BACHELORARBEIT

zum Thema

Zur Anwendung des Verfahrens der lokalen Suche auf die Berechnung stark regulärer Graphen

Autor:	Jakob Dunkel
Matrikel:	25451
Einrichtung:	Hochschule Merseburg
Studiengang:	Bachelor Angewandte Informatik
1. Prüfer:	Prof. Dr. Andreas Spillner
2. Prüfer:	Prof. Dr. Doreen Straß
eingereicht am:	20.01.2025

Abstract

In dieser Bachelorarbeit werden die stark regulären Graphen und das Verfahren der lokalen Suche vorgestellt. Ziel der Arbeit ist es, die Berechnung eines stark regulären Graphen ausgehend von seinen Parametern v, k, λ und μ in Form einer lokalen Suche umzusetzen. Besondere Schwerpunkte sind dabei die Bildung des Rahmens der lokalen Suche anhand der Eigenschaften der stark regulären Graphen, die effiziente Umsetzung des Verfahrens, sowie die Anwendung der lokalen Suche auf theoretische stark reguläre Graphen, deren Existenz noch unbestätigt ist.

Inhaltsverzeichnis

1 Einleitung

In der Graphentheorie sind die stark regulären Graphen (auch kurz: SRGs) eine Untergruppe der regulären Graphen. Diese mitunter als besonders regulär oder schön bezeichneten Strukturen sind von Interesse, da sie in vielen verschiedenen Bereichen Anwendung finden, darunter Statistik, euklidische Geometrie, Gruppentheorie, Kodierungstheorie, endliche Geometrie, extreme Kombinatorik und Kryptographie [1, S. iii].

Der Aufbau dieser stark regulären Graphen ist im Allgemeinen nicht trivial. Insbesondere gibt es einige solche Graphen, deren Existenz lediglich vermutet wird und daher Bestandteil ungelöster mathematischer Probleme ist (siehe Abschnitt 2.3). Daher beschäftigt sich diese Bachelorarbeit mit der Frage, inwiefern sich das Verfahren der lokalen Suche dazu eignet, stark reguläre Graphen auf Grundlage ihrer Parameter algorithmisch zu generieren. Der Fokus liegt dabei laut Aufgabenstellung unter Anderem auf folgenden Fragen: Wie lassen sich die Eigenschaften von stark regulären Graphen so interpretieren, dass man sie als Grundlage für eine lokale Suche verwenden kann? Welche Optimierungen sind nötig, um eine solche lokale Suche möglichst effizient programmatisch umzusetzen? Und welche stark regulären Graphen lassen sich auf diese Art und Weise generieren (und mit welchem Aufwand)?

Kapitel 2 fasst die im Rahmen dieser Arbeit benötigten Grundlagen von stark regulären Graphen zusammen, während Kapitel 3 in das Konzept der lokalen Suche einführt. In Kapitel 4 werden die beiden Themen zusammengeführt und die Suche nach SRGs als ein konkretes Problem der diskreten Optimierung beschrieben. Kapitel 5 beginnt mit der programmatischen Umsetzung und adressiert insbesondere den hohen Rechenaufwand als Hauptproblem des naiven Ansatzes. Im Anschluss werden in Kapitel 6 verschiedene Varianten der lokalen Suche implementiert, deren Ergebnisse in Kapitel 7 ausgewertet werden.

2 Stark reguläre Graphen

2.1 Eigenschaften

Die Definition eines Graphen und der Adjazenzmatrix, sowie die eingeführten Begriffe sind alle [2, Kapitel 1] entnommen. Ein Graph ist ein geordnetes Paar $G = (V, E)$ mit $E \subseteq [V]^2$. Die Elemente $v \in V$ werden als Knoten bezeichnet und die Elemente $e \in E$ als Kanten. Zwei Knoten x und y werden als benachbart bezeichnet, wenn sie durch eine Kante verbunden sind, also wenn gilt $\{x, y\} \in E$. Die Kante liegt an x und y an und diese Knoten werden daher als Endpunkte der Kante bezeichnet. Die Menge aller mit x benachbarten Knoten heißt Nachbarschaft von x und ihre Anzahl ist der Grad von x. Ein Graph wird als leer bezeichnet, wenn die Menge seiner Kanten leer ist und als vollständig, wenn jeder Knoten mit jedem anderen Knoten benachbart ist. Außerdem ist ein Graph k-regulär, wenn alle in ihm enthaltenen Knoten den Grad k haben. Der Komplementgraph $\bar{G}$ eines Graphen G hat die gleiche Knotenmenge wie G, aber die „entgegengesetzte“ Knotenmenge $E(\bar{G}) = [V]^2 \setminus E(G)$.

Jeder Graph G hat eine Adjazenzmatrix A. Diese Adjazenzmatrix ist eine $(n \times n)$-Matrix (n sei hier die Anzahl der Knoten in G), deren Einträge angeben, ob die Knoten mit den jeweiligen Indizes benachbart sind. Sie ist daher für ungewichtete Graphen folgendermaßen definiert:

$$a_{ij} \coloneqq \begin{cases} 1 \; wenn \; (v_i v_j) \in E \\ 0 \; sonst \end{cases} \tag{2.1}$$

Eine interessante Eigenschaft der n-ten Potenzen der Adjazenzmatrix A^n ist, dass ihre Einträge $(A^n)_{ij}$ angeben, wie viele n Kanten lange Pfade es zwischen den Knoten i und j gibt.

Sofern nicht anders angegeben, ist jeder in dieser Arbeit betrachtete Graph ein endlich großer einfacher Graph. Einfache Graphen sind ungerichtete ungewichtete Graphen ohne Schleifen oder Mehrfachkanten [3, S. 2]. „Ungerichtet“ bedeutet, dass (xy) und (yx) dieselbe Kante bezeichnen. „Ungewichtet“ heißt, Kanten haben keine unterschiedlichen Wertigkeiten. „Ohne Schleifen oder Mehrfachkanten“ verbietet Kanten (xx), die einen Knoten mit sich selbst verbinden sowie mehrere Kanten mit den gleichen Endpunkten. Diese Einschränkung bedeutet insbesondere, dass die Adjazenzmatrix eines Graphen nicht nur quadratisch, sondern auch symmetrisch ist und die Elemente ihrer Hauptdiagonale immer gleich 0 sind.

Die Definitionen eines stark regulären Graphen sind [1, S. 2] entnommen. Ein Graph mit v Knoten ist stark regulär, wenn er folgende drei Kriterien erfüllt:

1: Der Graph ist k-regulär.
2: Jedes Paar benachbarter Knoten hat λ gemeinsame Nachbarn.
3: Jedes Paar nicht benachbarter Knoten hat μ gemeinsame Nachbarn.

Ist dieses Kriterium erfüllt, nennt man (v, k, λ, μ) die Parameter des stark regulären Graphen. Dieser Zusammenhang lässt sich auch kombinatorisch beschreiben. Man wähle zwei Knoten x und y. Die Anzahl der gemeinsamen Nachbarn von x und y beträgt k, wenn x und y gleich sind, λ, wenn x und y benachbart sind und μ, wenn x und y nicht benachbart sind. Formal lassen sich diese Kriterien in Form von zwei Gleichungen beschreiben, sodass ein Graph genau dann stark regulär ist, wenn folgende Gleichungen erfüllt sind:

$$AJ = JA = kJ \tag{2.2}$$

$$A^2 = kI + \lambda A + \mu(J - I - A) \tag{2.3}$$

Dabei sind A die Adjazenzmatrix des Graphen, J die $(v \times v)$-Einsmatrix und I die $(v \times v)$-Einheitsmatrix. Gleichung (2.2) ist erfüllt, wenn der Graph k-regulär ist. Die Summanden von (2.3) ergeben sich aus den Definitionen der Parameter k, λ und μ, sowie der Eigenschaft, dass die Einträge $(A^2)_{ij}$ der Anzahl der 2 Kanten langen Pfade zwischen Knoten i und Knoten j entsprechen.

Der Komplementgraph $\bar{G}$ eines SRG G mit den Parametern (v, k, λ, μ) ist ebenfalls ein SRG mit den Parametern $(v, v - k - 1, v - 2k + \mu - 2, v - 2k + \lambda)$. Leere Graphen und vollständige Graphen zählen nicht zu den stark regulären Graphen. In einem leeren Graphen gibt es keine benachbarten Knoten, daher ist λ nicht definiert und in einem vollständigen Graphen gibt es keine nicht benachbarten Knoten, weshalb μ nicht definiert ist. Für die Größen der Parameter gelten folgende Zusammenhänge: $0 < k < v - 1$, $0 \leq \lambda \leq k - 1$, $0 \leq \mu \leq k$ und für $\mu \neq 0$ ebenfalls $\mu(v - k - 1) = k(k - \lambda - 1)$. Diese Einschränkungen der Parameter werden im Rahmen dieser Bachelorarbeit nicht benötigt, da die Parameter als Ausgangspunkt der lokalen Suche vorgegeben sind.

