]$ alle kodierten Instanzen der Menge J_Π des Problems Π beinhaltet (siehe Abschnitt 2.3).

[12] $\Pi_{\text{GVC}} \propto \Pi_{\text{VC}}$ gilt ebenfalls, aber die benötige Umwandlung ist nicht besonders anschaulich und soll aus diesem Grund hier nicht näher erläutert werden.

Cover I_{VC} für eine Instanz aus Π_{VC} existiert und $|I_{VC}| \leq B_{VC}$ ist, kann das Gesamtgewicht $\sum_{i \in I_{\text{GVC}}} w_i$ des Vertex Cover I_{GVC}, das aus denselben (umgewandelten) Knoten besteht, folglich nicht über der Grenze B_{GVC} liegen. Auf diese Weise kann eine Instanz aus Π_{VC} auf eine Instanz aus Π_{GVC} reduziert werden, ohne die Lösung zu verändern. M_f kann zudem in polynomieller Zeit ausgeführt werden, da die Anzahl der Operationen, die jeweils das Gewicht 1 zu jedem Knoten hinzuzufügen, linear von $|V|$ abhängt.

Damit sind beide Bedingungen erfüllt und es wurde bewiesen, dass das klassische Vertex-Cover-Problem auf das gewichtete Vertex-Cover-Problem reduziert werden kann. Mit dieser Erkenntnis lässt sich feststellen, dass Π_{VC} nicht schwerer als Π_{GVC} sein kann. Hätte man einen Algorithmus M_{GVC}, der Lösungen für Π_{GVC} finden kann, könnte man jede Instanz mit der kodierten Eingabe x aus Π_{VC} in M_f eingeben und dessen Ausgabe $f(x)$ als Eingabe für M_{GVC} verwenden, um die Lösung zu erhalten. Wenn man das Programm M_{VC} auf diese Weise aus der Komposition von M_f und M_{GVC} bildet, unterscheidet sich die Laufzeit von M_{VC} und M_{GVC} ebenfalls nur polynomiell um die zusätzliche Zeit von M_f und Π_{VC} ist mindestens so leicht wie Π_{GVC}. Diese Eigenschaft lässt sich verallgemeinern: Gilt $\Pi_1 \propto \Pi_2$, kann Π_1 nicht schwerer als Π_2 sein und Π_2 ist umgekehrt auch nicht leichter als Π_1. Ist zudem $\Pi_2 \in \text{P}$, so gilt auch $\Pi_1 \in \text{P}$. Wenn das Programm M_{GVC} aus dem vorherigen Beispiel in polynomieller Zeit ausführbar wäre, dann müsste die Komposition aus M_f und M_{GVC}, die als M_{VC} verwendet werden kann, dies ebenfalls sein.

Zwei Probleme können als gleichschwer bezeichnet werden, wenn sie jeweils aufeinander reduziert werden können. Man spricht in diesem Fall davon, dass die beiden Probleme (oder formeller die beiden Sprachen) *polynomiell äquivalent* sind. In den vorherigen Abschnitten wurde bereits erwähnt, dass ein durch verschiedene Kodierungsschemata verursachter polynomieller Zeitunterschied zwischen zwei Algorithmen die Einordnung eines Problems in eine Komplexitätsklasse nicht beeinflusst, solange das Kodierungsschema sinnvoll gewählt wurde. Der Grund dafür ist, dass für zwei sinnvolle Kodierungen e_1 und e_2 die Sprache $L[\Pi, e_1]$ polynomiell äquivalent zu $L[\Pi, e_2]$ ist.

Da polynomielle Reduktionen den Vergleich der Schwierigkeit von Problemen ermöglichen, können nun weitere Komplexitätsklassen definiert werden: Als *NP-schwer* wird ein Problem Π bezeichnet, wenn $\forall \Pi' \in \text{NP}$, $\Pi' \propto \Pi$ gilt. Ein NP-schweres Problem ist folglich mindestens so schwer zu lösen, wie alle Probleme in NP. Ein Problem ist als *NP-vollständig* definiert, wenn es NP-schwer ist und zusätzlich in NP liegt. Es ist weder offensichtlich, dass ein solches Problem existiert, noch wie bewiesen werden könnte, dass alle weiteren Probleme aus NP auf dieses reduziert werden können. Tatsächlich gibt es aber eine große Menge von Problemen, für die gezeigt werden können, dass sie NP-vollständig sind. Dieser Prozess ist vergleichsweise simpel, wenn man bereits eine Auswahl von NP-vollständigen Problemen zur Verfügung hat. Der Grund dafür ist, dass eine polynomielle Reduktion eines NP-vollständigen Problems Π_1 auf ein weiteres Problem $\Pi_2 \in \text{NP}$ beweist, dass Π_2 nicht leichter als Π_1 sein kann und

folglich ebenfalls NP-vollständig sein muss. Im Jahr 1971 haben die zwei Mathematiker Stephen Cook und Leonid Levin unabhängig voneinander das Erfüllbarkeitsproblem der Aussagenlogik (SAT)[13] als erstes NP-vollständiges Problem bewiesen (Garey & Johnson, 1979, S. 118 - 119). Dieser Fund ermöglichte es SAT auf weitere Probleme zu reduzieren, um deren NP-Vollständigkeit zu beweisen. Diese Methodik kann mit den folgenden Schritten beschrieben werden, die für den Beweis der NP-Vollständigkeit eines Problems angewandt werden können:

1. Es muss gezeigt werden, dass Π in NP liegt.
2. Ein bereits bekanntes NP-vollständiges Problem muss auf Π polynomiell reduziert werden.

Dadurch, dass alle NP-vollständigen Probleme ihrer Definition folgend jeweils aufeinander reduzierbar sein müssen, kann man schlussfolgern, dass sie polynomiell äquivalent sind und auf diese Weise die Klasse der NP-Vollständigkeit bilden. Durch ihre Definition erhalten Probleme dieser Klasse eine besondere Eigenschaft: Sobald ein NP-vollständiges Problem in P liegt, können alle Probleme aus NP auf dieses Problem reduziert werden und es gilt $P = NP$. Dementsprechend müssen alle NP-vollständigen Probleme in $NP - P$ liegen, wenn $P \neq NP$ gilt. Auf diese Weise lässt sich die Struktur der definierten Klassen unter der Bedinung $P \neq NP$ in Abbildung 2.5 darstellen.

Im Jahr 1972 zeigte Richard Karp, dass SAT auf das klassische Vertex-Cover-Problem Π_{VC} reduziert werden kann und bewies auf diese Weise die NP-Vollständigkeit des Problems (Karp, 1972, S. 97 - 98). Da zu Beginn dieses Kapitel bereits gezeigt wurde, dass Π_{VC} auf das gewichtete Vertex-Cover-Problem Π_{GVC} reduziert werden kann und Abschnitt 2.4 einen Überblick gibt, warum Π_{GVC} in NP liegen muss, sei an dieser Stelle festgestellt, dass Π_{GVC} ebenfalls NP-vollständig ist.

[13] Abkürzung für engl. Satisfiability

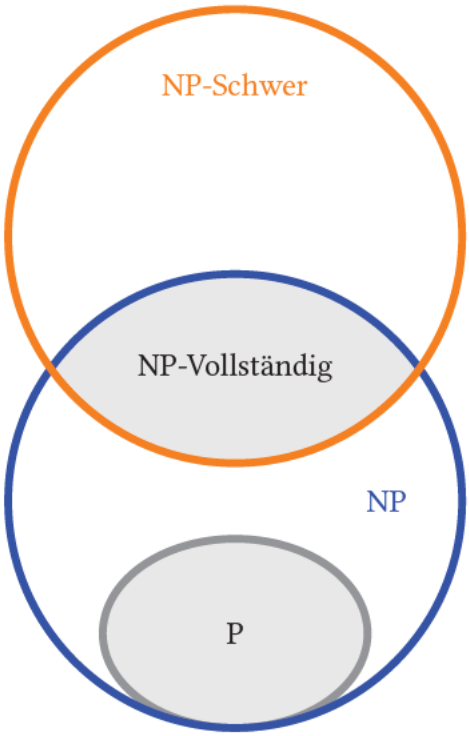

Abbildung 2.5: Struktur der beschriebenen Komplexitätsklassen unter der Bedinungung P ≠ NP [eigene Darstellung nach Garey und Johnson, 1979, S. 37, 109]

2.6 Approximationsalgorithmen

In ihrem Artikel haben Bar-Yehuda und Even einen Approximationsalgorithmus für das gewichtete Vertex-Cover-Problem vorgestellt (Bar-Yehuda & Even, 1981). *Approximationsalgorithmen* sind eine effiziente Alternative zu klassischen Algorithmen, wenn es um das Lösen von NP-vollständigen Problemen geht. Sie liefern eine suboptimale Lösung in polynomieller Zeit und eine Garantie dafür, wie weit die Lösung von dem Optimum maximal entfernt sein kann.[14] Allgemein spricht man von einem *δ-Approximationsalgorithmus*, solange der Algorithmus für jede Instanz I eines Minimierungsproblems Π eine Lösung liefert, die maximal das δ-fache des Optimums ist.[15] Der Faktor δ ist folglich eine Angabe über das theoretische Worst-Case eines Approximationsalgorithmus und wird auch als *Approximationsfaktor* bezeichnet (Hochbaum, 1996, S. XV). Bar-Yehuda und Even haben einen 2-Approximationsalgorithmus für das gewichtete Vertex-Cover-Problem entwickelt. Das bedeutet, dass jedes vom Algorithmus ausgegebene Vertex Cover maximal das doppelte Gewicht eines optimalen Vertex Covers der jeweiligen Instanz benötigt.

[14] Approximationsalgorithmen sind meistens auch in der Lage optimale Lösungen auszugeben, aber können diese nicht garantieren.

[15] In Fällen von δ-Approximationsalgorithmen für Maximierungsprobleme darf die Lösung minimal das δ-fache des Optimums sein, wobei $\delta < 1$ gilt.

Damit eine suboptimale Lösung akzeptabel ist, sollte ein Approximationsalgorithmus in Polynomialzeit arbeiten (Hochbaum, 1996, S. XV). Die Laufzeit des Algorithmus von Bar-Yehuda und Even kann mit $\mathcal{O}(|E|)$ mit E als Menge aller Kanten einer Instanz beschrieben werden und ist damit nicht nur von polynomieller, sondern auch von linearer Zeitkomplexität (Bar-Yehuda & Even, 1981).

2.7 Die Primal-Dual-Approximations-Methode

Bar-Yehuda und Even entwickelten ihren Algorithmus für das gewichtete Vertex-Cover-Problem, indem sie die *Primal-Dual-Methode* für die Anwendung in einem Approximationsalgorithmus modifizierten (Bar-Yehuda & Even, 1981). Die Primal-Dual-Methode ist ein Lösungsansatz für Probleme aus dem Bereich der *linearen Programmierung* (engl. linear programming) und wurde 1956 erstmals von Dantzig, Ford und Fulkerson (Dantzig et al., 1956) vorgestellt (Goemans & Williamson, 1996, S. 144). Die lineare Programmierung befasst sich mit der Optimierung von linearen Funktionen mit mehreren Variablen unter gegebenen Nebenbedingungen, wobei der Funktionsterm maximiert oder minimiert werden muss. Die Variablen der Funktion werden auch *Entscheidungsvariablen* genannt und repräsentieren bestimmte Entscheidungen, die in einem Problem getroffen werden, um eine Lösung zu erhalten. *Nebenbedingungen* grenzen den Bereich der Variablen ein und legen dadurch die „Regeln“ des Problems fest. Im Beispielfall des gewichtetes Vertex-Cover-Problems ist das Auslassen oder Hinzufügen eines Knotens in das zu bildende Vertex Cover eine Entscheidung und die Voraussetzung, dass jede Kante einen Endpunkt im Vertex Cover haben muss, ist eine Bedingung. Die Funktion, die es schließlich zu optimieren gilt, wird *Zielfunktion* genannt und ist für das gewichtete Vertex-Cover-Problem durch die Summe aller Knotengewichte des Vertex Covers festgelegt und soll dort minimiert werden (Williamson & Schmoys, 2011, S. 17).