2.2 Beispiele

Es gibt unendlich viele stark reguläre Graphen [1, S. 5]. Dieser Abschnitt soll dazu dienen, einen Überblick über einige Gruppen von SRGs zu geben sowie einige kleinere Exemplare zu visualisieren. Wie am Ende des letzten Abschnittes erwähnt, setzt diese Bachelorarbeit die Parameterkombinationen der gesuchten SRGs als gegeben voraus. Brouwer/Van Maldeghem haben eine Liste von „allen möglichen Parametern“ (vgl. [1]) zusammengestellt, in [1] Kapitel 12 bis 512 Knoten, unter https://aeb.win.tue.nl/graphs/srg/srgtab.html bis 1300 Knoten. Diese Liste enthält sowohl SRGs, deren Existenz bestätigt ist, als auch SRGs, deren Existenz widerlegt oder noch offen ist. Die nachfolgende Abbildung zeigt die ersten drei Einträge dieser Liste.

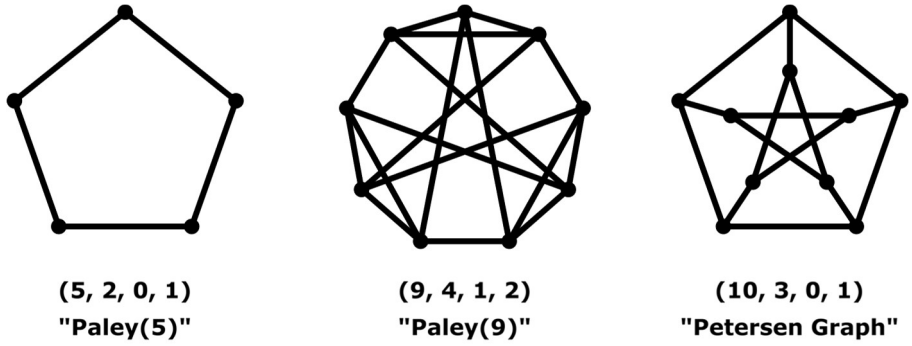

Abbildung 1: Grafische Darstellung einiger kleiner SRGs

Die Eigenschaften der folgenden Graphen und Graphenfamilien stammen aus [1, S. 2-5]. Der in Abbildung 1 dargestellte Petersen-Graph wurde 1898 von Julius Petersen als kleinstes Gegenbeispiel zu einer Behauptung bezüglich der Kantenfärbung von kubischen Graphen konstruiert. Donald Knuth bezeichnet ihn als „eine außergewöhnliche Konfiguration, die als Gegenbeispiel zu vielen optimistischen Voraussagen über die Eigenschaften von Graphen im Allgemeinen dient“ (vgl. [4]).

Der Paley-Graph Paley(q) ist genau dann ein stark regulärer Graph mit den Parametern $(4t+1, 2t, t-1, t)$, wenn $q = 4t+1$ entweder eine Primzahl oder eine ganzzahlige Potenz einer Primzahl ist. Paley-Graphen sind selbst-komplementär, sind also abgesehen von der Reihenfolge ihrer Knoten mit ihrem eigenen Komplementgraph identisch. Vertreter dieser Gruppe sind neben den Exemplaren aus Abbildung 1 beispielsweise $(13,6,2,3)$, $(17,8,3,4)$, $(25,12,5,6)$, $(29,14,6,7)$ und $(37,18,8,9)$.

Der $(n \times n)$-Turmgraph (engl. rook graph; in [1, S. 5] „$q \times q$ grid“ genannt) besteht aus einem Raster von Knoten mit n Zeilen und n Spalten. Zwei Knoten sind durch eine Kante verbunden, wenn sie sich entweder in der gleichen Zeile oder der gleichen Spalte des Rasters befinden. Der Name leitet sich davon ab, dass die Kanten eines solchen Graphs allen zulässigen Zügen eines Turms auf einem $n \times n$ Felder großen Schachfeld entsprechen. Diese Graphen sind für $n \geq 2$ stark regulär und haben die Parameter $(n^2, 2n-2, n-2, 2)$.

Als letzte Gruppen seien die Leitersprossengraphen (engl. ladder rung graph) und Cocktailpartygraphen (engl. cocktail party graph) genannt. Leitersprossengraphen sind Graphen des Typs nK_2, das heißt sie bestehen aus n isolierten Gruppen von benachbarten Knotenpaaren ($n > 1$). Diese sind stark regulär mit den Parametern $(2n, 1, 0, 0)$. Der Komplementgraph des Leitersprossengraphen nK_2 ist der vollständige n-partite Graph $K_{n\times 2}$, auch Cocktailpartygraph genannt. Als Komplementgraph eines SRG ist auch er stark regulär mit den Parametern $(2n, 2n-2, 2n-4, 2n-2)$. In [1, S. 2] werden diese beiden Gruppen als imprimitiv bezeichnet und daher nicht weiter betrachtet. Aus diesem Grund werden sie auch nicht in der Liste der möglichen Parameter aufgeführt. Ohne weiter auf den Status dieser Graphen einzugehen, werden sie an dieser Stelle explizit erwähnt, da sie in späteren Kapiteln dieser Arbeit interessante Grenzfälle darstellen (siehe Abschnitte 6.1 und 7.2).

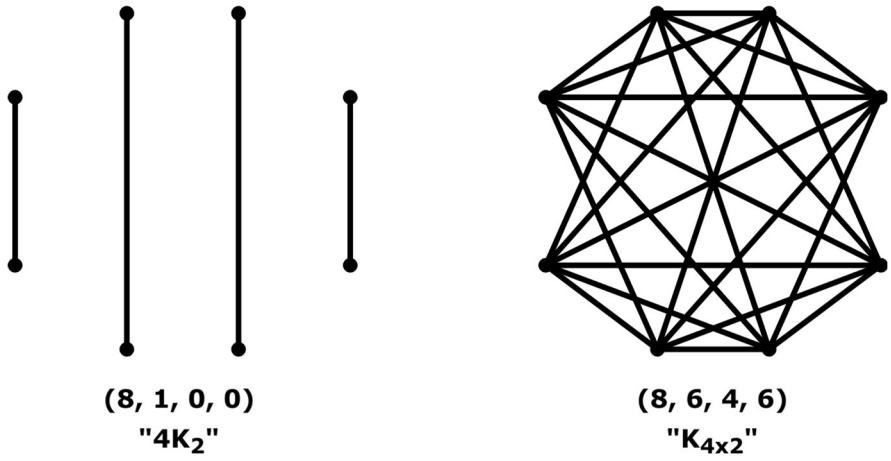

Abbildung 2: Leitersprossengraph und Cocktailpartygraph mit 8 Knoten

2.3 Offene Fragen

Wie in den meisten Bereichen der Mathematik gibt es auch um stark reguläre Graphen viele offene Fragen oder Probleme. Darunter einige, die nach der Existenz eines hypothetischen Graphen mit bestimmten Eigenschaften fragen. Beispiele für solche Fragen sind:

Ein Moore-Graph ist ein Graph mit dem Durchmesser d und dem Taillenumfang (engl. girth) $2d + 1$. Der Satz von Hoffman-Singleton besagt, dass ein Moore-Graph stark regulär ist, wenn er k-regulär ist mit $k \in \{2,3,7,57\}$. Für die ersten drei k existieren Paley(5) $(5,2,0,1)$, der Petersen-Graph $(10,3,0,1)$ und der Hoffman-Singleton-Graph $(50,7,0,1)$. Bisher ist kein stark regulärer Moore-Graph mit $k = 57$ bekannt. [5]

Es sind 7 dreiecksfreie ($\lambda = 0$) stark reguläre Graphen bekannt. Paley(5) $(5,2,0,1)$, der Petersen-Graph $(10,3,0,1)$, der Clebsch-Graph $(16,5,0,2)$, der Hoffman-Singleton-Graph $(50,7,0,1)$, der Gewirtz-Graph $(56,10,0,2)$, der M22-Graph $(77,16,0,4)$ und der Higman-Sims-Graph $(100,22,0,6)$. Die offene Frage ist, ob ein achter stark regulärer Graph mit $\lambda = 0$ existiert. [5]

Ein geodätischer Graph ist ein Graph, in dem es einen eindeutigen kürzesten Pfad zwischen jedem Knotenpaar gibt. Es ist nicht bekannt, ob es unendlich viele stark reguläre geodätische Graphen gibt. Ebenso ist nicht bekannt, ob es einen stark regulären geodätischen Graphen gibt, der kein Moore-Graph ist. [6]

Es gibt genau 5 mögliche SRG-Parameterkombinationen, bei denen jede Kante zu einem einzigartigen Dreieck gehört ($\lambda = 1$) und jede nicht vorhandene Kante die Diagonale eines einzigartigen Vierecks ist ($\mu = 2$). Zwei dieser Graphen, Paley(9) $(9,4,1,2)$ und der Berlekamp-Van Lint-Seidel-Graph $(243,22,1,2)$ wurden bereits gefunden. Zwei weitere Graphen haben sehr große Parameter, $(6273,112,1,2)$ und $(494019, 994,1,2)$. Der letzte Graph $(99,14,1,2)$ ist der kleinste solche Graph, dessen Existenz unbekannt ist [7]. Die Frage nach seiner Existenz ist heute als Conways 99-Graph-Problem bekannt. Es ist eines von fünf (seit 2017 vier) ungelösten Problemen, für die der Namensgeber John H. Conway im Jahr 2014 ein Preisgeld von je 1000 US-Dollar ausgeschrieben hat [8].

Der potenzielle SRG $(99,14,1,2)$ ist gemeinsam mit anderen unbestätigten SRGs wie $(69,20,7,5)$ oder $(85,14,3,2)$ im Rahmen dieser Bachelorarbeit von besonderem Interesse, da er wegen seiner (vergleichsweise) geringen Größe leichter algorithmisch handhabbar ist als andere in diesem Kapitel erwähnte Beispiele.

3 Lokale Suche

3.1 Grundkonzept

Die Berechnung eines stark regulären Graphen ausgehend von dessen Parametern (v, k, λ, μ) kann als kombinatorisches Problem verstanden werden. Der gesuchte SRG ist eine bestimmte Konfiguration von v Knoten und $(v \cdot k) \,/\, 2$ Kanten. Das Problem: Selbst im Falle eines kleinen SRG wie Paley(17) gibt es etwa $6 \cdot 10^{39}$ mögliche Graphen mit 17 Knoten und 136 Kanten[1]. Der „ausführliche“ Ansatz, der jede mögliche Lösung betrachtet, ist also bei kombinatorischen Problemen wie diesem nicht praktisch durchführbar.

Die lokale Suche ist ein Optimierungsverfahren, das dem Bereich der Heuristik zuzuordnen ist. Das bedeutet, dass keine Garantie besteht, die global optimale Lösung zu finden. Im Gegenzug ist das Verfahren in der Regel mit überschaubarem Aufwand (in Polynomialzeit) durchführbar und daher für die algorithmische Bewältigung des Problems, hier der Berechnung eines SRG, geeignet.

Der Begriff der lokalen Suche leitet sich von der Idee ab, dass zu jedem Zeitpunkt nur eine einzige Lösung und ihr unmittelbares Umfeld, Nachbarschaft[2] genannt, betrachtet wird. Die Lösungen in der Nachbarschaft werden hinsichtlich einer Zielfunktion evaluiert, die die Güte der Lösung quantifiziert. Anschließend wählt man die beste dieser Lösungen aus und betrachtet im nächsten Schritt deren Nachbarschaft. Diese Folge aus Schritten in Richtung der besten Lösung endet, wenn man einer Lösung angelangt ist, die besser ist als alle Lösungen aus ihrer Nachbarschaft. Diese Lösung stellt dann ein lokales Optimum hinsichtlich der Zielfunktion dar. Da die Zielfunktion aber mehrere lokale Optima haben kann, deren Eigenschaften aufgrund des lokalen Charakters des Verfahrens nicht bekannt sind, ist in der Regel nicht feststellbar, ob es sich auch um ein globales Optimum handelt.

[1] $(17 \cdot 16)$ über 136. Die tatsächliche Zahl der unterschiedlichen Graphen ist niedriger aufgrund von Isomorphismen. Das Beispiel dient lediglich der Veranschaulichung der Komplexität kombinatorischer Zusammenhänge.

[2] Nicht verwandt mit dem Nachbarschaftsbegriff aus Abschnitt 2.1.

3.2 Charakteristiken

Eine lokale Suche wird durch die folgenden Elemente charakterisiert: Suchraum, Zielfunktion, Startlösung und Schritt bzw. Nachbarschaft.

Der Suchraum L ist die Menge aller möglicher Lösungen l, die als potenzielles Ziel des Verfahrens in Frage kommen. Er kann auch als Menge der zulässigen Lösungen bezeichnet werden, denn Lösungen außerhalb dieser Menge kommen nicht als potenzielles Ziel in Frage und sind daher für die Optimierung irrelevant.

Die Zielfunktion ist das Zentrum des Optimierungsverfahrens, da sie das Kriterium ist, das es zu optimieren gilt. Sie ist eine Funktion $f: L \to \mathbb{R}$, die jeder Lösung im Suchraum einen Wert zuweist. Ziel der lokalen Suche ist es, Optima dieser Zielfunktion zu ermitteln.

Die Startlösung ist die Lösung $l \in L$, mit der die lokale Suche beginnt. Die Wahl der Startlösung hat einen großen Einfluss darauf, ob das am Ende der lokalen Suche gefundene Optimum ein globales Optimum ist. Lassen sich im Vorherein keine Aussagen über das Verhalten der Zielfunktion treffen, ist es naheliegend, die Startlösung willkürlich bzw. zufällig zu wählen.

In jedem Schritt der lokalen Suche wird die Nachbarschaft der aktuellen Lösung ermittelt und die Zielfunktion für jede Lösung in der Nachbarschaft berechnet. Die Nachbarschaft $N(l)$ einer Lösung l ist definiert als die Menge aller Lösungen, die man durch eine festgelegte Veränderung von l erhalten kann. Diese Veränderung wird Schritt genannt, denn sie definiert den Unterschied zwischen einer Lösung und ihren Nachbarn und somit die „Distanz“, die in einem Schritt der lokalen Suche in Richtung des Optimums zurückgelegt wird. Der Suchraum L darf während der lokalen Suche nicht verlassen werden, daher ist der Schritt so zu wählen, dass alle Nachbarn einer Lösung ebenfalls Teil des Suchraums sind, also: $\forall l \in L: (\forall l' \in N(l): l' \in L)$.

4 Stark reguläre Graphen als Problem der lokalen Suche

4.1 Zielfunktion

Um einen stark regulären Graphen mittels lokaler Suche zu berechnen, wird eine Zielfunktion benötigt, die jedem Graphen (im Suchraum) eine Zahl zuordnet. Dieser Funktionswert soll ein Maß dafür sein, wie stark regulär der jeweilige Graph ist. Insbesondere muss die Zielfunktion dazu geeignet sein, aus einer Menge von nicht stark regulären Graphen denjenigen zu identifizieren, der „am meisten“ stark regulär ist. Die Definitionen eines SRG sind nicht als Zielfunktion verwendbar, da sie einen Graphen lediglich als stark regulär oder nicht stark regulär bestimmen können, jedoch ohne den für die lokale Suche benötigten Gütegrad. Zur Lösung dieses Problems werden die Gleichungen aus Abschnitt 2.1 benötigt. Ein Graph ist demnach genau dann stark regulär mit den Parametern (v, k, λ, μ), wenn gilt:

$$AJ = JA = kJ \tag{4.1}$$

$$A^2 = kI + \lambda A + \mu(J - I - A) \tag{4.2}$$

Dabei sei A die Adjazenzmatrix des Graphen, J die $(v \times v)$-Einsmatrix und I die $(v \times v)$-Einheitsmatrix. Gleichung (4.1) entspricht der Voraussetzung, dass ein SRG k-regulär ist. Dieses Kriterium kann durch eine Anpassung des Suchraums erreicht werden und entfällt daher an dieser Stelle. Gleichung (4.2) führt nach einer beidseitigen Subtraktion von A^2 zu:

$$0 = kI + \lambda A + \mu(J - I - A) - A^2 \tag{4.3}$$

Diese Form der Gleichung veranschaulicht, dass es sich bei diesem Kriterium um eine Differenz handelt, die im Falle eines stark regulären Graphen 0 beträgt. Seien beispielsweise G_1 und G_2 zwei Graphen gleicher Größe ($|V_1| = |V_2|$ und $|E_1| = |E_2|$), betrachtet im Rahmen einer lokalen Suche nach dem SRG (v, k, λ, μ). Evaluiert man Gleichung (4.3) durch Einsetzen der SRG-Parameter und der Adjazenzmatrizen A_1 und A_2, erhält man als Differenz links des Gleichheitszeichens für G_1 betragsmäßig einen deutlich niedrigeren Wert als für G_2. Da in beiden Fällen k, λ, μ, J und I gleich sind und die Differenz daher nur von den Adjazenzmatrizen abhängt, liegt die Annahme nah, dass die Adjazenzmatrix A_1 und somit der Aufbau des Graphen G_1 deutlich weniger vom „Soll“ eines SRG abweicht als A_2 bzw. der Aufbau von G_2.

Basierend auf dieser Annahme wird die Zielfunktion folgendermaßen definiert:

$$f: L \to \mathbb{N}_0$$
$$f(l) = \sum_{i=1}^{v} \sum_{j=1}^{v} \left| (A^2)_{ij} - P_{ij} \right|$$
$$P = kI + \lambda A + \mu(J - I - A)$$

Da es sich bei dieser Zielfunktion um eine Differenz mit globalen Optima $f(l) = 0$ handelt, ist ihr Wert im Laufe der lokalen Suche zu minimieren. Die symmetrische $(v \times v)$-Matrix P wird im weiteren Verlauf dieser Bachelorarbeit aufgrund ihrer Abhängigkeit von den SRG-Parametern als Parametermatrix bezeichnet.