Die Primal-Dual-Methode formuliert zu dem ursprünglichen Problem, dem *Primal*, ein zweites Problem, das *Dual*, bei dem die Optimierungsrichtung vertauscht wird. Sucht das Primal nach einem Maximum, wird im Dual ein Minimum angestrebt und andersherum (Dantzig et al., 1956, S. 172). Die Methode löst die beiden Probleme gleichzeitig, indem sie iterative Verbesserungen durchführt. Eine wichtige Einschränkung in ihrer Anwendung ist, dass sie nur mit kontinuierlichen Variablen arbeiten kann. Probleme aus der linearen Programmierung, die ganzzahlige Entscheidungsvariablen verlangen, gehören zu der *ganzzahligen linearen Programmierung* (ILP) (engl. integer linear programming) und können nicht mit der Primal-Dual-Methode gelöst werden. Das gewichtete Vertex-Cover-Problem ist Teil der ILP, da die Entscheidungsvariable für das Hinzufügen eines Knotens nur 1 und 0 („Ja“ oder „Nein“) sein kann. Die ganzzahlige lineare Programmierung ist zudem NP-vollständig und lässt sich aktuell nicht in polynomieller Zeit lösen (Goemans & Williamson, 1996, S. 145).

Die von Bar-Yehuda und Even abgeänderte *Primal-Dual-Approximation-Methode* beruht auf den von Dantzig, Ford und Fulkerson vorgestellten Konzepten und liefert einen neuen Ansatz, Approximationsalgorithmen für NP-vollständige Probleme zu finden (Goemans & Williamson, 1996, S. 145). Anstatt die Methode zum Bestimmen einer optimalen Lösung zu verwenden, was bis heute nicht in polynomieller Zeit möglich ist, haben sie die Methode so modifiziert, dass ihr iterativer Prozess in die Konstruktion einer Lösung integriert wird, um eine gute Näherung zu erhalten. Ihre erstmalige Anwendung auf das gewichtete Vertex-Cover-Problem (Bar-Yehuda & Even, 1981) wird in dem in Abschnitt 3.4 ausformulierten Beweis beschrieben und analysiert.

3 Der Algorithmus

Es folgt die Beschreibung und die Analyse des Algorithmus von Bar-Yehuda und Even für das gewichtete Vertex-Cover-Problem. Sie haben im Jahr 1981 zum ersten Mal die Primal-Dual-Methode für einen Approximationsalgorithmus verwendet und dadurch eine Ausweitung der Methode auf viele weitere Probleme ermöglicht. Insbesondere für die Approximation von gewichteten NP-vollständigen Problemen findet sie eine breite Anwendung (Hochbaum, 1996).

Der Algorithmusablauf und die vorgestellten Beweise stammen aus der Publikation „A Linear-Time Approximation for the Weighted Vertex Cover Problem“ (Bar-Yehuda & Even, 1981). Soweit nicht anders gekennzeichnet beruhen sämtliche Inhalte aus Abschnitt 3.1, Abschnitt 3.2 und Abschnitt 3.4 auf ihrem Artikel. Im Folgenden werden diese Inhalte detailliert ausgeführt und aufgearbeitet. Der Algorithmus wird zudem in Abschnitt 3.3 in Python implementiert und die Ergebnisse der Laufzeitanalyse werden präsentiert.

3.1 Der Ablauf

Bar-Yehuda und Even haben ihren Algorithmus auf eine Weise entworfen, die es ermöglicht, ihre Anwendung auf ein weiteres Problem auszuweiten. Das gewichtete Vertex-Cover-Problem (GVC) kann als besonderer Fall des gewichteten Set-Cover-Problems (GSC) verstanden werden, welches wie folgt definiert werden kann:

Gewichtetes Set Cover (GSC)
Generische Instanz: Eine Menge von Elementen E, eine Mengenfamilie bestehend aus Mengen $S_1, S_2, ..., S_n$ mit zugewiesenen den Gewichten $w_1, w_2, ..., w_n$ und eine Grenze B.
Generische Frage: Gibt es eine Menge I der Mengen S, sodass jedes Element in E mindestens einmal in einer Menge aus I vorkommt und das Gesamtgewicht aller Mengen in I maximal B ist?

Ein Knoten V_1 des GVC kann als Menge S_1 des GSC verstanden werden, wenn die Kanten mit den Elementen gleichgesetzt werden. Reduziert man auf diese Weise eine Instanz des GVC auf eine Instanz des GSC, ist jedes Element zuletzt in genau zwei Mengen enthalten, die vor der Umwandlung den beiden anliegenden Knoten entsprachen. Nun enthält jede

Menge S_i die Elemente, die zuvor den Kanten entsprachen, die an den Knoten V_i anliegend waren. Die Grenze und die Gewichte können einfach übernommen werden und bilden so die neue Instanz des GSC. Diese Umwandlung ist in polynomieller Zeit möglich, allerdings bedeutet eine polynomielle Reduktion nicht zwangsläufig, dass eine Approximationsgarantie eines Algorithmus erhalten bleibt. Da die Umwandlung aber ausschließlich durch das unterschiedliche Interpretieren derselben mathematischen Konstrukte erfolgt, kann eine Instanz des GVC jeder Zeit als Instanz des GSC verstanden werden, ohne dafür eine Veränderung des Algorithmus oder der Beweise zu verlangen. Bar-Yehuda und Even haben ihren Approximationsalgorithmus für das GSC entworfen, jedoch besonders die Anwendung auf das GVC betont, da diese die beste Approximationsgarantie liefert. In den folgenden Abschnitten soll der Algorithmus und die zugehörigen Beweise aus diesem Grund hauptsächlich für die Anwendung auf das GVC erläutert werden.

Der Ablauf des Approximationsalgorithmus wird nun schrittweise erarbeitet und anhand des kleinen Subnetzwerks aus Veras Beispiel veranschaulicht. Der dazugehörige Graph ist erneut in Abbildung Abbildung 3.1 abgebildet. Vor dem Beginn des Algorithmus müssen ein paar Konstrukte definiert werden, die als Kodierung der Instanzen für die Eingabe in den Algorithmus verstanden werden können:

- $G = (V, E)$ ist der Graph mit der Menge von Knoten V und der Menge von Kanten E.
- $N = \{1, 2, ..., n\}$ ist eine Menge von Indizes, die jeweils einem Knoten zugeordnet werden können und alle Knoten in V referenziert.
- $T = \{1, 2, ..., t\}$ ist eine Menge von Indizes, die jeweils einer Kante zugeordnet werden können und alle Kanten in E referenziert.
- e_j ist definiert als eine Kante, die den Index j zugeordnet bekommen hat.
- S_i ist definiert als die Menge von Kanten, die von dem Knoten mit dem Index i gedeckt werden und auch als Referenz auf den Knoten selbst verstanden werden kann.
- $F(j) = \{i | e_j \in S_i\}$ ist eine Funktion, die jedem Index j einer Kante e_j eine Menge von Indizes der Knoten liefert, die an diese Kante anliegen.
- w_i ist definiert als das Gewicht, welches an einen Knoten S_i anliegt.

Zusätzlich müssen vier weitere Entitäten eingeführt werden, die zu Beginn des Algorithmus initialisiert werden und eine zentrale Rolle beim Bilden eines Vertex Covers spielen:

- I wird als eine leere Menge initialisiert, die zum Abschluss des Algorithmus das gefundene Vertex Cover beinhalten soll.
- J ist definiert als Menge der Indizes aller bisher ungedeckten Kanten und beinhaltet zu Beginn alle Elemente aus T. Sobald J leer ist endet der Algorithmus, da alle Kanten gedeckt und folglich ein Vertex Cover gebildet wurde.

- $SW(i)$ ist eine Funktion, die das aktuelle „Restgewicht“ eines Knotens S_i liefert. Das Restgewicht von S_i wird zu Beginn auf w_i für $\forall i \in N$ gesetzt und kann im Laufe des Algorithmus verändert werden.
- $EW(j)$ ist eine Funktion, die das aktuelle „Kantengewicht“ einer Kante e_j liefert. Sie bildet ein Hilfskonstrukt, dass zum Verständnis des Algorithmus beiträgt, aber nicht direkt von diesem benötigt wird. Zu Beginn werden alle Kantengewichte auf 0 gesetzt.[1]

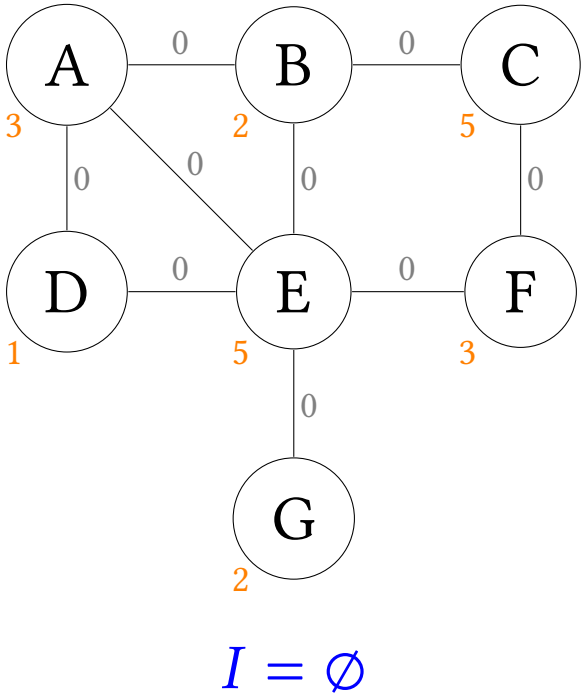

Abbildung 3.1: Anfangszustand des Graphen aus Veras Beispiel vor Anwendung des Algorithmus [eigene Darstellung]

Mit diesen Definitionen ist es nun möglich, den Ablauf schrittweise zu beschreiben. Abbildung 3.1 zeigt die Ausgangsposition des Graphen. In orange sind die Restgewichte der Knoten eingezeichnet, in grau sind die Kantengewichte zu sehen. Sobald ein Knoten dem zu bildenden Vertex Cover hinzugefügt werden soll, wird er in die Menge I eingefügt, die zum Start noch leer und unten auf der Abbildung notiert ist. Die ersten zwei Zeilen des Algorithmus sind Zuweisungen, die bereits beschrieben wurden:

$$\forall i\, SW(i) \leftarrow w_i, [\forall j\, EW(j) \leftarrow 0] \tag{1}$$

$$I \leftarrow \emptyset, J \leftarrow T \tag{2}$$

[1] Die Kantengewichte sollten nicht mit Gewichten eines kantengewichteten Graphen verwechselt werden, da das gewichtete Vertex-Cover-Problem weiterhin nur knotengewichtete Graphen in den Instanzen enthält. Die in diesem Algorithmus definierten Kantengewichte sind nicht Teil einer Instanz, sondern zuätzlich für den Algorithmus definierte Hilfsvariablen, die jeweils einer Kante zugewiesen werden.

Die Initialisierung der Funktion *EW* ist optional, da diese nur für das Speichern der erhöhten Kantengewichte verwendet wird. Ein Kantengewicht wird jedoch nach der ersten Erhöhung und der darauffolgenden Zuweisung nicht mehr benötigt und das Festhalten der Werte dient allein dem anschaulichen Protokollieren des Ablaufs. Nach den Zuweisungen beginnt der Kern des Verfahrens. Dieser wird durch eine Schleife gebildet, die jeden Durchlauf eine beliebige noch ungedeckte Kante aus J auswählt und endet, sobald J leer ist und folglich jede Kante von einem Knoten im Vertex Cover gedeckt wurde.

Do **while** $J \neq \emptyset$ (3)

Let $j \in J$ (4)

Für den ersten Durchlauf des Beispielszenarios wurde zufällig die Kante CF ausgewählt. Sie ist in Abbildung 3.2 magenta markiert.

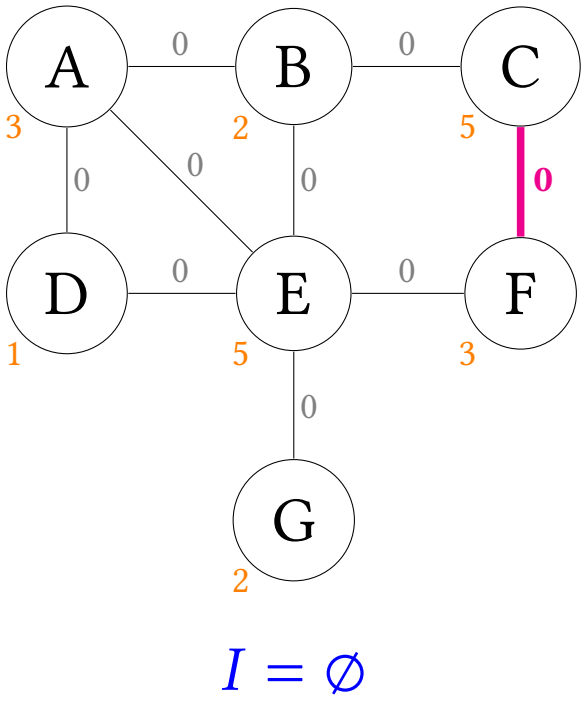

Abbildung 3.2: Die Kante CF wird für den ersten Schleifendurchlauf ausgewählt [eigene Darstellung].