4.2 Suchraum und Startlösung

Um die Zielfunktion aus der Definition eines stark regulären Graphen herzuleiten, wurde auf die Evaluation der Gleichung (4.1) verzichtet. Dabei handelt es sich um die Voraussetzung, dass der Graph k-regulär sein muss. Um dieses Kriterium dennoch zu erfüllen, wird es in die Definition des Suchraums der lokalen Suche übernommen. Somit ist der Suchraum formal definiert als die Menge aller k-regulären Graphen auf v Knoten. Die Fixierung der Knotenzahl dient dazu, die Veränderungen am Graphen während der lokalen Suche auf die Kantenmenge zu beschränken.

Die Startlösung als Ausgangspunkt der lokalen Suche muss ein Element des Suchraums sein. Diese Definition des Suchraums hat somit zur Folge, dass vor Beginn der lokalen Suche ein k-regulärer Graph auf v Knoten erzeugt werden muss (siehe Abschnitt 6.1).

4.3 Schritt und Nachbarschaft

In Abschnitt 3.2 wurde eingeführt, dass alle Nachbarn einer Lösung $l \in L$ ebenfalls Teil des Suchraums sein müssen. Die im vorhergehenden Kapitel eingeführte Definition des Suchraums als Menge aller k-regulären Graphen auf v Knoten führt daher zu Einschränkungen bezüglich der Frage, welche Operation sich als Schritt der lokalen Suche eignet. Der Schritt muss eine Operation sein, die die Kanten des Graphen (und somit dessen Adjazenzmatrix) verändert, während die Anzahl der Knoten v, die Anzahl der Kanten $(v \cdot k) \,/\, 2$ und die Grad k aller Knoten konstant bleibt.

Die kleinstmögliche Operation, die die Kantenmenge des Graphen verändert, ist die Verschiebung einer einzelnen Kante. Diese Operation ist äquivalent zum Entfernen einer Kante und dem anschließenden Hinzufügen einer anderen Kante. Ausgehend von einem k-regulären Graph resultiert diese Änderung jedoch unweigerlich[3] in einem nicht-regulären Graphen. Ein solcher Graph ist nicht Teil des Suchraums, weshalb die Operation als Schritt unbrauchbar ist.

Ein weiterer Ansatz besteht in der Entfernung zweier Kanten und dem anschließenden Hinzufügen zweier anderer Kanten. Die Auswahl zweier zu entfernender Kanten führt zu folgenden vier potenziellen Fällen, unterschieden durch die Anzahl der einzigartigen Endpunkte der beiden Kanten[4]:

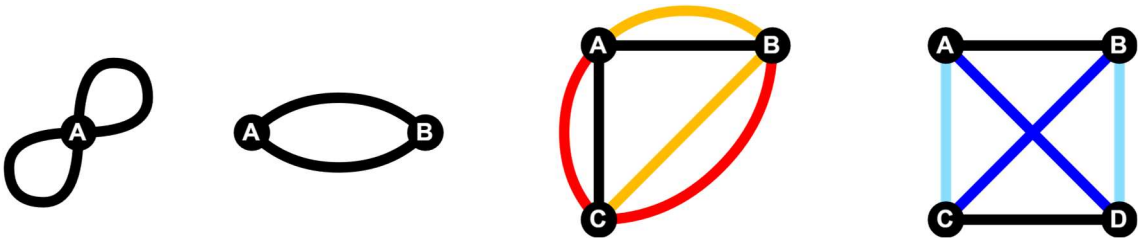

Abbildung 3: Auswahl von 2 Kanten (alte Kanten schwarz, neue farbig)

Fall 1 und 2 zeigen zwei Kanten mit einem bzw. zwei anliegenden Knoten. Die dafür nötigen Schleifen und Mehrfachkanten treten in den hier betrachteten einfachen Graphen nicht auf, daher können beide Fälle kategorisch ausgeschlossen werden.

Fall 3 zeigt zwei Kanten mit drei anliegenden Knoten. In der Ausgangsposition (schwarze Kanten) ist der Graph k-regulär. Es ist leicht zu erkennen, dass dieses Kriterium beim Versuch, zwei neue Kanten (orange oder rot) einzufügen, verletzt wird, da sich der Grad einiger Knoten ändert.

[3] Außer die neue Kante ist gleich der alten Kante, was jedoch den Graphen unverändert lässt.
[4] Die Abbildungen zeigen lediglich die sich verändernden Ausschnitte des Graphen. Kanten zu Knoten außerhalb der Abbildung werden der Übersichtlichkeit halber nicht dargestellt.

Fall 4 zeigt zwei Kanten mit vier anliegenden Knoten. In der Ausgangsposition hat jeder der vier Knoten genau einen Nachbarn (im abgebildeten Ausschnitt). Im Gegensatz zu Fall 3 ist es hier nach dem Entfernen zweier Kanten (schwarz) möglich, zwei neue Kanten (hell- oder dunkelblau) so einzufügen, dass alle anliegenden Knoten ihren ursprünglichen Grad behalten. Um Mehrfachkanten zu vermeiden, setzt dies voraus, dass die neuen Kanten nicht bereits Teil der Kantenmenge des Graphen sind. In Fall 4 ist es daher möglich, die Kantenmenge des Graphen (also auch seine Adjazenzmatrix) zu verändern, ohne die Anzahl der Knoten, Kanten oder Nachbarn eines einzelnen Knoten zu beeinflussen.

Der Schritt der lokalen Suche ist also folgende Operation: Wähle zwei Kanten des Graphen $\overline{AB}$ und $\overline{CD}$ mit 4 unterschiedlichen Endpunkten A, B, C und D. Entferne nun das Kantenpaar $\overline{AB}$ und $\overline{CD}$ und füge entweder das Kantenpaar $\overline{AC}$ und $\overline{BD}$ oder das Kantenpaar $\overline{AD}$ und $\overline{BC}$ hinzu[5]. Folglich ist die Nachbarschaft eines Lösungsgraphen l definiert als die Menge aller Graphen, die durch die Ausführung dieser Operation auf l erhalten werden können.

[5] Diese Operation ist der Tauschoperation sehr ähnlich, die die Grundlage der k-Opt-Algorithmen bildet, welche beispielsweise beim Problem des Handlungsreisenden angewandt werden.

5 Naiver Ansatz und Berechnung der Zielfunktion

In diesem Kapitel, dem ersten Teil der Umsetzung, wird die Programmiersprache Python verwendet. Diese Entscheidung wurde getroffen, weil Python sich mit seiner einfachen Syntax und übersichtlichen Funktionen wie Listenausdrücken gut für die Programmierung eines „ersten Entwurfs“ eignet.

5.1 Naive Implementierung der lokalen Suche

Die erste naive Umsetzung des im vergangenen Kapitel beschriebenen Verfahrens resultierte in *srg_naive.py* und *graph_naive.py*.

Die Datei *graph_naive.py* ist ein Modul, das einen Graphen als Klasse Graph implementiert. Objekte der Klasse Graph haben Attribute für die Knotenzahl (Datentyp int) sowie die Kantenmenge des Graphen (Datentyp list[tuple[int, int]]). Die Objekte haben außerdem Methoden für das Hinzufügen und Entfernen von Kanten (die lediglich die Kantenmenge anpassen) und für die Berechnung der Adjazenzmatrix, sowie der quadrierten Adjazenzmatrix.

srg_naive.py importiert das Modul *graph_naive.py* und implementiert die lokale Suche. Im Folgenden ist *srg_naive.py* als Pseudocode dargestellt:

```
1     Einlesen der SRG-Parameter
2     Generierung eines k-regulären Graphen G (Startlösung)
3     Schleife: lokale Suche
4           aktuellerWert = f(G)
5           falls aktuellerWert = 0:
6                 G ist SRG, beende lokale Suche
7           besterNachbar = NULL; besterNachbarWert = INF
8           Schleife: betrachte alle validen Kantenpaare
9                 ermittle alteKanten und neueKanten
10                lösche alteKanten aus G und addiere neueKanten
11                berechne Zielfunktion f(G‘)
12                falls f(G‘) < f(G):
13                      besterNachbar = (alteKanten, neueKanten)
14                      besterNachbarWert = f(G‘)
15                mache Zeile 10 rückgängig
16          falls besterNachbarWert < aktuellerWert:
17                lösche bzw. addiere Kanten aus besterNachbar
18          sonst:
19                G ist lokales Optimum, beende lokale Suche
```

Quelltext 1: Pseudocode von srg_naive.py

Die Ausführung dieses Programms zeigt, dass die in Kapitel 4 beschriebene lokale Suche tatsächlich in der Lage ist, stark reguläre Graphen zu erzeugen. Die folgende Tabelle stellt das Verhalten bei ausgewählten Eingabeparametern dar:

SRG-Parameter	**Ergebnis**	**Zeit**
(5, 2, 0, 1)	SRG gefunden	<0,1
(9, 4, 1, 2)	SRG gefunden	<0,1
(10, 3, 0, 1)	SRG gefunden	<0,1
(15, 6, 1, 3)	lokales Optimum, $f(G) = 76$	1,3
(25, 8, 3, 2)	lokales Optimum, $f(G) = 300$	62

Tabelle 1: Von srg_naive.py erzielte Ergebnisse; Zeit in Sekunden

SRGs mit niedrigen Parametern lassen sich also sehr schnell berechnen. An Zeile 4 und 5 erkennt man hingegen, dass die Suche nach dem SRG $(25{,}8{,}3{,}2)$ mit 25 Knoten und 100 Kanten fast 50-mal so lange dauert, wie die Suche nach dem SRG $(15{,}6{,}1{,}3)$ mit 15 Knoten und 45 Kanten. Dieser enorme Unterschied hinsichtlich der Laufzeit ist ein Anzeichen dafür, dass der implementierte Algorithmus eine sehr hohe Zeitkomplexität besitzt.