Das Kantengewicht von CF wird nun auf den Wert des kleineren Restgewichts der beiden anliegenden Knoten erhöht (hier auf den Wert 3 des Knotens F). Im Algorithmus wird dafür das minimale Restgewicht der anliegenden Knoten mit Hilfe der Funktion $F(j)$ ausfindig gemacht und in der Variable M gespeichert (siehe Zeile (5)).

$M \leftarrow \text{Min}\{SW(i)|i \in F(j)\}, [EW(j) \leftarrow M]$ (5)

M kann als Kantengewicht der augewählten Kante verstanden werden. Die zusätzliche Zuweisung von M in $EW(j)$ ist optional, da das Speichern des bereits in M liegenden Kantengewichts über den aktuellen Schleifendurchlauf hinaus nicht notwendig ist. Mit anderen Worten ausgedrückt, wird in Zeile (5) das Kantengewicht der ausgewählten Kante maximiert, ohne dabei das Restgewicht eines anliegenden Knotens zu überschreiten. Die Maximierung des Kantengewichts kann als die Optimierung, die im Rahmen des Duals stattfinden soll, verstanden werden. In Abschnitt 2.7 wurde bereits beschrieben, wie ein Primal für das gewichtete Vertex-Cover-Problem aussehen könnte, und das Dual kann nun ebenfalls informell definiert werden. Die Entscheidungsvariablen des zugehörigen Duals können als die Kantengewichte verstanden werden, die in der Zielfunktion maximiert werden sollen. Die Begrenzungen der Erhöhung durch die anliegenden Restgewichte bilden die Nebenbedingung. Diese Beschreibung des Duals ist noch ungenau; wie Primal und Dual mathematisch definiert und für die Beweise verwendet werden, wird in Abschnitt 3.4 erläutert. Allerdings kann im hier beschriebenen Ablauf des Algorithmus bereits betrachtet werden, inwiefern das Dual in die Konstruktion eines Vertex Covers eingebaut wurde.

In Zeile (6) wird der Knoten ausgewählt, dessen Restgewicht mit dem Kantengewicht übereinstimmt. Dies ist weiterhin der anliegende Knoten mit dem kleineren Restgewicht (hier F). Haben beide Knoten dasselbe Restgewicht und entsprechen folglich beide dem berechneten Wert, kann ein beliebiger der beiden Knoten ausgewählt werden.

$$\text{Let } k \in F(j) \text{ such that } SW(k) = M \tag{6}$$

Das in M gespeicherte Kantengewicht wird nun in Zeile (7) vom Restgewicht der beiden anliegenden Knoten abgezogen und folglich fällt der Wert aus $SW(i)$ des ausgewählten Knotens auf 0. Der andere Knoten erhält das neue durch die Differenz berechnete Restgewicht. Das Ergebnis dieser Berechnung ist in Abbildung 3.3 zu sehen.

$$\forall i \in F(j) \quad SW(i) \leftarrow SW(i) - M \tag{7}$$

Zum Abschluss eines Schleifendurchlaufs wird der ausgewählte Knoten, dessen Restgewicht auf 0 gefallen sein muss, dem entstehenden Vertex Cover zugeteilt. Alle an diesen Knoten anliegende Kanten, werden zudem aus der Menge der ungedeckten Kanten entfernt (siehe Zeile (8)). In Abbildung 3.4 sind die nun gedeckten Kanten und der zu I hinzugefügte Knoten F blau markiert. Der nächste Schleifendurchlauf, in dem eine neue ungedeckte Kante ausgewählt wird, kann nun beginnen. Sobald keine ungedeckte Kante mehr übrig ist, endet der Algorithmus und I beinhaltet ein Vertex Cover.

Im Rahmen dieser Arbeit wurde eine Animation des gesamten Ablaufs mit Hilfe der Python-Bibliothek Manim erstellt. Sie visualisiert alle Schleifendurchläufe des Algorithmus für Veras kleines Beispielnetzwerk. Der Verweis befindet sich im Anhang.

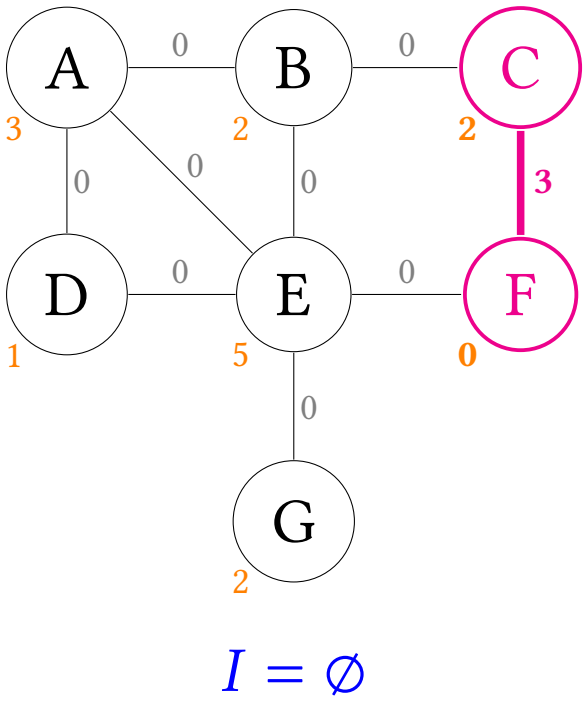

Abbildung 3.3: Das Kantengewicht von CF wird erhöht und von dem Restgewicht der beiden anliegenden Knoten abgezogen [eigene Darstellung].

$I \leftarrow I \cup \{k\}, J \leftarrow J - S_k$ (8)

end (9)

Stop. (10)

Dadurch, dass jede aus J ausgewählte ungedeckte Kante unvermeidbar am Ende jedes Durchlaufs durch einen der beiden anliegenden Knoten gedeckt wird, lässt sich schlussfolgern, dass I nach allen Schleifendurchläufen ein Vertex Cover beinhalten muss. Warum ein ausgegebenes Vertex Cover jedoch maximal das zweifache Gewicht eines optimalen Vertex Covers hat und warum der Algorithmus eine lineare Zeitkomplexität von $\mathcal{O}(|E|)$ besitzt, ist nicht sofort ersichtlich und wird aus diesem Grund in den folgenden Abschnitten genauer erläutert.

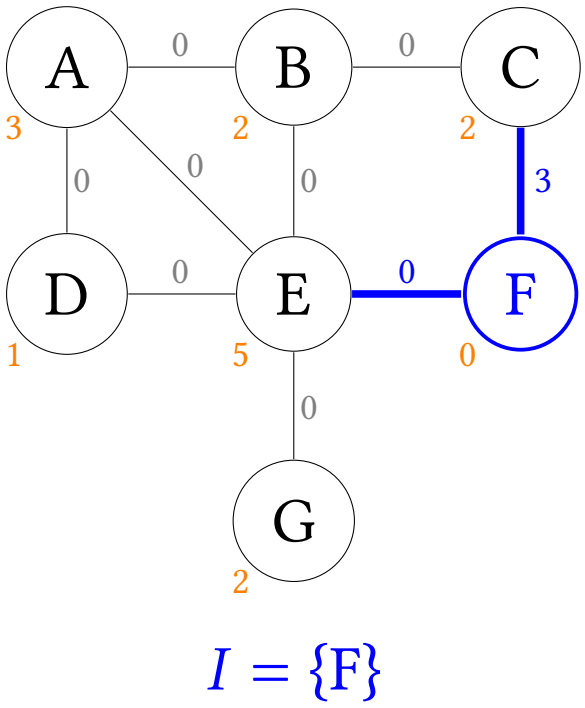

Abbildung 3.4: Der Knoten F wird zum entstehenden Vertex Cover I hinzugefügt und deckt alle anliegenden Kanten ab [eigene Darstellung].

3.2 Beweis der linearen Zeitkomplexität

In ihrem Artikel haben Bar-Yehuda und Even einen Beweis für die lineare Zeitkomplexität des Algorithmus präsentiert. Dieser ist sehr kurz und übersichtlich gehalten, erfordert dadurch aber einige gedankliche Zwischenschritte. In diesem Kapitel soll eine ausführliche Erklärung der gesamten Beweislage erfolgen, die diese Schritte konkretisiert. Für ihren Algorithmus formulieren Bar-Yehuda und Even den folgenden Satz, der – wie in diesem Kapitel herausgearbeitet wird – eine lineare Laufzeit von $\mathcal{O}(|E|)$ für die Anwendung auf das gewichtete Vertex-Cover-Problem garantiert.

Satz 1. *Die Zeitkomplexität des Algorithmus ist linear in* $\sum_{i=1}^{n} |S_i|$.

Um dieses Laufzeitverhalten zu beweisen, muss zunächst die Zeitkomplexität der einzelnen Zeilen analysiert werden. Der gesamte Algorithmus ist erneut unter Algorithmus 1 abgebildet. Zeile 1 und Zeile 2 benötigen eine Laufzeit in $\mathcal{O}(1)$, da es sich um Zuweisungen von Mengen und Funktionswerten handelt. Die Zuweisung von w_i auf $SW(i)$ in Zeile 1 darf man sich dabei nicht als Kopieren aller Werte w_i vorstellen, da dies eine Zeitkomplexität von $\mathcal{O}(|V|)$ benötigen würde. Da diese Laufzeitverhalten von der Knotenanzahl des Graphen abhängig ist, würde sich die Zeitkomplexität des gesamten Algorithmus mit dieser Veränderung von $\mathcal{O}(|E|)$ auf $\mathcal{O}(|E| + |V|)$ verschlechtern. Stattdessen muss man sich den Schritt als einmaligen

(1) $\forall i\, SW(i) \leftarrow w_i, [\forall j\, EW(j) \leftarrow 0]$
(2) $I \leftarrow \emptyset, J \leftarrow T$
(3) Do **while** $J \neq \emptyset$
(4) Let $j \in J$
(5) $M \leftarrow \text{Min}\{SW(i) | i \in F(j)\}, [EW(j) \leftarrow M]$
(6) Let $k \in F(j)$ such that $SW(k) = M$
(7) $\forall i \in F(j) \quad SW(i) \leftarrow SW(i) - M$
(8) $I \leftarrow I \cup \{k\}, J \leftarrow J - S_k$
(9) **end**
(10) Stop.

Algorithmus 1 : Der Approximationsalgorithmus von Bar-Yehuda und Even, 1981

Verweis der gesamten Funktion $SW(i)$ auf die Werte w_i vorstellen, da diese Aktion keine zusätzlichen Schleifendurchläufe benötigt. Dasselbe gilt für die Zuweisung der Menge J in Zeile 2.

Von Zeile 3 bis Zeile 9 verläuft eine Schleife, die über die ungedeckten Kanten interiert und genauer betrachtet werden muss. Für Zeile 5 bis Zeile 7 wird eine Laufzeit proportional zu $|F(j)|$ in jedem Schleifendurchlauf benötigt, da die Funktion F diktiert, wie viele Argumente zum Berechnen des Minimums in Zeile 5 benötigt werden und wie häufig das Minimum von einem Knoten-Restgewicht in Zeile 7 abgezogen werden muss. In diesem Fall darf man sich einen Aufruf von F nicht als das Durchsuchen aller Knoten darauf, ob sie die Kante e_j decken, vorstellen, da dies erneut eine Laufzeit von $\mathcal{O}(|V|)$ zur Folge hätte. Für einen erfolgreichen Durchlauf des Verfahrens müssen die Informationen einer gegebenen Instanz bereits in bestimmten Strukturen kodiert worden sein. Für Algorithmus 1 sind beispielsweise die Menge T der Kantenindizes und die Mengen S_i für alle Knoten ein paar der bereitsgestellten Elemente. Wie in Abschnitt 3.1 festgelegt wurde, gehört die Funktion F, die alle an eine Kante anliegenden Knoten wiedergibt, ebenfalls zu den vordefinierten Strukturen: Vor Beginn wird definiert, dass ein Aufruf der Funktion während des Programmablaufs jeweils einen direkten Verweis auf die zwei an j anliegenden Knoten liefern soll – ohne jedes Mal neu nach ihnen suchen zu müssen. Dadurch benötigt ein Funktionsaufruf von F nur eine konstante Laufzeit, bevor die resultierende Ausgabe im Algorithmus diktiert, wie häufig bestimmte Nachfolgeoperationen ausgeführt werden müssen. In der Praxis könnte die Vorbereitung einer Instanz für den Algorithmus auf diese Weise eine schlechtere Laufzeit als $\mathcal{O}(|E|)$ benötigen. Allerdings verlässt dies den Zuständigkeitsbereich des Algorithmus und soll in dieser Arbeit nicht näher betrachtet werden.[2] Mit diesen Bedingungen ergibt sich eine Laufzeit proportional zu $|F(j)|$ für Zeile 5 bis Zeile 7.