Daher ist eine Komplexitätsanalyse für Quelltext 1 notwendig. Zeilen 1, 6, 7, 9, 13 und 19 sind von konstantem Zeitaufwand ($O(1)$). Die Funktionen add_edge() und remove_edge() aus *graph_naive.py* bestehen aus elementaren Listenoperationen auf der Kantenliste. Sie sind daher $O(|E|) = O(v \cdot k)$; gleiches gilt für Zeilen 10, 15 und 17. Die Generierung des Startgraphen in Zeile 2 ist $O(v^2 \cdot k)$ (siehe Abschnitt 6.1). Der restliche Aufwand liegt in der Berechnung der Zielfunktion in Zeilen 4 und 11.

```
1     Berechne Adjazenzmatrix A anhand der Liste von Kanten
2     A² = A * A
3     Berechne Parametermatrix anhand von Adjazenzmatrix
4     ergebnis = 0
5     Schleife über alle Zeilen i:
6          Schleife über alle Spalten j:
7               ergebnis += |(A²(i,j) - P(i,j)|
8     return ergebnis
```

Quelltext 2: Pseudocode der Berechnung der Zielfunktion aus srg_naive.py

In Quelltext 2 haben Zeilen 4 und 8 konstanten Aufwand. Zeilen 1 und 3 sind jeweils die zellenweise Erstellung einer $(v \times v)$-Matrix und daher $O(v^2)$. Die Doppelschleife in Zeilen 5 bis 7 entspricht der Doppelsumme aus der Definition der Zielfunktion und ist ebenfalls $O(v^2)$. In Zeile 2 wird eine Matrixmultiplikation von A mit sich selbst durchgeführt, diese hat den Aufwand $O(v^3)$.

Durch den hohen Aufwand der Matrixmultiplikation ist die (naive) Berechnung der Zielfunktion $O(v^3)$. Die Zielfunktion muss laut Quelltext 1 in jedem Schritt der lokalen Suche in Zeile 4 einmal und in Zeile 11 einmal pro Kantenpaar durchgeführt werden. Damit hat die (naive) Durchführung einer lokalen Suche die Komplexität $O(schritte \cdot (|E|)^2 \cdot v^3) = O(schritte \cdot (v \cdot k \,/\, 2)^2 \cdot v^3) = O(schritte \cdot k^2 \cdot v^5)$[6].
Um die Laufzeit des Programms zu verbessern, muss die Berechnung der Zielfunktion vereinfacht werden.

5.2 Effizientere Berechnung der Zielfunktion

Im vergangenen Abschnitt entstand der hohe Rechenaufwand bei der Berechnung der Zielfunktion dadurch, dass nach jeder Veränderung des Graphen seine Adjazenzmatrix A, die quadrierte Adjazenzmatrix A^2 und die Parametermatrix P von Grund auf neu berechnet werden mussten. Daher stellt sich die Frage, ob es stattdessen möglich ist, lediglich schrittweise Veränderungen an diesen Matrizen vorzunehmen. Die folgenden Überlegungen beziehen sich auf das Hinzufügen einer Kante; das Verfahren beim Entfernen einer Kante ist analog.

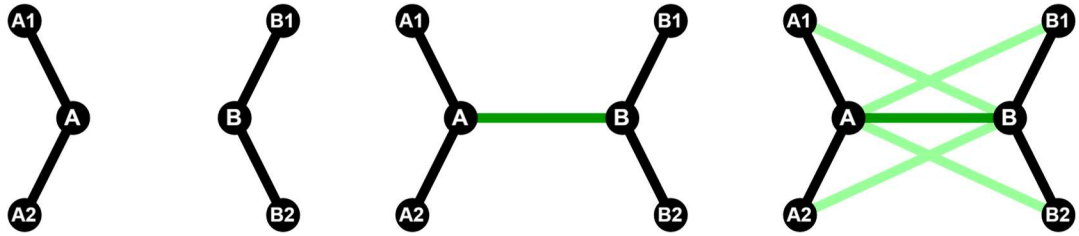

Abbildung 4: Visualisierung der neuen Pfade im Graphen beim Hinzufügen einer Kante (dunkelgrün)

Abbildung 4 zeigt: beim Hinzufügen einer Kante entsteht ein neuer Pfad der Länge 1 (A nach B, dunkelgrün) und mehrere neue Kanten der Länge 2 (A über B zu dessen Nachbarn Bn sowie B über A zu dessen Nachbarn An, hellgrün). In der Adjazenzmatrix ändern sich deshalb nur zwei Einträge, A_{AB} und A_{BA} werden von 0 zu 1. Die Einträge der Parametermatrix sind (abgesehen von der Hauptdiagonale) direkt von der Adjazenzmatrix abhängig. Daher ändern sich auch in dieser nur zwei Einträge, P_{AB} und P_{BA} werden von μ zu λ.

[6] Anmerkung des Autors: Eine Laufzeit abhängig von der fünften Potenz einer Eingabegröße ist mir noch nie begegnet. Im Studium werden selbst als „schlechte Beispiele“ für ineffiziente Algorithmen meist nur dritte Potenzen verwendet.

Die neuen Pfade der Länge 2 wurden als „A zu den Nachbarn von B“ und „B zu den Nachbarn von A“ identifiziert. Daher sind die Änderungen der quadrierten Adjazenzmatrix: Addiere die B-te Zeile/Spalte der Adjazenzmatrix (Nachbarn von B) auf die A-te Zeile/Spalte der quadrierten Adjazenzmatrix (Pfade mit Länge 2 von A) und die A-te Zeile/Spalte der Adjazenzmatrix (Nachbarn von A) auf die B-te Zeile/Spalte der quadrierten Adjazenzmatrix (Pfade der Länge 2 von B).

Wenn also A, A^2, P und $f(G)$ als Attribute des Graphen gespeichert werden, müssen mit jedem Hinzufügen/Entfernen einer Kante nur jeweils 2 Einträge in A und P, sowie $4v-2$ Einträge (2 Zeilen und 2 Spalten) in A^2 verändert werden. Wegen der $4v-2$ Änderungen in A^2 sind außerdem $4v-2$ Veränderungen an $f(G)$ nötig. Durch die Umgehung der Matrixmultiplikation mit Aufwand $O(v^3)$ sinkt der Aufwand der Berechnung der Zielfunktion auf $O(2+2+4v-2+4v-2) = O(8v) = O(v)$. Für die gesamte lokale Suche resultiert ein Aufwand von $O(schritte \cdot k^2 \cdot v^3)$.

Die Umsetzung dieses optimierten Ansatzes erfolgte in einer Modifikation der Dateien des letzten Kapitels zu *srg_opt.py* und dem Hilfsmodul *graph_opt.py*. Diese Modifikationen umfassen sowohl die Einführung von v, k, λ, μ und $f(G)$ (Datentyp int) sowie A, A^2 und P (Datentyp list[list[int]]) als Attribute des Graph-Objekts in *graph_opt.py*, als auch die Auslagerung der Berechnung der Zielfunktion aus *srg_naive.py* in die Funktionen add_edge() und remove_edge() aus *graph_opt.py*.

```
1     Füge (i,j) zur Kantenliste hinzu
2     A(i,j) = 1 und A(j,i) = 1
3     Schleife über alle Knotenindizes t:
4           f(G) -= |A²(t,i) - P(t,i)|; f(G) -= |A²(t,j) - P(t,j)|
5           f(G) -= |A²(i,t) - P(i,t)|; f(G) -= |A²(j,t) - P(j,t)|
6     P(i,j) und P(j,i) werden von my zu ld
7     Schleife über alle Knotenindizes t:
8           A²(i,t) += A(j,t); A²(j,t) += A(i,t)
9           A²(t,i) += A(t,j); A²(t,j) += A(t,i)
10    Schleife über alle Knotenindizes t:
11          f(G) += |A²(t,i) - P(t,i)|; f(G) -= |A²(t,j) - P(t,j)|
12          f(G) += |A²(i,t) - P(i,t)|; f(G) -= |A²(j,t) - P(j,t)|
```

Quelltext 3: Pseudocode der Berechnung der Zielfunktion aus graph_opt.py

Zeilen 2 und 6 sind die jeweils 2 Änderungen an A und P, Zeilen 7 bis 9 die $4v-2$ Änderungen an A^2. Da nur der Gesamtwert der Zielfunktion gespeichert wird (und nicht etwa die Differenzmatrix $|A^2-P|$), werden in Zeilen 3 bis 5 die alten Werte für alle betroffenen Matrixeinträge subtrahiert, dann P und A^2 angepasst und schließlich in Zeilen 10 bis 12 die neuen Werte für alle veränderten Matrixeinträge addiert.

SRG-Parameter	Ergebnis	Zeit
(5, 2, 0, 1)	SRG gefunden, $f(G) = 0$	<0,1
(9, 4, 1, 2)	SRG gefunden, $f(G) = 0$	<0,1
(10, 3, 0, 1)	SRG gefunden, $f(G) = 0$	<0,1
(15, 6, 1, 3)	lokales Optimum, $f(G) = 76$	1,0
(25, 8, 3, 2)	lokales Optimum, $f(G) = 300$	23,5

Tabelle 2: Von srg_opt.py erzielte Ergebnisse; Zeit in Sekunden

Wie zu erwarten, erzielt *srg_opt.py* mit steigenden SRG-Parametern schnellere Ergebnisse. Dass der Unterschied in der tatsächlichen Laufzeit kleiner ist als angenommen (ca. Faktor 3), liegt unter anderem daran, dass Python als in der Regel interpretierte Sprache mit dynamischer Typisierung einen hohen Overhead-Aufwand besitzt. Die Ausführung der beiden Versionen des Programms mit dem Just-in-Time-Compiler PyPy erfolgt für die Eingabe $(25{,}8{,}3{,}2)$ bei *srg_naive.py* in etwa 10,7 Sekunden und bei *srg_opt.py* in etwa 1,3 Sekunden. In dem Fall ist *srg_opt.py* also ca. 8-mal schneller.

6 Implementierung verschiedener Varianten

Aufgrund der im vergangenen Abschnitt erwähnten Probleme der Programmiersprache Python wurde für das weitere Vorgehen der optimierte Ansatz aus *srg_opt.py* und *graph_opt.py* in die Programmiersprache C „übersetzt“. Diese hat gegenüber Python eine aufwendigere und weniger übersichtliche Syntax, jedoch einen deutlich geringeren Overhead. Die Suche wird also (trotz gleichbleibender Komplexität) deutlich schneller ausgeführt. Die folgende Tabelle zeigt einen Laufzeitvergleich der verschiedenen Implementierungen am Beispiel des SRG $(25,8,3,2)$:

Implementierung	**Ergebnis**	**Schritte**	**Gesamtzeit**	**Zeit / Schritt**
srg_naive.py	$f(G) = 300$	17	62	ca. 3,65
srg_opt.py	$f(G) = 300$	17	23,5	ca. 1,38
srg_naive.py (PyPy)	$f(G) = 300$	17	10,7	ca. 0,63
srg_opt.py (PyPy)	$f(G) = 300$	17	1,3	ca. 0,08
srg_opt.c	$f(G) = 220$	22	0,4	ca. 0,02

Tabelle 3: Übersicht über die Laufzeit der verschiedenen Implementierungen; Zeit in Sekunden

srg_opt.c erreicht trotz gleicher Startlösung (siehe Abschnitt 6.1) ein anderes Ergebnis als die Python-Implementierungen. Das liegt daran, dass die Kantenmenge in Python den Datentyp list hat, in C jedoch als Array umgesetzt wurde, weshalb die Kantenindizes in C manuell gehandhabt werden müssen. Dadurch weicht die Reihenfolge der gespeicherten Kanten im Laufe der lokalen Suche ab, was sich auf die Auswahl des besten Nachbarn auswirkt.

Außerdem ist C keine objektorientierte Sprache, ein Graph und seine Eigenschaften können daher nicht wie bisher durch ein Objekt einer Klasse realisiert werden. Die Kantenmenge, sowie die für die Berechnung der Zielfunktion benötigten Matrizen werden daher als globale Variablen gespeichert.

6.1 Generierung des Startgraphen

In allen bisherigen Implementierungen erfolgte die Generierung des k-regulären Startgraphen mithilfe des gleichen Algorithmus. Erst wird ein leerer Graph erzeugt, indem die Kantenliste leer bleibt und die Matrizen A, A^2 und P, sowie der Wert der Zielfunktion $f(G)$ die jeweiligen Werte für den Fall eines leeren Graphen erhalten: $A_{ij} = 0; (A^2)_{ij} = 0; P_{ij} = k$, wenn $i = j$, sonst μ; $f(G) = v \cdot (v-1) \cdot \mu + v \cdot k$.

Anschließend werden so lange iterativ Kanten in den Graphen gelegt, bis jeder Knoten den gewünschten Grad k hat. Diese Kanten bilden visuelle „Ringe“ um den Graphen, die sich in der Differenz der Knotenindizes der Kantenendpunkte unterscheiden.

```
1     Schleife: distanz = 1; distanz < (k + 1) / 2; distanz++
2          Schleife: knoten = 0; knoten < v; knoten++
3               neue Kante von knoten zu (knoten + distanz) % v
4     falls k ungerade:
5          Schleife: knoten = 0; knoten < v / 2; knoten++
6               neue Kante von knoten zu knoten + (v / 2)
```

Quelltext 4: Pseudocode des Algorithmus zur Generation eines k-regulären Graphen auf v Knoten

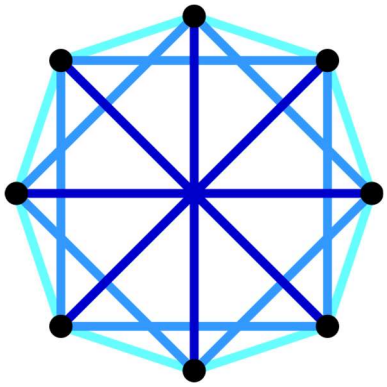

Abbildung 5: Algorithmisch generierter Startgraph mit v = 8 und k = 5; Knotenindizes analog zur Anordnung

Abbildung 5 zeigt die hinzugefügten „Ringe“ von Kanten. Hellblaue Kanten entstammen dem ersten Durchlauf der Schleife in Zeile 1 (distanz = 1) und mittelblaue Kanten dem zweiten Durchlauf (distanz = 2). In der Schleife in Zeile 5 werden die dunkelblauen Kanten hinzugefügt, um für jeden Graphen den geforderten ungeraden Grad zu erreichen.

Dieser Algorithmus hat den Vorteil, dass der Startgraph schnell berechnet werden kann. Mit einem Aufwand von $O(v)$ für das Hinzufügen einer Kante ergibt sich für die Berechnung des gesamten Graphen $O(k \cdot v^2)$. Das ist um den Faktor $k \cdot v$ schneller als ein einzelner Schritt der darauf folgenden lokalen Suche. Der große Nachteil dieses Verfahrens ist jedoch, dass (bei konstanten Eingaben v und k) immer der gleiche Startgraph generiert wird, sodass die mehrfache Ausführung der lokalen Suche immer zum gleichen Ergebnis führt. Zur Lösung dieses Problems bedarf es eines Algorithmus zur Generierung eines zufälligen k-regulären Graphen. In [9, S. 379] wird ein solcher Algorithmus beschrieben. Dessen Ansatz lautet grob:

```
1     Betrachte eine Liste U mit k Exemplaren jedes Knotenindexes
2     Schleife: Solange U nicht leer
3           Wähle zufällig zwei Einträge i und j aus U
4           Falls Kante (i,j) noch nicht im Graphen:
5                 Füge Kante (i,j) hinzu
6           Lösche i und j aus U
7     Falls resultierender Graph k-regulär:
8           Rückgabe Graph
9     sonst:
10          Gehe zurück zu Zeile 1
```

Quelltext 5: Pseudocode des Algorithmus zur Generation eines zufälligen k-regulären Graphen auf v Knoten

Der so erzeugte k-reguläre Startgraph ist im Gegensatz zum vorherigen Verfahren zufällig und hat im Best Case den gleichen Aufwand $O(k \cdot v^2)$. Jedoch hat das Verfahren einen großen Nachteil, der an Zeile 10 deutlich wird. Alle noch zu verbindenden Endpunkte werden paarweise zufällig miteinander verbunden. Dabei können die Kanten jedoch so gewählt werden, dass die letzten offenen Endpunkte keiner validen Kante entsprechen. Ist dies der Fall, wird diese Zuordnung noch einmal versucht. Durch den Zufallsaspekt ist nicht vorhersagbar, wie viele dieser Versuche durchgeführt werden müssen. Theoretisch kann es sich also um eine Endlosschleife handeln, weshalb der Aufwand im Worst Case nicht definiert ist.