[2] An dieser Stelle sei an die Erkenntnis aus Abschnitt 2.1 erinnert, die besagt, dass verschiedene sinnvolle Kodierungsschemata sich nur polynomiell voneinander unterscheiden und es aus diesem Grund nicht schwierig wäre eine Instanz in die von dem Algorithmus benötigte Form zu bringen.

Die benötigte Laufzeit für Zeile 8 hängt von der Berechnung der Mengendifferenz $J - S_k$ ab, da das Hinzufügen des Elements k zu I in $\mathcal{O}(1)$ erfolgen kann. Wenn nacheinander alle Elemente in S_k aus J entfernt werden, statt iterativ die Elemente aus J auf ihr Enthaltensein in S_k zu überprüfen, ist diese Zeile mit einer Laufzeit in $\mathcal{O}(|S_k|)$ umsetzbar. Betrachtet man nun den Kontext der äußeren Schleife, dann lassen sich die folgenden zwei Erkenntnisse ableiten. Erstens kann jede Kante in der Schleife maximal einmal auf j zugewiesen werden, bevor sie in Zeile 8 aus der Menge der ungedeckten Kanten entfernt wird. Lässt man die von Zeile 8 benötigte Zeit vorerst außen vor, ergibt sich damit für die Schleife eine maximale Zeit in $\sum_{j=1}^{t} |F(j)|$. Zweitens kann k in Zeile 6 nur maximal einmal als ein bestimmter Knoten festgelegt werden, bevor dieser dem neuen Vertex Cover I hinzugefügt wird. Denn sobald das geschieht, werden alle an k anliegenden Kanten aus J entfernt und können nicht mehr in der Schleife durchlaufen werden, um den Knoten mit der Funktion F an k zu übergeben. Da jeder Knoten innerhalb der Schleife maximal einmal vorkommen kann, kann für Zeile 8 im Kontext der Schleife eine maximale Zeit in $\sum_{i=1}^{n} |S_i|$ bestimmt werden. Die Zeitkomplexität der gesamten Schleife ergibt sich durch die Addition der beiden Laufzeitverhalten und ist $\sum_{j=1}^{t} |F(j)| + \sum_{i=1}^{n} |S_i|$.

Betrachtet man nun die beiden erarbeiteten Terme $\sum_{j=1}^{t} |F(j)|$ und $\sum_{i=1}^{n} |S_i|$ einzeln, kann man feststellen, dass beide ein identisches Ergebnis liefern. $\sum_{j=1}^{t} |F(j)|$ berechnet die Summe aller an einer Kante anliegenden Knoten für alle Kanten oder einfacher ausgedrückt, wie häufig im Graphen ein Knoten an einer Kante anliegt. In Abbildung 3.5 sind diese Verbindungspunkte mit kleinen orangenen Vierecken eingezeichnet. $\sum_{i=1}^{n} |S_i|$ berechnet die Summe aller an

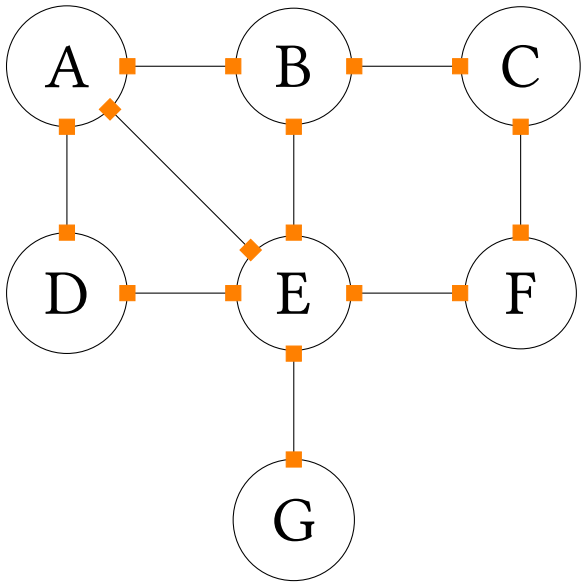

Abbildung 3.5: Die Verbindungspunkte zwischen Kanten und Knoten [eigene Darstellung]

einem Knoten anliegenden Kanten für alle Knoten oder einfacher, wie häufig eine Kante an

einen Knoten anliegt. Auch diese Verbindungspunkte können mit den orangen Vierecken aus Abbildung 3.5 visualisiert werden. Mit dieser visualisierten Übereinstimmung sollte erkennbar sein, dass das Anliegen einer Kante an einen Knoten und eines Knotens an eine Kante identisch ist und $\sum_{j=1}^{t} |F(j)| = \sum_{i=1}^{n} |S_i|$ gilt.[3] Die Laufzeit der gesamten Schleife lässt sich durch diese Erkenntnis mit $2 \cdot \sum_{j=1}^{t} |F(j)|$ beschreiben. Visuell ausgedrückt könnte die Summe bestimmt werden, indem man die in Abbildung 3.5 durch orangene Vierecke abgebildeten Verbindungspunkte zählt. Da Kanten im gewichteten Vertex-Cover-Problem an ihren beiden Endpunkte immer eine Verbindung zu einem Knoten haben müssen, können jeder Kante genau zwei Verbindungspunkte zugeordnet werden. $\sum_{j=1}^{t} |F(j)|$ lässt sich folglich mit $2 \cdot |E|$ berechnen. Konstante Faktoren können für die Beschreibung der Zeitkomplexität jedoch weggelassen werden und es bleibt eine Laufzeit von $\mathcal{O}(|E|)$ für die gesamte Schleife und im weiteren Sinne für den gesamten Algorithmus.

3.3 Die Implementation und Ergebnisse der Laufzeitanalyse

Vor der Analyse der Python-Implementation muss zunächst dargestellt werden, wie das für das Verfahren benötigte Setup übernommen werden kann. In Codeblock 3.1 ist eine mögliche Konfiguration für Veras kleines Beispielnetzwerk kodiert.

Codeblock 3.1: Python-Kodierung von Veras Beispielnetzwerk

```
1  T = set(i for i in range(9))
2  S = [[0, 2, 3], [0, 1, 4], [1, 5], [2, 6], [3, 4, 6, 7, 8], [5, 7], [8]]
3  w = [3, 2, 5, 1, 5, 3, 2]
4  F_helper = {}
5  for i, covered in enumerate(S):
6    for edge in covered:
7      if edge not in F_helper:
8        F_helper[edge] = []
9      F_helper[edge].append(i)
10 F = [F_helper.get(i) for i in T]
```

In der ersten Zeile wird die Menge der Kantenindizes T implementiert, die jeder Kante einen bestimmten Index zuweist (AB erhält beispielsweise den Index 0). Die verschiedenen Mengen S_i sind als Listen implementiert worden, da diese Änderung übersichtlicher ist und das

[3] Diese Eigenschaft gilt auch für das gewichtete Set-Cover-Problem, da die Funktion $F(j)$ in diesem Fall die Mengen zurückgibt, die das Element e_j beinhalten, und die beschriebene Symmetrie auf diese Weise erfüllt.

Verfahren nicht merkbar beeinflusst.[4] Die Funktion F ist nicht als Pythonfunktion, sondern ebenfalls als Liste initialisiert worden, um eine Ausgabe mit einer Zeitkomplexität $\mathcal{O}(1)$ liefern zu können. Statt einen Kantenindex als Eingabeparameter zu übergeben, wird dieser direkt für einen Elementzugriff verwendet. Derselbe Trick wird später auch für die Funktionen EW und SW angewandt.

Es folgt die Analyse der Implementation, die den Algorithmus mit einer Zeitkomplexität in $\mathcal{O}(|E|)$ umsetzt. Das gesamte Programm ist in Codeblock 3.2 abgebildet und wird nun schrittweise vorgestellt.[5]

Codeblock 3.2: Python-Implementation des Algorithmus von Bar-Yehuda und Even, 1981

```
1  SW = w
2  EW = [0 for i in range(len(T))] # optional
3  I = set()
4  J = T
5  while len(J) > 0:
6        j= J.pop() # alternativ j = next(iter(J))
7        covering_vertices = F[j]
8        M = min([SW[i] for i in covering_vertices])
9        EW[j] = M # optional
10       for i in covering_vertices:
11         SW[i] = SW[i] - M
12         if(SW[i] == 0):
13           k = i
14       I.add(k)
15       for edge in S[k]:
16         J.discard(edge)
```

Zu Beginn werden die Funktionen SW und EW sowie die Mengen I und J initialisiert. Die Implementierung von EW ist optional, benötigt eine Laufzeit proportional zu $|E|$ und verändert folglich die Zeitkomplexität des gesamten Algorithmus nicht. Die Menge J und die als Liste implementierte Funktion SW erhalten für ihre Initialisierung nur Verweise auf bereits definierte Werte, weil das Kopieren dieser Elemente keine konstante Laufzeit benötigen würde. In Abbildung 3.6 sind Laufzeittests der beiden Varianten abgebildet, die diesen Unterschied deutlich hervorheben. Besonders für die Zuweisung der Liste SW wäre das Kopieren der Gewichtsliste w problematisch, da die benötigte Laufzeit proportional zu $|V|$ wäre. Die direkten Zuweisung sind hingegen unproblematisch, da sie eine konstante Zeit benötigen und w sowie T in diesem Verfahren für keinen anderen Zweck verwendet werden.

[4] `S = [set([0, 2, 3]), set([0, 1, 4]), ...]` wäre die korrekte Implementation und eine mögliche Alternative, die dieselbe Zeitkomplexität verspricht.

[5] Eine allgemeine Implementation des Algorithmus als Python-Funktion ist im Anhang zu finden.

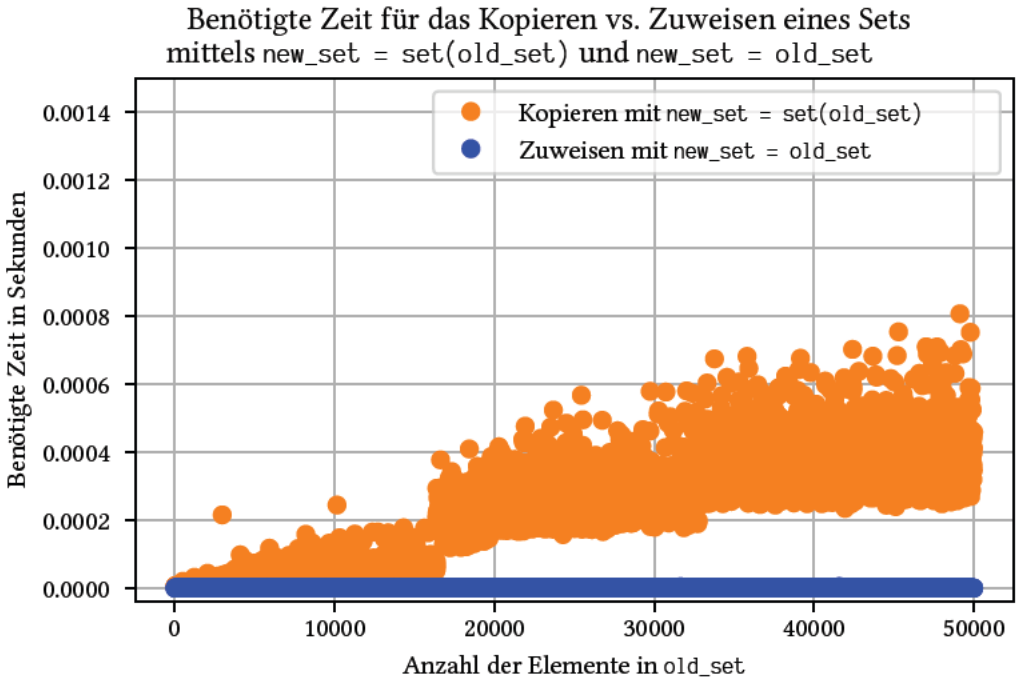

Abbildung 3.6: Unterschiedlicher Zeitaufwand des Zuweisen und Kopieren von Mengen in Python [eigene Darstellung]

Innerhalb der Schleife wird eine Kante mit dem Funktionenaufruf `J.pop()` ausgewählt und aus J entfernt. Da die ausgewählte Kante j am Ende eines Durchlaufs ohnehin aus J entfernt werden muss, wird der Algorithmus durch das vorschnelle Entfernen der Kante nicht beeinträchtigt. In einer alternativen Implementation werden die Funktionsaufrufe `next(iter(J))` verwendet, um eine Kante auszuwählen, ohne sie gleichzeitig aus J zu entfernen. Wie in Abbildung 3.7 zu sehen ist, benötigen diese Aufrufe in einem isolierten Kontext nur eine konstante Laufzeit. Allerdings fiel beim Laufzeittest auf, dass die benötigte Zeit von `next(iter(J))` sich durch das Löschen von Elementen aus J deutlich verschlechtert. Set-Objekte sind in Python intern mit Hash-Tabellen programmiert und durch das Entfernen von Kanten aus dem Set-Objekt J am Ende jedes Durchlaufs leert sich die zugehörige Hash-Tabelle stetig (Upadhyay, 2024). Dadurch entstehen Lücken in der internen Tabelle, die `next(iter(J))` in jedem neuen Aufruf erneut überspringen muss, um ein Feld zu erreichen, das noch nicht leer ist. Auf diese Weise entsteht im Worst-Case ein quadratisches Laufzeitverhalten für die gesamte Schleife, das in Abbildung 3.8 zu sehen ist. Auch ein Aufruf der Funktion `J.pop()` kann aufgrund der internen Programmierung ein schlechteres Laufzeitverhalten als $\mathcal{O}(1)$ erhalten. Allerdings läuft diese Funktion intern mit einem Zeiger über die Hash-Tabelle und merkt sich die Stelle, an der sie zuletzt ein Element entfernt hat (Upadhyay, 2024). Dadurch muss jedes Element (bzw. für das Objekt J folgich jede Kante) in der Hash Tabelle von `J.pop()` über alle Durchläufe der Schleife hinweg maximal einmal übersprungen werden und die Funktion benötigt so

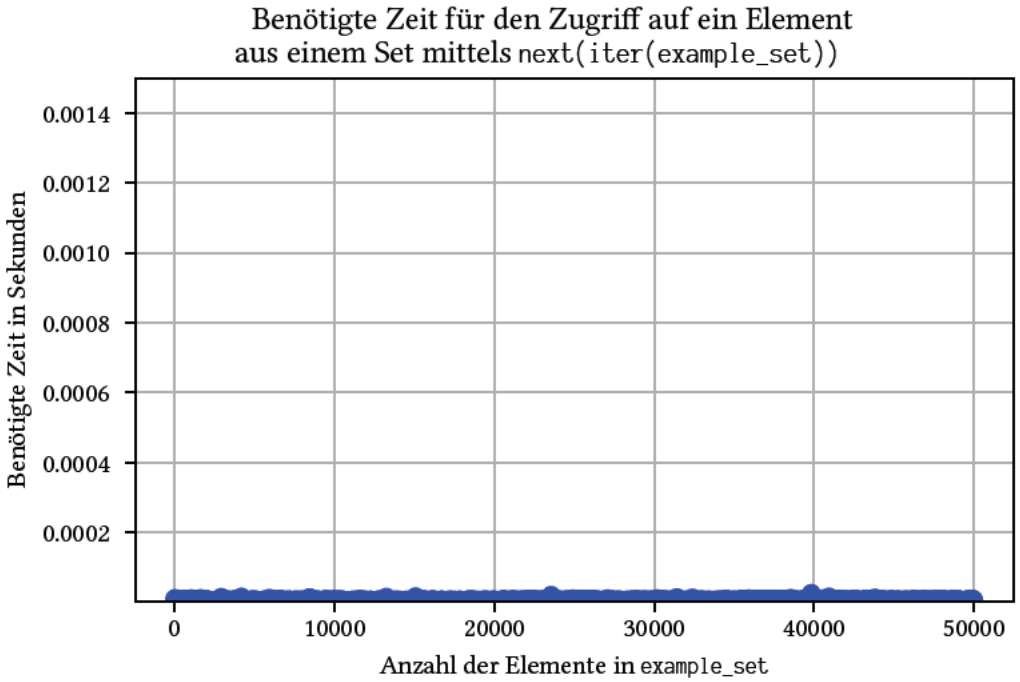

Abbildung 3.7: Zeitaufwand für die Auswahl einer Kante mittels `next(iter(J))` in Python [eigene Darstellung]

im Worst-Case maximal eine zusätzliche Laufzeit $\mathcal{O}(|E|)$ für die gesamte Schleife. Der Erhalt der linearen Laufzeit durch die Funktion ist in Abbildung 3.8 abgebildet. Der Aufruf `J.pop()` sollte aus diesem Grund für die Kantenauswahl von j verwendet werden, um die optimale Zeitkomplexität zu ermöglichen.

Die Funktionsaufrufe von F und SW in Zeilen 7 bis 12 können durch die sinnvolle Initialisierung als Listen-Objekte in konstanter Zeit abgeschlossen werden. Zeile 9 ist erneut optional und verändert die Zeitkomplexität des Verfahrens nicht. In der letzten Zeile werden die abgedeckten Kanten mittels der Funktion `set.discard()` entfernt, die nur eine Laufzeit von $\mathcal{O}(1)$ benötigt[6] und somit die Anforderungen des Algorithmus erfüllt.

Für die Laufzeitanalyse des gesamten Verfahrens musste zunächst eine Funktion implementiert werden, die für eine gegebene Anzahl von Kanten einen zufälligen Graphen mittels einer Adjazenzmatrix erstellt und an den Algorithmus übergibt. Um ein möglichst akkurates Ergebnis zu erzielen und zeitliche Ausreißer zu minimieren, wurde zudem das Garbage-Collector-Python-Modul, das automatisch im Hintergrund den Speicher verwaltet, deaktiviert. Stattdessen müssen die Elemente manuell zwischen Durchläufen gelöscht werden, um den Speicher nicht auszulasten. Die resultierende Laufzeit ist – wie im Verfahren vorausgesetzt

[6]Der Grund warum J nicht als Liste implementiert werden sollte, um die Kantenauswahl simpler zu gestalten, ist, dass das Entfernen eines Elements mittels `list.remove()` deutlich ineffizienter wäre.

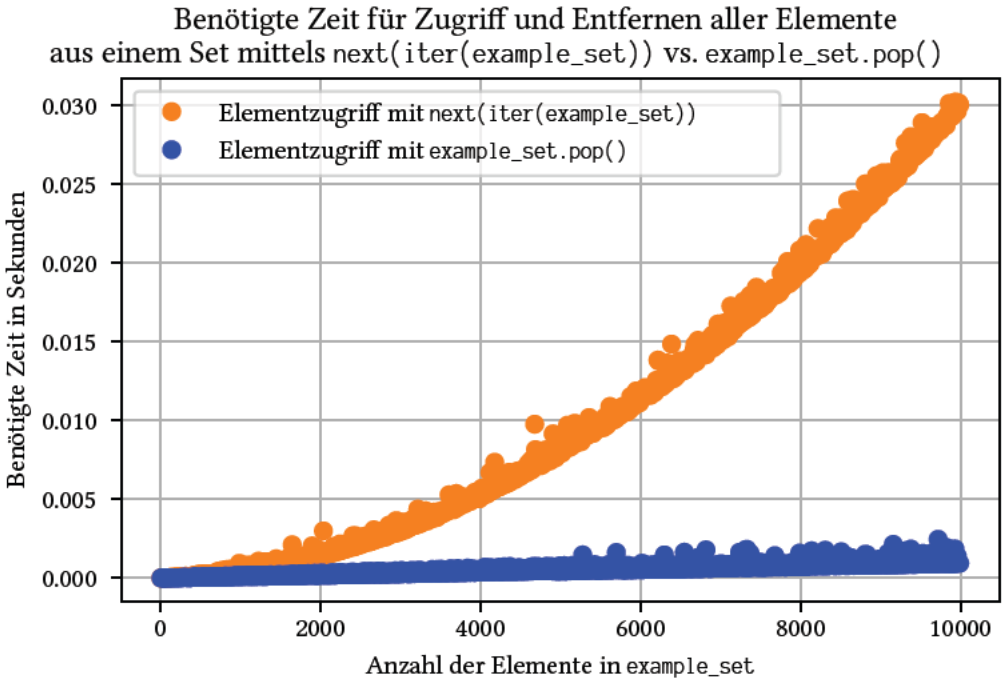

Abbildung 3.8: Unterschiedlicher Zeitaufwand der wiederholten Auswahl einer Kante mittels `next(iter(J))` und `J.pop()` in Python [eigene Darstellung]

– linear in $|E|$ und kann in Abbildung 3.9 betrachtet werden. Ein über zehn Laufzeittests gemittelter Graph ist in Abbildung 3.10 zu sehen.

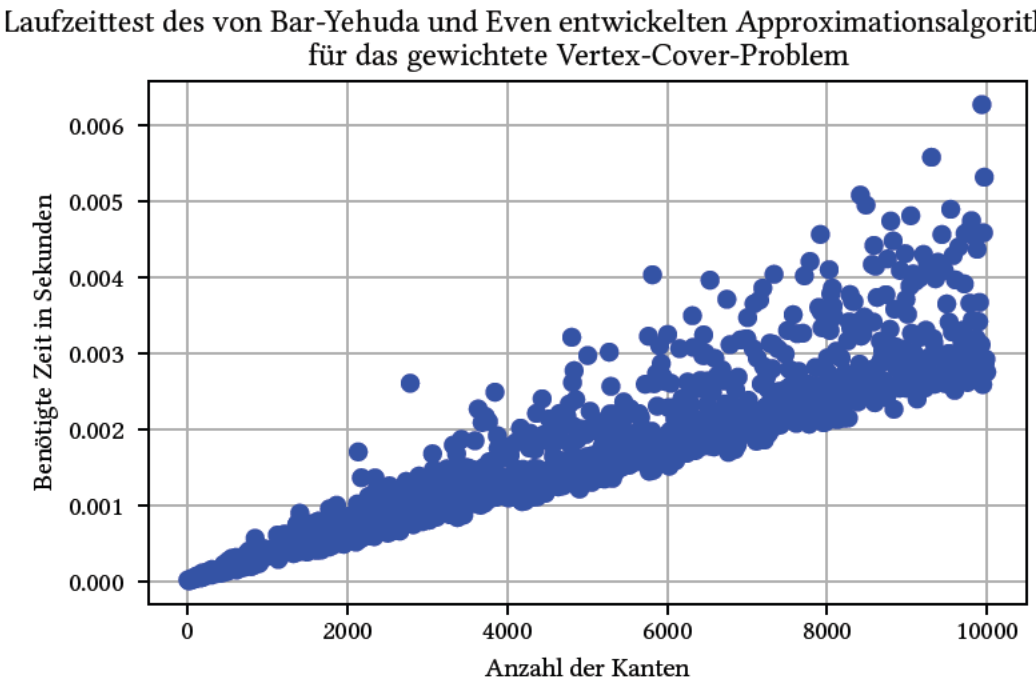

Abbildung 3.9: Zeitaufwand des Approximationsalgorithmus 3.2 in Python [eigene Darstellung]