Die Anzahl der von diesem Algorithmus benötigten Versuche ist scheinbar davon abhängig, wie „voll“ der Graph ist, also wie groß das Verhältnis zwischen v und k ist. Extrembeispiele dafür sind etwa der Leitersprossengraph $(30{,}1{,}0{,}0)$ und der Cocktailpartygraph $(30{,}28{,}26{,}28)$. Im Falle des Leitersprossengraphen wird im Durchschnitt ein einziger Versuch, für den Cocktailpartygraphen etwa 31000 Versuche für die Erstellung des Startgraphen benötigt[7]. Für die meisten Parameter ist die Anzahl jedoch deutlich niedriger, sodass der Algorithmus in der Praxis für die Generierung des Startgraphen problemlos verwendet werden kann[8].

[7] Durchschnitt aus jeweils 10 Messwerten

[8] So auch von Netzwerk- und Graphentheorielibraries wie NetworkX, siehe [10]

6.2 Variierung der lokalen Suche

Das Konzept der lokalen Suche kann durch verschiedene Verfahren umgesetzt werden. Welches Verfahren am besten geeignet ist, ist von Fall zu Fall unterschiedlich und hängt von der betrachteten Struktur und den Eigenschaften der dazugehörigen Zielfunktion ab. Im konkreten Fall der hier betrachteten Graphen handelt es sich um eine möglichst kleine und unveränderliche Schrittdefinition. Des Weiteren lassen sich keine allgemeinen Aussagen über das Verhalten der Zielfunktion treffen, die Zielfunktion ist also nur im lokalen Umfeld einer Lösung bestimmbar. Aufgrund dieser Eigenschaften sind einige lokale Suchverfahren nicht anwendbar und im Rahmen dieser Bachelorarbeit wird das Hill Climbing verwendet.

Dieses Verfahren kann bezüglich der Auswahl der für den nächsten Schritt benötigten Nachbarlösung variiert werden. Bisher wurden dabei in jedem Schritt alle Nachbarlösungen analysiert und diejenige ausgewählt, die den besten (niedrigsten) Zielfunktionswert aufwies. Dieser Bester-Schritt-Ansatz minimiert die Anzahl der für die Suche benötigten Schritte, allerdings müssen in jedem Schritt alle Nachbarlösungen betrachtet werden. Der Aufwand eines Schrittes ist daher immer $O(k^2 \cdot v^3)$.

Eine Alternative dazu ist der Erster-Schritt-Ansatz, implementiert in *srg_first.c*. Bei diesem werden ebenfalls alle Nachbarlösungen betrachtet, jedoch wird der Schritt bei der ersten validen (also die Zielfunktion reduzierenden) Nachbarlösung abgebrochen. Der Vorteil dieses Ansatzes ist, dass im Best Case nur sehr wenige oder sogar nur eine einzige Nachbarlösung berechnet werden müssen (Aufwand $O(v)$). Allerdings ist zu erwarten, dass die Zielfunktion in jedem Schritt weniger stark reduziert wird als beim Bester-Schritt-Ansatz. Es werden daher deutlich mehr Schritte für die Berechnung des gewünschten SRG benötigt.

Der Zufälliger-Schritt-Ansatz, implementiert in *srg_random.c*, betrachtet in jedem Schritt nur eine einzige zufällig gewählte Nachbarlösung. Da auf die restlichen Nachbarlösungen verzichtet wird, resultiert dieses Verfahren in schnellen Schritten ($O(v)$), hat aber den gleichen großen Nachteil, der auch im vorhergehenden Kapitel besprochen wurde. Ist die zufällig ausgewählte Nachbarlösung valide (der Wert ihrer Zielfunktion niedriger), bildet sie den Ausgangspunkt für den nächsten Schritt. Ist dies jedoch nicht der Fall, wird die aktuelle Lösung erneut betrachtet. Das kann zu Endlosschleifen führen.

Dieser Abschnitt diente dazu, einige Variierungsmöglichkeiten vorzustellen und auf ihre Vor- und Nachteile einzugehen. Da jeder Ansatz einen offensichtlichen Nachteil hat (hoher Rechenaufwand pro Schritt, hohe Schrittanzahl oder potenzielle Endlosschleifen), liegt die Vermutung nahe, dass man mehrere dieser Ansätze verbinden kann, um mit einem solchen Hybridansatz die „goldene Mitte“ zu finden (siehe Kapitel 8).

7 Ergebnisse

7.1 Gefundene und nicht gefundene Graphen

Die folgende Tabelle zeigt die Ergebnisse der Ausführung von *srg_opt.c* mit einem zufälligen Startgraphen und dem Bester-Schritt-Ansatz. Sie enthält alle Parameterkombinationen aus der in Abschnitt 2.2 erwähnten Parameterliste mit $v < 40$, abzüglich nachgewiesen unmöglicher SRGs wie $(21,10,4,5)$ und Komplementgraphen von enthaltenen Einträgen. So ist etwa $(28,12,6,4)$ enthalten, die Parameter seines Komplementgraphen $(28,15,6,10)$ hingegen nicht.

Parameter	Bestes Ergebnis	Schritte	Zeit	Lokale Suchen
(5,2,0,1)	gefunden	0	<0,1	1
(9,4,1,2)	gefunden	3	<0,1	1
(10,3,0,1)	gefunden	2	<0,1	1
(13,6,2,3)	gefunden	5	<0,1	3
(15,6,1,3)	gefunden	7	<0,1	4
(16,5,0,2)	gefunden	8	<0,1	1
(16,6,2,2)	gefunden	8	<0,1	2
(17,8,3,4)	gefunden	10	<0,1	11
(21,10,3,6)	gefunden	19	0,16	50
(25,8,3,2)	f(G)=64	25	0,4	>500
(25,12,5,6)	f(G)=188	16	0,4	>500
(26,10,3,4)	f(G)=188	22	0,4	>500
(27,10,1,5)	gefunden	29	0,9	3
(28,12,6,4)	gefunden	41	1,3	2
(29,14,6,7)	f(G)=280	24	1,0	>250
(35,16,6,8)	f(G)=548	33	3,4	>100
(36,10,4,2)	f(G)=680	35	3,0	>100
(36,14,4,6)	f(G)=556	40	4,7	>100
(36,14,7,4)	f(G)=936	51	6,0	>100
(36,15,6,6)	f(G)=584	34	4,2	>100
(37,18,8,9)	f(G)=536	42	6,7	>100

Tabelle 4: Ergebnisse von srg_opt.c; Spalten 2 bis 4 beziehen sich auf dieselbe lokale Suche; Zeit in Sekunden

Dass einige Graphen wie $(27,10,1,5)$ binnen weniger lokaler Suchen gefunden werden, während andere Graphen wie $(25,12,5,6)$ trotz ähnlich großer Parameter nahezu unauffindbar scheinen, liegt vermutlich am Verhältnis zwischen den lokalen und globalen Minima der Zielfunktion. Da die Startgraphen zufällig über den Suchraum verteilt sind und vergleichsweise viele dieser Startgraphen zum gesuchten SRG führen, liegt die Annahme nah, dass es im Suchraum nur wenige lokale Minima gibt. Im gleichen Zug würde eine hohe Anzahl von lokalen Minima im Suchraum im Falle von $(25,12,5,6)$ erklären, warum keiner der über 500 Startgraphen „nah genug" am globalen Minimum ist, um dieses mittels der lokalen Suche zu erreichen.

Tabelle 5 zeigt die Ergebnisse der Anwendung von *srg_opt.c* auf einige größere Parameterkombinationen, deren Existenz ein offenes Problem ist (siehe Abschnitt 2.3). Die Startgraphen wurden zufällig gewählt und die lokale Suche nach dem Bester-Schritt-Ansatz durchgeführt. Im Gegensatz zu Tabelle 4 entspricht hier jede Zeile einer lokalen Suche.

Parameter	f(G) Anfang	f(G) Ende	Schritte	Zeit	Zeit / Schritt
(69,20,7,5)	7552	3516	111	219	2,0
(69,20,7,5)	7376	3540	110	219	2,0
(69,20,7,5)	7736	3612	102	201	2,0
(85,14,3,2)	7320	3656	113	303	2,7
(85,14,3,2)	7204	3768	90	237	2,6
(85,14,3,2)	7192	3532	116	301	2,6
(99,14,1,2)	8960	4460	141	606	4,3
(99,14,1,2)	8988	4628	120	517	4,3
(99,14,1,2)	8952	4440	139	597	4,3

Tabelle 5: Ergebnisse bei Suche durch srg_opt.c nach unbestätigten Graphen; Zeit in Sekunden

Durchgeführt wurden je 3 lokale Suchen nach $(69,20,7,5)$ und $(85,14,3,2)$, den beiden kleinsten unbestätigten SRGs (gemessen an der Anzahl der Knoten), sowie nach $(99,14,1,2)$, Conways 99-Graph. Die Suchen waren erfolglos, zeigen aber, dass selbst Graphen dieser Größenordnung durch die in Kapitel 4 eingeführte und in Kapitel 5 und 6 optimierte lokale Suche bearbeitet werden können. Zum Vergleich benötigt die erste Implementierung *srg_naive.py* für Conways 99-Graph selbst mit dem Just-In-Time-Compiler PyPy für einen einzelnen Schritt der lokalen Suche etwa 1300 Sekunden.