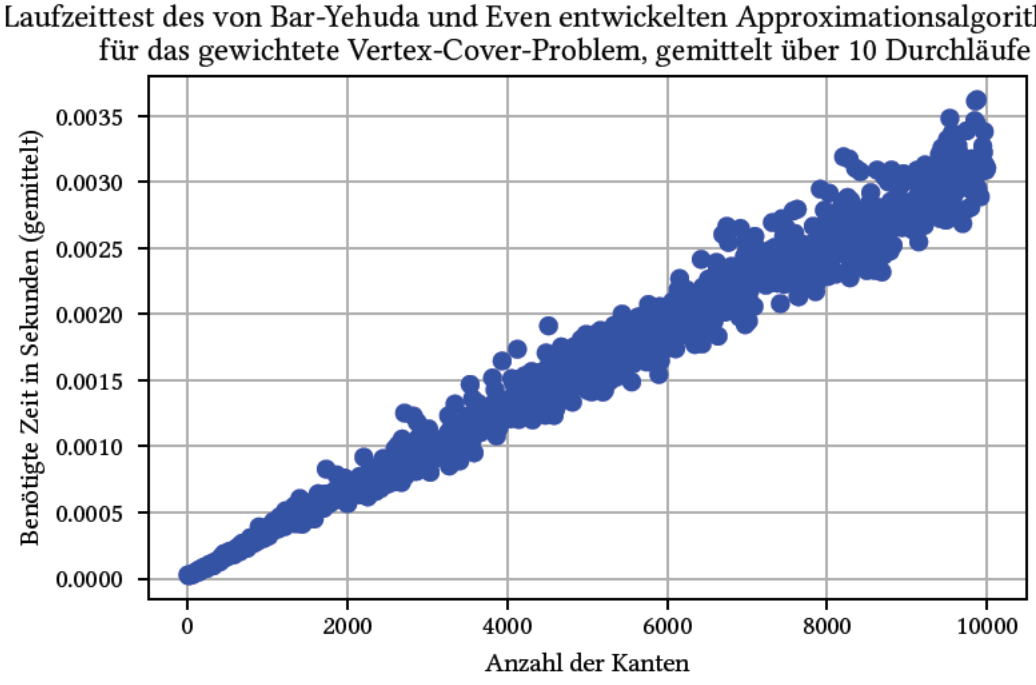

Abbildung 3.10: Über 10 Durchläufe gemittelter Zeitaufwand des Approximationsalgorithmus 3.2 in Python [eigene Darstellung]

3.4 Beweis der Approximationsgarantie

In diesem Kapitel wird der Beweis der Qualitätsgarantie des 2-Approximationsalgorithmus von Bar-Yehuda und Even schrittweise erklärt. Der gesamte Beweis ist in kürzerer Form als Variante für das gewichtete Set-Cover-Problem auch in der Hauptquelle Bar-Yehuda und Even, 1981 zu finden und kann dort nachgelesen werden.[7] Es wird der folgende Satz betrachtet:

Satz 2. *Das Gesamtgewicht W eines vom Algorithmus erhaltenen Vertex Cover I erfüllt*

$$W \leq W_{\text{opt}} \cdot \max_{j \in T} |F(j) \cap I|. \tag{1}$$

W ist das Gesamtgewicht des Vertex Cover I und kann mit

$$W = \sum_{i \in I} w_i \tag{2}$$

berechnet werden. $F(j)$ liefert die beiden an j anliegenden Knoten (siehe Abschnitt 3.1). $F(j) \cap I$ enthält alle Knoten des Vertex Covers, die e_j decken, und dem folgend liefert $\max_{j \in T} |F(j) \cap I|$ die maximale Anzahl, die eine Kante von anliegenden Knoten in I gedeckt wurde. In der Regel ist dieser Wert für Vertex Cover 2, weil eine Kante im gewichteten Vertex-Cover-Problem nicht von mehr als zwei Knoten gedeckt werden kann. Da Approximationsalgorithmen nach ihrem Worst-Case über allen Instanzen evaluiert werden (siehe Abschnitt 2.6), folgt, dass der Satz einen 2-Approximationsalgorithmus für das gewichtete Vertex-Cover-Problem verspricht.[8] Es folgen nun die benötigten Schritte, die dieser Beweis beinhaltet. Zunächst kann die Eigenschaft

$$\forall i \in N, \quad SW(i) \geq 0 \tag{3}$$

formuliert werden. Sie folgt aus Zeile 1, in dem $SW(i)$ für $\forall i \in N$ auf den positiven Wert w_i gesetzt wird und den Schritten Zeile 5 bis Zeile 7. Hier wird der Wert M, bevor er von $SW(i)$ abgezogen wird, auf den kleineren Wert der beiden anliegenden Restgewichte gesetzt und $SW(i)$ kann folglich höchstens auf 0 fallen. Als nächstes lässt sich

$$\forall j \in T, \quad EW(j) \geq 0 \tag{4}$$

beweisen, da $EW(j)$ in Zeile 1 für $\forall j \in T$ auf 0 gesetzt wird und in Zeile 5 durch das kleinere Restgewicht der beiden anliegenden Knoten ersetzt wird, die – wie in (3) bewiesen wurde – nicht negativ sein können. Es folgt

$$\forall i \in N, \quad SW(i) + \sum_{j \in S_i} EW(j) = w_i. \tag{5}$$

[7] Wie in Abschnitt 3.1 bereits erwähnt, kann der Beweis für das Set-Cover-Problem direkt auf das gewichtete Vertex-Cover-Problem übertragen werden, indem die Elemente als Kanten und die Mengen als Knoten interpretiert werden.

[8] Für das gewichtete Set-Cover-Problem wird ein f-Approximationsalgorithmus mit $f = \max_{j \in T} |F(j) \cap I|$ versprochen und ist folglich davon abhängig, wie häufig ein Element in einer Menge aus I vorhanden ist.

Diese Eigenschaft kann wie folgt beschrieben werden: Das Gewicht eines Knotens kann durch die Summe des Restgewichts und allen anliegenden Kantengewichten berechnet werden. In Abbildung 3.3 kann man dies beispielsweise für die beiden Knoten C und F beobachten, da der in Zeile 7 vom Restgewicht abgezogene Betrag bereits einem der anliegenden Kantengewichte in Zeile 5 hinzuaddiert wurde. Die Summe aus (5) verändert sich dadurch nicht, auch wenn dem Restgewicht eines Knotens erneut ein anliegendes Kantengewicht abgezogen wird (siehe Abbildung 3.11, die zeigt wie im zweiten Durchlauf die Kante BC vom Algorithmus ausgewählt und erneut ein Kantengewicht vom Restgewicht des Knotens C abgezogen wurde).

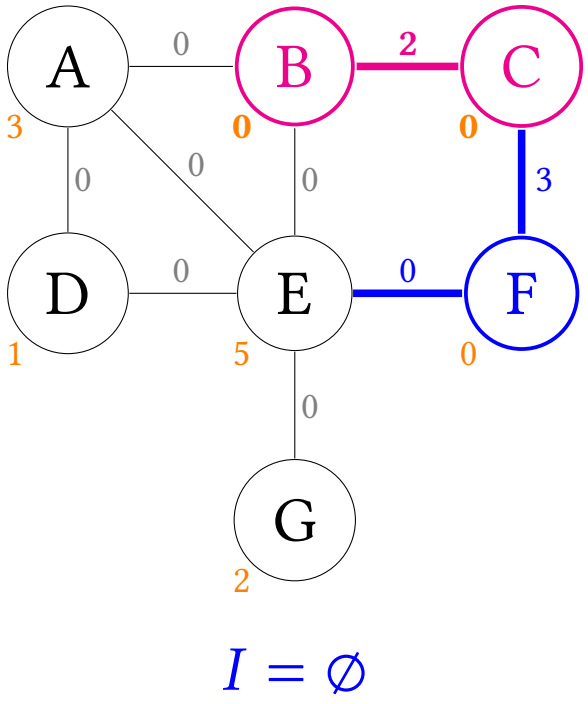

Abbildung 3.11: Im zweiten Durchlauf wird die Kante CF ausgewählt und das zugehörige Kantengewicht erhöht und von dem Restgewicht der anliegenden Knoten abgezogen [eigene Grafik].

$$\forall i \in N, \quad \sum_{j \in S_i} EW(j) \leq w_i \tag{6}$$

folgt durch (5) und (3), da die Summe der Kantengewichte in (5) nicht größer als das Gewicht des anliegenden Knotens werden kann, ohne dass $SW(i)$ negativ werden müsste, um die Gleichung weiterhin zu erfüllen. Wörtlich sagt (6) aus, dass alle an einen Knoten anliegenden Kantengewichte, summiert, maximal das Knotengewicht ergeben können. Auch diese Eigenschaft ist aus Abbildung 3.3 und Abbildung 3.11 erkennbar.

$$\forall i \in I, \quad \sum_{j \in S_i} EW(j) = w_i \tag{7}$$

beschreibt, dass für alle Knoten, die im Vertex Cover landen, die Summe der anliegenden Kantengewichte genau gleich dem Knotengewicht sein muss. Das Restgewicht eines Knotens muss vor dem Hinzufügen zum Vertex Cover I in Zeile 8 durch die Subtraktion in Zeile 7 auf 0 gefallen sein (siehe Abbildung 3.3), da M der Wert des Restgewichts in Zeile 5 zugewiesen wurde. Fügt man das Restgewicht von 0 in die Summe aus (5) für alle Knoten aus I ein, folgt (7). Die Aussage

$$W \leq \sum_{j \in T} EW(j) \cdot \max_{j \in T} |F(j) \cap I| \tag{8}$$

ist etwas komplizierter und lässt sich durch

$$W = \sum_{i \in I} \sum_{j \in S_i} EW(j) \tag{9}$$

beweisen. (9) folgt, indem man die linke Seite der Gleichung aus (7) für die Summanden w_i aus (2) einsetzt. Die resultierende Aussage (9) besagt, dass das Gesamtgewicht eines VC berechnet werden kann, indem für jeden Knoten aus I jeweils die Summe aller anliegenden Kantengewichte gebildet wird und alle resultierenden Werte zuletzt addiert werden. Diese Eigenschaft sollte durch die Herleitung von (7) wenig überraschend sein. Betrachtet man das

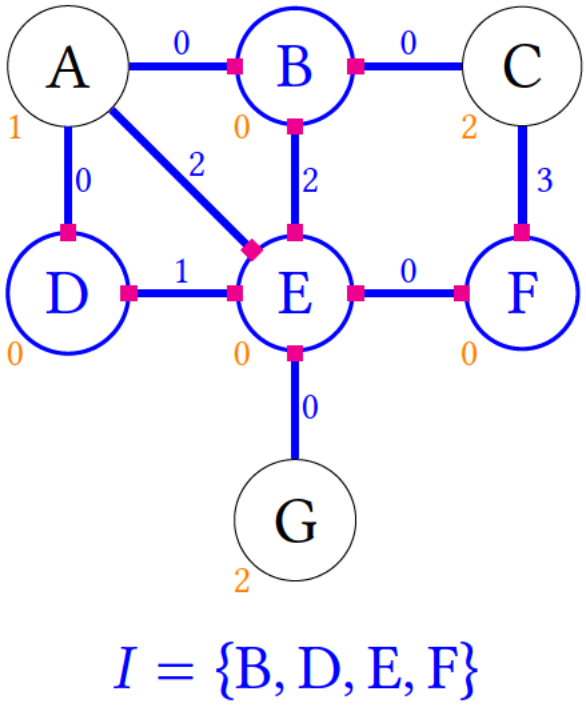

Abbildung 3.12: Visualisierung von $|F(j) \cap I|$ für jede Kante j im Vertex Cover $I = \{B, D, E, F\}$ [eigene Darstellung]

durch den Algorithmus ausgegebene Vertex Cover $I = \{B, D, E, F\}$ in Abbildung 3.12, kann

man feststellen, dass manche Kantengewichte (beispielsweise die von BE oder DE) für die Berechnung von W in (9) doppelt gezählt werden müssen, während andere nur einmal als Summand vorkommen (zum Beispiel das Kantengewicht von CF). Ein Kantengewicht muss genau dann doppelt gezählt werden, wenn beide anliegenden Knoten in I vorkommen (siehe (9)). Diese Häufigkeit kann durch die kleinen magentafarbenen Vierecke in Abbildung 3.12 abgelesen werden, die jeweils markieren, dass ein Knoten aus I eine anliegende Kante deckt. $|F(j) \cap I|$ beschreibt diese Anzahl mathematisch für jede Kante $j \in T$ (siehe Beschreibung von (1)). Daraus erschließt sich, dass W mit der Summe $\sum_{j \in T} EW(j) \cdot |F(j) \cap I|$ berechnet werden kann. Nimmt man stattdessen das Maximum dieser Häufigkeit aus allen Kanten,[9] kann der Faktor nur größer geworden sein und es folgt die Ungleichung aus (8). Dieser Schritt ändert den für jede Kante e_j unterschiedlichen Faktor $|F(j) \cap I|$ in den konstanten Wert $\max_{j \in T} |F(j) \cap I|$, der folglich nach Belieben aus der Summe herausgezogen werden kann. Dies mag zum aktuellen Zeitpunkt noch nicht zielführend wirken, aber er ermöglicht wichtige Schritte im weiteren Verlauf des Beweises.

Im nächsten Schritt kann das Dual der modifizierten Primal-Dual-Methode durch die Entscheidungsvariablen y_j für $\forall j \in T$, die Nebenbedingungen

$$\forall i \in N, \quad \sum_{j \in S_i} y_j \quad \leq w_i \quad \text{und} \tag{10}$$

$$\forall j \in T, \quad y_j \quad \geq 0 \tag{11}$$

und dem Optimierungsziel

$$\max \sum_{j \in T} y_j \tag{12}$$

für die Zielfunktion $\sum_{j \in T} y_j$ definiert werden. Die Werte y_j kann man sich als maximierte Kantengewichte des Algorithmus vorstellen, wobei die Symmetrie zwischen (4) und (11) sowie (6) und (10) betrachtet werden kann. Allerdings sind die Werte y_j nicht an den Ablauf des Algorithmus gebunden, sondern dienen lediglich als symbolische Platzhalter. Nach der Definition des Duals, ist das Ziel der Zielfunktion, alle (symbolischen) Kantengewichte y_i, für ausnahmslos alle anliegenden Knoten zu maximieren, statt dies nur für die Knoten aus I zu tun, wie es im Ablauf der Fall ist. Während die Kantengewichte $EW(j)$ im Algorithmus nur so lange erhöht werden, bis das Restgewicht aller Knoten, die für ein VC benötigt werden, auf 0 gefallen ist (siehe Abbildung 3.12, in der die Summe aller Kantengewichte 8 ist), würde die Zielfunktion (12) an diesem Punkt anknüpfen, um ausnahmslos alle Kantengewichte nach den Einschränkungen in (10) und (11) zu maximieren. In Abbildung 3.13 ist eine mögliche Optimierung aller (symbolischen) Kantengewichte in magenta abgebildet, die die Zielfunktion auf den Wert 10.5 maximiert. In orange sind die Knotengewichte eingezeichnet, die für die Bedingung (10) relevant sind. Diese Erklärung darf nicht zu wörtlich genommen werden, da

[9] Im Fall von VC wäre dies maximal 2, da eine Kante von maximal zwei Knoten gedeckt werden kann.

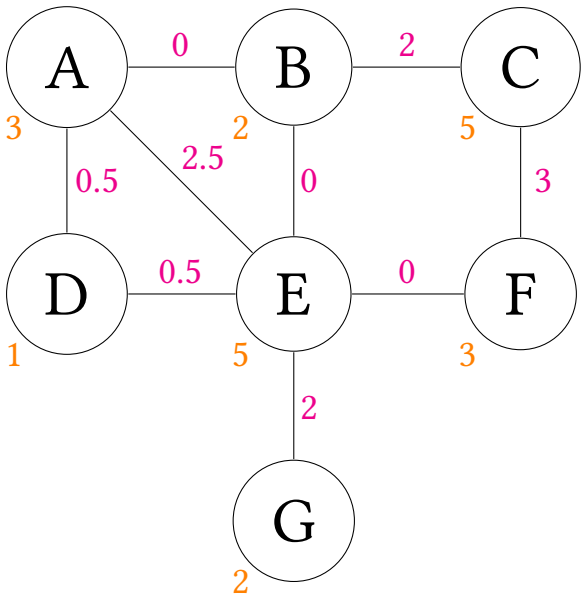

Abbildung 3.13: Theoretische Maximierung aller Kantengewichte unter den gegebenen Bedingungen [eigene Darstellung]

die Werte y_j weiterhin nur Platzhalter sind und nicht wirklich als Kantengewichte verwendet werden. Doch sie sollte die Erkenntnis ermöglichen, dass die Maximierung der Zielfunktion in jedem Fall mindestens so groß sein muss, wie die Summe aller Kantengewichte bei Halt des Algorithmus. Mathematisch ausgedrückt bedeutet dies

$$\sum_{j \in T} EW(j) \leq \max \sum_{j \in T} y_j. \tag{13}$$

Nach dem Dual wird nun das Primal mit den Entscheidungsvariablen x_i für $\forall i \in N$, den Nebenbedingungen

$$\forall j \in T, \quad \sum_{i \in F(j)} x_i \quad \geq 1 \quad \text{und} \tag{14}$$

$$\forall i \in N, \quad x_i \quad \geq 0 \tag{15}$$

und dem Optimierungsziel

$$\min \sum_{i \in N} x_i \cdot w_i \tag{16}$$

für die Zielfunktion $\sum_{i \in N} x_i \cdot w_i$ definiert. Diese Defintion ähnelt der Beschreibung aus Kapitel Abschnitt 2.7 und auch in diesem Fall kann man sich die Variablen x_i als Platzhalter für die Entscheidung vorstellen, ob ein Knoten v_i in dem zu bildenden Vertex Cover landen soll oder nicht. Allerdings gibt es einen entscheidenden Unterschied: Während ein Knoten nur die zwei Optionen zur Auswahl hat, im Vertex Cover enthalten zu sein oder nicht (symbolisiert durch

1 und 0), wird für die Variablen x_i definiert, dass sie auch zwischen ganzen Zahlen liegen dürfen und dementsprechend ihren zugehörigen Knoten nur teilweise hinzufügen können, solange die Bedingungen in (14) und (15) berücksichtigt werden. Diesen Akt, in dem die Bindung an ganzzahlige Werte aufgehoben wird, nennt man *Relaxation* eines ganzzahligen linearen Programms (Williamson & Schmoys, 2011, S. 18) (auch *LP-Relaxation* nach Goemans & Williamson, 1996, S. 145). Natürlich muss auch an dieser Stelle erneut darauf hingewiesen werden, dass die Werte x_i nur mathematisch definierten Entscheidungsvariablen des Primals sind, nicht wirklich halbe Knoten in ein Vertex Cover hinzufügen können und diesen Akt höchstens symbolisieren. Es wird nun das Gesamtgewicht eines optimalen Vertex Covers I_{opt} mit

$$W_{\text{opt}} = \sum_{i \in I_{\text{opt}}} w_i \tag{17}$$

definiert. I_{opt} ist ein realistisches Vertex Cover, folgt den Regeln des gewichteten Vertex-Cover-Problems und muss folglich eine klare Entscheidung für jeden Knoten treffen, ob dieser in der Menge enthalten sein soll oder nicht (und darf diesen nicht „teilweise“ hinzufügen). Ein optimales Vertex Cover für das Beispielnetzwerk mit $W_{\text{opt}} = 11$ ist in Abbildung 3.14 zu sehen.

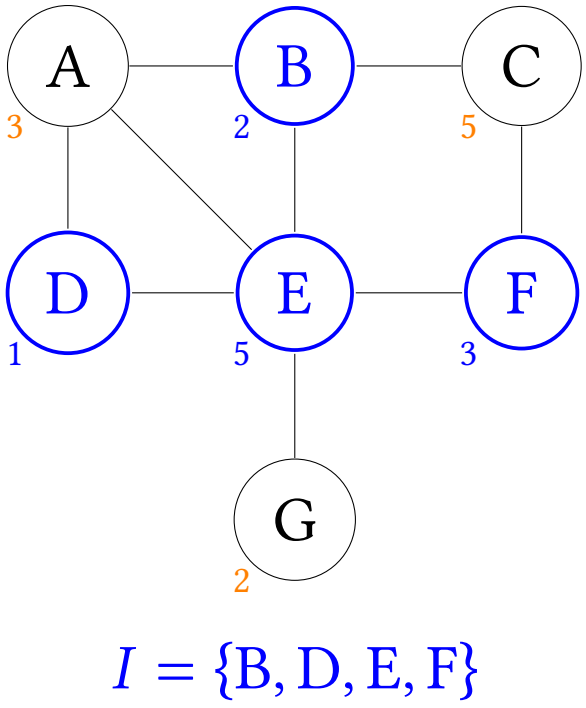

Abbildung 3.14: Abbildung eines optimalen Vertex Covers $I_{\text{opt}} = \{B, D, E, F\}$ für Veras kleines Teilnetzwerk [eigene Darstellung]

Eine Optimierung der Zielfunktion des Primals kann dieses Vertex Cover als Verteilung der Entscheidungsvariablen x_i berücksichtigen, indem $x_B = 1, x_D = 1, x_E = 1, x_G = 1,$

und alle weiteren x_i auf 0 gesetzt werden.[10] Durch die LP-Relaxation des Primals kann die Zielfunktion für den Graphen aus Abbildung 3.14 allerdings noch weiter optimiert werden. Das Ergebnis 10.5 mit $x_A = 0.5, x_B = 0.5, x_C = 0.5, x_D = 0.5, x_E = 0.5, x_F = 0.5$ und $x_G = 0.5$, ist beispielsweise eine optimale Lösung und kann von einem echten optimalen Vertex Cover I_{opt} nicht realisiert werden. Mit diesen beiden Erkenntnissen folgt

$$W_{opt} \geq \min \sum_{i \in N} x_i \cdot w_i. \tag{18}$$

Ein optimiertes Ergebnis der Zielfunktion des Primals entspricht folglich mindestens dem Gesamtgewicht eines realistischen optimalen Vertex Covers und unterbietet dieses meistens sogar. Es folgt nun die Definition einer Dualitäts-Eigenschaft mit

$$\sum_{j \in T} y_j \leq \sum_{j \in T} y_j \cdot \sum_{i \in F(j)} x_i = \sum_{i \in N} x_i \cdot \sum_{j \in S_i} y_j \leq \sum_{i \in N} x_i \cdot w_i. \tag{19}$$

Sie spielt eine wichtige Rolle in diesem Beweis und lässt sich in drei Schritte unterteilen. Die erste Ungleichung

$$\sum_{j \in T} y_j \leq \sum_{j \in T} y_j \cdot \sum_{i \in F(j)} x_i \tag{20}$$

kann mit der der Nebenbedingung (14) des Primals bewiesen werden. Dort wurde festgelegt, dass die Faktoren $\sum_{i \in F(j)} x_i$ größer als 1 sein müssen und (20) folgt. Durch Ausklammern der Werte x_i, erhält man die Summe $\sum_{i \in N} x_i \cdot \sum_{j \in S_i} y_j$ und es folgt

$$\sum_{j \in T} y_j \cdot \sum_{i \in F(j)} x_i = \sum_{i \in N} x_i \cdot \sum_{j \in S_i} y_j. \tag{21}$$

Eine Beispielrechnung kann in Gleichung 3.1 für den Beispielgraphen aus Abbildung 3.15 betrachtet werden.

$$\begin{aligned}
&y_a(x_A + x_B) \; + \\
&y_b(x_A + x_C) \; + \\
&y_c(x_B + x_C) \quad = \\
&\qquad x_A y_a + \; x_B y_a \qquad\quad + \\
&\qquad x_A y_b + \qquad\quad x_C y_b + \\
&\qquad\quad x_B y_c + \; x_C y_c \quad = \\
&\qquad\qquad\qquad\quad x_A(y_a + y_b) \; + \\
&\qquad\qquad\qquad\quad x_B(y_a + y_c) \; + \\
&\qquad\qquad\qquad\quad x_C(y_b + y_c)
\end{aligned} \tag{3.1}$$

[10] Zur Veranschaulichung wurden hier die Knotennamen als Indizes verwendet.

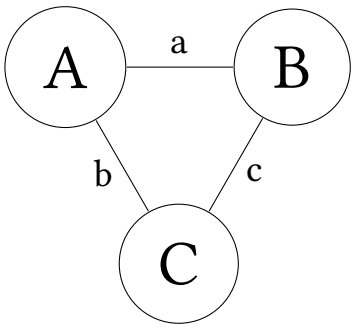

Abbildung 3.15: Beispielgraph für die Umformungen in Gleichung 3.1 [eigene Darstellung]

Wenn man nun die Nebenbedingung (10) des Duals betrachtet, kann man w_i für die einzelnen Faktoren, die jeweils mit x_i multipliziert werden, einsetzten und es folgt

$$\sum_{i \in N} x_i \cdot \sum_{j \in S_i} y_j \leq \sum_{i \in N} x_i \cdot w_i. \tag{22}$$

Setzt man (20), (21) und (22) zusammen, erhält man (19). Da für diese Beweisführung die Optimierungen der beiden Zielfunktionen in (12) und (16) mit einkalkuliert wurden, ergibt sich die Dualitäts-Eigenschaft

$$\max \sum_{j \in T} y_j \leq \min \sum_{i \in N} x_i \cdot w_i \tag{23}$$

aus (19).[11] Verbindet man die Aussagen aus (13), (23) und (18) in der Ungleichungskette

$$\sum_{j \in T} EW(j) \leq \max \sum_{j \in T} y_j \leq \min \sum_{i \in N} x_i \cdot w_i \leq W_{\text{opt}}, \tag{24}$$

folgt die Aussage

$$\sum_{j \in T} EW(j) \leq W_{\text{opt}}. \tag{25}$$

Die Primal-Dual-Methode hat dazu beigetragen diesen Zusammenhang herzustellen.[12] Durch (8) und (25) lässt sich nun die Ungleichungskette

$$W \leq \sum_{j \in T} EW(j) \cdot \max_{j \in T} |F(j) \cap I| \leq W_{\text{opt}} \cdot \max_{j \in T} |F(j) \cap I| \tag{26}$$

bilden und die Ungleichung (1) aus Satz 2 folgt.