7.2 Beobachtungen zu einigen Familien von Graphen

In Abschnitt 6.1 wurde erwähnt, dass die Leitersprossen- und Cocktailpartygraphen Grenzfälle bezüglich der Generierung des Startgraphen darstellen. Für die Leitersprossengraphen als „besonders leere“ Graphen ist demnach die Generierung des Startgraphen recht einfach, während für die „besonders vollen“ Cocktailpartygraphen viele Versuche nötig sind. In beiden Fällen bricht allerdings die anschließende lokale Suche sofort ab, denn der gesuchte SRG ist die einzige Konfiguration eines Graphen mit den entsprechenden Parametern. Somit ist der Startgraph, unabhängig vom Aufwand seiner Generierung, das Ende der lokalen Suche und es sind keine weiteren Schritte notwendig.

In Abschnitt 7.1 wurde als Schlussfolgerung aus Tabelle 4 die Vermutung aufgestellt, dass der Erfolg der lokalen Suche (das Finden des SRG anstatt eines lokalen Minimums) vom Verhältnis zwischen den globalen und lokalen Optima der Zielfunktion abhängig ist. Gibt es beispielsweise im Falle eines Graphen wie $(27{,}10{,}1{,}5)$ oder $(28{,}12{,}6{,}4)$ wenige lokale Optima, ist die Wahrscheinlichkeit höher, dass die lokale Suche ein globales Optimum findet und die Suche erfolgreich ist. Per Zufall konnte mit den vollständigen bipartiten Graphen $K_{n,n}$ eine Gruppe von Sonderfällen für diese Annahme gefunden werden. Ein solcher Graph besteht aus zwei Gruppen von je n Knoten, welche genau dann durch eine Kante verbunden sind, wenn sie zu unterschiedlichen Gruppen gehören. $K_{n,n}$ ist ein SRG mit Parametern $(2n, n, 0, n)$. Die Besonderheit bei dieser Familie von Graphen ist, dass die lokale Suche mit diesen Parametern scheinbar immer erfolgreich ist[9]. Ausgehend von der Vermutung über das Verhältnis der Optima würde das bedeuten, dass die Zielfunktion im Falle dieser Graphen keine nicht-globalen Optima besitzt.

[9] Mehrere hundert lokale Suchen für unterschiedliche n

8 Fazit und Ausblick

Das Hauptziel der Bachelorarbeit bestand darin, zu ermitteln, inwiefern das Verfahren der lokalen Suche dazu verwendet werden kann, stark reguläre Graphen zu berechnen. In Kapitel 7 wurde bewiesen, dass es durchaus möglich ist, SRGs auf diese Art und Weise zu berechnen, auch wenn der Erfolg der Suche insbesondere bei höheren Graphen aus verschiedenen Gründen oft ausbleibt.

Besonders interessant waren dabei die Erkenntnisse aus dem mittleren Teil der Arbeit. Das Ja/Nein-Kriterium aus der Definition eines SRG lässt sich so interpretieren/umwandeln, dass es die Grundlage einer Funktion bildet, die als Zielfunktion für die lokale Suche verwendet werden kann (Kapitel 4). Außerdem ist es bemerkenswert, wie stark die daraus resultierende lokale Suche durch Optimierungen hinsichtlich der Berechnung der Zielfunktion (Kapitel 5) und der allgemeinen Implementierung beschleunigt werden konnte. Während die erste intuitive Implementierung nur für sehr kleine Parameter verwendet werden konnte, waren die späteren Versionen auch in der Lage, mit deutlich größeren Parametern und insbesondere den in Abschnitt 2.3 beschriebenen ungeklärten Parametern, bzw. unbestätigten SRGs umzugehen.

Die wahrscheinlich größten Hürden des erarbeiteten Verfahren liegen wohl in der sehr grundlegenden Betrachtung der stark regulären Graphen selbst. Der Rahmen für die lokale Suche beruht hauptsächlich auf der Definition eines SRGs, sowie einigen Eigenschaften seiner Parameter. Viele Eigenschaften von SRGs, beispielsweise bezüglich der Eigenwerte seiner Adjazenzmatrix, fanden in dieser Arbeit keine Verwendung. Jedoch ist nicht auszuschließen, dass sich einige dieser Eigenschaften im diesem Rahmen als nützlich erweisen würden. Es stellt sich daher die Frage, ob man aus in dieser Arbeit nicht erwähnten Eigenschaften andere Zielfunktionen herleiten kann, um auf deren Basis womöglich effizientere lokale Suchverfahren zu konzipieren.

Das in dieser Arbeit verwendete Verfahren ist durch drei Aspekte limitiert: der Rechenaufwand eines einzelnen Schritts, die Anzahl der Schritte in der lokalen Suche und die Anzahl der lokalen Suchen, die für das Finden des SRG benötigt werden.

Die Komplexität des in Kapitel 5.2 beschriebenen optimierten Ansatzes ist $O(v)$ für das Hinzufügen/Entfernen einer Kante und damit auch $O(v)$ für die Betrachtung (inklusive Berechnung der Zielfunktion) einer Nachbarlösung. Da sich mit jeder Veränderung der Kantenmenge mehrere Zeilen und Spalten der quadrierten Adjazenzmatrix ändern, kann diese Komplexität scheinbar nicht weiter optimiert werden. Der Zeitaufwand für einen einzelnen Schritt ist dennoch weiter reduzierbar, beispielsweise durch einen in Abschnitt 6.2 erwähnten Hybridansatz. So könnte man etwa anfangs per Zufallsprinzip schnell viele Schritte machen und später zum Bester-Nachbar-Ansatz wechseln, wenn präzisere Schritte nötig sind.

Außerdem sind zwar die Schritte der lokalen Suche vom vorherigen Schritt abhängig, nicht aber die Reihenfolge, in welcher die Nachbarlösungen (innerhalb eines Schritts) betrachtet werden. Daher ließe sich die Analyse der $|E| = (v^2 \cdot k^2) / 4$ Nachbarlösungen beim Bester-Nachbar-Ansatz mittels Multithreading beschleunigen, sodass die Berechnung echt parallel auf mehreren CPU- oder auch GPU-Kernen durchgeführt wird.

Die Anzahl der für die lokale Suche benötigten Schritte ließe sich durch einen besseren Startgraph realisieren. Ein besserer Startgraph hätte einen niedrigeren Zielfunktionswert, weshalb weniger Änderungen nötig wären, um aus ihm einen SRG zu erzeugen. Da es im Rahmen dieser Arbeit nicht möglich ist, allgemeine Aussagen über das Verhalten der Zielfunktion zu treffen, ist nicht klar, wie ein solcher besser Startgraph gefunden werden kann. Daher werden hier (ab Abschnitt 6.1) nur zufällige Startgraphen verwendet.

Ebenso ist nicht offensichtlich, wie sich die Anzahl der benötigten lokalen Suchen, also das Verhältnis von erfolgreichen und Fehlversuchen, verbessern lässt. Tabelle 4 in Abschnitt 7.1 legt die Vermutung nahe, dass dieses Verhältnis unterschiedlich groß sein kann, der Mangel an Informationen über die Zielfunktion erschwert allerdings Aussagen über dessen Ursache.

Literaturverzeichnis

[1] A. E. Brouwer, H. Van Maldeghem, Strongly Regular Graphs. Cambridge, England: Cambridge University Press, 2022.

[2] R. Diestel, Graphentheorie. 4. Auflage. Berlin, Deutschland: Springer, 2010.

[3] A. Gibbons, Algorithmic Graph Theory. Cambridge, England: Cambridge University Press, 1985.

[4] D. E. Knuth, The Art of Computer Programming; volume 4, pre-fascicle 0A. A draft of section 7: Introduction to Combinatorial Searching. Addison-Wesley, 2008.

[5] C. D. Godsil, Problems in Algebraic Combinatorics, The Electronic Journal of Combinatorics 2, 1995.

[6] A. Blokhuis, A. E. Brouwer, Geodetic Graphs of diameter two, Geometriae Dedicata, volume 25, S. 527-533, 1988.

[7] A. A. Makhnev, I. M. Minakova, On Automorphisms of strongly regular graphs with parameters $\lambda = 1$, $\mu = 2$, Discrete Mathematics and Applications, volume 14, 2004.

[8] J. H. Conway, Five 1,000$ Problems (Update 2017), Online Encyclopedia of Integer Sequences, https://oeis.org/A248380/a248380.pdf, eingelesen am 12.01.2024.

[9] A. Steger and N. Wormald, Generating random regular graphs quickly, Probability and Computing 8 (1999), S. 377-396, 1999.

[10] NetworkX: Network Analysis in Python, Documentation, https://networkx.org/documentation/stable/_modules/networkx/generators/random_graphs.html#random_regular_graph, eingesehen am 13.01.2024.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit eigenständig und ohne Benutzung anderer als der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel verfasst habe. Alle Textstellen, die wortwörtlich oder sinngemäß anderen Werken oder sonstigen Quellen entnommen sind, habe ich unter Angabe der Herkunft als solche kenntlich gemacht.