[11] Mit den erfolgten Beispielrechnungen kann sogar $\max \sum_{j \in T} y_j = \min \sum_{i \in N} x_i \cdot w_i = 10.5$ für Veras Beispielnetzwerk festgestellen werden.

[12] Es gilt zu beachten, dass $EW(j)$ weiterhin die tatsächlich im Algorithmus vorkommenden Kantengewichte widerspiegelt und nicht mit den Werten y_j verwechselt werden sollte.

4 Fazit

In dieser Arbeit wurde der 2-Approximationsalgorithmus von Bar-Yehuda und Even analysiert und gezeigt, wie er für das gewichtete Vertex-Cover-Problem implementiert werden kann. Die zugehörigen Beweise und Erklärungen, sowie die dafür benötigten Grundlagen, konnten ausführlich erläutert werden. Diese Ausführungen ermöglichen die Anwendung des Algorithmus auf eine beliebige Problemstellung des gewichteten Vertex-Cover-Problems und vermitteln das dafür erforderliche Verständnis. Die Ergebnisse der Analyse können wie folgt zusammengefasst werden.

In Abschnitt 2.4 bis Abschnitt 2.5 wurde nachgewiesen, dass das gewichtete Vertex-Cover-Problem NP-vollständig ist und sich nicht in polynomieller Zeit lösen lässt, ohne gleichzeitig P = NP zu beweisen und das P-NP-Problem zu beantworten. In der Praxis bedeutet dies, dass große Instanzen des Problems sich aktuell nicht exakt lösen lassen. Bar-Yehuda und Even lieferten dafür den alternativen Ansatz eines 2-Approximationsalgorithmus, der nur eine lineare Laufzeit $\mathcal{O}(|E|)$ benötigt und folglich eine effiziente Möglichkeit bietet, gute Lösungen für das gewichtete Vertex-Cover-Problem zu berechnen. Der Beweis für das lineare Laufzeitverhalten konnte in Abschnitt 3.2 mit detaillierten Zwischenschritten anschaulich erläutert werden.

Das Verfahren ist für diese Arbeit in der Programmiersprache Python implementiert worden. Die lineare Laufzeit des Programms wurde in Abschnitt 3.3 schrittweise dargelegt und mit einem Laufzeittest bestätigt. Die Laufzeitanalyse ermöglichte zudem wesentliche Erkenntnisse über die interne Programmierung von Mengenobjekten in Python, die für eine erfolgreiche Implementation zu beachten sind. Um die Linearität der Laufzeit zu gewährleisten, wurden zusätzlich die für den Algorithmus definierten Funktionen als Listen-Objekte implementiert.

Der Algorithmus von Bar-Yehuda und Even ist ein 2-Approximationsalgorithmus für das gewichtete Vertex-Cover-Problem und gehört damit noch immer zu den besten Approximationsalgorithmen, die für dieses Problem gefunden wurden. Der Beweis für die Approximationsgarantie konnte in Abschnitt 3.4 mit passenden Visualisierungen vorgestellt werden. Ein wesentlicher Vorteil des Algorithmus ist die lineare Zeitkomplexität, wodurch er sich in der Praxis als deutlich effizienter gegenüber alternative Approximationsalgorithmen erweist (Williamson & Schmoys, 2011, S. 161). Statt ein lineares Programm vollständig zu lösen, nutzt der Algorithmus die Primal-Dual-Methode in der Konstruktion einer Lösung, bricht diese ab,

sobald alle Nebenbedingungen erfüllt sind, und erhält so ein effizienteres Laufzeitverhalten als alternative Ansätze (zum Beispiel aus dem *LP-Rounding*) (Williamson & Schmoys, 2011, S. 23). Der Ablauf dieser Konstruktion wurde in Abschnitt 3.1 beschrieben und in Abschnitt 3.4 durch die zugrunde liegenden mathematischen Definitionen bestätigt. Dabei konnte die Abweichung von der ursprünglichen Primal-Dual-Methode (Dantzig et al., 1956) beobachtet werden.

Abschließend lässt sich festhalten, dass die erste Realisierung der Primal-Dual-Approximations-Methode den Entwurf von weiteren effizienten Approximationsalgorithmen zu NP-vollständigen Problemen ermöglichte. Das Feedback Set Problem, das Warehause Location Problem und das Minimum Knapsack Problem gehören beispielsweise zu den Problemen, die einen guten Algorithmus durch die Methode erhielten (Williamson & Schmoys, 2011, S. 193 - 194). In ihrem Beitrag aus dem Buch „Approximation Algorithms for NP-Hard Problems“ stellten Goemans und Williamson die Formulierung einer generischen Primal-Dual-Approximations-Methode vor, die auf viele Netzwerk-Design-Probleme angewendet werden kann, und fassten auf diese Weise die Anwendung der Methode zusammen (Goemans & Williamson, 1996, S. 146). In Erinnerung an den im Jahr 2004 verstorbenen Shimon Even, entwickelten Reuven Bar-Yehuda et al. die *Local-Ratio-Methode*, die auf der von Bar-Yehuda und Even entwickelten Primal-Dual-Approximations-Methode beruht und auf eine breite Masse von NP-schweren Problemen angewendet werden kann, um gute Approximationsalgorithmen zu erhalten (Bar-Yehuda et al., 2004).

Neben Approximationsalgorithmen wird in der Forschung über NP-vollständige Probleme auch nach den effizientesten Algorithmen gesucht, die eine optimale Lösung liefern können, aber eine exponentielle Laufzeit benötigen. In einer Publikation aus dem Jahr 2010 stellten Chen et al. beispielsweise einen Algorithmus vor, der ein optimales Vertex Cover für das ungewichtete Vertex-Cover-Problem mit einer Laufzeit $\mathcal{O}(1.2738^B + Bn)$ mit B als Grenze der Instanz (siehe Abschnitt 2.5) berechnen konnte (Chen et al., 2010). In der Praxis werden diese Algorithmen häufig eingesetzt, da eine Worst-Case-Approximationsgarantie von höchstens dem Doppelten der optimalen Lösung in manche Anwendungen nicht besonders praktikabel erscheint. Allerdings konnten Goemans und Williamson die interessante Beobachtung machen, dass Approximationsalgorithmen, die auf der Primal-Dual-Approximations-Methode beruhen, in vielen Tests aus der theoretischen und praktischen Anwendung deutlich bessere Ergebnisse liefern, als die Approximationsgarantien versprechen (Goemans & Williamson, 1996, S. 186). Das bedeutet, dass die Approximationsqualität von Ausgaben dieser Algorithmen in der Praxis häufig noch besser ist, als erwartet. Mit der Primal-Dual-Approximations-Methode ermöglichten Bar-Yehuda und Even auf diese Weise bedeutende Fortschritte in der Entwicklung von Approximationsalgorithmen sowie deren Anwendungsfeldern. Seitdem hat die Methode sich als neuer Ansatz zur Entwicklung von Approximationsalgorithmen für NP-vollständige Probleme etabliert.

Literatur

Bar-Yehuda, R., & Even, S. (1981). A Linear-Time Approximation Algorithm for the Weighted Vertex Cover Problem. *Journal of Algorithms*, *2*(2), 198–203. https://doi.org/10.1016/0196-6774(81)90020-1

Bar-Yehuda, R., Bendel, K., Freund, A., & Rawitz, D. (2004). Local ratio: A unified framework for approximation algorithms. In Memoriam: Shimon Even 1935-2004. *ACM Comput. Surv.*, *36*(4), 422–463. https://doi.org/10.1145/1041680.1041683

Chen, J., Kanj, I. A., & Xia, G. (2010). Improved upper bounds for vertex cover. *Theoretical Computer Science*, *411*(40), 3736–3756. https://doi.org/https://doi.org/10.1016/j.tcs.2010.06.026

Cook, S. (2006). The P versus NP Problem. In J. Carlson, A. Jaffe & A. Wiles (Hrsg.), *The Millenium Prize Problems* (S. 87–104). Clay Mathematics Institue; American Mathematical Society.

Dantzig, G. B., Ford, L. R., & Fulkerson, D. R. (1956). A Primal-Dual Algorithm for Linear Programs. In H. W. Kuhn & A. W. Tucker (Hrsg.), *Linear Inequalities and Related Systems* (S. 171–182). Princeton University Press.

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman; Company.

Goemans, M. X., & Williamson, D. P. (1996). The Primal-Dual Method for Approximation Algorithms and Its Application to Network Design Problems. In D. S. Hochbaum (Hrsg.), *Approximation Algorithms for NP-Hard Problems* (S. 144–191). PWS Publishing Company.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. *arXiv preprint arXiv:quant-ph/9605043*. https://doi.org/10.48550/arXiv.quant-ph/9605043

Hochbaum, D. S. (1996). Introduction. In D. S. Hochbaum (Hrsg.), *Approximation Algorithms for NP-Hard Problems* (S. xiii–xxii). PWS Publishing Company.

Karp, R. M. (1972). Reducibility among Combinatorial Problems. In R. E. Miller, J. W. Thatcher & J. D. Bohlinger (Hrsg.), *Complexity of Computer Computations* (S. 85–103). Springer, Boston, MA. https://doi.org/10.1007/978-1-4684-2001-2_9

Statista Research Department. (2025). *Rechenleistung der leistungsstärksten Supercomputer weltweit im Juni 2025*. Verfügbar 5. August 2025 unter https://de.statista.com/statistik/daten/studie/193104/umfrage/rechenleistung-der-leistungsstaerksten-supercomputer-weltweit/

Upadhyay, A. (2024). *Looking Under the Hood of Python's Set Data Structure*. Verfügbar 4. September 2025 unter https://blog.codingconfessions.com/p/cpython-set-implementation

Weitz, E. (2021). *Konkrete Mathematik (nicht nur) für Informatiker: Mit vielen Grafiken und Algorithmen in Python* (2. Aufl.). Springer Spektrum.

Williamson, D. P., & Schmoys, D. B. (2011). *The Design of Approximation Algorithms*. Cambridge University Press.

Anhang

Eine im Rahmen dieser Arbeit erstellte Animation zu dem Approximationsalgorithmus von Bar-Yehuda und Even ist online verfügbar: https://youtu.be/Ozp0OVuEJj4

Der Approximationsalgorithmus von Bar-Yehuda und Even wurde im Rahmen dieser Arbeit in Python implementiert. Die in Abschnitt 3.1 eingeführten und in Abschnitt 3.3 genutzten Konstrukte T, S, w, und F wurden in der Implementation als `edges`, `vertices`, `weights` und `covering_vertices_for_given_edge_id` bezeichnet. Das Programm ist in allgemeiner Form hier abgebildet.

Codeblock A.1: Allgemeine Python-Implementation des Algorithmus von Bar-Yehuda und Even, 1981

```
def approximate_vertex_cover(edges,
                             vertices,
                             weights,
                             covering_vertices_for_given_edge_id):
  SW = weights
  I = set()
  J = edges
  while len(J) > 0:
        j = J.pop()
        covering_vertices = covering_vertices_for_given_edge_id[j]
        M = min([SW[i] for i in covering_vertices])

        for i in covering_vertices:
          SW[i] = SW[i] - M
          if(SW[i] == 0):
            k = i

        I.add(k)
        for edge in vertices[k]:
          J.discard(edge)
  return I
```

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit mit dem Titel

Analyse und exemplarische Implementation eines klassischen Approximationsalgorithmus für das NP-vollständige Problem Vertex Cover

selbstständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

Avery Laura Lönker
München, 5. September 2025